Кам’янець-Подільський національний університет  
імені Івана Огієнка

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

ЛАБОРАТОРНА РОБОТА #0106

з дисципліни «Об’єктно-орієнтоване програмування»

**Класи колекцій і протоколи ітерації**

**Виконала**:  
студентка 2-го курса  
гр. KN1-B23  
Серебрянський Артем

**Прийняв**:  
старший викладач,  
Слободянюк О.В.

Дата здачі 09.05.2025р.

Кам’янець-Подільський – 2025

**Відповіді на завдання:**

**1.1 Як у мові C# використовується узагальнення?**

Узагальнення (Generics) в C# використовуються для створення повторно використовуваних структур даних і алгоритмів, які можуть працювати з даними різних типів без втрати безпеки типів. Замість того, щоб оперувати типом object (що вимагає приведення типів і може призвести до помилок під час виконання), узагальнення дозволяють визначити параметри типу (позначаються <T>, <TKey, TValue> тощо) під час оголошення класу, структури, інтерфейсу, делегата або методу.

Під час використання узагальненого типу або методу, ви конкретизуєте ці параметри типу, вказуючи фактичний тип даних, з яким ви хочете працювати. Це забезпечує:

* **Безпеку типів:** Компілятор перевіряє, чи відповідають типи даних, що додаються до узагальненої колекції або передаються до узагальненого методу, вказаному параметру типу. Це запобігає помилкам приведення типів під час виконання.
* **Підвищення продуктивності:** Уникається необхідність упаковки (boxing) та розпаковки (unboxing) для типів значень (struct).
* **Покращення читабельності коду:** Код стає більш виразним, оскільки явно видно, з яким типом даних він працює.

**Приклад:**

Замість використання ArrayList, який зберігає об'єкти типу object, ви можете використовувати List<int>, який може зберігати лише цілі числа, забезпечуючи безпеку типів.

**1.2 Що робить ключове слово «where» при визначенні класу HumanContainer?**

Ключове слово where використовується для визначення **обмежень (constraints)** на параметри типу в узагальнених типах і методах. Якщо клас HumanContainer оголошено як узагальнений, наприклад, HumanContainer<T>, то where T: ... вказує, які вимоги повинен задовольняти тип T.

Наприклад:

* where T: class - T повинен бути посилальним типом (класом, інтерфейсом, делегатом або масивом).
* where T: struct - T повинен бути типом значення (структурою, перерахуванням або базовим вбудованим типом, таким як int, double, bool тощо).
* where T: new() - T повинен мати публічний конструктор без параметрів.
* where T: BaseClass - T повинен бути похідним від класу BaseClass або бути самим BaseClass.
* where T: IInterface - T повинен реалізовувати інтерфейс IInterface.

Ці обмеження дозволяють вам використовувати певні члени або функціональність типу T всередині узагальненого класу або методу, не турбуючись про те, що переданий тип не підтримує їх.

**1.3 З якою метою клас Human реалізує інтерфейс IComparable? Що описує цей інтерфейс?**

Клас Human реалізує інтерфейс IComparable для того, щоб об'єкти цього класу можна було **порівнювати** між собою. Це необхідно для сортування колекцій об'єктів Human, пошуку мінімального або максимального елемента, а також для використання в інших алгоритмах, що вимагають порівняння елементів.

Інтерфейс IComparable визначає **єдиний метод**:

C#

int CompareTo(object obj);

Цей метод порівнює поточний об'єкт з іншим об'єктом того ж типу, переданим як параметр. Він повинен повертати:

* Від'ємне ціле число, якщо поточний об'єкт менший за переданий об'єкт.
* Нуль, якщо поточний об'єкт дорівнює переданому об'єкту.
* Додатне ціле число, якщо поточний об'єкт більший за переданий об'єкт.

Реалізація цього інтерфейсу в класі Human визначає, за яким критерієм порівнюються два об'єкти Human (наприклад, за віком, ім'ям тощо).

**1.4 Поясніть призначення інтерфейсу IEnumerable. Які методи доведеться реалізувати для того, щоб скористатися цим інтерфейсом?**

Інтерфейс IEnumerable призначений для забезпечення **можливості ітерації** по колекції об'єктів. Він визначає, що об'єкт може надавати послідовність елементів, до яких можна послідовно звертатися (наприклад, за допомогою циклу foreach).

