МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з курсу «Комп'ютерні технології та програмування. Частина 1»

Харків

2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з курсу «Комп’ютерні технології та програмування. Частина 1»

для студентів спеціальності 174 – Автоматизація, комп’ютерно-інтегровані технології та робототехніка денної та заочної форми навчання

Затверджено
редакційно-видавничою
радою університету,
протокол № 1 від 16.03.2023

Харків

2023

Методичні вказівки до лабораторних робіт з курсу «Комп’ютерні технології та програмування. Частина 1» 174 – Автоматизація, комп’ютерно-інтегровані технології та робототехніка денної та заочної форми навчання /уклад. О. В. Пугановський, С. Д. Деменкова, О. Г. Шутинський – Харків: НТУ “ХПІ”, 2023. – 72 с .

Укладачі: О. В. Пугановський
С. Д. Деменкова
О. Г. Шутинський

Рецензент О. В. Дудник

Кафедра автоматизації технологічних систем та екологічного моніторингу

ВСТУП

Комп'ютерні технології стали основою сучасної цивілізації, їх інтегрували майже у всі області існування людини і у всі області виробництва. Постійно зростають вимоги до рівня знань комп'ютерної техніки і програмування. Особливі вимоги пред'являють до випускників вищих навчальних закладів.

Представлені методичні вказівки призначені для використання при виконанні лабораторного практикуму з курсу «Комп'ютерні технології та програмування». Методичні вказівки містять необхідний мінімум теоретичних відомостей, який забезпечує повне виконання поставлених завдань. Повний обсяг теоретичних знань, студенти отримують на лекціях.

При виконанні лабораторних робіт, студенти можуть вести як індивідуальну роботу так і створювати групи для вирішення поставлених завдань. Матеріал викладено таким чином, щоб студент мав змогу виконувати роботу як дистанційно так і у обладнаній лабораторії університету.

Для виконання робіт потрібно використовувати середовище Visual Studio C#. Для цього достатньо встановленої безкоштовної версії з офіційного додатку або роботи з будь-яким онлайн сервісом, що підтримує мову програмування C#.

Результатом виконання робіт є демонстрація робочого варіанту програми, що виконує задані функції. Захист робіт передбачає відповіді на теоретичні питання та демонстрація модифікацій розробленої програми у відповідності з завданням викладача.

Лабораторна робота №1
СЕРЕДОВИЩЕ РОЗРОБКИ *VISUAL STUDIO* 2019

Мета роботи: ознайомитись з середовищем розробки *Visual Studio* 2019. Набути практичних навичок із створення консольних проектів *C#*.

1.1. Основні відомості про *Visual Studio* 2019

Visual Studio 2019 (*VS*19) представляє собою потужний комплекс для розробки програмного забезпечення різними мовами програмування. Має платний та безкоштовний варіанти, що відрізняються функціональними можливостями. Програмний пакет встановлюється з офіційного сайту через обліковий запис користувача *Microsoft*. При інсталяції бажано обирати ті компоненти, які планується використовувати в подальшому або обрати стандартний набір компонентів і вподальшому за необхідності додавати необхідні компоненти.

В курсі «Комп'ютерні технології та програмування» буде використано консольні проекти *C#* у першій частині курсу та *Windows Forms C#* у другій частині курсу. Для виконання лабораторного практикуму можна використати будь-яку з версій *VS*, починаючи з 2009 року. Вимоги до обладнання та інструкції з установки знаходяться на сайті.

До переваг середовища *VS* відносяться розвинена система *Intelli Sense* та засобів відлагодження програм, можливість зберігання проектів на хмарному сервісі *GitHub*.

1.2. Створення консольного проекту

Після запуску *VS*, буде відкрите вікно привітання з варіантами наступних дій, рис. 1.1. У лівій частині показано останні проекти, у правій – варіанти відкриття чи створення нового проекту. Пункт «*Open a project or solution*» відкриває теку проектів в локальному репозиторії а пункт «*Clone a repository*» – у хмарних репозиторіях *GitHub* або *Azure DevOps*. Інші два пункти – відкриття

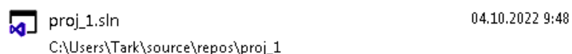
проекту з будь-якої теки на локальному компьютері чи створення нового проекту.

Visual Studio 2019

Open recent

Get started

Today



This week

Older

Continue without code →

Рисунок 1.1 – Вікно привітання *VS*

Для створення проекткту обирають пункт «*Create a new project*». За замовчуванням, налаштування проекту відповідає типу операційної системи, вибір мови програмування може бути одним з кількох, якщо при інсталяції встановлено кілька мов програмування. Для лабораторних робіт використовують мову *C#*, рис. 1.2.

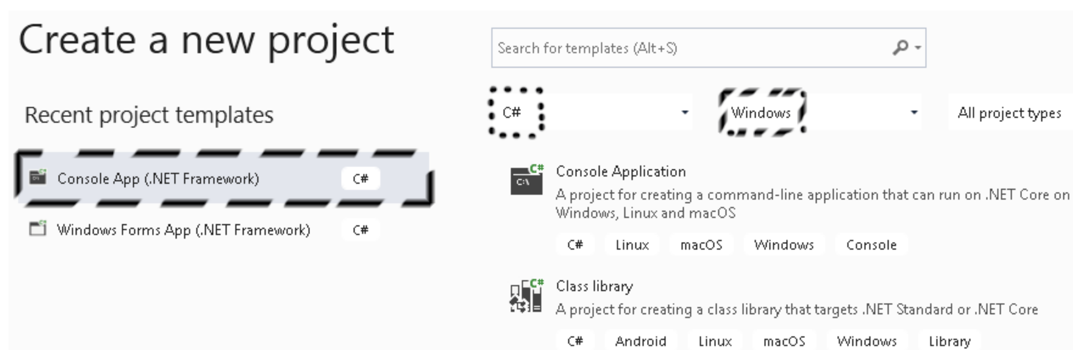


Рисунок 1.2 – Вікно створення проекту

Натискаючи кнопку «*NEXT*», переходять у вікно налаштувань проекту, рис. 1.3.

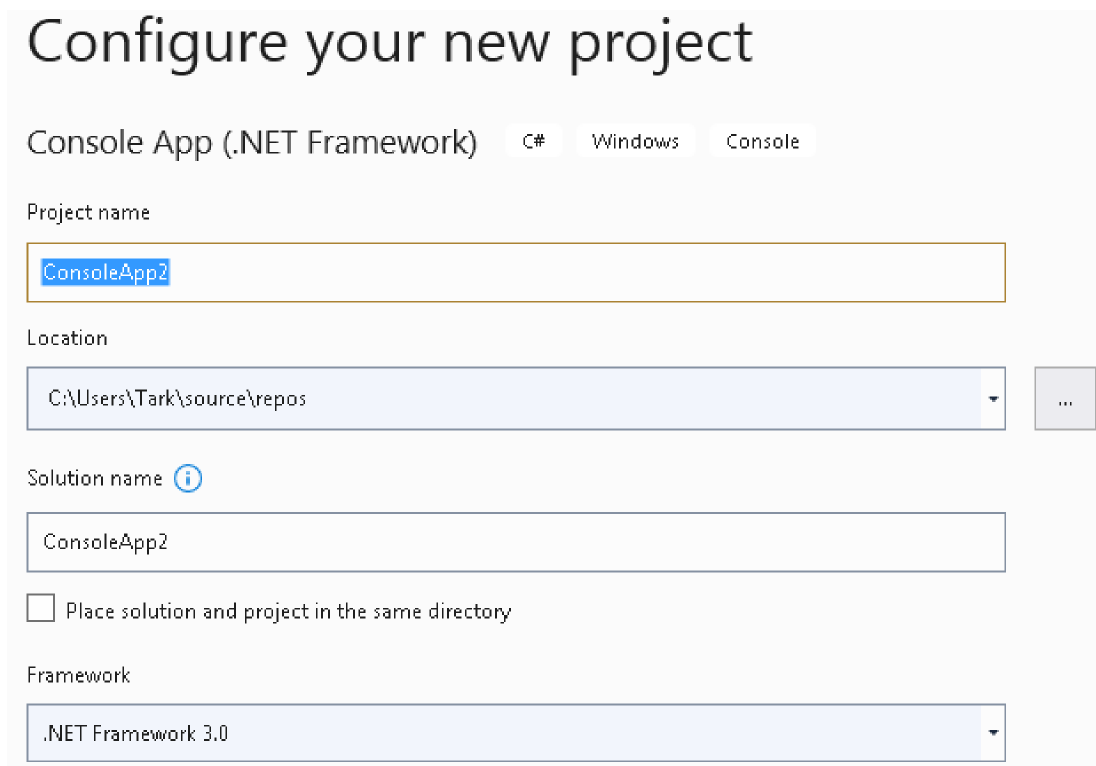


Рисунок 1.3 – Вікно створення проекту

Уважно обираємо опції на цій сторінці. Назву проекту бажано робити латиницею та такою, що однозначно описує ваш проект. Наприклад, «*Comp_tech_lab_1*» або «*Calc_component_costs*» та подібно. Не зважаючи, що сучасні додатки та операційні системи дозволяють використовувати в назвах різні значки, напрацюйте звичку використовувати тільки літери, цифри та знак підкреслення. Це допоможе уникати різних складнощів при роботі з файлами та файловою системою. Після зміни назви проекту, назва рішення буде змінена автоматично, але її можна змінити окремо. Це не вплине на назву проекту. Таким чином, для одного складного проекту можна створювати окремі рішення, що поєднуються в проект.

За замовчуванням, проекти зберігаються у теці «*repos*», що іноді не дуже зручно. Тому, за потреби, вказують свій шлях для збереження проекту.

Вибір версії *.NET Framework* є важливим моментом при створенні додатків. З одного боку, нові версії додають новий функціонал, з іншого – обмежують використання, так як на обладнані кінцевого користувача може бути встановленою більш стара версія. Тобто, якщо розроблюваний додаток не потребує новітнього функціоналу (опис є на сайтах розробників), то достатньо обрати версії 3.0 або 3.5.

Натискаючи кнопку «*Create*», створюємо новий проект.

1.3. Інтерфейс користувача *VS*

Як і всі сучасні програмні продукти, *VS* має стандартизований інтерфейс. У верхній частині розміщено меню, що дають доступ до найбільш вживаних функцій, рис. 1.4.

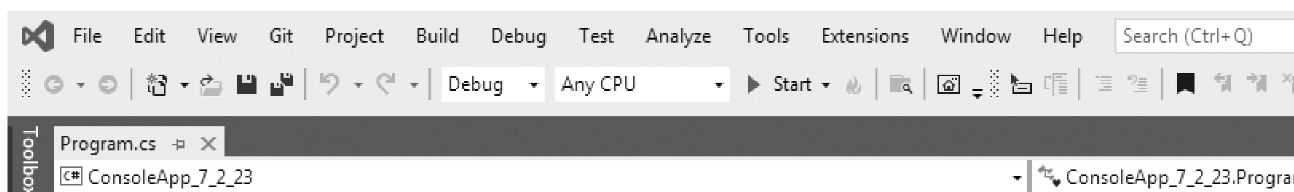


Рисунок 1.4 – Головне меню *VS*19.

