МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Інститут енергозбереження та енергоменеджменту

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання комп'ютерного практикуму з дисциплін

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ»
Частина 2. Об'єктно-орієнтоване програмування
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ»
Частина 2. Об'єктно-орієнтоване програмування
для студентів спеціальності
144 Теплоенергетика

(навчальне електронне видання)

Київ, НТУУ «КПІ»
2016

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Інститут енергозбереження та енергоменеджменту

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання комп'ютерного практикуму з дисциплін

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ»
Частина 2. Об'єктно-орієнтоване програмування
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ»
Частина 2. Об'єктно-орієнтоване програмування
для студентів спеціальності
144 Теплоенергетика

(навчальне електронне видання)

Рекомендовано Вченою радою
Інституту енергозбереження та енергоменеджменту НТУУ «КПІ»

Київ, НТУУ «КПІ»
2016

Методичні вказівки виконання комп’ютерного практикуму з дисциплін «Обчислювальна техніка та програмування» та «Інформаційні технології» для студентів спеціальностей 141 Електроенергетика, електротехніка та електромеханіка та 144 Теплоенергетика. Навчальне електронне видання [Електронний ресурс] / Уклад.: І.В. Притискач. – Київ: НТУУ «КПІ», 2016.

Гриф надано Вченою радою ІЕЕ НТУУ «КПІ»
(Протокол № від 2016 р.)

Навчальне електронне видання

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання комп’ютерного практикуму з дисциплін

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ»
Частина 2. Об'єктно-орієнтоване програмування
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ»
Частина 2. Об'єктно-орієнтоване програмування
для студентів спеціальності
144 Теплоенергетика

Укладачі: Притискач Іван Васильович, к.т.н., ст. викл.

Відповідальний редактор Штогрин Євген Андрійович, к.т.н., доц.

Рецензент Данілін Олександр Валерійович, к.т.н., доц.

За редакцією укладачів

ЗМІСТ

ВСТУП

Розвиток економіки, промисловості, науки і техніки, сфери освіти сьогодні значною мірою залежить від масового запровадження та використання обчислювальної техніки. Це вимагає підготовки і перепідготовки фахівців з програмування і використання персональних комп'ютерів.

Останнім часом в навчальних планах вищих навчальних закладів і різних курсів підвищення кваліфікації передбачається проведення практичних робіт на комп'ютерах. Тому в основу методичних вказівок покладено курс практичних робіт, які призначені для студентів, що вивчають одну з найпоширеніших мов програмування – C#. Вказівки містять практичні роботи з найважливіших тем, які необхідно засвоїти студенту при вивченні основ програмування.

Вибір мови програмування C# пояснюється такими чинниками:

- простотою і природністю основних конструкцій мови, що дозволяє швидко її освоїти і створювати алгоритмічно складні програми;
- можливістю використання розвинених засобів подання структур даних, що забезпечує зручність роботи як з числовою, так і з символьною інформацією;
- відповідністю принципам об'єктно-орієнтованого програмування, що робить програми наочними;
- наявністю бібліотеки процедур і функцій для роботи як з текстовою, так і з графічною інформацією, що дозволяє створювати досить складні програми.

ВИМОГИ БЕЗПЕКИ ПІД ЧАС РОБОТИ В ЛАБОРАТОРІЇ ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

1. Ці правила є обов'язковими для всіх студентів та інших осіб, які працюють в лабораторіях постійно чи тимчасово.
2. Перевірка знань цих правил проводиться:
3. обслуговуючого персоналу – завідуючим лабораторією;
4. студентів – викладачами, які є керівниками лабораторних робіт.
5. Після перевірки знань цих правил, кожен з тих, хто працює в лабораторії ставить свій підпис в спеціальному журналі. Без цього підпису ніхто до роботи в лабораторії не повинен бути допущеним.
6. Дотримання правил з техніки безпеки повинно гуртуватися на високій свідомості всіх, хто працює в лабораторіях. Кожен, хто помітить порушення правил, а також несправність, яка являє собою небезпеку для людей і обладнання, повинен сповістити про це керівника.
7. Робота в лабораторії по виконанню конкретного завдання може проводитися тільки після ретельного ознайомлення студентів з обладнанням і роботою, та чіткого уявлення про те, які елементи установки будуть під напругою та дотик до яких є небезпечними у стані роботи.
8. Забороняється виконання ремонтних робіт на обладнанні, яке знаходиться під напругою.
9. Забороняється наближатися або торкатися до струмоведучих частин, що обертаються, або усувати несправності без відключення установок.
10. Забороняється проводити переключення в схемі, яка знаходиться під напругою.
11. Перевірку наявності напруги дозволяється проводити тільки за допомогою вольтметра.

