Кам’янець-Подільський національний університет імені Івана Огієнка

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

Навчальна дисципліна «Об’єктно-орієнтоване програмування»

ЛАБОРАТОРНА РОБОТА #0106

Тема:

Класи колекцій і протоколи ітерації

Варіант №3

**Виконала**:  
студентка 1-го курсу  
групи KNms1-B24  
Некрасова Ю.Д.

**Прийняв**:  
Слободянюк О.В.

Кам’янець-Подільський – 2025

1. **Короткі теоретичні відомості**

У платформі .NET колекціями називаються структури, що дозволяють зберігати та опрацьовувати набори об'єктів. Хоча масиви (Array) також вважаються колекціями, вони є статичними за своєю природою — після створення їх розмір не можна змінити. Для більш гнучкої роботи з даними в .NET передбачено багато класів колекцій, які дозволяють зручно оперувати як списками елементів, так і парами ключ-значення. Усі вони зосереджені переважно в просторі імен System.Collections, а також у його розширенні System.Collections.Specialized.

Клас ArrayList є динамічною альтернативою звичайному масиву. Його можна автоматично розширювати під час додавання нових елементів. Ще одним класом є Stack, який працює за принципом «останній прийшов — перший пішов» (LIFO). Це означає, що останній доданий об’єкт буде вилучений першим. Колекція Queue натомість реалізує чергу за принципом FIFO, де першим буде видалено той елемент, який був доданий найпершим. Ці колекції не дозволяють індексованого доступу — користувачі не можуть звертатися до конкретного елемента за індексом.

Інші структури працюють з парами ключ-значення. Наприклад, SortedList зберігає елементи у вигляді пар, сортує їх за ключами та дозволяє звертатися як за ключем, так і за індексом. Клас Hashtable також зберігає пари ключ-значення, однак без сортування, і обидва — ключ і значення — мають тип object, що дозволяє зберігати елементи будь-яких типів, але потребує явного приведення при зчитуванні. Такий підхід створює ризик помилки типізації та знижує продуктивність через операції упаковки та розпаковки структурних типів.

Для подолання цих проблем у .NET з версії 2.0 було впроваджено узагальнені типи (Generics), які дозволяють створювати класи, методи та інтерфейси з параметрами типу. Завдяки Generics забезпечується безпека типів: наприклад, у колекцію List<int> не можна помістити рядкове значення, а спроба зчитати елемент як тип string спричинить помилку на етапі компіляції. Крім того, усувається потреба у приведенні типів і підвищується швидкодія — зокрема, при роботі з структурами, де упаковка (boxing) і розпаковка (unboxing) є дорогими за ресурсами.

Сучасні generic-колекції представлені, зокрема, такими типами: List<T>, що є типобезпечною версією ArrayList; Dictionary<TKey, TValue>, аналог Hashtable, але з фіксованими типами для ключів і значень; SortedList<TKey, TValue> — сортований словник; а також Stack<T> і Queue<T>, які працюють з типобезпекою в рамках відповідних структур даних. Microsoft офіційно рекомендує надавати перевагу узагальненим колекціям у всіх можливих випадках, залишаючи неузагальнені колекції лише для ситуацій, де потрібно працювати з об'єктами різних типів у межах однієї структури.

Розширення System.Collections.Specialized пропонує менш відомі, але іноді дуже корисні колекції. Наприклад, ListDictionary подібний до Hashtable, але значно ефективніший при невеликій кількості елементів (до 10). Клас HybridDictionary поєднує в собі ListDictionary і Hashtable, автоматично перемикаючись з одного на інший при збільшенні кількості елементів. Ці спеціалізовані колекції особливо ефективні при роботі з налаштуваннями, конфігураціями або невеликими обсягами даних.

Є також класи, що зручно працюють зі строковими значеннями. NameValueCollection дозволяє зберігати множину значень під одним ключем. StringCollection — це список рядків із підтримкою дублювання та null-значень, тоді як StringDictionary — словник, ключі якого автоматично зводяться до нижнього регістру.

Для ітерації через елементи колекцій найзручніше використовувати цикл foreach. Щоб підтримувати такий спосіб ітерації, потрібно реалізувати інтерфейси IEnumerable та IEnumerator. Проте ще зручніше створювати власні ітератори з використанням ключового слова yield. Такі ітератори можуть повертати значення поступово, і використовуються переважно в методах, які повертають IEnumerable. Це дозволяє організовувати потужні та зручні механізми перебору елементів навіть у складних структурах.

Окрему увагу заслуговує LINQ — технологія інтегрованих запитів, яка дозволяє звертатися до колекцій у вигляді запитів, подібних до SQL. Наприклад, можна написати: var selected = from item in collection where item > 10 select item;. LINQ працює не лише з колекціями в пам’яті, але й з базами даних, XML-документами та іншими джерелами даних, що робить його універсальним інструментом для обробки даних у межах єдиного синтаксису.

Узагальнені типи можна створювати не лише для класів, а й для методів, інтерфейсів, делегатів. Їх можна параметризувати кількома типами та накладати обмеження на ці типи — наприклад, змушуючи їх бути структурами, мати конструктори за замовчуванням або реалізовувати певні інтерфейси. Завдяки цьому generic-код стає ще гнучкішим, безпечнішим і ефективнішим.

