МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

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 ІП-11 Головатюк В.І.

Викладач: Бардін В.

Київ 2023

**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 репозиторій:** [**click**](https://github.com/aquaprogit/LinkedList)

**Лістинг коду:**

1. Node.cs

namespace CustomCollections;

internal class Node<T>

{

public T Value { get; set; }

public Node<T>? Next { get; set; } = null;

public Node(T value)

{

Value = value;

}

public void SetNext(T value)

{

var newNode = new Node<T>(value);

if (Next == null)

{

Next = newNode;

}

else

{

var current = Next;

while (current.Next != null)

{

current = current.Next;

}

current.Next = newNode;

}

}

public int ChildrenCount

{

get

{

var current = this;

int count = 0;

while (current.Next != null)

{

current = current.Next;

count++;

}

return count;

}

}

public override string ToString()

{

return $"{Value} with {ChildrenCount}";

}

}

1. **LinkedList.cs**

using System.Collections;

namespace CustomCollections;

public class LinkedList<T> : ICollection<T>

{

private Node<T>? \_root;

public int Count { get; private set; }

public bool IsReadOnly => false;

public event Action<NotifyCollectionChangedEventArgs<T>>? CollectionChanged;

public T this[int index]

{

get

{

if (index < 0 || index >= Count)

throw new IndexOutOfRangeException(nameof(index));

var i = 0;

var current = \_root;

while (current != null)

{

if (i == index)

{

return current.Value;

}

i++;

current = current.Next;

}

throw new IndexOutOfRangeException(nameof(index));

}

set

{

if (index < 0 || index >= Count)

throw new IndexOutOfRangeException(nameof(index));

var i = 0;

var current = \_root;

while (current != null)

{

if (i == index)

{

var previousValue = current.Value;

current.Value = value;

CollectionChanged?.Invoke(new NotifyCollectionChangedEventArgs<T>(NotifyCollectionChangedAction.Update, current.Value, previousValue));

return;

}

i++;

current = current.Next;

}

}

}

public void Add(T item)

{

var old = (ICollection<T>)MemberwiseClone();

if (\_root == null)

\_root = new Node<T>(item);

else

\_root.SetNext(item);

Count++;

CollectionChanged?.Invoke(new NotifyCollectionChangedEventArgs<T>(NotifyCollectionChangedAction.Add, (ICollection<T>)MemberwiseClone(), old));

}

public void Clear()

{

\_root = null;

Count = 0;

CollectionChanged?.Invoke(new NotifyCollectionChangedEventArgs<T>(NotifyCollectionChangedAction.Clear));

}

public bool Contains(T item)

{

if (\_root == null)

return false;

var current = \_root;

while (current != null)

{

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

return true;

current = current.Next;

}

return false;

}

public void CopyTo(T[] array, int arrayIndex = 0)

{

if (array == null)

throw new ArgumentNullException(nameof(array));

if (arrayIndex < 0)

throw new ArgumentOutOfRangeException(nameof(arrayIndex));

if (array.Length - arrayIndex < Count)

throw new ArgumentException("Not enough elements after arrayIndex in the destination array.");

int index = 0;

var current = \_root;

while (current != null)

{

array[index + arrayIndex] = current.Value;

index++;

current = current.Next;

}

}

public IEnumerator<T> GetEnumerator()

{

return \_root == null

? Enumerable.Empty<T>().GetEnumerator()

: new LinkedListEnumerator<T>(ref \_root);

}

public bool Remove(T item)

{

var old = (ICollection<T>)MemberwiseClone();

if (\_root == null)

return false;

if (\_root.Value!.Equals(item))

{

\_root = \_root.Next;

Count--;

CollectionChanged?.Invoke(new NotifyCollectionChangedEventArgs<T>(NotifyCollectionChangedAction.Remove, (ICollection<T>)MemberwiseClone(), old));

return true;

}

var next = \_root.Next;

var current = \_root;

while (next != null)

{

if (next.Value!.Equals(item))

{

current.Next = next.Next;

Count--;

CollectionChanged?.Invoke(new NotifyCollectionChangedEventArgs<T>(NotifyCollectionChangedAction.Remove, (ICollection<T>)MemberwiseClone(), old));

return true;

}

next = next.Next;

}

return false;

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

using System.Collections;

namespace CustomCollections;

internal class LinkedListEnumerator<T> : IEnumerator<T>

{

private Node<T>? \_currenctNode = null;

private readonly Node<T> \_root;

public LinkedListEnumerator(ref Node<T> node)

{

\_root = node;

}

public T Current => \_currenctNode.Value;

object? IEnumerator.Current => Current;

public bool MoveNext()

{

if (\_currenctNode == null)

{

\_currenctNode = \_root;

}

else

{

if (\_currenctNode.Next == null)

return false;

\_currenctNode = \_currenctNode.Next;

}

return true;

}

public void Reset()

{

\_currenctNode = null;

}

void IDisposable.Dispose() { }

}

using System.Collections;

namespace CustomCollections;

public class NotifyCollectionChangedEventArgs<T>

{

public NotifyCollectionChangedAction Action { get; set; }

public object? NewItem { get; set; }

public object? OldItem { get; set; }

public ICollection<T>? NewItems { get; set; }

public ICollection<T>? OldItems { get; set; }

public NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction action)

{

Action = action;

}

public NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction action, T newItem, T oldItem)

: this(action)

{

NewItem = newItem;

OldItem = oldItem;

}

public NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction action, ICollection<T> newItems, ICollection<T> oldItems)

: this(action)

{

NewItems = newItems;

OldItems = oldItems;

}

public NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction action, T newItem, T oldItem, ICollection<T> newItems, ICollection<T> oldItems)

: this(action, newItem, oldItem)

{

NewItems = newItems;

OldItems = oldItems;

}

}

namespace CustomCollections;

public enum NotifyCollectionChangedAction

{

Add,

Remove,

Update,

Clear

}

**Висновок:** в ході роботи я навчився проектувати та реалізовувати узагальнені типи, впровадження підтримку подій для типу.

**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.

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.