Інтерфейс IEnumerable містить **єдиний метод**:

C#

IEnumerator GetEnumerator();

Цей метод повертає об'єкт, який реалізує інтерфейс IEnumerator. Саме об'єкт IEnumerator відповідає за фактичну навігацію по колекції та доступ до її елементів.

Щоб скористатися інтерфейсом IEnumerable, клас повинен:

1. **Реалізувати інтерфейс IEnumerable**.
2. **Реалізувати метод GetEnumerator()**, який повертає об'єкт, що реалізує інтерфейс IEnumerator.

Інтерфейс IEnumerator визначає такі члени:

* **Current властивість (read-only):** Повертає поточний елемент у колекції.
* **MoveNext() метод:** Переміщує ітератор до наступного елемента в колекції. Повертає true, якщо переміщення було успішним, і false, якщо досягнуто кінець колекції.
* **Reset() метод (необов'язково до реалізації):** Встановлює ітератор на початкову позицію перед першим елементом колекції.

Часто для спрощення реалізації ітерації використовують **ітератори** (про які йдеться в наступному питанні) та ключове слово yield return, що автоматично генерує необхідний код для реалізації IEnumerable та IEnumerator.

**1.5 Що таке «Ітератор». Який інтерфейс описує властивості та поведінку об'єкта-ітератора? Поясніть принцип роботи ітераторів у C#.**

**Ітератор** в C# - це блок коду (метод, аксесор властивості або оператор), який використовує ключове слово yield для послідовного повернення елементів колекції. Ітератори надають спрощений спосіб реалізації патерну ітератора, дозволяючи клієнтському коду перебирати елементи колекції за допомогою циклу foreach без необхідності явно керувати станом ітерації.

Інтерфейс, який описує властивості та поведінку об'єкта-ітератора, є **IEnumerator** (а також його узагальнена версія IEnumerator<T>). Як було зазначено у попередній відповіді, цей інтерфейс визначає властивість Current та методи MoveNext() і Reset().

**Принцип роботи ітераторів у C#:**

1. Коли викликається метод, аксесор або оператор, що містить блок ітератора, код всередині нього не виконується одразу. Замість цього компілятор генерує клас, який реалізує інтерфейси IEnumerable (або IEnumerable<T>) та IEnumerator (або IEnumerator<T>).
2. При першому виклику методу GetEnumerator() (який неявно викликається циклом foreach), створюється екземпляр цього згенерованого класу і повертається.
3. Коли в циклі foreach викликається метод MoveNext() об'єкта-ітератора, виконується код у блоці ітератора до наступної інструкції yield return.
4. Значення виразу після yield return повертається як поточний елемент колекції (через властивість Current об'єкта-ітератора).
5. Виконання блоку ітератора призупиняється до наступного виклику MoveNext().
6. Коли в блоці ітератора зустрічається інструкція yield break або досягається кінець блоку, метод MoveNext() повертає false, і ітерація завершується.

Таким чином, ітератори приховують складність реалізації інтерфейсів ітерації, роблячи код більш чистим і зрозумілим.

**1.6 Поясніть принцип роботи індексатора.**

Індексатор в C# - це член класу, який дозволяє екземплярам класу індексуватися подібно до масивів. Індексатори визначаються за допомогою ключового слова this і квадратних дужок [], які містять тип індексу.

**Принцип роботи індексатора:**

1. Індексатор надає синтаксичний цукор для доступу до внутрішніх даних об'єкта за допомогою індексу (який може бути цілим числом, рядком або будь-яким іншим типом).
2. Він визначає методи get і set (або лише один з них), які виконуються при читанні або записі значення за вказаним індексом відповідно.
3. Коли ви використовуєте квадратні дужки з екземпляром класу (наприклад, myObject[index]), компілятор перетворює це на виклик відповідного методу get або set індексатора.
4. У методах get і set ви реалізуєте логіку для отримання або встановлення значення внутрішніх даних об'єкта на основі наданого індексу.

**Приклад:**

У класі, що представляє список, індексатор може використовувати ціле число як індекс для доступу до елемента за його позицією в списку. У класі, що представляє словник, індексатор може використовувати рядок як ключ для доступу до відповідного значення.

Індексатори роблять класи більш інтуїтивно зрозумілими та зручними у використанні, особливо при роботі з колекціями або об'єктами, що мають внутрішню структуру, до якої можна логічно звернутися за індексом або ключем.