12. Апарати управління та вимірювальні прилади слід розташовувати так, щоб було зручно вести спостереження за приладами, не перегинаючись через проводи та апарати.
13. У випадку виникнення будь-яких несправностей необхідно негайно вимкнути живленим установки та сповістити керівника занять про це.
14. Кнопки управління, рубильники встановлювати в легкодоступних місцях для швидкого виключення схеми.
15. У випадку припинення подачі електроенергії в лабораторію всі установки в лабораторії обов'язково вимикаються на робочих місцях.
16. В лабораторіях категорично забороняється:
    - Палити в усіх приміщеннях, крім спеціально відведених для цього місць;
    - Прокладати без дозволу постійні та тимчасові лінії;
    - Користуватися побутовими електронагрівальними приладами;
    - Користуватися зіпсованим електрообладнанням, саморобними запобіжниками, провідниками із зіпсованою ізоляцією та саморобними електросвітильниками;
    - Проводити в непристосованих приміщеннях обмивку та фарбування деталей горючими рідинами та фарбниками;
    - Зберігати паливно-мастильні матеріали, хімікати та інші горючі речовини;
    - Загромаджувати проходи в лабораторіях;
17. Всі, хто працює в лабораторії повинні знати, де знаходиться аптечка з медикаментами для наданим першої допомоги.
18. При ураженні людини електричним струмом треба негайно вимкнути напругу, надати першу допомогу та покликати лікаря.

19. Порятунок осіб, які постраждали, залежить від тою, як швидко вони будуть звільнені від електричного струму та як швидко їм буде надано першу допомогу.

20. Першу допомогу необхідно надати негайно на місці події.

21. Переносити людину, яка постраждала в інше місце необхідно тільки в тих випадках, коли небезпека продовжує загрожувати або надання допомоги на місці неможливе.

22. При відсутності у постраждалого дихання, серцебиття, пульсу ніколи не треба ставити під сумнів необхідність першої допомоги, тому що при ураженні електричним струмом смерть часто буває несправжньою. Тільки лікар може дати висновок про смерть постраждалого.

23. До приїзду лікаря постраждалому необхідно надати допомогу і провести штучне дихання з дотриманням всіх правил надання першої допомоги.

24. Про випадок негайно треба сповістити керівництво кафедри, деканату та інституту.

25. Недотримання цих вимог не дозволяється. Якщо розпорядження суперечить діючим правилам, необхідно надати роз'яснення з приводу неухильною виконання цих правил і довести це до відома керівництва.

ЗАГАЛЬНІ ВИМОГИ ДО ВИКОНАННЯ ПРАКТИЧНИХ РОБІТ

Підготовка до кожної роботи проводиться в поза аудиторний час. Студенти знайомляться із загальними відомостями, пишуть необхідні програми відповідно до отриманого варіанта індивідуального завдання.

Усі роботи виконуються на мові програмування C#. Під час занять студенти проводять тестування написаних програм, тобто запускають їх в інтегрованому середовищі розробки на персональних комп’ютерах, займаються налагодженням і виконують необхідні розрахунки.

Після виконання роботи студент оформляє звіт, який складається з таких розділів:

1. Назва, тема і мета роботи
2. Індивідуальне завдання
3. Текст програми
4. Результати виконання програми

Шаблон звіту про виконання практичної роботи наведено в додатку А. Робота оформлюється в електронному або друкованому вигляді.

Під час захисту роботи необхідно відповісти на контрольні питання і вміти пояснити роботу програми.

ПРАКТИЧНА РОБОТА №1.
КЛАСИ

Мета

Засвоїти основні поняття концепції об’єктно-орієнтованого програмування. Вивчити основні принципи реалізації класів в мові програмування C#. Отримати досвід створення простих класів з полями базових типів даних.

