ЗАТВЕРДЖЕНО

Науково-методичною радою
Державного університету
«Житомирська політехніка»

протокол від 12 червня 2025 р.
№ 4

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ
для проведення лабораторних робіт з навчальної дисципліни «Кросплатформерна розробка .Net Core»

для здобувачів вищої освіти освітнього ступеня «бакалавр»
спеціальності 122 «Комп'ютерні науки»
освітньо-професійна програма «Комп'ютерна графіка
та розробка ігор»
факультет інформаційно-комп'ютерних технологій
(назва факультету)
кафедра комп'ютерних наук
(назва кафедри)

Рекомендовано на засіданні кафедри комп'ютерних наук
(назва кафедри)
15 травня 2025 р., протокол №05

Розробники: ст. викладач кафедри комп'ютерних наук БЕЙРАК Дмитро,
асистент кафедри комп'ютерних наук УКРАЇНЕЦЬ Микола
(науковий ступінь, посада, ПРІЗВИЩЕ, власне ім'я)

Житомир
2025

ЗМІСТ

Лабораторна робота №1. Робота з рядками і датами в .NET

Мета роботи: поглибити розуміння роботи із рядками, символами та датами, набути навичок форматування та аналізу, ознайомитись з механізмами перетворення в .NET за допомогою C#.

Теоретичні відомості

У мові C# рядки представлені типом string, який є reference-типом і фактично є обгорткою над масивом символів (char[]). Рядки в C# є незмінними (immutable), тобто будь-яка операція, що змінює рядок, створює новий об’єкт у пам’яті.

До основних операцій з рядками належать:

- **Визначення довжини рядка**: властивість Length повертає кількість символів у рядку.
- **Зміна регістру символів**: методи `ToUpper()` і `ToLower()` дозволяють перетворити всі символи рядка відповідно у верхній або нижній регістр.
- **Інверсія рядка (перевертання)**: здійснюється шляхом перетворення рядка у масив символів за допомогою `ToCharArray()`, інверсії через `Array.Reverse()` або LINQ, і формування нового рядка.

У C# кожен символ типу `char` можна проаналізувати за допомогою методів класу `Char`, що визначають його тип:

- `Char.IsLetter(c)` – визначає, чи є символ літерою.
- `Char.IsDigit(c)` – чи є символ цифрою.

- `Char.IsWhiteSpace(c)` – чи є символ пробільним.
- Усі інші символи, які не належать до наведених категорій, можна вважати спеціальними.

Кодування – це процес перетворення рядка (який складається з символів Unicode) у масив байтів для збереження або передачі. У C# для цього використовується простір імен System.Text, зокрема клас Encoding.

- `Encoding.UTF8.GetBytes(string)` – перетворює рядок у масив байтів відповідно до кодування UTF-8.
- `Encoding.UTF8.GetString(byte[])` – виконує зворотне перетворення: байти назад у символи.

У деяких задачах важливо розрізняти символи верхнього та нижнього регістрів. В C# рядки можна порівнювати з урахуванням або без урахування регістру.

- Порівняння з урахуванням регістру виконується за допомогою `Equals(str1, str2)`.
- Якщо потрібно ігнорувати регістр, застосовується перевантаження `Equals` з параметром `StringComparison.OrdinalIgnoreCase`.

У C# метод String.Compare або Equals підтримують різні варіанти порівняння:

- `StringComparison.Ordinal` – порівняння за порядковими значеннями Unicode, враховує регістр.
- `StringComparison.CurrentCultureIgnoreCase` – порівняння відповідно до поточної культури, без урахування регістру.

Розбиття рядка на частини здійснюється за допомогою методу `Split`, який приймає символ(и)-роздільник(и). Після розбиття:

- Можна визначити кількість слів як довжину отриманого масиву.
- Метод `String.Join(separator, array)` дозволяє з'єднати елементи масиву в один рядок із вказаним роздільником.

Методи `IndexOf` та `LastIndexOf` дозволяють знайти індекси входження підрядка у рядок:

- `IndexOf` – перше входження.
- `LastIndexOf` – останнє входження.
- Метод `Contains` перевіряє наявність підрядка.

Параметр `StringComparison` визначає, чи враховувати регістр при пошуку.

Клас `StringBuilder` дозволяє ефективно створювати та змінювати рядки без створення нових об'єктів у пам'яті. Основні методи:

- `Append(string)` – додає рядок.
- `Insert(index, string)` – вставляє рядок у задану позицію.
- `Remove(start, length)` – видаляє частину рядка.
- `ToString()` – повертає фінальний рядок.

C# підтримує стандартні форматні рядки, які дозволяють виводити числові значення з певним форматуванням:

- "C" – формат валюти.
- "P" – відображає число як відсоток.
- "F2" – формат фіксованої точки з двома знаками після коми.

Перетворення рядка у числове значення:

- `int.Parse(string)` або `double.Parse(string)` – кидає виняток, якщо рядок некоректний.
- `int.TryParse(string, out int result)` – безпечна альтернатива, повертає true/false.

У C# формат чисел залежить від культури (регіональних налаштувань):

- `CultureInfo` з простору імен `System.Globalization` дозволяє задати культуру.
- Наприклад, "en-US" використовує крапку як роздільник дробу, "uk-UA" – кому.

C# підтримує кілька способів форматування рядків:

- **Композитне форматування**: `String.Format("Ім'я: {0}, Вік: {1}", name, age)`
- **Інтерполяція рядків**: `$"Ім'я: {name}, Вік: {age}"`

Інтерполяція є сучаснішим і зручнішим методом.

Для перетворення рядка у дату використовується:

- `DateTime.Parse(string)` – розпізнає рядок за поточною культурою.
- Для форматування – метод `ToString("формат")`.

Формати:

- "d" – коротка дата (наприклад, 19.05.2025).
- "D" – повна дата (понеділок, 19 травня 2025 р.).
- "F" – повна дата і час.

`Enum` – це перелік іменованих констант цілого типу. Наприклад:

```
enum Status { Active, Inactive, Pending }
```

Перетворення рядка у значення `enum`:

- `Enum.Parse(typeof(Status), input)` – повертає об’єкт enum.
- `Enum.TryParse<Status>(input, out Status result)` – безпечна версія з обробкою помилок.