1. **Приклад**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Lab4

{

    class Program

    {

        static void Main()

        {

            try

            {

                var st1 = new Student

                {

                    Weight = 60,

                    Height = 190,

                    FirstName = "Marie",

                    LastName = "Little",

                    University = "BSTU"

                };

                var st2 = new Student

                {

                    Weight = 54,

                    Height = 172,

                    FirstName = "Sue",

                    LastName = "Jackson",

                    University = "BSTU"

                };

                var st3 = new Student

                {

                    Weight = 54,

                    Height = 181,

                    FirstName = "Lance",

                    LastName = "Knight",

                    University = "BSU"

                };

                var st4 = new Student

                {

                    Weight = 78,

                    Height = 184,

                    FirstName = "Lance",

                    LastName = "Stepth",

                    University = "BSU"

                };

                var st5 = new Student

                {

                    Weight = 81,

                    Height = 184,

                    FirstName = "Wesley",

                    LastName = "Jackson",

                    University = "BSTU"

                };

                var wr1 = new Worker

                {

                    Weight = 67,

                    Height = 190,

                    FirstName = "Douglas",

                    LastName = "Collins",

                    Salary = 578.4

                };

                var wr2 = new Worker

                {

                    Weight = 67,

                    Height = 190,

                    FirstName = "Lynn",

                    LastName = "Gibson",

                    Salary = 976.5

                };

                var wr3 = new Worker

                {

                    Weight = 55,

                    Height = 172,

                    FirstName = "Olivi",

                    LastName = "Smith",

                    Salary = 493

                };

                var container1 = new HumanContainer<Human> { st1, st2, wr1, wr2 };

                container1.Remove(wr2);

                container1.Remove(st1);

                //container1[-1] = st1;

                //container1[6] = st1;

                //container1[1] = st1;

                foreach (var human in container1)

                {

                    Console.WriteLine(human.ToString());

                }

                var container2 = new HumanContainer<Human>();

                container2.Add(st3);

                container2.Add(st4);

                container2.Add(st5);

                container2.Add(wr3);

                container2.Sort();

                foreach (var human in container2)

                {

                    Console.WriteLine(human.ToString());

                }

                var list = new List<HumanContainer<Human>>();

                list.Add(container1);

                list.Add(container2);

                //orderBy

                Console.WriteLine("\nLinq To objects: OrderBy, ThenBy");

                var orderRes = container1.OrderBy(h => h.Height).ThenBy(h => h.Weight);

                foreach (var human in orderRes)

                    Console.WriteLine(human);

                //where

                Console.WriteLine("\nLinq To objects: Where");

                var whereRes = container1.Where(h => (h.Height > 170 && h.Weight >= 58) || h.FullName.StartsWith("L"));

                foreach (var human in whereRes)

                    Console.WriteLine(human.ToString());

                //select

                Console.WriteLine("\nLinq To objects: Select");

                var selectRes = container1.Select((h, i) => new { Index = i + 1, h.FullName });

                foreach (var el in selectRes)

                {

                    Console.WriteLine(el);

                }

                //selectMany

                Console.WriteLine("\nLinq To objects: SelectMany");

                var selectManyRes = container1.SelectMany(h => h.FullName.Split(' '));

                foreach (var el in selectManyRes)

                    Console.WriteLine(el);

                //Skip

                Console.WriteLine("\nLinq To objects: Skip");

                var skipRes = container1.Skip(2);

                foreach (var human in skipRes)

                {

                    Console.WriteLine(human);

                }

                //SkipWhile

                Console.WriteLine("\nLinq To objects: SkipWhile");

                var skipWhileRes = container1.SkipWhile(h => h.Height < 190);

                foreach (var human in skipWhileRes)

                {

                    Console.WriteLine(human);

                }

                //Take

                Console.WriteLine("\nLinq To objects: Take");

                var takeRes = container1.Take(2);

                foreach (var human in takeRes)

                {

                    Console.WriteLine(human);

                }

                //TakeWhile

                Console.WriteLine("\nLinq To objects: TakeWhile");

                var takeWhileRes = container1.TakeWhile(h => h.Height < 190);

                foreach (var human in takeWhileRes)

                {

                    Console.WriteLine(human);

                }

                //Concat

                Console.WriteLine("\nLinq To objects: Concat");

                var concatRes = container1.Concat(container2);

                foreach (var human in concatRes)

                {

                    Console.WriteLine(human);

                }

                //GroupBy

                Console.WriteLine("\nLinq To objects: GroupBy");

                var groupByRes = concatRes.Where(h => h is Student).GroupBy(h => ((Student)h).University);

                foreach (var group in groupByRes)

                {

                    Console.WriteLine($"Group: {group.Key}, Count: {group.Count()}");

                    foreach (var human in group) Console.WriteLine(human);

                }

                //First

                Console.WriteLine("\nLinq To objects: First");

                var firstRes = concatRes.First(h => h.FullName.Length > 12);

                Console.WriteLine(firstRes);

                //FirstOrDefault

                Console.WriteLine("\nLinq To objects: FirstOrDefault");

                var firstOrDefRes = concatRes.FirstOrDefault(h => h.FullName.Length > 14);

                if (firstOrDefRes != null)

                    Console.WriteLine();

                //DefaultIfEmpty

                Console.WriteLine("\nLinq To objects: DefaultIfEmpty");

                var defaultIfEmptyRes = container2.Where(c => c.FirstName == "Eleanor")

                .DefaultIfEmpty(new Human

                {

                    FirstName = "Eleanor",

                    LastName = "Fuller"

                })

                .First();

                Console.WriteLine(defaultIfEmptyRes);

                //Min

                Console.WriteLine("\nLinq To objects: Min");

                var minRes = container1.Min(h => h.Weight);

                Console.WriteLine(minRes);

                //Max

                Console.WriteLine("\nLinq To objects: Max");

                var maxRes = container1.Max(h => h.Height);

                Console.WriteLine(maxRes);

                //Join

                Console.WriteLine("\nLinq To objects: Join");

                var joinRes = container1.Join(container2, o => o.Height, i => i.Height, (o, i) => new Human

                {

                    FirstName = o.FirstName + " " + i.FirstName,

                    LastName = o.LastName + " " + i.LastName,

                    Height = o.Height,

                    Weight = (o.Weight + i.Weight) / 2

                });

                foreach (var human in joinRes)