**1.7 Навіщо використовується мова інтегрованих запитів (Language Integrated Query)?**

Мова інтегрованих запитів (LINQ) використовується для надання **єдиного та декларативного синтаксису** для запиту даних з різних джерел безпосередньо в коді C#.

Основні цілі використання LINQ:

* **Спрощення доступу до даних:** LINQ усуває необхідність вивчати різні синтаксиси запитів для різних джерел даних (наприклад, SQL для баз даних, XML XPath для XML-документів, спеціальні API для колекцій).
* **Підвищення читабельності та зручності супроводу коду:** Запити LINQ є більш виразними та схожими на SQL, що полегшує їх розуміння та модифікацію.
* **Забезпечення безпеки типів на етапі компіляції:** LINQ використовує узагальнення, що дозволяє компілятору перевіряти правильність запитів та типів даних, зменшуючи кількість помилок під час виконання.
* **Підтримка різних джерел даних:** LINQ-провайдери дозволяють виконувати запити до різних типів джерел даних, включаючи колекції .NET, бази даних (LINQ to SQL, LINQ to Entities), XML-документи (LINQ to XML), ADO.NET DataSets та інші.
* **Розширені можливості обробки даних:** LINQ надає широкий набір операторів для фільтрації, сортування, групування, проектування та об'єднання даних.

**1.8 Наведіть приклад відкладених запитів та тих, що виконуються одразу, у мові LINQ;**

У LINQ існує два основних типи виконання запитів: **відкладене (deferred)** та **негайне (immediate)**.

**Відкладені запити:**

Відкладені запити не виконуються одразу після їх визначення. Замість цього вони зберігають логіку запиту, яка буде виконана лише тоді, коли ви почнете ітеруватися по результату запиту (наприклад, у циклі foreach або при виклику методів, що вимагають негайної оцінки).

**Приклад відкладеного запиту:**

C#

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

var evenNumbers = numbers.Where(n => n % 2 == 0); // Запит не виконується зараз

// Запит виконується лише при ітерації

foreach (var number in evenNumbers)

{

Console.WriteLine(number);

}

Оператор Where() є відкладеним. Він повертає об'єкт, який знає, як фільтрувати елементи колекції numbers, але фактична фільтрація відбувається лише під час ітерації по evenNumbers.

**Негайні запити:**

Негайні запити виконуються одразу після їх визначення, і їх результат зберігається в новій колекції. Оператори, що викликають негайне виконання, зазвичай повертають конкретне значення або нову колекцію.

**Приклад негайного запиту:**

C#

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

List<int> evenNumbersList = numbers.Where(n => n % 2 == 0).ToList(); // Запит виконується одразу, результат у evenNumbersList

int firstEvenNumber = numbers.First(n => n % 2 == 0); // Запит виконується одразу, повертається перше парне число

Оператори ToList(), ToArray(), First(), Last(), Single(), Count(), Sum(), Average() та багато інших викликають негайне виконання запиту.

**1.9 У чому переваги відкладених запитів?**

Переваги відкладених запитів:

* **Ефективність:** Запит виконується лише тоді, коли його результати дійсно потрібні. Якщо вам потрібна лише частина результатів, не буде оброблятися вся колекція.
* **Композиція запитів:** Можна будувати складні запити, об'єднуючи кілька відкладених операторів, і вони будуть оптимізовані для одноразового виконання.
* **Відображення змін:** Якщо ви модифікуєте вихідну колекцію після визначення відкладеного запиту, зміни будуть відображені при наступній ітерації по результату запиту.

**1.10 Як LINQ використовує лямбда-вирази?**

LINQ активно використовує **лямбда-вирази** для представлення делегатів (анонімних методів). Багато стандартних операторів LINQ (такі як Where, Select, OrderBy тощо) приймають делегати як параметри, які визначають логіку операції (наприклад, умову фільтрації, спосіб проектування елементів, критерій сортування).

Лямбда-вирази надають компактний синтаксис для написання цих делегатів без необхідності явно оголошувати окремі методи.

**Приклад:**

У виразі numbers.Where(n => n % 2 == 0), n => n % 2 == 0 є лямбда-виразом.

* n - це вхідний параметр (представляє кожен елемент колекції numbers).
* => - це оператор лямбда.
* n % 2 == 0 - це тіло лямбда-виразу, яке повертає булеве значення (істина, якщо число парне, інакше - хиба).