Зміст роботи

Завдання 1. Робота з рядками

1. Створіть консольну програму C# під назвою StringManipulationLab.
2. Попросіть користувача ввести три різні речення.
3. Збережіть ці речення як елементи колекції.
4. Виведіть такі властивості кожного рядка на консоль:
   - Довжина рядка
   - Кількість пробілів у рядку
   - Рядок у верхньому регістрі
   - Рядок у нижньому регістрі
   - Рядок у зворотньому порядку

Завдання 2. Робота з символами

Реалізуйте метод, який приймає рядок як вхідний параметр і повертає кількість:

- Алфавітних символів
- Числових символів

- Пробільних символів
- Спеціальних символів

Завдання 3. Робота з кодуванням

Напишіть метод для демонстрації кодування та декодування тексту за допомогою System.Text.Encoding

- Реалізуйте метод, який перетворює рядок у масив байтів за допомогою кодування UTF-8
- Реалізуйте метод, який декодує масив байтів в кодуванні UTF-8 назад у рядок
- Продемонструйте роботу реалізованих методів, виводячи на екран як масив байтів, так і декодований рядок

Завдання 4. Регістри

Створіть метод, який перевіряє чи передані рядки є однаковими урахування регістру символів.

Завдання 5. Порівняння рядків

Реалізуйте два методи:

- Метод, який порівнюватиме два рядки з використанням StringComparison.Ordinal
- Метод, який порівнюватиме два рядки з використанням StringComparison.CurrentCultureIgnoreCase

Підберіть тестові дані, які демонструватимуть різницю між цими опціями і поясніть як вони працюють

Завдання 6. Розбиття

1. Створіть метод, який приймає речення від користувача.
2. Розбийте речення на слова
3. Порахуйте та виведіть кількість слів у реченні.
4. Запросіть у користувача символ і з'єднайте слова назад в один рядок, використовуючи цей символ як роздільник.
5. Виведіть отриманий рядок у консоль.

Завдання 7. Пошук

Реалізуйте метод для пошуку підрядка у введеному користувачем рядку:

- Метод повинен реалізовувати case sensitive і case insensitive пошук. Тип пошуку задає користувач.
- Поверніть та виведіть індекс першого входження, останнього входження та інформацію про наявність підрядка у рядку

Завдання 8. StringBuilder

Створіть метод для побудови рядка, використовуючи StringBuilder. Запросіть у користувача N додаткових рядків і додайте їх у StringBuilder.

Реалізуйте метод для вставки рядка у певну позицію, використовуючи StringBuilder:

- Користувач вводить індекс, куди потрібно вставити новий рядок.

Реалізуйте метод для видалення підрядка:

- Користувач вводить текст, який потрібно видалити з рядка

Завдання 9. Форматування чисел

Запитайте у користувача десяткове число.

Відформатуйте це число за допомогою кількох стандартних форматів чисел:

- Формат валюти.
- Формат відсотків.
- Формат фіксованої точки з двома знаками після коми.

Виведіть відформатовані версії числа разом з їхніми описами

Завдання 10. Парсинг чисел

Реалізуйте метод, який приймає рядкове представлення числа і повертає значення відповідного типу (int, double)

Завдання 11. Числа і культура

Продемонструйте, як відрізняється форматування числових значень для різних культур.

Завдання 12. Форматування

Запитайте інформацію про користувача (ім’я, прізвище, вік, баланс тощо). Використайте String.Format та композитне форматування для створення рядка з наданої інформації з коректним форматуванням. Реалізуйте ідентичне форматування з використанням інтерполяції рядків.

Завдання 13. Standard format strings

Запитайте у користувача числове значення та дату.

Продемонструйте використання різних standard format strings для введених значень типу double та DateTime:

- Для чисел: C, P, N та E.
- Для дат: d, D, t, T та f.

Поясніть значення кожного формату.

Завдання 14. Форматування дати

Запитайте у користувача дату у вигляді рядка. Перетворіть рядок на об'єкт DateTime за допомогою DateTime.Parse. Відформатуйте дату за допомогою різних стандартних рядків формату дати (наприклад, коротка дата, довга дата, повна дата).

Завдання 15. Форматування дати 2

Напишіть програму, яка дозволяє користувачу вводити дату у форматі рядка:

- Дозвольте користувачеві вказати користувацький рядок формату (наприклад, «yyyy-MM-dd HH:mm:ss»).
- Розбийте цей рядок на об'єкт типу DateTime.
- Переформатуйте дату відповідно до користувацького рядку формату і виведіть на екран.

Завдання 16. Робота з enum

Реалізуйте простий enum, що представляє різні статуси (наприклад, Status зі значеннями «Active», «Inactive», «Pending»). Запитайте у користувача ввести одне зі значень enum у вигляді рядка. Приведіть значення рядка до відповідного значення enum з обробкою можливих помилок приведення.

Звіт та вихідний код завантажте на корпоративний git-репозиторій.

Лабораторна робота №2. Робота з колекціями

Мета роботи: поглибити знання про колекції в .NET, набути навичок роботи з різними типами колекцій.

Теоретичні відомості

У C# для того, щоб дозволити об'єкту використовуватись у циклі foreach, клас має реалізовувати інтерфейс IEnumerable<T>. Цей інтерфейс вимагає реалізації методу GetEnumerator(), який повертає об'єкт-ітератор (IEnumerator<T>).

Ключовим полегшенням реалізації є оператор yield return, який автоматично створює ітератор:

```
public IEnumerator<Book> GetEnumerator()
{
  foreach (Book book in books)
  {
  yield return book;
  }
}
```

Інтерфейси ICollection<T> та IList<T> розширюють функціональність колекцій:

- ICollection<T> забезпечує базові операції: Add, Remove, Clear, Contains, Count.

- IList<T> додає можливість доступу за індексом (this[int index]), вставки та видалення елементів за індексом.

Масиви в C# — це фіксовані колекції елементів одного типу. У цьому завданні використовується низка вбудованих методів класу Array:

- Array.ConvertAll<TInput, TOutput>() — перетворює елементи масиву.
- Array.FindAll() — дозволяє фільтрувати елементи.
- Array.BinarySearch() — виконує бінарний пошук (потрібно спочатку відсортувати масив).
- **Зубчасті масиви** (jagged arrays) — це масиви масивів, де кожен підмасив може мати різну довжину: int[][].

.NET надає розширену підтримку стандартних колекцій:

- List<T> – динамічний масив. Дозволяє додавання, вставку, видалення за індексом.
- Queue<T> – реалізує принцип **FIFO (перший прийшов – перший обслуговується)**. Методи: Enqueue, Dequeue.
- Stack<T> – реалізує принцип **LIFO (останній прийшов – перший обслуговується)**. Методи: Push, Pop.
- HashSet<T> – зберігає лише унікальні значення. Підтримує операції множин: Union, Intersect.

