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»

Виконала:

студентка гр. ІС-11

Гаврильчик Яна

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 4:**

****

**Посилання на код GitHub:**

<https://github.com/melancholiya/dotnet-lw1-generic.git>

Код бібліотеки:

**MyDequeNode:**

public class MyDequeNode<T>  
{  
 public T Value { get; internal set; }  
 public MyDequeNode<T> Previous { get; internal set; }  
 public MyDequeNode<T> Next { get; internal set; }  
 public MyDequeNode(T value)  
 {  
 Value = value;  
 }  
  
}

**MyDeque:**

using System;  
using System.Collections;  
using System.Collections.Generic;  
using LW1.MyCollectionLogic;  
  
  
namespace LW1.MyCollection;  
  
public class DoubleEndedQueue<T>:IList<T>  
{  
 private MyDequeNode<T> \_head;  
  
 private MyDequeNode<T> \_tail;  
  
 public MyDequeNode<T> Head  
 {  
 get => \_head;  
 set => \_head = value;  
 }  
  
 public MyDequeNode<T>Tail  
 {  
 get => \_tail;  
 set => \_tail = value;  
 }  
  
 /// <summary>  
 /// Gets the number of elements contained in the deque  
 /// </summary>  
 public int Count { get; private set; }  
 /// <summary>  
 /// Gets a value indicating whether the deque is read-only  
 /// </summary>  
 public bool IsReadOnly => false;  
 public event EventHandler<T> ElementAdded;  
 public event EventHandler<T> ElementRemoved;  
 public event EventHandler<EventArgs> CollectionCleared;  
 public event EventHandler<T> AddedToBeginning;  
 public event EventHandler<T> AddedToEnd;  
   
 public DoubleEndedQueue() { }  
 public DoubleEndedQueue(IEnumerable<T>collection)  
 {  
 if (collection is null)  
 {  
 throw new ArgumentNullException(nameof(collection));  
 }  
  
 foreach (var item in collection)  
 {  
 AddLast(item);  
 }  
 }  
   
 /// <summary>  
 /// Removes a specific item from the deque  
 /// </summary>  
 public bool Remove(T item)  
 {  
 var node = Head;  
 while (node!=null)  
 {  
 if (node.Value.Equals(item))  
 {  
 if (node == Head)  
 {  
 RemoveFirst();  
 }  
 else if (node ==Tail )  
 {  
 RemoveLast();  
 }  
   
 else  
 {  
 node.Previous.Next = node.Next;  
 node.Next.Previous = node.Previous;  
 }  
 Count--;  
 ElementRemoved?.Invoke(this,item);  
 return true;  
 }   
 node = node.Next;  
 }  
 return false;  
 }  
   
 /// <summary>  
 /// Removes and returns the item at the front of the deque  
 /// </summary>  
 /// <exception cref="InvalidOperationException">If collection is empty</exception>  
 public T RemoveFirst()  
{  
 if (Count == 0)  
 {  
 throw new InvalidOperationException("Deque is empty.");  
 }  
  
 T item = Head.Value;  
 if (Count == 1)  
 {  
 Head = null;  
 Tail = null;  
 }  
 else  
 {  
 Head=Head.Next;  
 Head.Previous = null;  
 }  
  
 Count--;  
 ElementRemoved?.Invoke(this,item);  
 return item;  
   
}  
 /// <summary>  
 /// Removes and returns the item at the end of the deque  
 /// </summary>  
 public T RemoveLast()  
 {  
 if(Count==0)  
 throw new InvalidOperationException("Deque is empty.");  
 var item = Tail.Value;  
 if (Count == 1)  
 {  
 Head = null;  
 Tail = null;  
 }  
 else  
 {  
 Tail = Tail.Previous;  
 Tail.Next = null;  
 }  
 Count--;  
 ElementRemoved?.Invoke(this,item);  
 return item;  
   