                    Console.WriteLine(human);

                //GroupJoin

                Console.WriteLine("\nLinq To objects: GroupJoin");

                var groupJoinRes = container2.GroupJoin(container2, o => o.Height, i => i.Height, (o, i) => new

                {

                    FullName = $"{o.FirstName} {o.LastName}",

                    Count = i.Count(),

                    TotalWeight = i.Sum(s => s.Weight)

                });

                foreach (var human in groupJoinRes)

                {

                    Console.WriteLine($"{human.FullName}: Count = {human.Count}, TotalWeight: {human.TotalWeight}");

                }

                //All and Any

                Console.WriteLine("\nLinq To objects: All/Any");

                var allAnyRes = list.First(c => c.All(h => h.Height > 160) && c.Any(h => h is Worker))

                .Select(h => h.FirstName)

                .OrderByDescending(s => s);

                foreach (var name in allAnyRes)

                    Console.WriteLine(name);

                //Contains

                Console.WriteLine("\nLinq To objects: Contains");

                var containsRes = list.Where(c => c.Contains(wr3))

                .SelectMany(c => c.SelectMany(h => h.FullName.Split(' ')))

                .Distinct()

                .OrderBy(s => s)

                .ToList();

                foreach (var name in containsRes)

                    Console.WriteLine(name);

            }

            catch (Exception ex)

            {

                Console.WriteLine(ex.Message);

            }

        }

    }

    public interface IHuman

    {

        string FirstName { get; set; }

        string LastName { get; set; }

        int Height { get; set; }

        double Weight { get; set; }

    }

    public class Human : IHuman, IComparable<Human>

    {

        #region Propeties

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public int Height { get; set; }

        public double Weight { get; set; }

        public string FullName

        {

            get { return string.Format("{0} {1}", FirstName, LastName); }

        }

        #endregion

        #region Methods

        public int CompareTo(Human other)

        {

            return string.Compare(other.FullName, FullName, StringComparison.InvariantCultureIgnoreCase);

        }

        public override string ToString()

        {

            return string.Format("Class Human: \n FullName: {0}, Height: {1}, Width: {2}", FullName,

            Height, Weight);

        }

        #endregion

    }

    public class Worker : Human

    {

        #region Properties

        public double Salary { get; set; }

        #endregion

        #region Methods

        public void DoWork() { }

        public override string ToString()

        {

            return string.Format(

            "Class Worker: \n FullName: {0}, Height: {1}, Width: {2}, Salary: {3}",

            FullName,

            Height,

            Weight,

            Salary

            );

        }

        #endregion

    }

    public class Student : Human

    {

        #region Properties

        public string University { get; set; }

        #endregion

        #region Methods

        public void DoStudy() { }

        public override string ToString()

        {

            return string.Format(

            "Class Student: \n FullName: {0}, Height: {1}, Width: {2}, University: {3}",

            FullName,

            Height,

            Weight,

            University

            );

        }

        #endregion

    }

    public class HumanContainer<T> : IEnumerable<T> where T : Human

    {

        #region Fields

        private readonly List<T> \_container;

        #endregion

        #region Constructors

        public HumanContainer()

        {

            \_container = new List<T>();

        }

        #endregion

        #region Properties

        public int Count

        {

            get { return \_container.Count; }

        }

        #endregion

        #region Indexers

        public T this[int index]

        {

            get

            {

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

                    throw new IndexOutOfRangeException();

                return \_container[index];

            }

            set

            {

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

                    throw new IndexOutOfRangeException();

                \_container[index] = value;

            }

        }

        #endregion

        #region Methods

        public T GetByName(string name)

        {

            return

            \_container.FirstOrDefault(

            h => string.Compare(h.FirstName, name, StringComparison.InvariantCultureIgnoreCase) == 0);

        }

        public void Add(T human)

        {

            \_container.Add(human);

        }

        public T Remove(T human)

        {

            var element = \_container.FirstOrDefault(h => h == human);

            if (element != null)

            {

                \_container.Remove(element);

                return element;

            }

            throw new NullReferenceException();

        }

        public void Sort()

        {

            \_container.Sort();

        }

        public IEnumerator<T> GetEnumerator()

        {

            return \_container.GetEnumerator();

        }

        IEnumerator IEnumerable.GetEnumerator()

        {

            return GetEnumerator();

        }

        #endregion

    }

}

1. **Умови завдання**

Дослідіть вихідний код прикладу:

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

Узагальнення (generics) дозволяють створювати класи, методи чи інтерфейси, які працюють з будь-яким типом даних без втрати типобезпеки. У прикладі узагальнення використовується в класі:

**public class HumanContainer<T> : IEnumerable<T> where T : Human**

Це означає, що HumanContainer може працювати з будь-яким типом T, але лише якщо T наслідує клас Human. Це дає гнучкість, дозволяючи створювати контейнер як для Student, так і для Worker.

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

Ключове слово **where T : Human** задає обмеження для типу T. Воно вказує, що тип T, який буде використовуватися у контейнері, обов’язково має бути або класом Human, або його нащадком. Це гарантує, що в контейнері будуть зберігатися лише об’єкти, які мають властивості Height, Weight, FirstName, LastName, та метод CompareTo.

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

Інтерфейс IComparable<Human> використовується для порівняння об’єктів одного типу. Він визначає метод:

**public int CompareTo(Human other)**

У прикладі цей метод порівнює людей за повним ім’ям (FullName). Це потрібно, щоб колекції людей могли сортуватися методом .Sort().

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

IEnumerable<T> дозволяє перебирати колекцію у циклі foreach. Щоб реалізувати цей інтерфейс, потрібно реалізувати метод:

public IEnumerator<T> GetEnumerator()

А також неявну реалізацію для неузагальненого інтерфейсу:

IEnumerator IEnumerable.GetEnumerator()

Це дозволяє класу HumanContainer бути сумісним із усіма LINQ-операціями та зручно ітеративно працювати з даними.