Ці структури даних широко застосовуються в алгоритмах, чергах подій, системах відміни/повторення, перевірках на унікальність тощо.

Клас Dictionary<TKey, TValue> дозволяє зберігати пари **ключ – значення**:

- Швидкий доступ до елементів за ключем.
- Метод TryGetValue() дозволяє безпечно отримати значення, перевіривши наявність ключа.
- Ітерація через foreach (var kvp in dictionary) дозволяє перебрати всі пари.

ImmutableList<T> – частина пакету **System.Collections.Immutable**. Основна особливість: **всі зміни створюють новий екземпляр колекції**, оригінал лишається незмінним.

- Add(), Remove() не змінюють існуючий список, а повертають новий.
- Забезпечує **безпечну роботу у багатопотоковому середовищі**, де зміни не повинні впливати на інші частини програми.

FrozenDictionary<TKey, TValue> — це високоефективна незмінна колекція, що оптимізована для **швидкого доступу та читання**.

- Створюється з наявного словника: FrozenDictionary.ToFrozenDictionary().
- Після створення вона **не підтримує змін**: не можна додавати чи видаляти елементи.
- Ідеальна для конфігурацій або часто використовуваних довідників, де потрібна максимальна продуктивність.

Зміст роботи

Завдання 1. Enumeration

1. Створіть клас Book з наступними властивостями: Title, Author, Year.

2. Створіть клас-колекцію Library, який зберігатиме об'єкти типу Book і імплементує інтерфейс IEnumerable<Book>.

3. Використайте yield return для реалізації підтримки циклу foreach для класу Library.

4. Перевірте вашу реалізацію шляхом додавання об’єктів типу Book до Library та виведення інформації про книжку в консоль, наприклад:

```
var books = new Book[]
{
    new Book() { Author = "123", Title = "sdfsdf", Year = 2025 },
    new Book() { Author = "456", Title = "vb", Year = 2025 },
    new Book() { Author = "456", Title = "cvb", Year = 2025 }
};
var library = new Library(books);

foreach (Book book in library)
{
          Console.WriteLine("{0}  {1}  {2}", book.Year, book.Title,
book.Author);
}
```

Завдання 2. Інтерфейси ICollection і IList

1. Створіть клас CustomCollection<T>, який реалізує ICollection<T>.

2. Реалізуйте необхідні методи та властивості (Add, Remove, Count, тощо).

3. Розширте його, щоб клас також реалізував інтерфейс IList<T> і реалізуйте підтримку індексування.

4. Перевірте свою реалізацію, зберігаючи та маніпулюючи цілими числами в CustomCollection<int>.

Завдання 3. Просунуті операції з масивами

1. Створіть одновимірний масив цілих чисел розміром 10 і заповніть його випадковими значеннями від 1 до 100.
2. Використайте Array.ConvertAll(), щоб перетворити масив, зводячи кожен елемент у квадрат.
3. Використайте Array.FindAll(), щоб фільтрувати лише парні числа.
4. Використайте Array.BinarySearch(), щоб знайти задане число після сортування масиву.
5. Створіть зубчастий масив (int[][]) з різною довжиною для кожного рядка.
6. Заповніть кожен рядок випадковими числами.
7. Реалізуйте метод для сортування кожного рядка окремо в порядку зростання.

Завдання 4. Списки, Черги, Стеки і Множини

1. Списки: створіть List<string> і виконайте операції: додавання елементів, вставка в певні позиції та видалення елементів.
2. Черги: імітуйте чергу підтримки клієнтів за допомогою Queue<string>. Реалізуйте операції Enqueue і Dequeue.
3. Стеки: реалізуйте систему Undo Redo за допомогою Stack<string>.
4. Множини: використайте HashSet<int> для зберігання унікальних випадкових чисел від 1 до 100 і демонстрації об’єднання та перетину з іншим набором.

Завдання 5. Словники

1. Створіть Dictionary<string, int>, де ключі — це назви країн, а значення — їх населення в мільйонах.

2. Реалізуйте метод для знаходження країни за чисельністю населення.

3. Використайте TryGetValue(), щоб перевірити, чи існує країна, перш ніж отримати до неї доступ.

4. Ітеративно пройдіться по словнику і виведіть його вміст.

Завдання 6. Настроювані колекції і проксі

1. Створіть проксі-клас LoggedList<T>, який обгортає List<T> і логує кожну операцію (Додати, Видалити тощо).

2. Використовуйте Console.WriteLine(), щоб відстежувати кожну зміну списку.

3. Протестуйте створений проксі, демонструючи використання LoggedList<int>.

Завдання 7. Незмінювані колекції

1. Використовуйте ImmutableList<T> із System.Collections.Immutable.

2. Додайте та видаліть елементи за допомогою Add() і Remove() (зауважте, що щоразу повертається нова колекція).

3. Продемонструйте, що оригінальна колекція залишається незмінною.

Завдання 8. Заморожені колекції

1. Використайте FrozenDictionary<string, int> для збереження населення країни.

2. Покажіть, що елементи не можна додавати або видаляти після заморожування.

3. Порівняйте ефективність пошуку між FrozenDictionary і Dictionary за допомогою Stopwatch.

Завдання 9. Перевірка рівності і впорядкування

1. Створіть клас Person із властивостями Name та Age.
2. Створіть IEqualityComparer<Person> для порівняння людей на основі імені та віку.
3. Реалізуйте IComparer<Person>, щоб упорядкувати людей за віком.
4. Використовуйте ці компаратори з HashSet<Person> (для рівності) і SortedSet<Person> (для впорядкування).

Лабораторна робота №3. LINQ. Unit-тестування.

Мета роботи: набути навичок роботи з LINQ. Набути навичок написання unit-тестів.

Теоретичні відомості

LINQ, або Language Integrated Query — це набір можливостей мови та середовища виконання для написання структурованих типобезпечних запитів до локальних колекцій об’єктів і віддалених джерел даних.

LINQ дає змогу надсилати запити до будь-якої колекції, що реалізує IEnumerable<T>, будь то масив, список або об’єктна модель документа XML (DOM), а також до віддалених джерел даних, таких як таблиці в базі даних SQL Server. LINQ має такі переваги, як перевірка типу під час компіляції та динамічна композиція запиту.