 }  
   
/// <summary>  
/// Adds an item to the front of the deque  
/// </summary>  
 public void AddFirst(T item)  
 {  
 MyDequeNode<T> newNode = new MyDequeNode<T>(item);  
 if (Count == 0)  
 {  
 Head = newNode;  
 Tail = newNode;  
 }  
 else  
 {  
 newNode.Next = Head;  
 Head.Previous = newNode;  
 Head = newNode;  
 }  
  
 Count++;  
 AddedToBeginning?.Invoke(this,item);  
 }  
  
/// <summary>  
/// Adds an item to the end of the deque  
/// </summary>  
 public void AddLast(T item)  
 {  
 MyDequeNode<T> newNode = new MyDequeNode<T>(item);  
   
 if (Count == 0)  
 {  
 Head = newNode;  
 Tail = newNode;  
 }  
 else  
 {  
 Tail.Next = newNode;  
 newNode.Previous = Tail;  
 Tail = newNode;  
 }  
  
 Count++;  
 AddedToEnd?.Invoke(this,item);  
 }  
  
private void AddItems(T value)  
{  
 if (Head == null)  
 {  
 AddFirst(value);  
 }   
 AddLast(value);  
}  
public void Add(T item)  
{  
 AddItems(item);  
 ElementAdded?.Invoke(this,item);  
}  
/// <summary>  
/// Clears the contents of the deque  
/// </summary>  
public void Clear()  
{  
 while (Head != null)  
 {  
 var nextNode = Head.Next;  
 Head = null;  
 Head = nextNode;  
 }  
   
 Count = 0;  
   
 CollectionCleared?.Invoke(this, EventArgs.Empty);  
}  
  
/// <summary>  
/// Checks if the deque contains a specific item  
/// </summary>  
public bool Contains(T item)  
{  
 MyDequeNode<T>current = Head;  
 while (current!=null)  
 {  
 if (current.Value.Equals(item))  
 {  
 return true;  
 }  
 current = current.Next;  
 }  
 return false;  
}  
/// <summary>  
/// Copies the elements of the deque to an array, starting at a particular array index  
/// </summary>  
/// <param name="array"></param>  
/// <param name="arrayIndex"></param>  
/// <exception cref="ArgumentNullException">If array is empty</exception>  
/// <exception cref="ArgumentOutOfRangeException">Value of an argument is outside the allowable range of values as defined by the invoked method</exception>  
/// <exception cref="ArgumentException">Explains the reason for the exception</exception>  
public void CopyTo(T[] array, int arrayIndex)  
{  
 if (array is null)  
 {  
 throw new ArgumentNullException(nameof(array));  
 }  
  
 if (arrayIndex < 0)  
 {  
 throw new ArgumentOutOfRangeException(nameof(arrayIndex), "Index must be non-negative");  
 }  
  
 if (array.Length - arrayIndex < Count)  
 {  
 throw new ArgumentException("The number of elements in the source deque is " +  
 "greater than the available space from arrayIndex to the " +  
 "end of the destination array.");  
 }  
  
 MyDequeNode<T>current = Head;  
 while (current!=null)  
 {  
 array[arrayIndex++] = current.Value;  
 current = current.Next;  
 }  
}  
  
private class MyEnumerator : IEnumerator<T>  
 {  
 private readonly DoubleEndedQueue<T> \_doubleEndedQueue;  
 private MyDequeNode<T> \_node;  
 private T \_currentElement;  
 public T Current => \_currentElement;  
 object IEnumerator.Current => \_currentElement;   
   
 public MyEnumerator(DoubleEndedQueue<T>doubleEndedQueue)  
 {  
 \_doubleEndedQueue = doubleEndedQueue;  
 \_currentElement = default(T);  
 \_node = doubleEndedQueue.Head;  
 }  
  