Цей лямбда-вираз передається методу Where як делегат, який визначає, які елементи повинні бути включені до результату.

**1.11 Поясніть принцип роботи всіх LINQ-операцій, які у прикладі.**

Оскільки ви не надали приклад коду LINQ, я поясню принцип роботи деяких найпоширеніших LINQ-операцій, які могли б бути використані в прикладі:

* **Where(predicate):** Фільтрує послідовність на основі заданого предиката (функції, яка повертає булеве значення для кожного елемента). Повертає нову послідовність, що містить лише ті елементи, для яких предикат поверну

**Варіант: 2**Створіть узагальнений клас CollectionType<T>. Визначити в класі конструктори, методи додавання та видалення елементів, інші необхідні методи та, якщо потрібно, перевантажені операції. Визначити індексатори та властивості. CollectionType можна реалізувати на основі стандартних колекцій (List, Stack, Array тощо).  
**Покликання на Git Hub:**

[**https://github.com/Arhangell228/-0106---**](https://github.com/Arhangell228/-0106---)

**Код програми коли все заддано:**using System;

using System.Collections;

using System.Collections.Generic;

public class CollectionType<T> : IEnumerable<T>

{

private List<T> \_items;

public CollectionType()

{

\_items = new List<T>();

}

public CollectionType(IEnumerable<T> collection)

{

\_items = new List<T>(collection);

}

public int Count => \_items.Count;

public void Add(T item)

{

\_items.Add(item);

}

public bool Remove(T item)

{

return \_items.Remove(item);

}

public T this[int index]

{

get

{

if (index < 0 || index >= \_items.Count)

throw new IndexOutOfRangeException("Недiйсний iндекс.");

return \_items[index];

}

set

{

if (index < 0 || index >= \_items.Count)

throw new IndexOutOfRangeException("Недiйсний iндекс.");

\_items[index] = value;

}

}

public static CollectionType<T> operator +(CollectionType<T> a, CollectionType<T> b)

{

return new CollectionType<T>(a.\_items.Concat(b.\_items));

}

public bool Contains(T item)

{

return \_items.Contains(item);

}

public IEnumerator<T> GetEnumerator()

{

return \_items.GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

class Program

{

static void Main()

{

var col1 = new CollectionType<int>();

col1.Add(1);

col1.Add(2);

col1.Add(3);

var col2 = new CollectionType<int>();

col2.Add(4);

col2.Add(5);

var merged = col1 + col2;

Console.WriteLine("Об'єднана колекцiя:");

foreach (var item in merged)

{

Console.WriteLine(item);

}

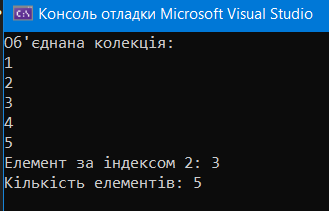
Console.WriteLine($"Елемент за iндексом 2: {merged[2]}");

Console.WriteLine($"Кiлькiсть елементiв: {merged.Count}");

}

}

**Результат:**

****

**Варіант: 3**Візьміть, створений тип (клас) з лабораторної №1, та реалізувати в ньому інтерфейс IComparable<T>. Використовуйте цей клас як параметр вашого узагальненого класу. Створіть кілька колекцій. Виконайте сортування, LINQ-запити відповідно до варіанта.  
**Покликання на Git Hub:**

[**https://github.com/Arhangell228/-0106---**](https://github.com/Arhangell228/-0106---)

**Код програми коли все заддано:**using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text; // Для підтримки української мови в консолі

// Узагальнений клас CollectionType<T>

public class CollectionType<T> : IEnumerable<T>

{

private List<T> \_items;

public CollectionType()

{

\_items = new List<T>();

}

public CollectionType(IEnumerable<T> collection)

{

if (collection == null)

throw new ArgumentNullException(nameof(collection));

\_items = new List<T>(collection);

}

public int Count => \_items.Count;

public void Add(T item)

{

\_items.Add(item);

}

public bool Remove(T item)

{

return \_items.Remove(item);

}

public void Clear()

{

\_items.Clear();

}

public bool Contains(T item)

{

return \_items.Contains(item);

}

public T this[int index]

{

get

{

if (index < 0 || index >= \_items.Count)

throw new IndexOutOfRangeException("Недійсний індекс.");

return \_items[index];