Основними одиницями даних у LINQ є послідовності та елементи. Послідовність — це будь-який об’єкт, який реалізує IEnumerable<T>, а елемент — це кожен елемент у послідовності.

Оператор запиту — це метод, який перетворює послідовність. Типовий оператор запиту приймає вхідну послідовність і видає перетворену вихідну послідовність. У класі Enumerable у System.Linq є близько 40 операторів запитів — усі вони реалізовані як статичні методи розширення.

Запит — це вираз, який під час перерахування перетворює послідовності за допомогою операторів запиту. Коли оператори запиту об’єднані в ланцюг, вихідна послідовність одного оператора є вхідною послідовністю наступного.

Порядок елементів у вхідній послідовності є важливим у LINQ. Деякі оператори запитів покладаються на цей порядок, наприклад Take, Skip і Reverse.

Не всі оператори запиту повертають послідовність. Такі оператори, як First, Last і ElementAt виділяють один елемент із вхідної послідовності. Оператори агрегації повертають скалярне значення, зазвичай числового типу.

Оператори запитів забезпечують відкладене виконання, повертаючи послідовності декораторів. На відміну від традиційного класу колекції, такого як масив або список, послідовність декоратора (загалом) не має власної структури підтримки для зберігання елементів. Замість цього він обертає іншу послідовність, що надається під час виконання, до якої він підтримує постійну залежність. Кожного разу, коли ви запитуєте дані від декоратора, він, у свою чергу, повинен запитувати дані з упакованої послідовності введення. Виклик Where, наприклад, просто створює послідовність обгортки декоратора, яка містить посилання на вхідну послідовність, лямбда-вираз та будь-які інші надані аргументи. Вхідна послідовність перераховується лише тоді, коли перераховується декоратор.

Зміст роботи

Виконання кожного завдання в даній лабораторній роботі має супроводжуватися написанням unit-тесту для перевірки коректності реалізації. Ознайомитися з xUnit (фреймворк для тестування в .NET) та техніками написання тестів можна за посиланням.

Завдання 1.

Реалізуйте метод, який фільтрує список імен на основі мінімальної довжини і повертає відповідні імена.

1. Реалізуйте метод FilterNames, який приймає масив рядків та ціле число minLength.

2. Метод повинен повертати IEnumerable<string>, що містить імена, довжина яких більша або дорівнює minLength.

Завдання 2.

Напишіть метод для підрахунку парних чисел з заданого списку цілих чисел, використовуючи LINQ.

1. Реалізуйте метод CountEvenNumbers, який отримує масив цілих чисел.

2. Метод повинен повертати ціле число - кількість парних чисел.

3. Використати оператор Where для фільтрації парних чисел та Count для отримання їх загальної кількості.

Завдання 3.

Створіть метод, який фільтрує імена на основі наявності заданої літери, сортує їх за довжиною та перетворює їхні символи у верхній регістр.

1. Реалізуйте метод SortAndTransformNames, який отримує масив рядків та символ filterLetter.

2. Метод повинен повертати IEnumerable<string>, що містить імена, які містять в собі filterLetter, відсортовані за довжиною та перетворені у верхній регістр.

Завдання 4.

Розробити клас кошика товарів з використанням LINQ для фільтрації та розрахунку загальної вартості відфільтрованих товарів.

1. Описати клас CartItem з властивостями Name (string) та Price (decimal).

2. Реалізувати клас ShoppingCart з методом GetTotalPrice, який

отримує список CartItem та значення minPrice.

3. Метод повинен повертати загальну вартість товарів, які коштують більше заданої мінімальної ціни.

Завдання 5.

Розробити рішення для управління запасами товарів, яке використовує LINQ для пошуку товарів на основі певних умов.

1. Визначити клас Product з властивостями ID (int), Name (string), Category (string) та Stock (int).

2. Реалізувати метод GetProductsByCategory, який отримує на вхід список продуктів і рядок категорії.

3. Метод повинен повертати список товарів у заданій категорії, запас яких більше нуля.

Завдання 6.

Створити рішення для аналізу оцінок студентів та пошуку студентів, які відповідають певним критеріям успішності.

1. Створити клас Student з властивостями Name (string) та Grade (double).

2. Реалізувати метод GetHighPerformers, який отримує список студентів та порогову оцінку.

3. Цей метод повинен повертати імена студентів з оцінками вище порогової, відсортовані за алфавітом (від А-Я або A-Z).

Звіт та вихідний код завантажте на корпоративний git-репозиторій.

Лабораторна робота №4. Просунуті LINQ-запити. Unit-тестування.

Мета роботи: набути навичок створення складних LINQ-запитів. Навчитися виконувати операції об’єднання, групування та сортування даних у колекціях. Закріпити навички написання unit-тестів.

Теоретичні відомості

LINQ (Language INtegrated Query) — компонент .NET, який додає нативні можливості виконання запитів даних до мов, що входять у .NET. LINQ розширює можливості мови, додаючи до неї вирази запитів, що є схожими на твердження SQL та можуть бути використані для зручного отримання та обробки даних масивів, XML документів, реляційних баз даних та сторонніх джерел. LINQ також визначає набір імен методів (що називаються стандартними операторами запитів, або стандартними операторами послідовностей), а також правила перекладу, що має використовувати компілятор для перекладу fluent виразів у звичайні, використовуючи їх назву, лямбда-вирази та анонімні типи.

Однією з важливих особливостей LINQ є можливість виконання агрегацій над колекціями. До них належать обчислення загальної кількості елементів (Count()), суми (Sum()), середнього значення (Average()), мінімального або максимального значення (Min(), Max()), а також довільні операції через метод Aggregate().

У випадках, коли необхідно об’єднати об'єкти за спільною ознакою, наприклад, студентів за групами або замовлення за клієнтами, використовується метод GroupBy(). Це дозволяє побудувати структуру «ключ – колекція елементів», де ключем виступає ознака групування.