 public bool MoveNext()  
 {  
 if (\_node == null &&\_node!=\_doubleEndedQueue.Tail)  
 {  
 return false;  
 }  
 \_currentElement = \_node!.Value;  
 \_node = \_node.Next;  
 if (\_node == \_doubleEndedQueue.Head)  
 {  
 \_node = null;  
 }  
  
 return true;  
 }  
   
 public void Reset()  
 {  
 \_currentElement = default(T);  
 }  
  
 public void Dispose()  
 {   
 }  
 /\*Implement GetEnumerator using yield  
 // public IEnumerator<T> GetEnumerator()  
 // {  
 // MyDequeNode<T> current = \_head;  
 // while (current != null)  
 // {  
 // yield return current.Value;  
 // current = current.Next;  
 // }  
 \*/  
  
 }  
 public IEnumerator<T> GetEnumerator()  
 {  
 return new MyEnumerator(this);  
 }  
 IEnumerator IEnumerable.GetEnumerator()  
 {  
 return GetEnumerator();  
 }  
/// <summary>  
/// Gets the index of the first occurrence of a specific item in the deque  
/// </summary>  
 public int IndexOf(T item)  
 {  
 int index = 0;  
 MyDequeNode<T> current = Head;  
 while (current != null)  
 {  
 if (current.Value.Equals(item))  
 {  
 return index;  
 }  
  
 index++;  
 current = current.Next;  
 }  
  
 return -1;  
 }  
/// <summary>  
/// Inserts an item to the deque at the specified index  
/// </summary>  
public void Insert(int index, T item)  
{  
 if (index < 0 || index > Count)  
 {  
 throw new ArgumentOutOfRangeException(nameof(index), "Index is out of range.");  
 }  
  
 var newNode = new MyDequeNode<T>(item);  
  
 if (index == 0)  
 {  
 newNode.Next = Head;  
 Head = newNode;  
 }  
 else if (index == Count)  
 {  
 newNode.Previous = Tail;  
 Tail.Next = newNode;  
 Tail = newNode;  
 }  
 else  
 {  
 var currentNode = Head;  
 for (int i = 0; i < index - 1; i++)  
 {  
 currentNode = currentNode.Next;  
 }  
  
 newNode.Previous = currentNode;  
 newNode.Next = currentNode.Next;  
 currentNode.Next.Previous = newNode;  
 currentNode.Next = newNode;  
 }  
  
 Count++;  
}  
  
/// <summary>  
/// Removes the item at the specified index from the deque  
/// </summary>  
 public void RemoveAt(int index)  
 {  
 if (index < 0 || index >= Count)  
 {  
 throw new ArgumentOutOfRangeException(nameof(index), "Index is out of range.");  
 }  
  
 if (index == 0)  
 {  
 RemoveFirst();  
 }  
 else if (index == Count - 1)  
 {  
 RemoveLast();  
 }  
 else  
 {  
 var currentNode = Head;  
 for (int i = 0; i < index - 1; i++)  
 {  
 currentNode = currentNode.Next;  
 }  
  
 currentNode.Previous.Next = currentNode.Next;  
 currentNode.Next.Previous = currentNode.Previous;  
 ElementRemoved?.Invoke(this,currentNode.Value);  
 Count--;  
 }   
 }  
/// <summary>  
/// Accesses the item at the specified index  
/// </summary>  
 public T this[int index]  
 {  
 get  
 {  
 if (index < 0 || index >= Count)  
 {  
 throw new ArgumentOutOfRangeException("Index is out of range.");  
 }  
  
 var node = Head;  
 for (int i = 0; i < index; i++)  
 {  
 node = node.Next;  
 }  
  
 return node.Value;  
 }  
 set  
 {  
 if (index < 0 || index >= Count)  
 {  
 throw new ArgumentOutOfRangeException("Index is out of range.");  
 }  
  
 var node = Head;  
 for (int i = 0; i < index; i++)  
 {  
 node = node.Next;  
 }  
  
 node.Value = value;  
 }  
 }  
   
   
}

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