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

Ітератор — це об'єкт, який забезпечує поетапний доступ до елементів колекції. В C# ітератори описуються через інтерфейс IEnumerator<T>. Він має такі ключові члени:

* bool MoveNext() — переходить до наступного елемента.
* T Current — повертає поточний елемент.
* void Reset() — скидає позицію на початок (рідко використовується).

Під час реалізації GetEnumerator() фактично надається дозвіл працювати з ітератором. Зазвичай використовується yield return, але у прикладі повертається вже готовий List<T>.Enumerator.

* ***Поясніть принцип роботи індексатора.***

Індексатор дозволяє звертатись до об'єктів класу, як до масиву:

**public T this[int index]**

У HumanContainer індексатор дозволяє отримувати чи змінювати елемент за вказаним індексом, при цьому перевіряється, чи індекс не виходить за межі.

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

LINQ — це мова інтегрованих запитів, що дозволяє писати SQL-подібні запити безпосередньо в C#. Це спрощує обробку колекцій: фільтрацію, сортування, агрегацію тощо. Приклад:

**var result = container1.Where(h => h.Height > 170).OrderBy(h => h.Weight);**

Такий підхід є читабельним, безпечним для типів і дозволяє зручно працювати з колекціями.

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

У коді прикладу є відкладений запит:

**var whereRes = container1.Where(h => h.Height > 170);**

Цей запит не виконується одразу, а лише коли починають його перебирати, наприклад через foreach.

А приклад негайного запиту — це додавання ToList():

**var immediate = container1.Where(h => h.Height > 170).ToList();**

Такий запит виконується одразу після виклику і повертає список результатів.

* ***У чому переваги відкладених запитів?***

Відкладене виконання дозволяє оптимізувати роботу з колекціями. Дані обробляються тільки тоді, коли вони реально потрібні. Це економить ресурси, дозволяє уникати зайвих обчислень і динамічно реагувати на зміни в джерелі даних до моменту виконання запиту. Також це дозволяє створювати ланцюжки запитів, де остаточне рішення виконується лише один раз.

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

Лямбда-вирази — це компактний спосіб оголосити анонімні функції. У LINQ вони використовуються для визначення умов, за якими відбираються чи трансформуються елементи. Наприклад:

**container1.Where(h => h.Height > 170)**

Тут **h => h.Height > 170** — лямбда-вираз, який приймає об'єкт h і повертає булеве значення. Також лямбди використовуються в Select, OrderBy, Join, GroupBy та інших LINQ-операціях.

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

У прикладі реалізовано понад 20 LINQ-операцій:

* **OrderBy, ThenBy** — сортування;

**var orderRes = container1.OrderBy(h => h.Height).ThenBy(h => h.Weight);**

OrderBy сортує елементи колекції container1 за зростом (від меншого до більшого). Після цього ThenBy додає другий рівень сортування — за вагою, якщо значення зросту збігаються. Обидва методи використовують лямбда-вираз як ключ сортування.

* **Where** — фільтрація;

**var whereRes = container1.Where(h => (h.Height > 170 && h.Weight >= 58) || h.FullName.StartsWith("L"));**

Where відбирає лише ті об’єкти з container1, які:

* або мають зріст > 170 та вагу ≥ 58,
* або чиє повне ім’я починається на літеру “L”.

Результат — підмножина Human, яка відповідає умові. Це часто використовується для динамічного фільтрування.

* **Select** — трансформація елементів;

**var selectRes = container1.Select((h, i) => new { Index = i + 1, h.FullName });**

Select формує новий анонімний тип — для кожного Human створюється об’єкт з:

* індексом (Index),
* повним ім’ям (FullName).

Використовується перегляд елементів зі збереженням порядку.

* **SelectMany** — розгортання вкладених колекцій;

**var selectManyRes = container1.SelectMany(h => h.FullName.Split(' '));**

SelectMany розгортає кожне FullName у масив слів (тобто розділяє на FirstName та LastName), і потім з’єднує їх усі в одну **пласку колекцію**. Наприклад, з кількох імен отримаємо просто послідовність всіх імен і прізвищ.

* **Skip, Take, SkipWhile, TakeWhile** — робота з підмножинами;

**var skipRes = container1.Skip(2);**

**var skipWhileRes = container1.SkipWhile(h => h.Height < 190);**

**var takeRes = container1.Take(2);**

**var takeWhileRes = container1.TakeWhile(h => h.Height < 190);**

Skip(2) пропускає перші 2 елементи контейнера.

SkipWhile(...) пропускає елементи, поки умова **Height < 190** — істинна.

Take(2) бере перші два елементи.

TakeWhile(...) бере елементи, доки виконується умова.

Ці операції зручні для пагінації, обрізання або відсікання даних.

* **Concat** — об'єднання двох послідовностей;

**var concatRes = container1.Concat(container2);**

Concat з’єднує два контейнери (container1 і container2) в одну послідовність **IEnumerable<Human>**. Порядок елементів зберігається.

* **GroupBy** — групування студентів за університетом;

**var groupByRes = concatRes**

**.Where(h => h is Student)**

**.GroupBy(h => ((Student)h).University);**

GroupBy формує групи студентів за назвою університету. Ключем групування є University, а значенням — список студентів.

Це дає змогу переглядати дані по групах:

**Group: BSTU**

**- Marie Little**

**- Sue Jackson**

**- Wesley Jackson**

* **Join, GroupJoin** — поєднання елементів двох колекцій за спільним ключем (наприклад, Height);

**var joinRes = container1.Join(**

**container2,**

**o => o.Height,**

**i => i.Height,**

**(o, i) => new Human**

**{**

**FirstName = o.FirstName + " " + i.FirstName,**

**LastName = o.LastName + " " + i.LastName,**

**Height = o.Height,**

**Weight = (o.Weight + i.Weight) / 2**

**});**

Join поєднує елементи з container1 і container2, якщо вони мають однаковий Height. Результатом є комбінований об’єкт Human, який містить ім’я і прізвище з обох елементів, а також середню вагу.