}

set

{

if (index < 0 || index >= \_items.Count)

throw new IndexOutOfRangeException("Недійсний індекс.");

\_items[index] = value;

}

}

public static CollectionType<T> operator +(CollectionType<T> a, CollectionType<T> b)

{

return new CollectionType<T>(a.\_items.Concat(b.\_items));

}

public IEnumerator<T> GetEnumerator()

{

return \_items.GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

public override string ToString()

{

return string.Join(", ", \_items);

}

}

// Клас Student з реалізацією IComparable

public class Student : IComparable<Student>

{

public string Name { get; set; }

public int Age { get; set; }

public double GPA { get; set; }

public int CompareTo(Student other)

{

if (other == null) return 1;

return GPA.CompareTo(other.GPA);

}

public override string ToString()

{

return $"{Name}, Вік: {Age}, Середній бал (GPA): {GPA}";

}

}

// Демонстрація в Main

class Program

{

static void Main()

{

// Увімкнення підтримки української мови в консолі

Console.OutputEncoding = Encoding.UTF8;

Console.InputEncoding = Encoding.UTF8;

var students1 = new CollectionType<Student>();

students1.Add(new Student { Name = "Аліса", Age = 20, GPA = 3.5 });

students1.Add(new Student { Name = "Богдан", Age = 22, GPA = 3.2 });

students1.Add(new Student { Name = "Чарлі", Age = 21, GPA = 3.8 });

var students2 = new CollectionType<Student>();

students2.Add(new Student { Name = "Діана", Age = 23, GPA = 3.9 });

students2.Add(new Student { Name = "Єва", Age = 20, GPA = 3.6 });

var allStudents = students1 + students2;

Console.WriteLine("== Усі студенти ==");

foreach (var s in allStudents)

Console.WriteLine(s);

Console.WriteLine("\n== Відсортовано за середнім балом (GPA) ==");

var sorted = allStudents.OrderBy(s => s).ToList();

foreach (var s in sorted)

Console.WriteLine(s);

Console.WriteLine("\n== Студенти з GPA > 3.5 ==");

var topStudents = allStudents.Where(s => s.GPA > 3.5);

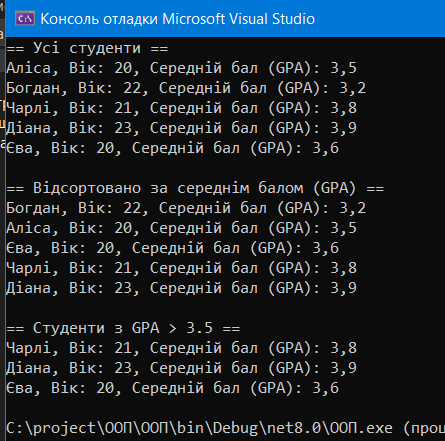
foreach (var s in topStudents)

Console.WriteLine(s);

}

}

**Результат:**

****

**Варіант: 4**Виконайте кілька складних LINQToObject запитів (мінімум 5) до колекції об'єктів, використовуючи одночасно більше трьох операцій (приклад: where + select + orderBy, first + any + min).  
**Покликання на Git Hub:**

[**https://github.com/Arhangell228/-0106---**](https://github.com/Arhangell228/-0106---)

**Код програми коли все заддано:**using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text; // Додано для Encoding

public class CollectionType<T> : IEnumerable<T>, IComparable<CollectionType<T>>

{

private List<T> \_items;

public CollectionType() => \_items = new List<T>();

public CollectionType(IEnumerable<T> collection) => \_items = new List<T>(collection);

public int Count => \_items.Count;

public void Add(T item) => \_items.Add(item);

public bool Remove(T item) => \_items.Remove(item);

public T this[int index]

{

get => \_items[index];

set => \_items[index] = value;

}

public IEnumerator<T> GetEnumerator() => \_items.GetEnumerator();

IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

public int CompareTo(CollectionType<T> other) => Count.CompareTo(other.Count);

}

class Program

{

static void Main()

{

Console.OutputEncoding = Encoding.UTF8; // Додано

Console.InputEncoding = Encoding.UTF8; // Додано

var numbers = new CollectionType<int> { 10, 5, 8, 20, 15, 3, 7, 12, 6, 25 };