Стислі теоретичні відомості

Клас є типом даних, визначеним користувачем. Він повинен представляти собою одну логічну сутність, наприклад, бути моделлю реального об'єкта або процесу. Елементами класу є дані і функції, призначені для їх обробки.

Опис класу містить ключове слово **class**, за яким слідує його ім'я, а далі у фігурних дужках - тіло класу, тобто список його елементів. Крім того, для класу можна задати його базові класи (предки) і ряд необов'язкових атрибутів і специфікаторів, які визначають різні характеристики класу:

```
[атрибути] [специфікатор] class ім'я_класу [:предки]
тіло класу
```

Обов'язковими є тільки ключове слово **class**, а також ім'я та тіло класу. Ім'я класу задається програмістом за загальними правилами C #. Тіло класу - це список описів його елементів, укладений у фігурні дужки. Список може бути порожнім, якщо клас не містить жодного елемента.

Специфікатори визначають властивості класу, а також доступність класу для інших елементів програми. Можливі значення специфікаторів перераховані в Таблиця 1.1. Клас можна описувати безпосередньо всередині простору імен або всередині іншого класу. В останньому випадку клас називається вкладеним. Залежно від місця опису класу деякі з цих специфікаторів можуть бути заборонені.

Таблиця 1.1 Специфікатори класів

№	Специфікатор	Опис
1	new	Використовується для вкладених класів. Задає новий опис класу, натомість успадкованого від предка. Застосовується в ієрархіях об'єктів
2	public	Доступ не обмежений
3	protected	Використовується для вкладених класів. Доступ лише з елементів даного і похідних класів
4	internal	Доступ лише з даної програми
5	protected internal	Доступ лише з даного і похідних класів або з даної програми
6	private	Використовується для вкладених класів. Доступ лише з елементів класу, всередині якого описаний даний клас
7	abstract	Абстрактний клас. Застосовується в ієрархіях об'єктів
8	sealed	Закритий клас. Застосовується в ієрархіях об'єктів
9	static	Статичний клас

Специфікатори 2-6 називаються специфікаторами доступу. Вони визначають, звідки можна безпосередньо звертатися до даного класу.

У цій роботі вивчаються класи, які описуються в просторі імен безпосередньо (тобто не вкладені класи). Для таких класів допускаються тільки два специфікатора: **public** і **internal**. За замовчуванням, тобто якщо жоден специфікатор доступу не вказаний, мається на увазі специфікатор **internal**.

Клас є узагальненим поняттям, що визначає характеристики і поведінку всіх конкретних об'єктів цього класу, які називають екземплярами, або об'єктами, класу.

Об'єкти створюються явним чи неявним чином, тобто або програмістом, або системою. Програміст створює екземпляр класу за допомогою операції **new**, наприклад:

```
Book a = new Book();
var b = new Book();
```

Для кожного об'єкта при його створенні в пам'яті виділяється окрема область, в якій зберігаються його дані. Крім того, в класі можуть бути присутніми статичні елементи, які існують в єдиному екземплярі для всіх об'єктів класу. Часто статичні дані називають даними класу, а решта - даними екземпляра.

Функціональні елементи класу не тиражуються, тобто завжди зберігаються в єдиному екземплярі. Для роботи з даними класу використовуються методи класу (статичні методи), для роботи з даними примірника – методи примірника, або просто методи.

До цього часу ми використовували в програмах тільки один вид функціональних елементів класу – методи. Поля і методи є основними елементами класу. Крім того, в класі можна задавати цілу гаму інших елементів: властивості, події, індексатори, операції, конструктори, деструктори, а також типи.

Дані, що містяться в класі, можуть бути змінними або константами. Змінні, описані в класі, називаються полями класу.

При описі елементів класу можна також вказувати атрибути і специфікатори задають різні характеристики елементів. Синтаксис опису елемента даних наведений нижче:

```
[атрибути] [специфікатор] [const] тип ім'я
[= початкове значення];
```

Можливі специфікатори полів і констант перераховані в Таблиця 1.2. Для констант можна використовувати тільки специфікатори 1-6.