**var groupJoinRes = container2.GroupJoin(**

**container2,**

**o => o.Height,**

**i => i.Height,**

**(o, i) => new**

**{**

**FullName = $"{o.FirstName} {o.LastName}",′**

**Count = i.Count(),**

**TotalWeight = i.Sum(s => s.Weight)**

**});**

GroupJoin створює об’єкт для кожного елемента з container2, додаючи до нього:

* кількість інших людей з таким самим Height,
* суму їх ваг.

Це схоже на ліве з’єднання з групуванням — зручно для статистики.

* **First, FirstOrDefault, DefaultIfEmpty** — отримання окремих елементів;

**var firstRes = concatRes.First(h => h.FullName.Length > 12);**

**var firstOrDefRes = concatRes.FirstOrDefault(h => h.FullName.Length > 14);**

**var defaultIfEmptyRes = container2**

**.Where(c => c.FirstName == "Eleanor")**

**.DefaultIfEmpty(new Human { FirstName = "Eleanor", LastName = "Fuller" })**

**.First();**

First() — повертає перший елемент, який відповідає умові. Якщо такого немає — виняток.

FirstOrDefault() — повертає перший або null, якщо нічого не знайдено.

DefaultIfEmpty() — якщо запит повертає порожній результат, додає об’єкт за замовчуванням.

* **Min, Max** — агрегація;

**var minRes = container1.Min(h => h.Weight);**

**var maxRes = container1.Max(h => h.Height);**

Min() та Max() знаходять найменше/найбільше значення в колекції відповідно до лямбда-виразу. Тут — мінімальна вага і максимальний зріст у container1.

* **All, Any** — перевірка умов для всієї колекції або частини;

**var allAnyRes = list.First(c => c.All(h => h.Height > 160) && c.Any(h => h is Worker));**

All(...) перевіряє, чи всі елементи задовольняють умову.

Any(...) перевіряє, чи є хоча б один, що задовольняє умову.

Тут ми шукаємо перший контейнер, де всі люди мають зріст >160, і є хоча б один Worker.

* **Contains** — перевірка наявності елемента.

**var containsRes = list**

**.Where(c => c.Contains(wr3))**

**.SelectMany(c => c.SelectMany(h => h.FullName.Split(' ')))**

**.Distinct()**

**.OrderBy(s => s)**

**.ToList();**

**Contains(wr3)** перевіряє, чи міститься об’єкт wr3 у кожному контейнері. Якщо так — ми витягуємо всі імена й прізвища в контейнері, розбиваємо, залишаємо лише унікальні слова (Distinct) і сортуємо їх.

Представлені вище запити дозволяють ефективно обробляти дані у стилі функціонального програмування, що робить код читабельнішим і виразнішим.

1. **Умова завдання.**

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).

1. **Послідовність виконання роботи**

* Створення класу CollectionType<T>
* Використання вбудованого класу List<T> як основи.
* Оголошено приватне поле List<T> items.
* Додано метод Add(T item) для додавання елемента.
* Додано метод Remove(T item) для видалення елемента.
* Реалізовано властивість Count для отримання кількості елементів.
* Реалізовано індексатор this[int index] для доступу до елементів по індексу.
* Метод PrintAll() виводить усі елементи колекції.
* Реалізація класу Program з методом Main
* Створено об'єкт колекції типу string: var collection = new CollectionType<string>();
* Додаються два рядки: "Hello" і "World".
* Виводяться всі елементи.
* Виводиться кількість елементів через властивість Count.
* Видаляється елемент "Hello".
* Повторно виводиться список елементів.

1. **Код програми**

using System;

using System.Collections.Generic;

class CollectionType<T>

{

private List<T> items = new List<T>();

public void Add(T item) // Додати елемент

{

items.Add(item);

}

public bool Remove(T item) // Видалити елемент

{

return items.Remove(item);

}

public int Count => items.Count; // Кількість елементів

public T this[int index] // Індексатор

{

get => items[index];

set => items[index] = value;

}

public void PrintAll() // Вивести всі елементи

{

foreach (var item in items)

{

Console.WriteLine(item);

}

}

}

class Program

{

static void Main()

{

var collection = new CollectionType<string>();

collection.Add("Hello");

collection.Add("World");

Console.WriteLine("Elements ");

collection.PrintAll();

Console.WriteLine($"\nNumber {collection.Count}");

collection.Remove("World");

Console.WriteLine("\nAfter remove ");

collection.PrintAll();

}

}

1. **Приклад виконання програми**

Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.

1. **Умова завдання**

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.

1. **Послідовність виконання роботи**

* Створення класу ViddilKadriv.
* Реалізовано всі необхідні поля: прізвище, ініціали, рік народження, оклад.
* Додані конструктори: без параметрів, з параметрами, копіювання.
* Додано властивості з перевірками (валідацією).
* Додано метод Show() для виведення даних.
* Додано метод ToString() — для зручного виведення у колекціях.
* Реалізовано інтерфейс IComparable<ViddilKadriv>, який дозволяє сортувати об’єкти за окладом (від більшого до меншого).
* Створення узагальненого класу CollectionType<T>.
* Використано стандартний контейнер List<T>.
* Додано методи: Add(T item) — додати елемент, AddRange(IEnumerable<T>) — додати одразу декілька, PrintAll() — вивести всі елементи, Sort() — сортувати, якщо тип реалізує IComparable<T>, Query(Func<T, bool>) — реалізація LINQ-запитів, ToList() — перетворення на звичайний список.
* Створення об’єктів класу ViddilKadriv (5 працівників з різними значеннями окладу та років народження).
* Додано у колекцію типу CollectionType<ViddilKadriv>.
* Виведення початкової колекції. За допомогою методу PrintAll() виведено список усіх працівників у консолі.
* Сортування. Викликано метод Sort(). Колекція сортується за окладом у спадному порядку (завдяки CompareTo в ViddilKadriv).
* Виконання LINQ-запитів. Запит 1: всі працівники з окладом > 500 грн. Запит 2: всі працівники, народжені до 1970 року.
* Результати запитів виводяться в консоль.
* Завершення програми
* Після виконання всіх операцій виводиться повідомлення, що очікує натискання клавіші (Console.ReadKey()).