При виконанні лабораторних робіт в основному використовуватимуться наступні розділи меню.

Група «*File*» – доступ до пунктів збереження та відкривання проектів.

Група «*Build*» – для перевірки та створення виконуваних файлів.

Група «*Debug*» – для відлагодження та запуску програм.

Більш детально склад і призначення цих груп розглянуто на лекціях та у виконанні робіт.

У центральній частині екрану відображається текст програми а у нижній частині екрану – вікно помилок та вихідних параметрів, рис. 1.5.

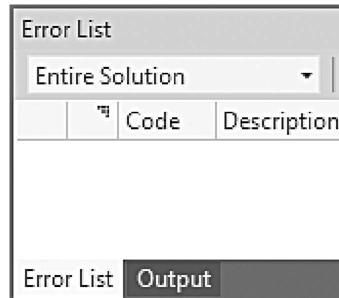


Рисунок 1.5 – Вікно помилок

Для запуску проекту на виконання потрібно натиснути F5, кнопку «Start» у меню або відкрити меню «*Debug*» і обрати пункт «*Start debugging*». За необхідності пошуку помилок у програмі, зручно використовувати точки зупинки. Після запуску програми буде виконано увесь код до цієї точки (рядка програми) і програма зупиниться. Після цього можна переглянути значення змінних у нижній частині екрану (вкладка «*Output*»), обираючи відповідні змінні зі списку. Більш зручний спосіб – навести курсор на змінну у тексті програми. Буде показане її поточне значення а для масивів можна відкрити список із усіма його елементами.

Для встановлення/вимкнення точок зупинки необхідно встановити курсор у відповідний рядок і натиснути клавішу *F*9. Після цього зліва з’явиться червона мітка і сам рядок буде виділено кольором. Повторне натискання *F*9 знімає точку зупинки. Таких точок можна встановити декілька а для подальшого виконання програми натискають *F*11.

1.4. Файли проекту

Консольні проекти складаються з багатьох файлів для яких створюється каталог проекту, шлях до якого було вказано раніше. Приклад дерева каталогу проекту показано на рис. 1.6. У середині теки з файлами проекту є тека з ідентичною назвою. У ній розміщено файл з назвою рішення і розширенням «*.csproj*». У наведеному прикладі – «*ConsoleApp_7_2_23.csproj*». Це головний

файл, що дає змогу коректно відкрити проект з усіма компонентами та налаштуваннями. Файл із розширенням «*.cs*» містить тільки текст програми.

Ще однією важливою текою є «Debug», що містить виконуваний файл із розширенням «*.exe*». Цей файл створюється після відлагодження та запуску програми і **може бути виконаний окремо !** Навіть на іншому ПК без *VS*19, за умови, що в операційній системі присутня необхідна версія фреймворку. Для даного прикладу шлях до файлу – «*ConsoleApp_7_2_23\ ConsoleApp_7_2_23\ bin\ Debug\ ConsoleApp_7_2_23.exe*»

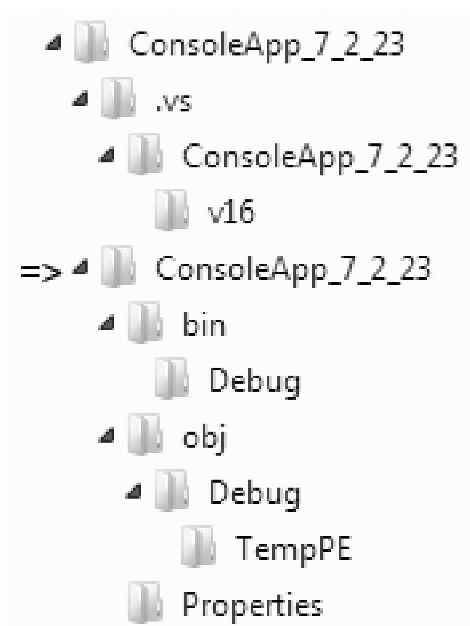


Рисунок 1.6 – Дерево каталогу проекту

1.5. Порядок виконання роботи

Необхідно створити проект *C#* із наступними налаштуваннями:

- Ім'я проекту «*My_proj*».
- Ім'я рішення «*Simle_proj*».
- Версія *.NET Framework* 3.5
- Шлях до проекту «*D:\temp\My_lab_proj*»

Після чого буде створено проект із початковим текстом програми, рис. 1.7.

Увага! Редагувати рядки тексту, що створені автоматично можна за умови розуміння наслідків ваших дій. На даному етапі їх не редагують.

Програма складається з двох секцій. Перша – з рядками «*using*», підключає необхідні компоненти до проекту. Друга – простір імен «*name space*», власне текст програми. Структура програми і її синтаксис розглядаються на лекції.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace ConsoleApp10
8  {
       0 references
9      class Program
10     {
           0 references
11         static void Main(string[] args)
12         {
13         }
14     }
15 }
16
```

Рисунок 1.7 – Стартовий текст програми.

Для подальшого виконання роботи необхідно в секцію (метод) «Main» додати наступні рядки:

int a = 5*;*

int b = 10*;*

int c = *a* + *b;*

Console.WriteLine("c = *"* + *c);*

Console.ReadKey();

Після чого встановити точку останову на рядку з «*Console.WriteLine*». Натиснувши *F*5 запустити програму на виконання. Коли програма зупиниться у вказаному рядку, наведіть курсор на імена змінних і затримайте на кілька секунд. Перегляньте значення змінних. Натисніть *F*11 кілька раз – поки не з'явиться чорний екран консолі з результатом виконання програми. Закрийте консоль.

Виберіть у меню «*File*» пункт «SaveAll» та закрийте *VS*19.

Перейдіть у теку з проектом. Перегляньте назви та розширення файлів. Запустіть на виконання файл з розширенням «*.exe*».

Для захисту лабораторної роботи необхідно розуміти виконані дії та пояснювати отримані результати.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Попередні налаштування проекту. На що впливає версія .NET?
2. Як коректно відкрити збережений проект?
3. Де розміщуються файли проекту?
4. Як відлагодити програму? Що таке точки зупинки?
5. Чи можна запускати програму без *VS*. Яким чином.

Лабораторна робота №2
ОСНОВИ СТВОРЕННЯ ПРОГРАМ МОВОЮ *C*#

Мета роботи: навчитись створювати програми арифметичних обчислень, використовувати базові типи даних та засвоїти принципи синтаксису мови C#

2.1. Структура програми

Як видно з рис. 1.7, програма складається щонайменше з двох секцій. Секція «*using*» містить бібліотеки та компоненти, що необхідні для роботи. Секція «*name space*» – це простір імен програми. За означенням, це структурний еле-

мент області дій, що поєднує в собі пов'язані елементи коду. У найпроситішому випадку можна вважати, що простір імен містить набір інструкцій програми.

Так, як мова програмування відноситься до об'єктно орієнтованих, то клас є структурним елементом, що містить самодостатній набір інструкцій (код програми). За замовчуванням, його іменують «*Program*» але це іменування можна змінити за потреби.

Основним елементом, з якого починається виконання інструкцій є метод «*Main*». Його також називають точкою входу у програму. Якщо в програмі відсутній метод *Main*, то програма не буде виконуватись. **Увага!** Не змінюємо і не видаляємо рядок методу *Main.* А при перенесені тексту програми, слідкуємо за наявністю цього рядка.

Весь текст програми розміщується у межах, що позначені фігурними дужками. Вони дозволяють оформлювати окремі структурні елементи. Кожен структурний елемент впливає на «видимість» змінних і без спеціальних модифікаторів, змінні будуть видимі у межах фігурних дужок. Правила застосування будуть роз'яснені у відповідних розділах.

Найпростіші програми – це набори інструкцій, що написані в межах методу *Main.* Виконання починається з першої іеструкції. Кожен рядок закінчується символом « **;** ».

Увага! Не всі інструкції повинні закінчуватись крапкою з комою. Для деяких складних операторів початком і кінцем є фігурні дужки. Крапка з комою перериває дію таких інструкцій.

2.2. Типи змінних

Для обробки інформації у програмах використовують різні типи даних. Перед використанням кожне значення необхідно пов'язати з його асоціативним відображенням (іменем). Розрізняють змінні значення та константи. Значення константи не може бути змінено в процесі виконання програми (властивість – read only). Для констант використовують відповідну інструкцію. У загальному

вигляді оголошення змінних та констант має однакову структуру:

```
const int a=3;
bool d = false;
string s = "This is my string";
char ch = ‘g’;
float ft;
```

Найбільш вживані типи:

bool – логічний, представляє значення *true/false*;

byte – 8-розрядний цілочисловий без знаку;

int – цілі числа;

decimal – числовий тип для фінансових обчислень;

float – з плаваючою точкою;

double – з плаваючою точкою подвійної точності;

long – Тип для представлення довгого цілого числа;

char– символьний;

string – рядковий.

Окремим типом даних є масиви. Наприклад, синтаксис для об’явлення масивів цілих чисел:

```
int[] z = new int [10];
int[,] z = new int [10,15];
```

Вимірність масивів позначається кількістю ком у квадратних дужках. Одновимірний – без коми, двовимірний – одна кома, тривимірний – три коми.

2.3. Арифметичні, математичні та логічні операції

Для виконання арифметичних дій використовують прості та складені оператори. Прості мають загальновживаний вигляд:

+ додавання;

- віднімання, унарний мінус;

* множення;

/ ділення;

% ділення по модулю;

-- декремент (зменшення на одиницю);

++ інкремент (збільшення на одиницю).

Складені оператори присвоюють значення результату дії змінній, над якою проводиться дія. Наприклад:

Оператор « += » записують, як $x += 10$.

Це аналогічно запису $x = x + 10$.

Для інших операцій записи подібні:

-= ; *=; /=; %=

Зверніть увагу! Оператори співвідношення, співставляють значення і у результаті дають значення *true* або *false*. Логічні оператори визначають різні способи поєднання істинних та хибних значень. Оскільки оператори співвідношення генерують значення *true/false* результати, то вони часто виконуються в поєднанні з логічними операторами.

== дорівнює;

! = не дорівнює;

> більше ;

< менше ;

>= більше чи дорівнює;

<= менше чи дорівнює.

До складних логічних виразів застосовуються звичайні правила математики.

Щоб використати складні математичні функції, у C# використовують бібліотеку «*Math*». Наприклад:

int x = Math.Sin(a); Отримання значення синусу числа *a*.

Переглянути доступні функції модулю можна у документації або поста-

вивши крапку після *Math.* – система *IntelliSense* відкриє список доступних службових слів з поясненнями.

2.4. Уведення і виведення даних в консольних програмах

Так, як *C#* зчитує консольну інформацію у вигляді рядків, то для уведення даних інших типів використовують операцію перетворення *Convert* або *Parse*. Формат запису при уведенні з консолі:

*Convert.To***ТИП***(Console.ReadLine()),*

де **ТИП** – назва типу. Наприклад, Int16, Int32, Double, Char, Float та інші. Перелік видно із випадаючого списку, приклади використання:

int a = Convert.ToInt16(Console.ReadLine());
double d = Convert.ToDouble(Console.ReadLine());

Формат перетворення за допомогою *Parse*:

ТИП.*Parse(Console.ReadLine())*

Для виведення даних у консоль використовують метод *Console.WriteLine().* У дужках записують ім'я змінної або текст. За необхідності, можна поєднувати кілька одиниць даних через знак « + »:

Console.WriteLine(a); – виведення значення змінної a.

Console.WriteLine("myval = " + myval); – виведення тексту *myval =* і значення змінної *myval.*

Console.WriteLine("уведіть значення числа a "); – виведення тексту.

2.5. Порядок виконнання роботи

1) Створіть новий проект з назвою *Lab*_2.

2) У метод *Main* додайте наступні інструкції:

```
Console.WriteLine("ente the number a (integer):");
int a = Convert.ToInt16(Console.ReadLine());


Console.WriteLine("ente the number b (double):");
double b = Convert.ToDouble(Console.ReadLine());


Console.WriteLine("ente the number c (double):");
double c = Convert.ToDouble(Console.ReadLine());
double x = a + b;
double y = a - b;

c =  x / y ;
Console.WriteLine("result:");
Console.WriteLine(c);

c = Math.Cos(c);
Console.WriteLine("result Math.Cos(c):");
Console.WriteLine(c);

c = a * (a + b) / b;
Console.WriteLine("result a * (a + b) / b:");
Console.WriteLine(c);

Console.ReadKey();
```

3) Запустіть програму на виконання. Якщо виникають помилки, уважно

перегляньте уведені інструкції та скорегуйте за необхідності.

4) Напишіть програми обчислення виразів:

$k = x*sin(f*a);$

$x = a^2 + b^2;$

$abc = sin(a) + 2*sin(b) – cos(a*b);$

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Як уводиться текстова та числова інформація в програмах на *C*#?

2. У чому відмінність використання математичних та арифметичних операцій?

3. У чому сенс використання рядка «*Console.ReadKey();*» ?

Лабораторна робота №3
СТВОРЕННЯ ПРОГРАМ З РОЗГАЛУЖЕННЯМИ

Мета роботи: Отримати навички створення програм з умовними операторами та використання логічних виразів

3.1. Основні відомості

Умовні оператори дозволяють перенаправляти виконання програми у залежності від результату логічної операції. Якщо умова істина, то виконується один блок коду 1, якщо хибна – блок коду 2. Цим виключається виконання обох блоків. Таким чином програма розгалужується на гілки виконання. Повний формат:

if (**умова**)
{ блок коду 1 }
else { блок коду 2}

Можливе скорочене використання конструкції, коли виконується або не виконується тільки блок 1. Наприклад:

```
if (a<b) c=a+b;
x = k / c;
```

Операція «*c=a+b*» буде виконана за умови «*a<b*» а операція «*x = k / c*» буде виконана завжди. У наступному коді, буде виконана тільки одна з операцій.

```
if (a < b) {c = a + b ;}
else {c = a - b ;}
```

Зверніть увагу на синтаксис – використання фігурних дужок і крапки з комою.

Допускається використання складених умов, їх синтаксис:

```
string name = "myName";
if (name == "Carl")
    Console.WriteLine("you name Carl");
else if (name == "July")
    Console.WriteLine("you name July");
else if (name == "")
    Console.WriteLine("you name Bob");
else
    Console.WriteLine("you name Unknown");
```

Зверніть увагу! В даному прикладі не використовують фігурні дужки. Крапки з комою завершують дію операторів *else* та *if*.

Для випадків, коли необхідно обрати один з кількох варіантів, зручніше використовувати не *else* та *if* а оператор *switch – case*. В ньому аналізується значення змінної і виконується одна з умов *case*. Обов’язковою інструкцією, що забезпечує роботу є *break*.

```
switch (змінна)
    {
    case { блок коду 1 }
    break;
    case { блок коду 2 }
    break;
      ……………..
    case { блок коду N }
    break;
    }
```

Кількість секцій *case* може бути значною. Допускається використовувати не блоки коду а одну операцію. Наприклад:

```
switch (a)
      {case 5: x = 5;
            break;
```

3.2. Умови та логічні вирази

Під умовами розуміють логічне значення чи результат співвідношення двох величин. Логічне значення може бути *true* або *false,* таке саме значення і результату співвідношення. Логічні вирази це кілька умов, що поєднані операторами співвідношення. Результат такого виразу також приймає одне з двох значень – *true* або *false.*

Приклади використання у операціях розгалуження:

*if(a < 0) {*блок 1*} else {*блок 2*}* Якщо *a* менше 0, то результат співвідношення буде *true* і виконається блок 1.

*if(((a < 0)|(b > 10)) & (c==k)) {*блок 1*} else {*блок 2*}* Блок 1 буде виконано, якщо результат виконання логічного виразу буде *true*.

Правила складання логічних виразів подібні до арифметичних. Якщо потрібно виконати обидві умови, то використовують оператори логічного множення (*AND*). Позначається « & » та « && ». Коли необхідно, щоб виконалась будь-яка з двох умов, використовують оператори логічного складання (*OR*). Позначається « | » та « || ».

Оператор заперечення (*NOT*) використовують для зміни значення логічної змінної на протилежне. Тобто, якщо результат *true* то він прийме значення *false* і навпаки. Позначається « ! ».

Операція *XOR* . Повертає *true*, якщо перший, або другий операнд (але не одночасно) рівні *true*, інакше повертає *false.* Позначається « ^ ».

Операції « | » та « || » (а також & і &&) виконують схожі дії, проте вони не рівнозначні. У виразі $h = x \mid y$; обчислюватимуться обидва значення – *x* та *y*. У виразі $h = x \,||\, y$; спочатку буде обчислюватися значення *x*, і якщо воно дорівнює *true*, то обчислення значення *y* вже не має сенсу, так як *h* у будь-якому випадку вже буде *true*. Значення *y* буде обчислюватися тільки в тому випадку, якщо *x* дорівнює *false.* Те саме стосується пари операцій & і &&.

Тому операції « || » і « && » більш зручні у обчисленнях, оскільки дозволяють скоротити час на обчислення значення виразу, і тим самим підвищують продуктивність. А операції « | » і « & » більше підходять для виконання порозрядних операцій над числами.

3.3. Порядок виконнання роботи

1) Створіть новий проект з назвою *Lab*_3.

2) Перегляньте приклад програми для обчислення коренів квадратного рівняння і проаналізуйте використання умовних операторів.

```
Console.Write("a = ");
var a = double.Parse(Console.ReadLine());
Console.Write("b = ");
var b = double.Parse(Console.ReadLine());
Console.Write("c = ");
var c = double.Parse(Console.ReadLine());

double x1, x2;
var discriminant = Math.Pow(b, 2) - 4 * a * c;
if (discriminant < 0)  //не має коренів
{
   Console.WriteLine("do not solve");
}
else
{
   if (discriminant == 0)  //два однакові корені
   {
      x1 = -b / (2 * a);
      x2 = x1;
   }
   else //два різні корені
   {
      x1 = (-b + Math.Sqrt(discriminant)) / (2 * a);
      x2 = (-b - Math.Sqrt(discriminant)) / (2 * a);
   }
   //вивід результатів
   Console.WriteLine($"x1 = {x1}; x2 = {x2}");
}

Console.WriteLine("Для выхода нажмите любую клавишу...");
```

Console.ReadKey(true);

3) Напишіть програму обчислення виразів:

а) $x = a + b$, якщо $a>0$ і $b>0$

$x = a - b$, якщо $a>0$ і $b<0$

б) $x = a / b$, якщо b не дорівнює 0

$x = a / 2$, якщо b дорівнює 0

в) $x = (a + c) / b$, якщо b не дорівнює 0 та $c < 0$

$x = a + c$, якщо b дорівнює 0

КОНТРОЛЬНІ ЗАПИТАННЯ

1) Що таке логічний вираз?

2) Як порівняти дві змінні?

3) Які оператори розгалуження ви знаєте?

4) Коли доцільно використання *switch – case?*

5) Перелічіть логічні оператори.

Лабораторна робота №4
ВИКОРИСТАННЯ ЦИКЛІВ У КОНСОЛЬНИХ ПРОГРАМАХ

Мета роботи: Отримати навички роботи з циклами при створенні програм мовою *C*#.

4.1. Цикли з лічильником

Використовують для багатократного повторення обчислень з блоку коду, що називають тілом циклу. Загальний формат запису циклу *for* має наступний вигляд:

for(ініціалізація; умова; ітерація) {блок коду (тіло циклу)}

Цикл повторюється від значення змінної (лічильника), що задана в ініціалізації поки виконується умова з кроком, що заданий ітерацією. Приклади:

*for (int i = 0; i < 10; i++){*блок коду *}*
*for (int i = 100; i > 0; i = i - 5){*блок коду *}*

Особливістю даного циклу є необов'язковість опцій, зазначених в дужках. У граничному випадку, має сенс запис:

*for (;;) {*блок коду*} //* безкінечний цикл.

При ініціалізації лічильника за межами оператора *for*, його можна не описувати в дужках. Відсутність умови робить цикл нескінченним і для його переривання, в тілі циклу необхідно передбачити умову виходу з оператом *break.* Ітерація може бути відсутньою, якщо в тілі циклу є механізм обчислення значення змінної лічильника. Найчастіше використовують у випадках, коли перед початком циклу відомі його параметри – початкове і кінцеве значення, крок ітерації.

Для переривання усіх циклів можна використовувати оператор *break* а для пропуску інструкцій з блоку коду – оператор *continue*.

```
for (int i = 0; i < 9; i++)
{
        if (i == 5)
        continue;
        Console.WriteLine(i);
}
```

4.2. Цикли з передумовою

Використовують для випадків, коли потрібно провести циклічні обчислення не оголошену кількість разів. Цикл виконується, поки дійсна умова на початку. Часто у якості умови використовують складні інструкції. Формат:

*whi1e (*умова*) {*блок коду (тіло циклу)*}*

Наприклад, уведення з консолі, поки не буде уведено слово «*stop*»:

*while ((s = Console.ReadLine()) != "stop") {*блок коду (тіло циклу)*}*

4.3. Цикли з післяумовою

На відміну від попередніх циклів, цикл *do-while* перевіряє умову після виконання дій. Це означає, що цикл *do-while* завжди виконується хоча б один раз. Тому всі змінні що задіяні в циклі повинні мати початкове значення і в обчисленнях не повинно бути невизначених ситуацій.

Загальний формат :

*do {*блок коду (тіло циклу)*} while (*умова*);*

Незважаючи на те що фігурні дужки необов'язкові, якщо елемент інструкції складається тільки з однієї інструкції, вони часто використовуються для поліпшення читабельності конструкції *do-while*, не допускаючи тим самим плутанини з циклом *while*. Цикл *do-while* виконується, поки залишається істинною умова.

4.4. Цикли *foreach*

Призначений для перебору всіх елементів масиву або колекції. Формат:

foreach(тип_даних змінна *in* колекція) *{*блок коду (тіло циклу)*}*

Змінній покроково присвоюється значення елементів колекції від першого до останнього. Перевага такого циклу – немає необхідності визначати кількість елементів в колекції та задавати умови.

Наприклад для набору (масиву), що містить назви машин, роздрукуються всі назви.

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars) { Console.WriteLine(i); }
```

4.5. Порядок виконання роботи

1) Створіть новий проект з назвою *Lab*_4

2) Напишіть програму підсумовування 10 цілих чисел, що вводять з консолі за допомогою циклу *for*.