Наприклад:

```
var groups = students.GroupBy(s => s.GroupName);
foreach (var group in groups)
{
    Console.WriteLine($"Group: {group.Key}");
    foreach (var student in group)
    {
        Console.WriteLine($" - {student.Name}");
    }
}
```

LINQ дозволяє будувати запити, які містять інші запити як частину виразів where, select тощо. Такий підхід дозволяє будувати гнучкі й динамічні умови відбору або обчислення. Наприклад:

```
var topCustomers = customers
    .Where(c => c.Orders.Sum(o => o.Total) > 1000);
```

LINQ надає засоби об'єднання колекцій за спільними ключами, що відповідає поняттям JOIN у SQL. Метод Join() виконує внутрішнє з'єднання, а GroupJoin() — групове з'єднання, яке часто використовується для реалізації лівого приєднання (Left Outer Join). Наприклад:

```
var result = orders.Join(customers,
                         o => o.CustomerId,
                         c => c.Id,
                         (o, c) => new { c.Name, o.Total });
```

Метод OrderBy() виконує первинне сортування. Але якщо існує потреба сортувати за додатковими критеріями (наприклад, спочатку за курсом, потім за прізвищем), використовують ThenBy().

- ThenBy() – додаткове сортування за зростанням;
- ThenByDescending() – додаткове сортування за спаданням.

Розглянемо наступний лістинг коду:

```
var sortedBooks = books
    .OrderBy(b => b.Genre)
```

```
        .ThenByDescending(b => b.Rating);
```

В даному випадку, сортування з ThenBy працює лише після виклику OrderBy або OrderByDescending. Воно не замінює попереднє сортування, а доповнює його — для елементів з однаковим значенням першого критерію.

Метод Zip() дозволяє поєднувати дві послідовності у третю, де кожен елемент є результатом застосування заданої функції до відповідних елементів обох послідовностей. При цьому довжина результату дорівнює мінімальній довжині серед обох колекцій. Візьмемо до уваги наступний код:

```
string[] names = { "Alice", "Bob", "Charlie" };
int[] scores = { 95, 82, 91 };

var paired = names.Zip(scores, (name, score) => $"{name}: {score}");

foreach (var item in paired)
{
    Console.WriteLine(item);
}
```

Після його виконання отримаємо наступний результат в консолі:

```
Alice: 95
Bob: 82
Charlie: 91
```

Зміст роботи

Виконання кожного завдання в даній лабораторній роботі має супроводжуватися написанням unit-тесту для перевірки коректності реалізації. Ознайомитися з xUnit (фреймворк для тестування в .NET) та техніками написання тестів можна за посиланням.

Завдання 1.

Дано список працівників, де кожен працівник має:

- Id (int)
- Name (string)
- Age (int)
- Salary (decimal)
- DepartmentId (int)

Також дано список департаментів, де кожен департамент має

- Id (int)
- Name (string)

Реалізуйте LINQ-запит, який:

- Робить вибірку з усіх співробітників старше 30 років, які працюють в департаменті “IT” або “HR”
- Проектує результат в анонімний тип { EmployeeName, DepartmentName, IncreasedSalary } де IncreasedSalary - це заробітна плата співробітника збільшена на 10%

Завдання 2.

Дано список продуктів, кожен продукт містить:

- Id (int)
- Name (string)
- Category (string)
- Price (decimal)

Реалізуйте LINQ-запит, який:

- Групує продукти за властивістю Category
- Рахує загальну ціну продуктів в кожній категорії

- Повертає список типу { Category, TotalPrice }, відсортований за TotalPrice в порядку спадання

Завдання 3.

Дано список замовлень, кожне замовлення має:

- OrderId (int)
- CustomerId (int)
- OrderDate (DateTime)
- TotalAmount (decimal)

Також дано список клієнтів, кожен клієнт має:

- CustomerId (int)
- Name (string)

Реалізуйте LINQ-запит, який:

- Об’єднує список замовлень зі списком клієнтів (Join)
- Повертає список { CustomerName, NumberOfOrders, TotalAmountSpent } який містить дані клієнтів, які зробили більше ніж 3 замовлення.

Завдання 4.

Дано список студентів, кожен студент має:

- Id (int)
- Name (string)
- Subjects (List<string>)

Реалізуйте LINQ-запит, який:

- Знаходить студентів, які зараховані на предмет “.NET Core” і “ASP.NET” одночасно
- Сортує студентів спочатку за кількістю предметів, на які вони зараховані за спаданням, потім за ім’ям в алфавітному порядку.

Завдання 5.

Дано список банківських транзакцій, кожна транзакція містить:

- TransactionId (int)
- AccountId (int)
- Amount (decimal)
- TransactionType (string) // "Deposit" або "Withdrawal"

Напишіть LINQ-запит, який:

- Визначає поточний баланс для кожного AccountId
- Повертає список { AccountId, Balance } де баланс більший за 1000

Завдання 6.

Дано два списки:

List<int> list1 = { 1, 2, 3, 4, 5 };

List<int> list2 = { 10, 20, 30, 40, 50 };

Реалізуйте LINQ-запит за допомогою методу Zip, який повертатиме список елементів типу { FirstNumber, SecondNumber, Sum, Product }, де FirstNumber - число з першого списку, SecondNumber - число з другого списку, Sum - їхня сума, Product - їхній добуток.

Завдання 7.

Дано список книжок, кожна книжка містить:

- Id (int)
- Title (string)
- Author (string)
- Price (decimal)
- Genre (string)

Реалізуйте LINQ-запит, який групує книжки за полем Genre та повертає дві найдорожчі книжки з кожної групи.

Звіт та вихідний код завантажте на корпоративний git-репозиторій.

Лабораторна робота №5. Робота з Garbage Collector та IDisposable. Вивільнення ресурсів.

Мета роботи: ознайомитися з механізмом Garbage Collection в .NET, набути навичок роботи з вивільненням ресурсів та знищенням об’єктів в .NET.

Теоретичні відомості

У Common Language Runtime (CLR) Garbage Collector (GC) виконує функцію автоматичного менеджера пам'яті. Збирач сміття керує розподілом та звільненням пам'яті для програми. Тому розробникам, які працюють з керованим кодом, не потрібно писати код для виконання завдань управління пам'яттю. Автоматичне управління пам'яттю допомагає усунути поширені проблеми, такі як забуття про звільнення об'єкта та спричинення витоку пам'яті або спроба доступу до звільненої пам'яті для об'єкта, який вже звільнено.

Збирач сміття має такі переваги:

- Звільняє розробників від необхідності вручну звільняти пам'ять.
- Ефективно розподіляє об'єкти в керованій купі.
- Повертає об'єкти, які більше не використовуються, очищає їхню пам'ять і зберігає пам'ять доступною для майбутніх розподілів. Керовані об'єкти автоматично отримують чистий вміст спочатку, тому їхнім конструкторам не потрібно ініціалізувати кожне поле даних.
- Забезпечує безпеку пам'яті, гарантуючи, що об'єкт не може використовувати для себе пам'ять, виділену для іншого об'єкта.