1. **Код програми**

using System;

using System.Collections.Generic;

using System.Linq;

class CollectionType<T> //Узагальнений клас

{

private List<T> items = new List<T>();

public void Add(T item) => items.Add(item);

public void AddRange(IEnumerable<T> collection) => items.AddRange(collection);

public void PrintAll()

{

foreach (var item in items)

Console.WriteLine(item);

}

public List<T> ToList() => new List<T>(items);

public void Sort()

{

if (typeof(IComparable<T>).IsAssignableFrom(typeof(T)))

items.Sort();

else

Console.WriteLine("Клас не реалізує IComparable<T>, сортування недоступне.");

}

public IEnumerable<T> Query(Func<T, bool> predicate) => items.Where(predicate);

}

class ViddilKadriv : IComparable<ViddilKadriv> //Клас з реалізацією IComparable

{

private string pr;

private string inic;

private int rik;

private double okl;

public ViddilKadriv()

{

Prizv = "Не вказано";

Inicialy = "Не вказано";

RikNarodg = 2006;

Oklad = 0.0;

}

public ViddilKadriv(string prizv, string inicialy, int rikNarodg, double oklad)

{

Prizv = prizv;

Inicialy = inicialy;

RikNarodg = rikNarodg;

Oklad = oklad;

}

public ViddilKadriv(ViddilKadriv other)

{

Prizv = other.Prizv;

Inicialy = other.Inicialy;

RikNarodg = other.RikNarodg;

Oklad = other.Oklad;

}

public string Prizv

{

get => pr;

set => pr = !string.IsNullOrWhiteSpace(value) ? value : "Не вказано";

}

public string Inicialy

{

get => inic;

set => inic = !string.IsNullOrWhiteSpace(value) ? value : "Не вказано";

}

public int RikNarodg

{

get => rik;

set => rik = (value > 1950 && value < 2006) ? value : 2006;

}

public double Oklad

{

get => okl;

set => okl = (value > 0) ? value : 0.0;

}

public void Show()

{

Console.WriteLine(this);

}

public override string ToString()

{

return $"Прізвище: {Prizv}, Ініціали: {Inicialy}, Рік: {RikNarodg}, Оклад: {Oklad} грн.";

}

public int CompareTo(ViddilKadriv other) // Сортування за окладом (від більшого до меншого)

{

return other.Oklad.CompareTo(this.Oklad);

}

}

class Program

{

static void Main()

{

var col1 = new CollectionType<ViddilKadriv>(); //Створення колекцій

col1.Add(new ViddilKadriv("Іванов", "І.І.", 1975, 517.50));

col1.Add(new ViddilKadriv("Петренко", "П.П.", 1956, 219.10));

col1.Add(new ViddilKadriv("Паніковський", "П.П.", 1967, 300.00));

col1.Add(new ViddilKadriv("Сидоренко", "С.С.", 1980, 800.00));

col1.Add(new ViddilKadriv("Коваленко", "К.К.", 1960, 650.00));

Console.WriteLine("Початкова колекція:");

col1.PrintAll();

col1.Sort(); //Сортування

Console.WriteLine("\nПісля сортування за окладом (спадання):");

col1.PrintAll();

Console.WriteLine("\nПрацівники з окладом понад 500 грн:"); //LINQ-запит "оклад > 500"

var highSalary = col1.Query(x => x.Oklad > 500);

foreach (var emp in highSalary)

Console.WriteLine(emp);

Console.WriteLine("\nПрацівники, народжені до 1970 року:"); //LINQ-запит: народжені до 1970 року

var older = col1.Query(x => x.RikNarodg < 1970);

foreach (var emp in older)

Console.WriteLine(emp);

Console.WriteLine("\nНатисніть будь-яку клавішу для завершення.");

Console.ReadKey();

}

}

1. **Приклад виконання програми**

**Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.**

1. **Умова завдання**

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

1. **Послідовність виконання роботи**

* Створення основного класу ViddilKadriv.
* Реалізовано поля: прізвище, ініціали, рік народження, оклад.
* Додано конструктори, властивості з перевіркою значень.
* Реалізовано метод ToString() для зручного виводу.
* Клас реалізує інтерфейс IComparable<ViddilKadriv> для порівняння за окладом.
* Розробка узагальненого класу CollectionType<T>.
* Визначено колекцію на базі List<T>.Додано методи: Add, AddRange, Sort, PrintAll, ToList, Query.
* Створення колекцій об'єктів:
* Створено одну або кілька колекцій типу CollectionType<ViddilKadriv>.
* Заповнено колекцію об’єктами з різними параметрами.
* Виконано сортування колекції за окладом методом Sort() (спадання, через CompareTo).
* Реалізація та виконання LINQ-запитів.
* Виконано 5 складних запитів LINQ, що включають по 3 і більше методів, таких як: Where, Select, OrderBy; FirstOrDefault, Any, Min; GroupBy, Average, Take, тощо.
* Результати кожного запиту виведено на консоль.
* Після виводу програма завершується очікуванням натискання клавіші.