3) Переробіть попередній код так, щоб підсумовувались значення цілих чисел, що вводять з консолі, поки не буде уведено слово «*stop*». Підказка: значення вводять у циклі *while* у вигляді рядкових значень а потім конвертують у цілий тип.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Для чого призначені цикли?

2. Які особливості керування роботою циклу *for*.

3. Переваги і недоліки використання циклів з передумовою та післяумовою.

4. Призначення циклу *foreach*

Лабораторна робота №5
РОБОТА З ТЕКСТОВОЮ ІНФОРМАЦІЄЮ

Мета роботи: закріпити знання про літерали, отримати навички роботи з рядковими змінними, їх уведенням, перетворенням і виведенням.

5.1. Літерали, плейсхолдери та управляючі послідовності

У *C#* літералами називаються постійні значення, представлені у зручній для сприйняття формі і можуть бути будь-якого простого типу – ціле число, текст, символ та інші. Подання кожного літералу залежить від конкретного типу.

Шістнадцяткові літерали повинні починатися із символів « ***0x*** ». Приклади шістнадцяткових літералів:

count = 0xFF; // дорівнює 255 у десятковій системі
incr = 0x1a; // дорівнює 26 у десятковій системі

Значення, задані послідовністю біт, записують через « ***0b*** »:

int b = 0b0010;

Інформація, що вводиться за допомогою методів *Read()* та *Readline()*, повинна бути примусово перетворена у необхідний тип за допомогою *Parse* або *Convert,* як було показано в попередніх лабораторних роботах. Виключенням є уведення рядкових (текстових) літералів – вони не потребують перетворення.

Виведення інформації супроводжується автоматичним перетворенням у текст. Методи *Write()* та *WriteLine()* відрізняються тільки автоматичним переведенням рядка для останнього. Для обох методів доступні опції для візуального оформлення (форматування) за допомогою плейсхолдерів – спеціальних послідовностей у середині тексту, що виводиться:

```
int x = 10;
int y = 55;
Console.WriteLine("Текст {0} текст {1}", x, y);
```

У прикладі, що наведено, буде роздруковано текст із вставками значень змінних *x, y* на місцях, позначених фігурними дужками (плейсхолдерів) – «Текст 10 текст 55». Цифри у плейсхолдерах відповідають номерам змінних, що перелічені після тексту. Нумерація автоматична і починається з нуля. В прикладі, змінній *x* відповідає номер 0 а *y* – номер 1.

Увага ! Номери в плейсхолдерах не обов'язково виставляти послідовно. На відміну від автоматичної нумерації, в плейсхолдерах може бути будь-який порядок. Наприклад:

```
Console.WriteLine("Текст {3} текст {0} Текст {1} текст {2} ", x, y, k, d);
```

Буде виведено текст із вставками значень змінних у послідовності *d, x, y, k.*

У версіях *VS*19 і вище, додано спосіб виводу з посиланням на відповідні змінні без їх дописування у методі *WriteLine().* Для цього використовують знак *$* перед рядком:

```
string name = "Tom";   int age = 34;   double height = 1.7;
Console.WriteLine($"Ім'я: {name} Вік: {age}  Зріст: {height}м");
```

Додатково, у плейсхолдерах можна вказувати формати виводу. Всі можливі формати розглядають на лекціях або можна звернутись до документації. Приклади формату запису плейсхолдерів і результату виводу числа 23:

{0:d}   результат 23, десятковий формат.

{0:d4} результат 0023, десятковий формат.

{0:f} результат 23,00, дробовий з фіксованою точкою.

{0:f4} результат 23,0000, дробовий з фіксованою точкою.

{0:f1} результат 23,0, дробовий з фіксованою точкою.

{0:e} результат 2,300000e+001, експоненційний.

{0:e2} результат 2,30e+001, експоненційний.

5.2. Розбивання та об’єднання рядків

Двома важливими операціями обробки рядків є розбиття (декомпозиція) та складання. При розбитті рядок поділяється на складові. А при складанні рядок "збирається" з окремих частин. Для розбиття рядків у класі *String* визначено метод *Split()*, а для складання – метод *Join()*. Формати використання методу *Split()*:

```
public string[] Split(params char[] seps);
public string[] Split(params char[]seps, int count);
```

Наприклад, масив *seps,* містить набір символів (пробіл і кома), по яким буде розбиватись вхідний рядок *str*. Результат розбивання буде передано у рядковий масив *parts*:

```
char[] seps = {‘ ‘, ‘,’};
string[] parts = str.Split(seps);
```

Увага! Розмір масиву *parts* система визначає автоматично. Якщо розмір масиву наперед заданий, то можна використати метод *Split* з опцією *count,* що задає кількість елементів після розбивання. Величина *count* не повинна перевищувати розмір масиву *parts* .

Більш прийнятний формат методу, дозволяє видалити пусті значення, що утворюються, якщо вхідний рядок містить кілька символів розбивки, що йдуть підряд. Наприклад, пробіли.

```
string[] parts = text.Split(new char[] {‘ ‘},
StringSplitOptions.RemoveEmptyEntries);
```

Для об’єднання рядкових змінних можна використати метод *Join*, що дозволяє з рядкового масиву отримати один рядок з розділовим символом *sep*, що заданий:

```
public static string Join(string sep, string[] strs)
public static string Join(string sep, string[] strs, int start, int count)
```

Метод *Join()* у першому форматі повертає рядок, який містить конкатеновані рядки, передані в масиві *strs*. Другий формат дозволяє зібрати рядок, який міститиме *count* конкатенованих слів, починаючи з елементу масиву *start*.

Приклад використання методу *Split*.

```
string[] input = {"Це рядок, що буде розбито_на_слова"}
char[] seps = {‘ ‘, ‘,’, ‘_’} ;
string[] parts = input[i].Split(seps, StringSplitOptions.RemoveEmptyEntries);
```

Приклад програми, що демонструє розбивання і об’єднання рядків.

```
using System;
class SplitAndJoinDemo
{
public static void Main()
{
```

```
string str = "This is my line of text to break into words. Then, I will connect the words into a line.";

char[] seps = {' ', ','}; // Увага при копіюванні!!! Символи у лапках – пробіл та кома

string[] parts = str.Split(seps);   // Розділення

Console.WriteLine("Spit result : " ) ;
for(int i=0; i < parts.Length; i++)
Console.WriteLine(parts[i]);

string whole = String.Join(" | ", parts);  // Об'єднання

Console.WriteLine("Joint result : " ) ;
Console.WriteLine(whole);
}
}
```

5.3. Порядок виконання роботи

Завдання 1. Створити програму-калькулятор, що запитує у користувача рядок з двох чисел і арифметичного оператора. Наприклад, «12 + 35». Після виконання заданої дії, результат виводиться в консоль. Кількість арифметичних операторів обмежена і наперед відома. Наприклад, «+», «-», «/», «*».

Зауваження і пояснення. Зверніть увагу – числа відділені від оператору пробілами. Для розбивки вхідного рядка використати метод *Split*. Для аналізу оператора – оператор *switch*.

Завдання 2. Створити програму, що запитує у користувача кілька слів і виводить на консоль фразу, слова якої розділені дефісами. Використати метод *Join.*

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що означає термін «плейсхолдер»?

2. Для чого потрібні літерали?

3. Як вивести результати у вигляді таблиці, використовуючи форматування?

4. Як визначається розмір вихідного масиву при розбиванні рядка?

5. Які символи можна використовувати при розбиваннях і об'єднаннях рядків.

Лабораторна робота № 6
РОБОТА З ФАЙЛОВОЮ СИСТЕМОЮ

Мета роботи: набути навичок використання файлів у консольних програмах *C*#. Ознайомитись з відмінностями різних методів, призначених для роботи з файлами.

6.1. Основні відомості про потоки уведення/виведення

Файл – це впорядкована та іменована послідовність байтів, що має постійне сховище. При роботі з файлами використовуються шляхи до каталогів, пристрої, що запам'ятовують, а також імена файлів і каталогів. На відміну від файлу, потік – це послідовність байтів, яку можна використовувати для запису або читання з допоміжного пристрою, що є одним з пристроїв зберігання інформації (наприклад, дисків або пам'яті). Існує кілька видів запам'ятовуючих пристроїв, відмінних від дисків, і існує кілька видів потоків, крім файлових потоків, наприклад мережеві потоки, потоки пам'яті та потоки каналів.

Базовий клас *Stream* підтримує читання та запис байтів, його похідні класи забезпечують загальний спосіб перегляду джерел даних та сховищ об'єктів, а також ізолюють програміста від специфічних особливостей операційної системи та базових пристроїв.

Потоки включають три основні операції. Читання – перенесення інформації з потоку структуру даних, таку як масив байтів. Запис – перенесення даних у потік із джерела даних. Пошук – Визначення та зміна поточної позиції всередині потоку.

Часто використовувані класи потоку:

FileStream – для байтового читання та запису у файл.

MemoryStream – для читання та запису в пам'ять як резервне сховище.

BufferedStream – для підвищення швидкодії операцій читання та запису.

NetworkStream – для читання та запису на мережеві сокети.

Часто використовувані класи для читання та запису:

BinaryReader та *BinaryWriter* — для читання та записування простих типів даних, таких як двійкові значення.

StreamReader і *StreamWriter* — для читання та запису символів за допомогою закодованого значення для перетворення символів на байти або з байтів.

StringReader та *StringWriter* — для читання та запису символів у рядки або з рядків.

TextReader та *TextWriter* використовуються як абстрактні базові класи для інших засобів читання та запису, які зчитують і записують символи та рядки, а не двійкові дані.

6.2. Робота з файлами на основі байтового потоку *FileStream*

Увага! При роботі з файлами обов'язково дотримуватись наступних правил:

1) Додати бібліотеку уведення-виведення *using System.IO*.

2) Зв'язати змінну файлового потоку з файлом.

3) Провести необхідні операції зчитування/запису.

4) Закрити файловий потік.

Так як при роботі з файлами створюється їх копія з якою працюють, то остання операція оновлює файл даними, над якими проводились операції зчитування/запису. Без закривання потоку, данні не будуть внесені у файл.