Збирання сміття відбувається, коли виконується одна з наступних умов:

- Система має низький обсяг фізичної пам'яті. Розмір пам'яті визначається або сповіщенням про низький обсяг пам'яті від операційної системи, або низьким обсягом пам'яті, про який вказує хост.

- Обсяг пам'яті, що використовується виділеними об'єктами в керованій купі, перевищує допустимий поріг. Цей поріг постійно коригується під час виконання процесу.
- Викликається метод GC.Collect. Майже у всіх випадках цей метод не потрібно викликати самостійно, оскільки збирач сміття працює безперервно. Цей метод в основному використовується для унікальних ситуацій та тестування.

Для більшості об'єктів, створених вашою програмою, ви можете покладатися на збирання сміття для автоматичного виконання необхідних завдань керування пам'яттю. Однак некеровані ресурси потребують явного очищення. Найпоширенішим типом некерованого ресурсу є об'єкт, який обгортає ресурс операційної системи, такий як дескриптор файлу, дескриптор вікна або мережеве підключення. Хоча збирач сміття може відстежувати час життя керованого об'єкта, який інкапсулює некерований ресурс, він не має конкретних знань про те, як очистити цей ресурс.

Коли ви визначаєте об'єкт, який інкапсулює некерований ресурс, рекомендується надати необхідний код для очищення некерованого ресурсу у публічному методі Dispose. Надаючи метод Dispose, ви дозволяєте користувачам вашого об'єкта явно звільняти ресурс після завершення роботи з об'єктом. Коли ви використовуєте об'єкт, який інкапсулює некерований ресурс, обов'язково викликайте Dispose().

Фіналізатори (або старіша назва – деструктори) використовуються для виконання будь-якого необхідного остаточного очищення, коли екземпляр класу збирається збирачем сміття. Важливими особливостями фіналізаторів є наступне:

- Фіналізатори не можна визначати в структурах. Вони використовуються лише з класами.

- Клас може мати лише один фіналізатор.
- Фіналізатори не можна успадковувати або перевантажувати.
- Фіналізатори не можна викликати. Вони викликаються автоматично.
- Фіналізатор не може мат модифікаторів доступу і не може мати параметрів.

Програміст не контролює, коли викликається фіналізатор; збирач сміття вирішує, коли його викликати. Збирач сміття перевіряє наявність об'єктів, які більше не використовуються програмою. Якщо він вважає об'єкт придатним для фіналізації, він викликає фіналізатор (якщо такий є) та звільняє пам'ять, яка використовується для зберігання об'єкта. Можна примусово виконати збирання сміття, викликавши GC.Collect, але здебільшого цього виклику слід уникати, оскільки це може створити проблеми з продуктивністю.

Метод Dispose в реалізується для звільнення некерованих ресурсів. Збирач сміття .NET не виділяє та не звільняє некеровану пам'ять. Шаблон для утилізації об'єкта, який називається dispose pattern, задає життєвий цикл об’єкта. Dispose pattern використовується для об'єктів, що реалізують інтерфейс IDisposable. Цей шаблон поширений під час взаємодії з дескрипторами файлів та каналів, дескрипторами реєстру, дескрипторами очікування або вказівниками на блоки некерованої пам'яті, оскільки збирач сміття не може повернути некеровані об'єкти.

Зміст роботи

Завдання 1.

Створити клас, який містить конструктор та фіналізатор (деструктор). Створіть набір екземплярів класу з використанням циклу. Після цього викличте метод GC.Collect() для ініціації збору сміття вручну. Дослідіть, коли викликається фіналізатор (деструктор) на створених об’єктах. Для

відслідковування порядку викликів додайте Console.WriteLine у фіналізатор та конструктор. Поясніть поведінку фіналізатора в звіті.

Завдання 2.

Реалізуйте клас FileHandler, який виконує відкриття файлу для запису з використанням FileStream або StreamWriter. Імплементуйте інтерфейс IDisposable для даного класу. Реалізуйте метод Dispose() таким чином, щоб він виконував закриття файлового потоку. У викликаючому коді (н-д Main()) створіть екземпляр класу FileHandler з використанням ключового слова using. Дослідіть порядок викликів реалізованих методів. Переконайтеся, що всі ресурси було вивільнено після виходу з блоку using.

Завдання 3.

Модифікуйте створений в завданні 2 клас FileHandler, додавши реалізацію фіналізатора. У фіналізаторі реалізуйте логіку для закриття файлу в випадку, якщо він не був закритий належним чином. Створіть декілька екземплярів класу FileHandler без використання using. Ініціюйте збір сміття GarbageCollector’ом та відслідкуйте порядок виклику методів. Порівняйте його з порядком викликів з завдання 2. Опишіть різницю між Dispose() та фіналізатором у звіті.

Завдання 4.

Проведіть симуляцію створення великої кількості об’єктів. Використайте GC.GetTotalMemory() для визначення зайнятої пам’яті до створення об’єктів. Ініціюйте збір сміття шляхом виклику GC.Collect() і GC.WaitForPendingFinalizers() та знову визначте кількість зайнятої пам’яті. Опишіть вплив збору сміття на використання пам’яті.

Звіт та вихідний код завантажте на корпоративний git-репозиторій.

Лабораторна робота №6. Багатопотоковість в .NET Core

Мета роботи: ознайомитися з основами конкурентного програмування в середовищі .NET, навчитися застосовувати механізми багатопотоковості та асинхронного виконання, отримати навички створення потокобезпечних додатків, використовуючи засоби синхронізації.

Теоретичні відомості

Сучасні додатки часто потребують виконання кількох операцій паралельно: обробка подій, робота з мережею, фонові обчислення тощо. У .NET багатопоточність (multithreading) дозволяє ефективно використовувати ресурси процесора шляхом одночасного виконання кількох потоків.

Клас Thread, який знаходиться у просторі імен System.Threading, є базовим інструментом для створення та керування потоками (threads) у .NET. Потік — це незалежна одиниця виконання, що дозволяє виконувати частини програми паралельно з основним потоком або з іншими потоками.

Thread надає низькорівневий контроль над виконанням коду в окремих потоках, зокрема: запуск, зупинка, пріоритет, пауза, завершення. Але через свою складність, Thread у сучасній розробці використовується рідше, поступаючись високорівневим абстракціям, таким як Task.