// \*\*1. Вибір парних чисел, відсортованих за зростанням, і перевірка, чи є такі\*\*

var evenNumbers = numbers

.Where(x => x % 2 == 0)

.OrderBy(x => x)

.ToList();

bool hasEvenNumbers = evenNumbers.Any();

Console.WriteLine($"Парні числа (відсортовані): {string.Join(", ", evenNumbers)}");

Console.WriteLine($"Чи є парні числа? {hasEvenNumbers}");

// \*\*2. Перше число > 10, але менше 20, або -1, якщо такого немає\*\*

int firstInRange = numbers

.FirstOrDefault(x => x > 10 && x < 20, -1);

Console.WriteLine($"Перше число в діапазоні (10-20): {firstInRange}");

// \*\*3. Сума всіх чисел, які більші за середнє значення\*\*

double avg = numbers.Average();

var sumAboveAvg = numbers

.Where(x => x > avg)

.Sum();

Console.WriteLine($"Середнє: {avg}, Сума чисел > середнього: {sumAboveAvg}");

// \*\*4. Вибір унікальних чисел, відсортованих за спаданням, і перевірка на наявність дублікатів\*\*

var uniqueSorted = numbers

.Distinct()

.OrderByDescending(x => x)

.ToList();

bool hasDuplicates = numbers.Count() != uniqueSorted.Count;

Console.WriteLine($"Унікальні числа (спадання): {string.Join(", ", uniqueSorted)}");

Console.WriteLine($"Чи є дублікати? {hasDuplicates}");

// \*\*5. Мінімальне та максимальне число серед тих, що діляться на 5\*\*

var divisibleBy5 = numbers

.Where(x => x % 5 == 0)

.ToList();

int minDiv5 = divisibleBy5.Min();

int maxDiv5 = divisibleBy5.Max();

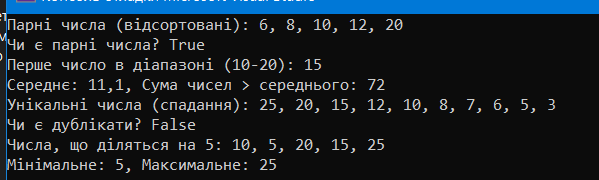
Console.WriteLine($"Числа, що діляться на 5: {string.Join(", ", divisibleBy5)}");

Console.WriteLine($"Мінімальне: {minDiv5}, Максимальне: {maxDiv5}");

}

}

**Результат:**

**  
Варіант: 5**Створіть узагальнену стандартну колекцію з простору імен System.Collections вказану у варіанті з рядками та виконайте введення-виведення, пошук рядків, що містять певне значення, підрахунок кількості рядків довжини n, сортування у зростаючому та спадному порядку.  
**Покликання на Git Hub:**

[**https://github.com/Arhangell228/-0106---**](https://github.com/Arhangell228/-0106---)

**Код програми коли все заддано:**using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

class Program

{

static void Main()

{

Console.OutputEncoding = Encoding.UTF8;

Console.InputEncoding = Encoding.UTF8;

List<string> strings = new List<string>();

Console.WriteLine("Введіть кількість рядків:");

int n = int.Parse(Console.ReadLine());

Console.WriteLine($"Введіть {n} рядків:");

for (int i = 0; i < n; i++)

{

string input = Console.ReadLine().Trim();

Console.WriteLine($"(DEBUG) Довжина рядка {i + 1}: {input.Length}");

strings.Add(input);

}

Console.WriteLine("\nВведіть значення для пошуку в рядках:");

string searchValue = Console.ReadLine();

var foundStrings = strings.Where(s => s.Contains(searchValue)).ToList();

Console.WriteLine($"\nРядки, що містять '{searchValue}':");

foreach (var s in foundStrings)

{

Console.WriteLine(s);

}

Console.WriteLine("\nВведіть довжину рядка для підрахунку:");

int length = int.Parse(Console.ReadLine());

int count = strings.Count(s => s.Length == length);

Console.WriteLine($"\nКількість рядків, які мають довжину {length} символів: {count}");

var ascendingOrder = strings.OrderBy(s => s).ToList();

Console.WriteLine("\nРядки у зростаючому порядку:");

foreach (var s in ascendingOrder)

{

Console.WriteLine(s);

}

var descendingOrder = strings.OrderByDescending(s => s).ToList();

Console.WriteLine("\nРядки у спадному порядку:");

foreach (var s in descendingOrder)