Найчастіше використовується наступний формат:

FileStream (string filename, FileMode.mode, FileAccess.access);

Тут елемент *filename* означає ім'я файлу, який необхідно відкрити, причому воно може включати повний шлях до файлу. Елемент *mode* означає, як саме повинен бути відкритий цей файл, або режим відкриття. Елемент mode може приймати одне із значень: *Append*, *Create*, *CreateNew*, *Open*, *OpenOrCreate*, *Truncate*.

Найчастіше використовують опцію *OpenOrCreate* або *Append*. У першому випадку використовується вказаний файл а якщо він не існує, то створюється новий. У другому випадку, дані будуть дописані в кінець файлу, що вже існує.

Опція *FileAccess* не обов'язкова і дозволяє задати режим доступу *Read*, *Write*, *ReadWrite*. Приклад використання:

FileStream fin = new FileStream(@"D:\test.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

Увага! Символ « @ » вказує на керуючу послідовність та є обов'язковим.

Приклад зчитування даних з файлу з виведенням на консоль.

```
{
int i;
FileStream fin = new FileStream(@"D:\test.txt", FileMode.Open);
do
```

```
{
    i = fin.ReadByte();
    if(i != -1) Console.Write((char) i);
}
while(i != -1);
fin.Close();
}
```

6.3. Робота з файлами на основі *StreamReader* і *StreamWriter*

Для роботи з текстовою інформацією найчастіше використовують потоки *StreamReader* і *StreamWriter*.

Для запису текстового файлу використовується клас *StreamWriter.* Найбільш вживаний формат запису:

```
StreamWriter(string path);
StreamWriter(string path, bool append);
StreamWriter(string path, bool append, System.Text.Encoding encoding);
```

Через параметр *path* передається шлях до файлу, який буде пов'язаний із потоком, параметр *append* вказує, чи треба додавати в кінець файлу дані або перезаписувати файл. Якщо дорівнює *true*, нові дані додаються в кінець файлу. Якщо одно *false*, то файл перезаписується наново. Параметр *encoding* вказує на кодування, яке застосовуватиметься під час запису

Як і у разі використання *FileStream*, після використання або перед завершенням роботи програми, необхідно використати метод *Close()*. За необхідності оновлення файлу під час роботи програми можна скористатись методом *Flush()*. Файловий потік не буде закрито але всі зміни будуть перенесені з віртуального потоку у фізичне сховище – файл.

Для запису у файл використовують методи, схожі на консольні. Наприклад, для змінної *myDat* відбувається запис даних з лічильника у циклі. На кожному кроці файл оновлюється, після завершення циклу – закривають файловий потік.

```
StreamWriter myDat = new StreamWriter(@"D:\test.txt");
    for (int i = 0; i < 10; i++)
    {
        myDat.WriteLine("i = " + i);
        myDat.Flush();
    }
myDat.Close();
```

Увага! Обов'язково дописуємо рядок *using System.IO*.

Для зчитування інформації з файлу використовують методи класу *StreamReader*. Найбільш вживані формати використання:

```
StreamReader(string path);
StreamReader(string path, System.Text.Encoding encoding);
```

Опції аналогічні тим, що розглянуті для *StreamWriter*. Так, як зчитування не модифікує дані у файлі, то використовувати метод *Close()* не обов'язково але бажано. Зв'язок з файлом буде перервано після завершення програми і система автоматично «прибере сміття» з пам'яті.

Приклад зчитування інформації з файлу наведено нижче. Дані зчитуються з першого по останній рядок. Для цього використано цикл з передумовою. Відбувається перевірка зчитаного значення на відповідність значенню *null* – система нічого не зчитала.

```
StreamReader myDat = new StreamReader(@"D:\test.txt", Encoding.Default);
string str;
while((str = myDat.ReadLine()) != null)
    {
    Console.WriteLine(str);
    }
```

6.4. Робота з файлами на основі *BinaryWriter* та *BinaryReader*

Для роботи з бінарними файлами призначені класи *BinaryWriter* та *BinaryReader*. Ці класи дозволяють читати та записувати дані у двійковому форматі і дуже зручні для роботи з бінарними даними. На відміну від попередніх, ці класи зберігають і зчитують числа і текст не у зручному для людини, символьному представлені а у машинному. Відповідно, якщо у файл записують числа в певному форматі, то для зчитування потрібно використовувати такий формат. Наприклад, *Int*16, записується двома байтами і спроба зчитати інформацію як *Int*32 (чотири байти) приведе до отримання некоректних даних.

Для створення відповідного екземпляру, використовують формат:

```
BinaryWriter(Stream stream);
```

У якості вхідного/вихідного потоку використовують потоки *FileStream*:

```
FileStream fout = new FileStream(@"D:\test.txt", FileMode.OpenOrCreate);
BinaryWriter bw = new BinaryWriter(fout);
```

Крім стандартних методів *Write(),* для запису даних та *Close()* для завершення роботи з потоком, є метод *Seek()*, що встановлює позицію у записуваних/зчитаних даних. Так як потоки можуть бути представлені як віртуальний одновимірний масив, то позиція – це індекс масиву, що розраховується шляхом

розділення потоку на основі формату даних. Наприклад, по два або чотири байти.

```
FileStream fin = new FileStream(@"D:\test.txt", FileMode.Open);
BinaryReader br = new BinaryReader(fin);
```

6.5. Порядок виконання роботи

1) Завдання. Написати програму для запису рядкових даних у файл *fileOut*, що вводять з консолі.

2) Написати програму, що записує у файл *fileOutKey* дані про натиснені клавіші.

Пам'ятка. Для використання даних про натиснені клавіші використовують пару *ConsoleKeyInfo* та *Console.ReadKey()*.

Приклад виводу на консоль:

```
ConsoleKeyInfo ki;
for (int i = 0; ; i++)
    {
    ki = Console.ReadKey();
    char c = ki.KeyChar;
    int d = Convert.ToInt16(ki.Key);
    Console.WriteLine("character -> " + c + "  number -> " + d);
    }
```

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Які є методи для роботи з файлами?

2. Чим відрізняються символьний потік від байтового?

3. Чим відрізняється збереження чисел у символьному форматі та у бінарному?

4. Як забезпечити виведення тексту у національному кодуванні?

5. З якою метою використовують потік із збереженням у оперативній пам'яті?

Лабораторна робота № 7

РОЗРОБКА ПРОГРАМ НА ОСНОВІ БАЗОВИХ АЛГОРИТМІВ. Ч1

Мета роботи: ознайомитись з алгоритмами і прикладами реалізації базових алгоритмів сортування.

7.1. Загальні відомості про алгоритми сортування

Алгоритми сортування у поєднанні з алгоритмами пошуку – це основа обробки сучасних масивів інформації. Для сортування було розроблено велику кількість алгоритмів, що обумовлено їх обмеженною ефективністю для різних типів даних та умов використання. До умов використання відносять в першу чергу, спосіб роботи з пам'ятю – чи потрібна додаткова пам'ять для роботи та збереження результатів. Також можливе розміщення масиву як в пам'яті так і на пристроях збереження даних. У сучасних системах можливе одночасне використання кількох потоків.

Під ефективністю алгоритму мається на увазі час, за який повинен виконуватись даний алгоритм. Цей час складається з часу виконання кожного оператора та частоти виконання кожного оператора. Існує наступна класифікація алгоритмів за ефективністю для N елементів:

- константний, позначають $O(1)$;
- логарифмічний $O(logN)$;
- лінійний $O(N)$;
- лінійно-логарифмічний $O(NlogN)$;
- квадратичний $O(N^2)$;
- кубічний $O(N^2)$.

Наглядна демонстрація ефективності роботи алгоритмів у залежності від кількості елементів масиву, наведена на рис. 8.1

Більшість алгоритмів сортування розглядаються на лекціях. В лабораторній роботі розглянемо кілька простих алгоритмів.

7.2. Сортування бульбашкою

Сутність алгоритму – порівняння сусідніх елементів. Якщо їх розміщення відповідає заданому (зростання чи спадання), елементи залишають на своїх місцях. У протилежному випадку – міняють місцями. Якщо після такого аналізу не зроблено жодної перестановки, масив вважають відсортованим. Це самий повільний алгоритм сортування.

Варіант найпростішої реалізації, без аналізу кількості перестановок:

```
int temp;
```

```
int len = a.Length;
for (int i = 1; i < len; i++)
{
   for (int j = 0; j < len - i; j++)
   {
      if (a[j] > a[j + 1])
      {
         //Swap a1 - a2
         temp = a[j];
         a[j] = a[j+1];
         a[j+1] = temp;
      }  }  }
```

7.3. Шейкерне сортування

Шейкерне сортування є модифікацією попереднього алгоритму. Відмінність полягає в тому, що аналіз і перестановки роблять в обох напрямках – від меншого до більшого і потім від більшого до меншого. Це прискорює роботу, так як відбувається впорядкування з двох напрямків.

Реалізація програми показана нижче.

```
int[] arr = new int[10000];
         Random rnd = new Random();
         int temp;
         for (int i = 0; i < 10000; i++)
         { arr[i] = rnd.Next(1000); }
         var sw = new Stopwatch();    // запуск таймеру діагностики
         sw.Start();

         for (var i = 0; i < 10000 / 2; i++)
         {
```

```
    var swapFlag = false;
    // прохід з ліва направо
    for (var j = i; j < 10000 - i - 1; j++)
    {
        if (arr[j] > arr[j + 1])
        {
            //Swap a1 - a2
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
            swapFlag = true;
        }
    }
    // прохід з права на ліво
    for (var j = 10000 - 2 - i; j > i; j--)
    {
        if (arr[j - 1] > arr[j])
        {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
            swapFlag = true;
        }
    }
    //якщо обмінів не було, то завершуємо
    if (!swapFlag)
    {
        break;
    }
}
```

```
sw.Stop(); // зупинка таймеру
Console.WriteLine(sw.Elapsed); // друк часу виконання
Console.ReadKey();
```

Програма доповнена двома опціями. Перша – аналіз перестановок. Якщо їх не було, то масив відсортований, можна завершити роботу.

Друга опція – визначення часу виконання програми. Для цього у секцію *using* додають рядок *System.Diagnostics*. Це дозволяє використати спеціальний таймер *Stopwatch* та визначити проміжок часу між його запуском і зупинкою.

7.4. Порядок виконання роботи

Завдання. Використавши наведені програми, створити нову. Вона повинна включати в себе обидва методи сортування та відображати час виконання. Для визначення часу, фрагменти коду, що відповідають за сортування розміщують між рядками запуску і зупинки таймеру

Використати окремі масиви на 10000 елементів для обох методів.

Після відлагодження програми, доповнити її виводом на консоль п'яти перших і п'яти останніх елементів масивів.

Порівняти час виконання обох алгоритмів сортування.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Які алгоритми сортування ви знаєте?
2. Що таке «ефективність алгоритму»?
3. У чому відмінність сортування бульбашкою та шейкерного?
4. Як працює сортування бульбашкою?
5. Як відрізняється час сортування бульбашкою та шейкерним методом?