Якщо декілька потоків або задач мають доступ до спільних ресурсів (наприклад, змінна або колекція), виникає ризик конфлікту доступу — "race condition".

Для захисту спільних ресурсів використовують механізми синхронізації.

lock — конструкція, яка неявно викликає методи класу Monitor, а сам Enter та Exit. Забезпечує, що лише один потік може виконувати критичну секцію коду одночасно.

Клас Monitor (простір імен System.Threading) — це базовий механізм синхронізації у .NET, який дозволяє керувати одночасним доступом до ресурсів з боку кількох потоків. Його мета — гарантувати, що лише один потік одночасно виконує критичну секцію коду, яка взаємодіє з розділюваними ресурсами (наприклад, змінними, файлами, колекціями).

Ключова відмінність від lock — гнучкість та розширені можливості, такі як:

- Очікування (Wait)
- Сповіщення (Pulse, PulseAll)
- Можливість задавати таймаут при захопленні монітора

Клас Task із простору імен System.Threading.Tasks є більш високорівневою абстракцією, яка замінює ручну роботу з Thread. Задачі добре підходять для паралельних обчислень, які не блокують основний потік.

Task Parallel Library (TPL) базується на концепції Task, яке представляє асинхронну операцію. У певному сенсі завдання нагадує Thread або робочий елемент ThreadPool, але на вищому рівні абстракції. Термін «паралелізм завдань» стосується одного або кількох незалежних завдань, що виконуються одночасно. Task забезпечують дві основні переваги:

- Більш ефективне та масштабоване використання системних ресурсів.

За лаштунками завдання ставляться в чергу до пулу потоків, який було вдосконалено алгоритмами, що визначають та налаштовуються на кількість потоків. Ці алгоритми забезпечують балансування навантаження для максимізації пропускної здатності. Цей процес робить завдання відносно легкими, і ви можете створити багато з них, щоб забезпечити дрібнозернистий паралелізм.

- Більше програмного контролю, ніж це можливо з потоком або

робочим елементом.

Task'и та фреймворк, побудований навколо них, надають багатий набір API, які підтримують очікування, скасування, продовження, надійну обробку винятків, детальний стан, налаштовуване планування тощо.

З цих причин TPL є кращим API для написання багатопотокового, асинхронного та паралельного коду в .NET.

Зміст роботи

Завдання 1.

Створіть метод PrintNumbers(), який виводить на екран цифри від 1 до 5 з затримкою в одну секунду. Щоб симулювати затримку використайте метод Thread.Sleep(). Для виконання методу PrintNumbers() створіть окремий потік (Thread). В основному потоці програми реалізуйте код, який очікуватиме закінчення роботи потоку, в якому було запущено метод PrintNumbers() за допомогою методу Join.

Завдання 2.

У багатопотокових сценаріях часто доводиться очікувати виконання певної умови, наприклад, доступності даних. Для синхронізації доступу до об'єктів та сигналізації про готовність використовується клас Monitor.

1. Створіть чергу Queue<int>, яка буде спільною для потоків
2. Створіть ThreadA (Producer), в якому відбуватиметься додавання елементів (від 1 до 5) в чергу кожну секунду
3. Створіть ThreadB (Consumer), який очікуватиме наявності елементів в черзі, забиратиме їх з черги (Dequeue) та виводитиме їх на екран
4. Використайте lock, Monitor.Wait для очікування наявності елементів, Monitor.Pulse для сигналізації про додавання елементів в чергу.

Завдання 3.

Під час роботи над спільними змінними потоки можуть читати та модифікувати вміст змінної до того як інші потоки закінчили роботу з нею. Для перевірки роботи багатьох потоків зі спільною змінною без використання механізмів синхронізації виконайте наступні кроки:

1. Створіть змінну-лічильник.
2. Створіть 10 потоків, кожен з яких інкрементуватиме значення лічильника 1000 разів.
3. Запустіть створені потоки та дочекайтеся їхнього виконання. Виведіть значення лічильника по завершенню роботи потоків. Поясніть отримане значення у звіті.
4. Повторіть ті ж дії, але з використанням lock. Поясніть різницю в отриманих результатах у звіті.

Завдання 4.

Колекції, такі як List<T>, за замовчуванням не є потокобезпечними. Одночасне додавання елементів може призвести до неочікуваних значень в пам’яті або викинути винятки.

1. Створіть спільний List<int>.
2. Запустіть кілька потоків, кожен з яких додає числа до списку. Перевірте кількість доданих елементів. Дочекайтеся їхнього виконання та поясніть отриманий результат.
3. Обгорніть list.Add() в lock. Запустіть декілька потоків та дочекайтеся їхнього виконання. Перевірте кількість доданих елементів та

поясніть різницю результатів виконання.

Завдання 5.

Необроблені винятки всередині потоків не призводять до збою основного потоку, але вони часто невидимі.

1. Створіть потік, який кидає виняток.
2. Зловіть цей виняток всередині методу потоку та надрукуйте зрозуміле повідомлення.

Завдання 6.

Task представляє асинхронну операцію і є сучасною альтернативою Thread. ContinueWith дозволяє одному Task запускати інший після свого завершення.

1. Створіть Task, який обчислює значення з певною затримкою.
2. Використовуйте Task.Result або await, щоб отримати результат.
3. Додайте Task, який повинен початися одразу після закінчення виконання першого. Використайте результат виконання першого Task в другому.

Звіт та вихідний код завантажте на корпоративний git-репозиторій.

Лабораторна робота №7. I/O операції в .NET Core

Мета роботи: ознайомитися з основами роботи з файловою системою в .NET Core та навчитися виконувати операції вводу/виводу (читання, запис, копіювання та видалення файлів і директорій) з використанням класів простору імен System.IO

Теоретичні відомості

.NET Core надає набір різних потоків (streams) і високорівневі утиліти для зчитування та запису даних із файлів, пам’яті, мережі, каналів тощо. Вони знаходяться в просторі імен System.IO.

Для базових операцій з файлами (створення, видалення, копіювання, перевірка існування тощо) .NET надає два основних класи: File і FileInfo.

File — це статичний клас, який дозволяє швидко виконувати базові дії з файлами. Його методи зручні у випадках, коли не потрібне збереження інформації про сам об’єкт файлу.

FileInfo — клас, який працює з конкретним файлом як об’єктом. Це дозволяє отримувати додаткову інформацію про файл (наприклад, розмір, час створення) та багаторазово працювати з ним без повторного зазначення шляху.