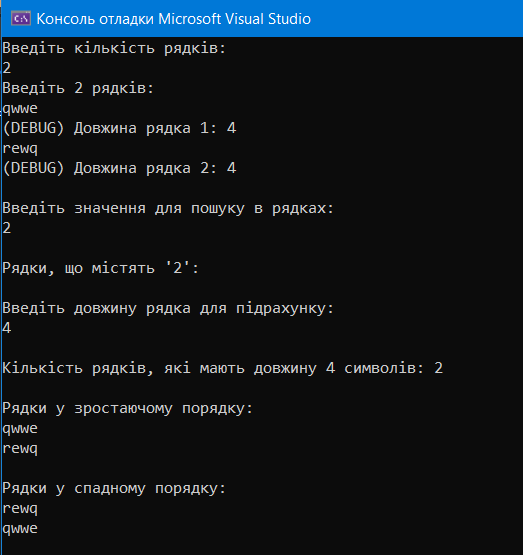
{

Console.WriteLine(s);

}

}

}

**Результат:  
**

**Варіант: 9**створити масив об'єктів CollectionType, запити – знайти колекції з негативними елементами (вибрати будь-яке поле об'єкта), знайти максимальну та мінімальну колекцію в масиві, що містить зазначений елемент. Узагальнена колекція - Dictionary<T>.

**Покликання на Git Hub:**

[**https://github.com/Arhangell228/-0106---**](https://github.com/Arhangell228/-0106---)

**Код програми коли все заддано:**using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

public class CollectionType

{

public string Name { get; set; }

public Dictionary<int, int> Items { get; set; }

public CollectionType(string name, Dictionary<int, int> items)

{

Name = name;

Items = items;

}

public override string ToString()

{

return $"Колекція: {Name}, Елементи: [{string.Join(", ", Items.Select(kv => $"{kv.Key}:{kv.Value}"))}]";

}

}

class Program

{

static void Main(string[] args)

{

Console.OutputEncoding = Encoding.UTF8;

Console.InputEncoding = Encoding.UTF8;

CollectionType[] collections = new CollectionType[]

{

new CollectionType("Колекція1", new Dictionary<int, int> { {1, 5}, {2, -3}, {3, 10} }),

new CollectionType("Колекція2", new Dictionary<int, int> { {1, 2}, {2, 7}, {3, -8}, {4, 4} }),

new CollectionType("Колекція3", new Dictionary<int, int> { {1, 1}, {2, 6} }),

new CollectionType("Колекція4", new Dictionary<int, int> { {1, -1}, {2, -5}, {3, 3}, {4, 9}, {5, -2} })

};

Console.WriteLine("Колекції з від'ємними значеннями:");

var collectionsWithNegatives = collections

.Where(c => c.Items.Values.Any(v => v < 0))

.ToList();

if (collectionsWithNegatives.Any())

{

collectionsWithNegatives.ForEach(c => Console.WriteLine(c));

}

else

{

Console.WriteLine("Колекцій з від'ємними значеннями не знайдено.");

}

int targetKey = 3;

Console.WriteLine($"\nКолекції, що містять ключ {targetKey}:");

var collectionsWithTargetKey = collections

.Where(c => c.Items.ContainsKey(targetKey))

.ToList();

if (collectionsWithTargetKey.Any())

{

var maxCollection = collectionsWithTargetKey

.OrderByDescending(c => c.Items.Count)

.FirstOrDefault();

Console.WriteLine($"Максимальна колекція: {maxCollection}");

var minCollection = collectionsWithTargetKey

.OrderBy(c => c.Items.Count)

.FirstOrDefault();

Console.WriteLine($"Мінімальна колекція: {minCollection}");

}

else

{

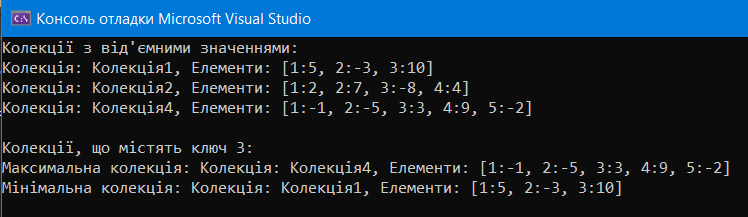
Console.WriteLine($"Колекцій з ключем {targetKey} не знайдено.");

}

}

}

**Результат:**

****