1. **Код програми**

using System;

using System.Collections.Generic;

using System.Linq;

class ViddilKadriv : IComparable<ViddilKadriv>

{

public string Prizv { get; set; }

public string Inicialy { get; set; }

public int RikNarodg { get; set; }

public double Oklad { get; set; }

public ViddilKadriv(string prizv, string inic, int rik, double oklad)

{

Prizv = string.IsNullOrWhiteSpace(prizv) ? "Не вказано" : prizv;

Inicialy = string.IsNullOrWhiteSpace(inic) ? "Не вказано" : inic;

RikNarodg = rik < 1950 || rik > 2006 ? 2006 : rik;

Oklad = oklad < 0 ? 0 : oklad;

}

public int CompareTo(ViddilKadriv other)

{

return other.Oklad.CompareTo(this.Oklad); // За спаданням

}

public override string ToString()

{

return $"{Prizv}, {Inicialy}, {RikNarodg}, {Oklad} грн";

}

}

class CollectionType<T> where T : IComparable<T>

{

private List<T> items = new List<T>();

public void Add(T item) => items.Add(item);

public void AddRange(IEnumerable<T> collection) => items.AddRange(collection);

public void PrintAll() => items.ForEach(Console.WriteLine);

public void Sort() => items.Sort();

public List<T> ToList() => items.ToList();

public IEnumerable<T> Query(Func<T, bool> predicate) => items.Where(predicate);

}

class Program

{

static void Main()

{

var col = new CollectionType<ViddilKadriv>();

col.AddRange(new[]

{

new ViddilKadriv("Іванов", "І.І.", 1975, 517.50),

new ViddilKadriv("Петренко", "П.П.", 1956, 219.10),

new ViddilKadriv("Паніковський", "П.П.", 1967, 300.00),

new ViddilKadriv("Шевченко", "Т.Г.", 1980, 900.00),

new ViddilKadriv("", "", 2025, -100.00)

});

Console.WriteLine("Початкова колекція:");

col.PrintAll();

col.Sort();

Console.WriteLine("\nВідсортована колекція за окладом (спадання):");

col.PrintAll();

var list = col.ToList();

var query1 = list

.Where(p => p.Oklad > 400)

.OrderBy(p => p.RikNarodg)

.Select(p => p.Prizv);

Console.WriteLine("\nОклад > 400, за роком народження:");

foreach (var name in query1)

Console.WriteLine(name);

var firstOld = list

.Where(p => p.RikNarodg < 1970)

.OrderByDescending(p => p.Oklad)

.FirstOrDefault();

bool hasSimilar = list.Any(p => p.Oklad == firstOld?.Oklad && p != firstOld);

Console.WriteLine($"\nПерший < 1970: {firstOld?.Prizv}, оклад: {firstOld?.Oklad}");

Console.WriteLine($"Є інші з таким самим окладом: {(hasSimilar ? "так" : "ні")}");

var query3 = list

.Where(p => p.Prizv != "Не вказано")

.OrderByDescending(p => p.Oklad)

.Take(3)

.Select(p => new { p.Prizv, p.Oklad });

Console.WriteLine("\nТоп-3 за окладом:");

foreach (var p in query3)

Console.WriteLine($"{p.Prizv}: {p.Oklad}");

var minP = list

.Where(p => p.Inicialy.StartsWith("П"))

.Select(p => p.Oklad)

.DefaultIfEmpty()

.Min();

Console.WriteLine($"\nМінімальний оклад з ініціалами на 'П': {minP}");

var grouped = list

.GroupBy(p => p.RikNarodg)

.OrderBy(g => g.Key)

.Select(g => new { Rik = g.Key, Sredniy = g.Average(p => p.Oklad) });

Console.WriteLine("\nСередній оклад по роках народження:");

foreach (var g in grouped)

Console.WriteLine($"{g.Rik}: {g.Sredniy:F2} грн");

Console.WriteLine("\nНатисніть клавішу для завершення...");

Console.ReadKey();

}

}

1. **Приклад виконання програми**

**Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.**

1. **Умова завдання (Варіант №3)**

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.

Створити масив об'єктів CollectionType, запити - знайти кількість колекцій, рівних заданому розміру, знайти максимальну та мінімальну колекцію в масиві. Узагальнена колекція – List<T>.

1. **Послідовність виконання роботи**

* Створення узагальненого класу CollectionType<T> реалізованого на основі List<T>.
* Додано методи для додавання елементів (Add), виведення (Print), властивість Count.
* Ініціалізація колекції рядків.
* Створення об'єкту типу CollectionType<string>.
* Заповнення значеннями, які представляють назви навчальних дисциплін (наприклад: Maths, Physics).
* Виведення вмісту колекції методом Print().
* Застосування LINQ-запитів до рядків: пошук рядків, що містять літеру “s”; підрахунок кількості рядків довжиною 7 символів; сортування рядків у зростаючому порядку; сортування рядків у спадному порядку.
* Створено масив об’єктів CollectionType<string> (кілька незалежних колекцій рядків).
* Запити до масиву колекцій: підраховано кількість колекцій з розміром 3; знайдено колекцію з мінімальною кількістю елементів; знайдено колекцію з максимальною кількістю елементів.
* Виведення результатів. Результати всіх операцій виведено на екран.
* Додано очікування натискання клавіші (Console.ReadKey()) перед завершенням.