Аналогічно, для роботи з каталогами використовуються класи Directory і DirectoryInfo. Directory — статичний, призначений для швидких операцій, а DirectoryInfo — об’єктно-орієнтований, дозволяє працювати з каталогами як з об’єктами.

Ці класи дозволяють реалізовувати повний цикл роботи з файловою системою: створення структури каталогів, маніпуляції з файлами, аналіз вмісту папок тощо.

Потік (stream) у .NET — це абстракція джерела або приймача послідовних

байтів. Потоки можуть бути пов’язані з файлами, пам’яттю, мережею, буферами або міжпроцесним зв’язком. Абстрактний базовий клас для більшості потоків — це System.IO.Stream

FileStream використовується для читання та запису байтів у файл. Він забезпечує низькорівневий доступ до файлу, дозволяє вказати режим відкриття (створення нового файлу, відкриття на читання, додавання тощо), доступ (читання, запис, або обидва), а також підтримує асинхронну роботу.

MemoryStream дозволяє записувати дані у вигляді байтів безпосередньо в оперативну пам’ять. Це зручно для тимчасового зберігання інформації або для передачі даних у форматі, який не потребує збереження на диск. MemoryStream часто використовується для проміжної обробки даних перед серіалізацією або стисненням.

BufferedStream додає проміжний буфер між джерелом даних і споживачем. Це дозволяє зменшити кількість фізичних звернень до диска або іншого пристрою, що виводить або приймає дані. Використовується як обгортка навколо інших потоків, наприклад FileStream.

StreamReader та StreamWriter надають зручні засоби для читання та запису текстової інформації. StreamReader використовується для зчитування тексту з потоку, тоді як StreamWriter — для запису тексту. Вони підтримують кодування (наприклад UTF-8, ASCII) і дозволяють працювати з рядками без необхідності ручної обробки байтів.

Для збереження дискового простору або передачі даних у стислому вигляді використовується клас GZipStream. Він дозволяє стискати або розпаковувати потік даних відповідно до формату GZip. Цей клас обгортає інший потік (наприклад, FileStream, MemoryStream) і підтримує два режими роботи: CompressionMode.Compress (стиснення) і CompressionMode.Decompress (розпакування).

Іменовані канали (Named Pipes) дозволяють реалізовувати обмін даними між процесами на одній машині або в межах локальної мережі. У .NET для цього призначені класи NamedPipeServerStream і NamedPipeClientStream:

- NamedPipeServerStream створює канал і очікує підключення з боку клієнта.
- NamedPipeClientStream підключається до сервера за іменем каналу.

Після встановлення з’єднання обидві сторони можуть читати і записувати дані через потік так само, як у випадку з файлами або пам’яттю. Зазвичай ці потоки обгортають за допомогою StreamReader або StreamWriter.

Зміст роботи

Завдання 1.

Неймспейс System.IO надає набір класів для роботи з файлами та директоріями, такі як File, FileInfo, Directory, DirectoryInfo. Виконайте наступні дії, використовуючи згадані класи:

1. Створіть папку з назвою LabIO
2. Всередині цієї папки створіть текстовий файл log.txt
3. Запишіть рядок “Hello, world!” в створений файл
4. Допишіть у файл поточну дату і час
5. Після збереження файлу, виведіть список всіх файлів, які є в папці.

Завдання 2.

Використовуючи клас FileStream виконайте наступні дії з файлом, створеним в завданні 1:

1. Відкрийте файл log.txt використовуючи FileStream
2. Прочитайте його вміст побайтово та виведіть ASCII символи на екран

3. Допишіть рядок "Stream Test Complete." в кінець файлу.

Завдання 3.

Виконайте наступні дії з використанням класу MemoryStream, який дозволяє записувати дані в оперативну пам’ять:

1. Створіть екземпляр класу MemoryStream
2. Запишіть в нього рядок “Data for the memory stream”
3. Прочитайте записаний рядок з MemoryStream і виведіть його на екран.

Завдання 4.

Реалізуйте комунікацію між двома процесами використовуючи NamedPipeServerStream і NamedPipeClientStream. Для цього зробіть наступне:

1. Створіть два окремих консольних додатки: один із них має виступати у ролі сервера та генерувати повідомлення, інший має ці повідомлення отримувати.
2. Додаток-клієнт повинен виводити отримані повідомлення на екран.

Завдання 5.

При роботі з великими обсягами даних, які записані у файл або які треба записати в файл, можна стикнутися з проблемами з продуктивністю через постійну взаємодію з диском. Для ефективної роботи з великими обсягами даних використовують BufferedStream. Порівняйте продуктивність запису і читання даних з файлу з використанням BufferedStream та без нього:

1. Створіть FileStream
2. Запишіть довгий рядок у файл з його використанням та виміряйте час, який пішов на виконання даної операції

3. Прочитайте дані з файлу та виміряйте затрачений на це час
4. Повторіть операцію запису і читання тих самих даних у файл, але при цьому використайте BufferedStream. Виміряйте час, який пішов на виконання операції читання та запису даних з використанням BufferedStream.
5. Порівняйте швидкодію операцій з використанням BufferedStream та без.

Завдання 6.

Такі класи як StreamReader та StreamWriter дозволяють конвертувати потоки (streams) в текстовий ввід/вивід. Використайте StreamReader та StreamWriter для:

1. Запису множини рядків тексту у файл
2. Читання рядків один за одним і виведення їх на екран.

Завдання 7.

Для стиснення та розпаковування потоків даних можна використати клас GZipStream.

1. Створіть рядок і стисніть його з використанням GZipStream
2. Запишіть стиснені дані в файл
3. Прочитайте стиснені дані, розпакуйте їх та виведіть на екран оригінальний рядок.

Звіт та вихідний код завантажте на корпоративний git-репозиторій.

Список рекомендованої літератури

1. Albahari J. C# 12 in a nutshell: the definitive reference. First edition. Sebastopol, CA: O'Reilly Media, 2023. 1066 p.
2. Troelsen A. Pro C# 10 With . NET 6: Foundational Principles and Practices in Programming. 11th ed. Berkeley, CA: Apress L. P, 2022.
3. Richter J. CLR via C#. 4th ed. Redmond, Wash: Microsoft, 2012.
4. Офіційна документація .NET: https://learn.microsoft.com/en-us/dotnet/
5. Туторіали по C#: https://www.csharpcorner.com/
6. Офіційний репозиторій .NET Core: https://github.com/dotnet
7. Довідник з LINQ: https://linqpad.com/