Лабораторна робота № 8

РОЗРОБКА ПРОГРАМ НА ОСНОВІ БАЗОВИХ АЛГОРИТМІВ. Ч2

Мета роботи: ознайомитись з алгоритмами і прикладами реалізації базових алгоритмів пошуку.

8.1. Загальні відомості про алгоритми пошуку

Пошук елементів із заданими властивостями у масивах інформації є однією із найрозповсюдженіших задач і відноситься до базових алгоритмів. Алгоритми пошуку можна розділити на дві групи. Перша – пошук у не відсортованих даних, друга – у попередньо відсортованих. Не сортовані дані практично завжди аналізуються перебором всіх елементів масиву. Для відсортованих даних розроблено кілька алгоритмів пошуку, що відрізняються своєю ефективністю.

Прості алгоритми пошуку – це лінійний (послідовний) та бінарний. Лінійний пошук використовують для не відсортованих даних. Його сутність у порівнянні кожного елементу масиву із заданим (ключем). При співпаданні, найчастіше повертають індекс елемента. Як варіант – аналіз всього масиву із підраховуванням кількості елементів, що відповідають заданому ключу.

Бінарний пошук – це пошук елементу у попередньо відсортованих масивах даних для встановлення його наявності чи індексу. Масив поділяють на дві частини і спочатку проводять пошук в одній половині а потім в другій. Як варіант, масив ділять на дві нерівні частини. Наприклад, за правилом золотого перетину.

Для більшості мов програмування, існують вбудовані методи пошуку в масивах. В їх основі стандартні алгоритми.

Алгоритм бінарного пошуку для масиву відсортованого за зростанням складається з наступних кроків:

1) Визначають значення елемента в середині робочої області масиву даних та порівнюємо його з тим, що шукають. Можливі варіанти – обрати не се-

редину масиву (дихотомія) а у якомусь співвідношенні. Наприклад, 2/3 і 1/3 або золотий перетин;

2) Якщо значення співпало із пошуковим, повертаємо його індекс;

3) Якщо отримане значення більше ніж шукане, то пошук продовжується в нижній (лівій, з меншими елементами) частини масиву, інакше аналізуємо верхню (праву, з більшими елементами);

4) Встановлюємо нові межі робочої області. Перевіряємо, чи не дорівнює верхня (права) межа нижній (лівій) , якщо так – шуканого значення немає, інакше – переходимо на перший крок.

Приклад реалізації.

```
while (left <= right)
{
    var middle = (left + right) / 2;
    if (searchedValue == array[middle])
    {
        return middle;
    }
    else if (searchedValue < array[middle])
    {
        right = middle - 1;
    }
    else
    {
        left = middle + 1;
    }
}
return -1;
```

8.2. Порядок виконання роботи

Завдання 1. Розробити програми лінійного та бінарного пошуку, що проводять пошук заданого цілого числа у файлі.

Виконання завдання складається з наступних підзавдань:

- зчитування даних з файлу, перетворення у цілі числа і запис у масив;
- запит ключа у користувача;
- пошук елементу за одним із алгоритмів і відображення у консолі його індексу.

Перед початком розробки програми пошуку, рекомендується створити програму генерації файлу з випадковими цілими числами. Приклад такої програми, що генерує файл із ста числами в діапазоні від 0 до 100:

```
StreamWriter sw = new StreamWriter(@"Q:\temp\arr.txt");
         Random rnd = new Random();
         int k;
         for (int i = 0; i < 100; i++)
         {
            k = rnd.Next(1000);
            sw.WriteLine(k);
         }
         sw.Close();
```

Для бінарного пошуку необхідно відсортувати зчитані дані, як у попередній лабораторній роботі.

Завдання 2. Розробити програму лінійного пошуку з відображенням в консолі індексів усіх елементів, що співпадають із заданим. Для роботи програми використати масив, що містить 10000 цілих, випадкових чисел у діапазоні 0 – 500.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Які алгоитми пошуку ви знаєте?
2. Для якого алгоритму не потрібне попереднє сортування масиву?
3. З якою метою використовують розділення масиву на частини?

Лабораторна робота № 9
МЕТОДИ КОРИСТУВАЧА

Мета роботи: отримати практичні навички створення методів користувача з використанням перевантажень.

9.1. Загальні відомості

Методи представляють собою завершений блок інструкцій, який має власне ім'я, перелік параметрів (змінних, що надходять у метод) та тип значення яке повертається. У загальному випадку, синтаксис наступний:

static ТИП ІМ'Я (тип 1 парам1, … тип *N* парам *N*)
{
блок інструкцій
return ЗНАЧЕННЯ
}

Увага! При визначенні методу вказують імена та типи будь-яких необхідних параметрів. Коли метод буде викликано з коду програми, йому передають конкретні значення, які називаються аргументами для кожного параметра. Аргументи мають бути сумісними з типом параметра, але ім'я аргументу, що використовується в коді виклику, не обов'язково повинно збігатися з іменем параметра, визначеного в методі.

У наведеному нижче прикладі, описано метод *sum*, що повертає результат типу *int* , підсумовування двох цілих чисел.

```
static void Main(string[] args)
    {
       int num_1 = 10;
       int num_2 = 15;
       Console.WriteLine("сума чисел = " + sum(num_1, num_2));
    }

static int sum(int a, int b)
    {
    int x = a + b;
    return x;
    }
```

Аргументи можуть бути обов’язковими як наведено вище і не обов’язковими. Їх використання потребує додаткової уваги при написанні коду. Необов’язкові аргументи об’являють при створенні методів. Наприклад:

```
double SV(double rad = 1.0)
{
  return 4.0 / 3.0 * 3.1415 * rad * rad * rad;
}
```

При викликах такого методу можна використовувати формати:

```
double vol; // деяка змінна

// виклик без аргументу
vol = SV();
```

```
// виклик з аргументом
vol = SV(3.0);
```

9.2. Перевантаження методів та рефакторинг

У випадках, коли потрібно робити однотипні дії з різною кількістю вхідних даних або різні дії, що відносяться до одного і того ж кроку роботи основного коду, можна спростити код за рахунок використання методів з однаковим ім’ям але різною сигнатурою. Сигнатура – це набір, що складається з ТИПу методу і вхідних параметрів. Наприклад:

1) *static int sum(int a, int b)*
2) *static string sum()*
3) *static int sum(string a, string b)*

Метод *sum* має різні сигнатури. Вибір того чи іншого з описаних методів відбувається автоматично, на основі переданих параметрів. Якщо при виклику методу буде передано дві рядкові змінні (або безпосередні значення), то виконається код з методу 3. Якщо при виклику методу параметрів не буде – то код методу 2.

Використання методів для скорочення чи спрощення коду та їх перевантаження, називають **рефакторингом**.

Зазвичай, при розробці програм, пишуть фрагмент в основному коді а коли є можливість його локалізувати – виносять у метод. При необхідності змінити внутрішні обчислення методів із збереженням їх призначення – роблять перевантаження.

В результаті рефакторингу, основний код методу *Main* тає більш лаконічним а сама програма набуває структурованості. Це полегшує аналіз і модифікацію програм при їх подальшому доопрацюванню.

Приклад перевантажених методів.

```
static void Main(string[] args)
    {
        Console.WriteLine("сума чисел = " + sum(5, 10));
        Console.WriteLine(sum());
        Console.WriteLine(sum("1.25","3.68"));
    }

    static int sum(int a, int b)
    {
    int x = a + b;
    return x;
    }
    static string sum()
    {
        return "пустий метод для прикладу перевантаження";
    }

    static double sum(string a, string b)
    {
        double d = Convert.ToDouble(a) / Convert.ToDouble(b);
        return d;
    }
```

9.3 Порядок виконання роботи

Завдання. Створити програму, що містить перевантаження методу для обчислення периметру фігур – квадрату, прямокутника, трикутника. Ім’я та сигнатури методів обрати довільно. Пояснити вибір сигнатур.

Рекомендації. В методі *Main* створити код, що надає запрошення для користувача з поясненням формату і кількості даних, які необхідно увести. Наприклад, пропонувати уведення одного значення для обчислення периметру квадрату, два – для прямокутника та три – для трикутника. Найпростіший спосіб – двокроковий. На першому кроці запитують скільки даних буде уведено, на другому реалізують уведення. Для цього можна скористатись інструкціями *if – else*.

Після операції уведення чисел, викликають відповідний метод.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Загальний синтаксис методу.
2. Що таке параметр і аргумент?
3. Що таке рефакторінг?
4. Що означає «перевантаження методу»?
5. Який механізм використання не обов'язкових параметрів?

Лабораторна робота № 10
СТВОРЕННЯ ТА ВИКОРИСТАННЯ КЛАСІВ

Мета роботи: набути практичних навичок створення та використання класів

10.1 Основні поняття

Поняття об'єктно-орієнтоване програмування (ООП) означає один з найефективніших підходів до сучасного програмування. ООП дозволяє об'єднати дані і методи, що відносяться до однієї сутності, і працювати з ними, як з одним цілим.

Ключовим елементом ООП є *клас* і *об'єкт*. Клас – це абстрактний тип даних. За допомогою класу описується деяка сутність, її характеристики і

можливі дії. Наприклад, клас може описувати студента, автомобіль і т.д. Описавши клас, його безпосередньо не використовують. Для використання класів, створюють його *екземпляр* – об'єкт, що є вже конкретним представником класу.

В простих випадках, клас представляє собою набір *полів* і методів. Поля класу – змінні, що об'явлені всередині класу і призначені для зберігання необхідних даних. З ними пов'язанні *властивості* – методи, що призначені для безпосередньої роботи з полями.

Методи, як і в інших частинах програми, є автономними частинами коду, що можуть використовуватись для обробки даних як у середині класу так і ззовні. Особливими методами класів є *конструктори* – методи, що виконуються безпосередньо при створенні екземпляру класу.

Для оперування змінними, що мають однакові імена або для звернення до змінних базового класу при спадкуванні класів, використовують покажчик *this* Ключове слово *this* забезпечує доступ до поточного екземпляру класу. Класичний приклад використання *this*, це як раз в конструкторах, при однакових іменах полів класу і аргументів конструктора.

10.2 Правила опису класів

У *C#* класи оголошуються за допомогою ключового слова *class*. Загальна структура оголошення виглядає наступним чином:

[модифікатор доступу] *class* [ім'я_класу] {   // тіло класу }

У мові *C#* застосовуються такі модифікатори доступу:

private – приватний компонент доступний лише у межах свого класу.

private protected – компонент класу доступний з будь-якого місця у своєму класі або похідних класах, які визначені в тій же збірці.

file – доданий у версії *C#* 11, з таким модифікатором доступні тільки з поточного файлу коду.

protected – такий компонент класу доступний з будь-якого місця у своєму класі або похідних класах. При цьому похідні класи можуть розміщуватись в інших збірках.

internal – компоненти класу або структури доступні з будь-якого місця коду в тій самій збірці, однак він недоступний для інших програм та збірок.

protected internal – поєднує функціонал двох модифікаторів *protected* та *internal*. Такий компонент класу доступний з будь-якого місця в поточній збірці та похідних класів, які можуть розташовуватися в інших збірках.

public – публічний, загальнодоступний компонент класу чи структури. Такий компонент доступний з будь-якого місця в коді, а також інших програм і збірок.