1. **Код програми**

using System;

using System.Collections.Generic;

using System.Linq;

class CollectionType<T>

{

public List<T> Items { get; set; } = new List<T>();

public CollectionType() { }

public CollectionType(IEnumerable<T> items)

{

Items.AddRange(items);

}

public void Add(T item) => Items.Add(item);

public void Print()

{

Console.WriteLine("Collection elements:");

foreach (var item in Items)

Console.WriteLine(item);

}

public int Count => Items.Count;

}

class Program

{

static void Main()

{

var collection = new CollectionType<string>();

collection.Add("Maths");

collection.Add("Physics");

collection.Add("Computer Science");

collection.Add("History");

collection.Add("English");

Console.WriteLine("Initial collection:");

collection.Print();

var containsS = collection.Items.Where(s => s.ToLower().Contains("s")).ToList();

Console.WriteLine("\nStrings containing the letter 's':");

containsS.ForEach(Console.WriteLine);

int countLength7 = collection.Items.Count(s => s.Length == 7);

Console.WriteLine($"\nNumber of strings with length 7: {countLength7}");

var asc = collection.Items.OrderBy(s => s).ToList();

Console.WriteLine("\nSorted in ascending order:");

asc.ForEach(Console.WriteLine);

var desc = collection.Items.OrderByDescending(s => s).ToList();

Console.WriteLine("\nSorted in descending order:");

desc.ForEach(Console.WriteLine);

var col1 = new CollectionType<string>(new[] { "Philosophy", "Logic", "Ethics" });

var col2 = new CollectionType<string>(new[] { "Biology", "Chemistry", "Physics", "Geography" });

var col3 = new CollectionType<string>(new[] { "Art", "Music" });

var collections = new[] { col1, col2, col3 };

int countSize3 = collections.Count(c => c.Count == 3);

Console.WriteLine($"\nNumber of collections with size 3: {countSize3}");

var minCollection = collections.OrderBy(c => c.Count).First();

var maxCollection = collections.OrderByDescending(c => c.Count).First();

Console.WriteLine("\nCollection with the minimum size:");

minCollection.Print();

Console.WriteLine("\nCollection with the maximum size:");

maxCollection.Print();

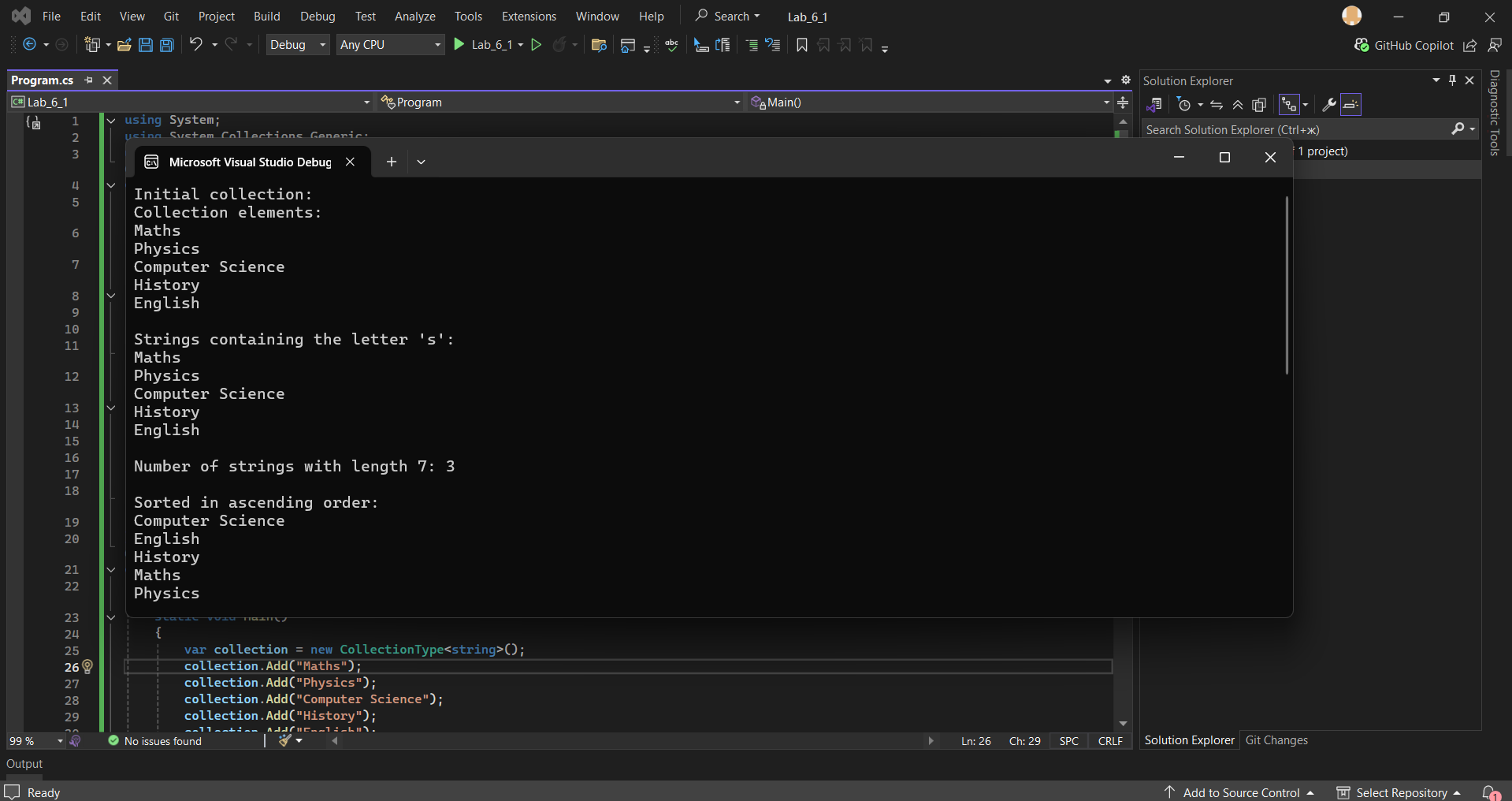
Console.WriteLine("\nPress any key to exit...");

Console.ReadKey();

}

}

1. **Приклад роботи програми**

****

**Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.**