Збірка (*assembly*) – це готовий функціональний модуль у вигляді *exe* або *dll* файлу (файлів), який містить скомпільований код для *.NET*. Збірка надає можливість повторного використання коду.

При оголошенні класу модифікатор доступу можна не вказувати, при цьому буде застосовуватися режим за замовчуванням *internal*.

Клас слід оголошувати всередині простору імен *namespace*, але за межами іншого класу. Можливо також оголошення класу всередині іншого – як вкладений тип але це більш складний вид використання.

Приклад оголошення класів *Student* і *Pupil*:

```
namespace HelloWorld
{
  class Student // без вказівки модифікатор доступу, клас буде internal
  {     // тіло класу     }
  public class Pupil
  {    // тіло класу     }
}
```

Членами класу можуть виступати:

- поля;

- константи;
- властивості;
- конструктори;
- методи;
- події;
- оператори;
- індексатори;
- вкладені типи.

Всі члени класу, як і сам клас, мають свій рівень доступу, що описується модифікаторами. Без модифікаторів доступу за замовчуванням йому буде присвоєно режим *private*. За допомогою модифікаторів доступу в *C#* реалізується один з базових принципів ООП – інкапсуляція.

Приклад оголошення полів в класі:

```
class Student
{
  private string firstName; // поле, доступне в середині класу Student
  private string lastName; // поле, доступне в середині класу Student
  private int age; // поле, доступне в середині класу Student
  public string group; // поле, доступне для використання будь-де у пронрамі
}
```

Створення об’єктів. Оголосивши клас, створюють об’єкти. Робиться це за допомогою ключового слова *new* і імені класу. Наприклад, на основі попереднього класу *Student*:

```
static void Main (string [] args)
{
```

Student student1 = new Student (); // створення об'єкта *student1*, що є нащадком класу *Student*

Student student2 = new Student (); // аналогічно

}

}

}

З об'єктами можна працювати, як і з іншими змінними. Доступ до членів об'єкта здійснюється за допомогою оператора крапка « . ». Для попереднього прикладу, доступ до поля з модифікатором *public* :

```
student1.group = "Group1";
student2.group = "Group2";
Console.WriteLine(student1.group); // виводить на екран Group1
Console.Write(student2.group);
```

Поля класу *Student* такі, як *firstName*, *lastName* і *age* вказані з модіфікатором доступу *private*, тому доступ до них буде заборонений поза класом.

Використання у якості полів констант не відрізняється від звичайного їх використання в програмах. Константа оголошується за допомогою ключового слова *const* і має властивість *readonly*.

10.3 Конструктори та покажчик this

Конструктор – це метод класу, призначений для ініціалізації об'єкта при його створенні. Ініціалізація – це встановлення початкових параметрів об'єктів при їх створенні. Особливістю конструктора, як методу, є те, що його ім'я завжди збігається з ім'ям класу, в якому він оголошується. При цьому, при оголошенні конструктора, не потрібно вказувати тип значень, що повертається. Конструктор слід оголошувати як *public*, інакше об'єкт не можна буде створити.

У класі, в якому не оголошений жоден конструктор, існує неявний конструктор за замовчуванням, який викликається при створенні об'єкта за допомогою оператора new. Синтаксис оголошення конструктора:

```
public [ім'я_класу] ([аргументи])
{  // тіло конструктора  }
```

Наприклад, є клас *Car*. Створюючи новий автомобіль, значення пробігу та кількості палива в баку є сенс поставити рівними нулю:

```
class Car
{
  private double mileage;
  private double fuel;

  public Car () // оголошення конструктора
  {
   mileage = 0;
   fuel = 0;
  }
}
class Program
{
  static void Main (string [] args)
  {
   Car Audi = new Car (); // створення об'єкта і виклик конструктора
  }
}
```

Як і будь-який метод, конструктор також може мати параметри. Попередній приклад але при створенні об'єкта можемо поставити початкові значення:

```
class Car
{
  private double mileage;
  private double fuel;

  public Car (double mileage, double fuel) //  створення конструктора
  {
    this.mileage = mileage; // полю класу присвоєно значення параметру
    this.fuel = fuel; // полю класу присвоєно значення параметру
  }
}
class Program
{
  static void Main (string [] args)
  {
  Car Mers = new Car (100, 50); // виклик конструктора з параметрами  }}
```

Зверніть увагу на використання інструкції *this*. Через однакове іменування полів класу та параметрів методу, виникає необхідність явного вказування на поля класу.

Кілька конструкторів. У класі може бути декілька конструкторів, головне щоб вони відрізнялися сигнатурами. Сигнатура, в разі конструкторів, – це набір параметрів. Наприклад, не можна створити два конструктора, які приймають два аргументи типу int.

Приклад використання декількох конструкторів для класу *Car*:

```
  public Car ()
  {
    mileage = 0;
```

```
  fuel = 0;
 }
public Car (double mileage, double fuel)
{
  this.mileage = mileage;
  this.fuel = fuel;
}
```

Використання конструкторів для створення нового екземпляру:

Car newCar = new Car (); // з параметрами за замовчуванням, 0 і 0

Car newCar2 = new Car (100, 50); // з вказаними параметрами

Увага ! Якщо в класі визначено один або кілька конструкторів з параметрами, то не можна створити об’єкт через неявний конструктор за замовчуванням. Необхідно описати конструктор без параметрів.

10.4 Порядок виконання роботи

Завдання. Створити клас «Student», що має наступні поля, методи і перевантажені конструктори.

Поля:

- прізвище;
- ім’я;
- група;
- середній бал;

Методи:

- метод, що розраховує середній бал за сесію на основі параметру, тип – масив дробових чисел;
- метод, що друкує дані про студента – прізвище, ім’я, група, середній бал.

Конструктори:

- без параметрів, пусті поля заповнюються автоматично «*no name*» для рядкових значень та «0» для числових;

- з параметрами прізвище та ім'я, без масиву оцінок;

- з параметрами прізвище та ім'я і масив оцінок.

Продемонструвати виклик і використання об'єктів через різні конструктори.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке «клас» у програмуванні?
2. З яких елементів складається клас?
3. Що таке «конструктор класу»?
4. Чи обов'язковий конструктор класу?
5. Який механізм роботи покажчика *this*?

Лабораторна робота № 11
СТВОРЕННЯ БАГАТОПОТОКОВИХ ПРОГРАМ

Мета роботи: навчитись створювати програми з використанням кількох потоків, засвоїти теоретичний матеріал по створенню багатопотокових додатків.

11.1 Основні відомості

Одночасне використання декількох потоків можливе за рахунок програмних та апаратних переривань. Фактично, процесор, на одному арифметико-логічному пристрої може виконувати тільки одну послідовність операцій. Для паралельного виконання двох і більше потоків, необхідно перервати виконання поточної послідовності, зберегти її стан (адреси, дані, результати) і завантажити інший потік інструкцій. На кожен процес (програму) відводиться певний час,

що називають квантом. Таким чином можна виконувати велику кількість програм а через велику швидкодію процесорів (сучасні мають кілька ядер з кількома потоками), користувач сприйматиме це як паралельне виконання.

Всі процеси можна представити у вигляді конвеєрної стрічки, що прокручується через процесор. Квант часу залежить від важливості процесу – наприклад, системні мають вищий пріоритет та більше часу на виконання. Після запуску додатку чи програми, вона розміщується у послідовності процесів та отримує квант часу. Якщо в програмі присутні декілька потоків, то вони отримають окремі кванти. Послідовність виконання програмних потоків та кількість квантів на одиницю реального часу залежить від роботи системи.

Що б почати працювати з потоками, необхідно підключити простір імен *System.Threading*, додавши в початок файлу з кодом наступну директиву:

using System.Threading;

Будь-який потік в *C #* це функція. Функції не можуть бути самі по собі, вони обов'язково є методами класу. Тому, що б створити окремий потік, знадобиться клас до необхідного методу. Наприклад, створимо найпростіший варіант методу, що повертає *void* значення і не сприймає жодного аргументу:

void MyThreadFunction() { ... }

Для використання цього методу як окремого потоку, необхідно створити потік (у прикладі – *thr*), передаючи у якості аргументу, ім’я методу:

Thread <u>thr</u> = new Thread(MyThreadFunction);
thr.Start();

Після виклику методу *Start* для створеного потоку, управління до основного потоку (програми) повернеться відразу, але в цей момент вже почне пра-

цювати новий потік. Новий потік виконає тіло функції *MyThreadFunction* і завершиться. Після виклику *Start*, управління передається далі, при цьому створений потік може працювати ще тривалий час. Що б обмінюватися даними між потоками, можна користуватися змінними класу.

11.2 Створення багатопотокових програм

Приклад програми, що створює окремий (дочірній) потік.

```
static void Main(string[] args)
    {
       Thread thread = new Thread(ThreadFunction);
       thread.Start();//Запускаємо потік
       for (int i = 0; i < 5; i++)
       { Console.WriteLine(i + "Це основний процес"); }
       Console.ReadKey();
    }

static void ThreadFunction()
    {         for (int i = 0; i < 5; i++)
       { Console.WriteLine(i + "Це дочірній процес"); }
    }
```

Для візуалізації роботи потоків, використано цикли, що друкують інформацію про те, де вони виконуються. У результаті роботи такої програми, послідовність друку з основного і дочірнього потоків матиме непередбачувану періодичність.

Якщо потрібно запустити кілька потоків з одним методом, можна використати цикли.

```
for (int i = 0; i < 10; ++i)
        { Thread thread = new Thread(ThreadFunction);
          thread.Start(); }
```

Для створення кількох незалежних потоків, що виконують різні функції, їх запускають окремо. Для кожного потоку вказують свій метод.

```
Thread thr_1 = new Thread(Thr_Function_1);
        thr_1.Start();//Запускаємо потік 1
Thread thr_2 = new Thread(Thr_Function_2);
        thr_2.Start();//Запускаємо потік 2
Thread thr_3 = new Thread(Thr_Function_3);
        thr_3.Start();//Запускаємо потік 3
```

Кожен з потоків, що виконується, має окремий стек і відповідно, власні локальні змінні. Тобто, якщо одночасно запущено кілька екземплярів одного методу, це не приведе до збоїв і плутанини у значеннях одноіменних змінних. Доступ і значення глобальних змінних відбувається як звичайно. Увага! Так як порядок виконання потоків незалежний, не потрібно зловживати такою можливістю.

Для створення одного потоку з різним функціоналом можливо і інше рішення. Наприклад, використання двійкового ідентифікатора – прапорця або іншого ідентифікатора, який аналізується у методі і перемикає порядок виконання. Нижче наведено фрагмент такої програми, яка визначає значення прапорця. Саме значення – це аргумент методу.

```
static void Main(string[] args)
    {
        Thread thread1 = new Thread(TFnc);
        thread1.Start(true);
```

```
    Thread thread2 = new Thread(TFnc);
    thread2.Start(false);
    Console.Read();
}

static void TFnc(Object input)
{ bool flag = (bool)input;
    if (flag)
    {Console.WriteLine("this is first flow!");    }
    else
    {Console.WriteLine("this is second  flow!");    }
}
```

11.3 Порядок виконання роботи

Завдання. Розробити програму, яка керує курсором у нижній частині екрану та одночасно переміщує об'єкт у верхній частині. Для керування курсором використовувати клавіші зі стрілками, об'єкт – літера «О».

Програма складатиметься з двох методів, які запускаються як окремі потоки і виконуються до їх завершення за умовою. Наприклад, натискання клавіші «q».

Для керування курсором за допомогою клавіш зі стрілками можна скористатись їх кодовими значеннями. На першому кроці розробіть програму, що визначає код клавіші і друкує його на екрані. Нижче наведено фрагмент для визначення коду натисненої клавіші.

```
ConsoleKeyInfo _key;
while ((_key = Console.ReadKey(true)).KeyChar != 'q')
        {
        int k  = Convert.ToInt16(_key.Key); // перетворення коду у число
        Console.WriteLine(k);
```

```
}
Console.ReadKey(true);
```

У методі *ReadKey,* модифікатор *true* приховує відображення символів на екрані.

Наступний крок. Модифікуйте програму так, щоб отриманий код клавіші переставляв позицію курсора вліво або вправо.

Властивість курсора *Console.CursorLeft.*

Створіть потік, об'єднавши ці два фрагменти.

Створіть ще один потік, що змінює позицію курсору для друку символу у верхній частині екрану. Наприклад, циклічно від 10 до 40 позиції.

Для уникнення колізій, кожен метод для потоків повинен містити локальні змінні позиції курсора. На початку роботи ці змінні встановлюють курсор у потрібну позицію. Після виконання нова позиція записується у змінну.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Поясніть поняття «потік» щодо програмування.
2. Який механізм роботи багатопотокових додатків?
3. Як створюють додаткові потоки?
4. Чи можна створити декілька потоків з одного методу?
5. Чи можна створити кілька методів у одному потоці?

Лабораторна робота № 12
СИНХРОНІЗАЦІЯ ПОТОКІВ БАГАТОПОТОКОВИХ ПРОГРАМ

Мета роботи: засвоїти методи синхронізації потоків при створені багатопотокових програм.

12.1 Основні відомості

У багатопотокових програмах часто виникає необхідність синхронізувати роботу потоків через обмін даними. Коли два або більше потоків, намагаються взаємодіяти з будь-якими даними одночасно, може виникнути помилка (колізія). У середовищі C# закладено кілька методів синхронізації потоків. Найпростіший – використання оператора *lock*.

Синтаксис оператору дуже простий:

lock (object1) { ... }

Принцип використання – коли один із потоків, доходить до оператора *lock*, він перевіряє, чи не заблоковано *object1*. Якщо ні – виконує вказані у дужках оператори. Якщо заблоковано — чекає, коли *object1* буде розблоковано. Таким чином, оператор *lock* запобігає одночасному обігу кількох потоків до одних і тих же даних. Приклад.

```
class Program
   {static void Main(string[] args)
      {
         clasW _flow = new clasW();
         _flow.Run(); // метод з класу clasW
         Console.Read();
      }
   }
   class clasW
   {
      private int _val = 0; // змінна для демонстрації доступу
      private object vLock = new object(); // об’єкт для перевірки
      public void Run()
      {
```

```
        for (int i = 0; i < 5; ++i)
        {
            Thread trd = new Thread(TFc);
            trd.Start();
        }
    }

    private void TFc()
    {
        lock (vLock) // перевірка доступу до потоку
        {   // якщо доступ є – виконуємо код
            Console.WriteLine(_val);
            ++_val;
        }
    }
}
```

Недолік цього способу – при гальмуванні чи «зависанні» одного з потоків, що отримав доступ до *object1,* весь процес виконання потоку що очікує, теж зупиняється. Кращий спосіб уникнути колізій – продовжити виконання а коли ресурс розблокується передати сигнал у поток, що намагався отримати доступ.

Такий підхід спирається на міжпоточні засоби спілкування, які дозволяють одному потоку повідомити інший про те, що він блокується, а потім повідомити про те, що він може відновити виконання. *C#* підтримує міжпоточну взаємодію з допомогою методів *Wait*(), *Pulse*() і *PulseAll*().

Методи *Wait*(), *Pulse*() та *PulseAll*() визначені в класі *Monitor*. Ці методи можна викликати лише усередині *lock*-блоку коду. Коли виконання потоку тимчасово блокується, викликається метод *Wait*. Він переходить у режим очікування ("засинає") і знімає блокування з об’єкта, дозволяючи іншому потоку ви-

користовувати цей об'єкт. Пізніше, коли інший потік входить в аналогічний стан блокування і викликає метод *Pulse* () або *PulseAll* (), потік що засинав прокидається. Звернення до методу *Pulse* () відновлює виконання потоку, що стоїть першим у черзі потоків, що перебувають у режимі очікування. Звернення до методу *PulseAll* () повідомляє про зняття блокування всім потокам, що очікують.

Синтаксис методів:

public static bool Wait(object waitOb)

public static bool Wait(object waitOb, int milliseconds)

public static void Pulse(object waitOb)

public static void PulseAll(object waitOb)

Перший формат означає очікування до повідомлення. Другий – передбачає очікування до повідомлення або до закінчення періоду часу, заданого в мілісекундах. В обох випадках параметр *waitOb* визначає об’єкт, до якого очікується доступ.

Для *Pulse та PulseAl,l* параметр *waitOb* означає об’єкт, що звільняється від блокування. Якщо метод *Wait*(), *Pulse*() або *PulseAll*() викликається з коду, який знаходиться поза *lock*-блоком, генерується виняток типу *SynchronizationLockException*.

11.3 Порядок виконання роботи

Створіть новий проєкт. Використовуючи метод *lock,* розробіть програму з двома потоками. Один друкує значення змінної *_printV* на екрані у незкінечному циклі, другий очікує уведення символу з клавіатури і записує його у змінну *_printV* першого потоку також без кінця. Для гальмування першого потоку використовуйте один з методів:

System.Threading.Thread.Sleep(500);

await Task.Delay(TimeSpan.FromSeconds(0.5), source.Token);

Поясніть результати роботи програми.

Створіть другий новий проєкт і перенесіть у нього код «годинника тік-так», наведний нижче. Відлагодіть програму та поясніть її роботу.

```
class TickTock
{
   public void tick(bool running)
   {
   lock(this)
   {
   if(!running)
   { // Зупинка годинника.
   Monitor.Pulse(this); // повідомлення для потоків, що очікують

   return;
   }
   Console.WriteLine("tick");
   Monitor.Pulse(this); // дозвіл виконання методу tock()

   Monitor.Wait(this); // очікування завершення методу tock()


   }
   }
   public void tock(bool running)
   {
   lock(this)
   {
   if(!running)
   { // Зупинка годинника.
```

```
    Monitor.Pulse(this); // повідомлення для потоків, що очікують

    return;
    }
    Console.WriteLine("tock");
    Monitor.Pulse(this); // дозвіл виконання методу tick()

    Monitor.Wait(this); // очікування завершення методу tick()
    }
    }
}

class MyThread
{
    public Thread thrd;
    TickTock ttOb;
    // новий потік.
    public MyThread(string name, TickTock tt)
    {
        thrd = new Thread(new ThreadStart(this.run));
        ttOb = tt;
        thrd.Name = name;
        thrd.Start();
    }
    // Початок нового потоку,
    void run()
    {
        if (thrd.Name == "тик")
        {
            for (int i = 0; i < 5; i++) ttOb.tick(true);
```

```
            ttOb.tick(false);
        }
        else
        {
            for (int i = 0; i < 5; i++) ttOb.tock(true);
            ttOb.tock(false);
        }
    }
}
class TickingClock
{
public static void Main()
{
    TickTock tt = new TickTock();
    MyThread mtl = new MyThread("tick", tt) ;
    MyThread mt2 = new MyThread("tock", tt) ;
    mtl.thrd.Join();
    mt2.thrd.Join();
    Console.WriteLine("Clock stunned");
    Console.ReadKey();
}
}
```

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Для чого потрібна синхронізація потоків?
2. Як працює оператор *lock* ?
3. Який механізм роботи методів *Wait*() і *Pulse*()?
4. У чому недолік оператору *lock* ?

ПЕРЕЛІК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Дібрівний О. А., Гребенюк В. В. Вступ до об’єктно орієнтованого програмування C#: Навчальний посібник. – К.: Державний університет телекомунікацій, 2018, – 190 с.

2. A. Hejlsberg, M. Torgersen, S.Wiltamuth, P. Gold, “The C# Programming Language (Covering C# 4.0) (4th Edition)”, Addison-Wesley Professional; 4 edition – 864 p., 2010.

3. J. Albahari, B. Albahari, “C# 7.0 in a Nutshell”, O'Reilly Media – 1090 p., 2017.

4. H. Schildt, “C# 4.0 The Complete Reference McGraw Hill Education; 1st edition” – 984 p., 2017.

5. J. Sharp, “Microsoft Visual C# Step by Step”, Pearson Education – 878p., 2018.

6. A. Troelsen, “Pro C# 7 With .NET and .NET Core” - Apress, – 1372 p., 2017.

7. C# Reference, сайт розробників MSDN. [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/uk-ua/dotnet/csharp/language-reference/index

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ

Методичні вказівки до лабораторних робіт з курсу «Комп'ютерні технології та програмування. Частина 1»

для студентів спеціальності 174 – Автоматизація, комп'ютерно-інтегровані технології та робототехніка денної та заочної форми навчання

Укладачі: ПУГАНОВСЬКИЙ Олег Валентинович
ДЕМЕНКОВА Світлана Дмитрівна
ШУТИНСЬКИЙ Олексій Григорович

Відповідальний за випуск : Пугановський О. В.
Роботу до видання рекомендував: доц. Дудник О. В.

Редактор О.І.Шпільова

План 2023 р., поз 147.Підп. до друку 23.04.21. Формат 60x84 1/16.
Папір офсет-ний. ГарнітураTimes.
Ум. друк. арк. 5,9. Наклад 25 прим. Зам. №__. Ціна договірна.

Видавець НТУ"ХПІ", 61002, Харків вул. Кипичова, 2
Свідоцтво про державну реєстрацію ДК № 5478 от 21.08.2017 г.

Надруковано з готового оригінал-макету у друкарні ФОП В.В. Петров Єдиний державний реєстр юридичних осіб та фізичних осіб – підпри-ємців.
Запис №24800000000106167 від 08.01.2009 р.
61144, м. Харків, вул. Гв. Широнінців, 79в, к. 137, тел. (057)78-17-137.
e-mail: bookfabrik@mail.ua