За замовчуванням елементи класу вважаються закритими (**private**). Для полів класу цей вид доступу є кращим, оскільки поля визначають внутрішню

будову класу, яке має бути прихована від користувача. Всі методи класу мають безпосередній доступ до його закритих полів.

Таблиця 1.2 Специфікатори полів і констант

№	Специфікатор	Опис
1	new	Нове опис поля, яке приховує успадкований елемент класу
2	public	Доступ до елементу не обмежений
3	protected	Доступ лише з даного і похідних класів
4	internal	Доступ лише з даної збірки
5	protected internal	Доступ лише з даного і похідних класів і з даної збірки
6	private	Доступ лише з даного класу
7	static	Одне поле для всіх екземплярів класу
8	readonly	Поле доступне тільки для читання
9	volatile	Поле може змінюватися іншим процесом або системою

Поля, описані зі специфікатором **static**, а також константи існують в єдиному екземплярі для всіх об'єктів класу, тому до них звертаються не через ім'я екземпляра, а через ім'я класу. Якщо клас містить тільки статичні елементи, екземпляр класу створювати не потрібно.

Звернення до полю класу виконується за допомогою операції доступу (точка). Праворуч від точки задається ім'я поля, ліворуч - ім'я екземпляра для звичайних полів або ім'я класу для статичних.

Всі поля спочатку автоматично ініціалізуються нулем відповідного типу (наприклад, полям типу **int** присвоюється 0, а посиланням на об'єкти - значення **null**). Після цього полю присвоюється значення, задане при його явній ініціалізації. Завдання початкових значень для статичних полів виконується при ініціалізації класу, а звичайних – при створенні екземпляру.

Поля зі специфікатором **readonly** призначені тільки для читання. Встановити значення такого поля можна або при його описі, або в конструкторі.

Робоче завдання

Навчитися створювати прості класи з полями даних базових типів.

Хід роботи

Написати програму на мові C#, в якій описано клас, що відповідає об’єкту відповідно до варіанту індивідуального завдання. Створити три екземпляри описаного класу з різними значеннями полів та вивести ці значення на екран.

Індивідуальне завдання

1. Створити клас, що містить відомості про студента: прізвище, ім’я, по батькові, рік народження, зріст.
2. Створити клас, що містить відомості про області України: назва, площа, населення, адміністративний центр, населення в адміністративному центрі.
3. Створити клас, що містить відомості про модель телефону: виробник, діагональ екрана, кількість процесорних ядер, ціна.
4. Створити клас, що містить відомості про електродвигун: тип, номінальна потужність, кількість фаз, маса.
5. Створити клас, що містить відомості про монітор комп’ютера: модель, розширення по горизонталі, розширення по вертикалі, яскравість, ціна.
6. Створити клас, що містить відомості про лінію електропередачі: тип кабелю, довжина, активний опір, реактивний опір, кількість фаз.
7. Створити клас, що містить відомості про будинок: адреса, кількість поверхів, загальна площа, кількість мешканців.
8. Створити клас, що містить відомості про футбольну команду: назва, місто, країна, кількість набраних очок за сезон, середня зарплата футболістів.

9. Створити клас, що містить відомості про літак: модель, максимальна швидкість, максимальна висота польоту, кількість пасажирів.
10. Створити клас, що містить відомості про навчальний предмет: назва, кількість семестрів, кількість лекцій, прізвище викладача.
11. Створити клас, що містить відомості про планету сонячної системи: назва, маса, діаметр, кількість супутників.
12. Створити клас, що містить відомості про провід ліній електропередачі: марка, площа перерізу, питомий активний опір, матеріал.
13. Створити клас, що містить відомості про квартиру: номер, площа, кількість кімнат, споживання електроенергії за місяць.
14. Створити клас, що містить відомості про модель ноутбука: назва, виробник, рік випуску, час автономної роботи.
15. Створити клас, що містить відомості про студента: прізвище, номер телефону та екзаменаційні оцінки з трьох предметів.
16. Створити клас, що містить відомості про прізвища, табельний № та зарплату працівника заводу.
17. Створити клас, що містить відомості про автора, назву книги, кількість сторінок та рік її видання.
18. Створити клас, що містить коефіцієнти квадратного рівняння, та його запис у текстовому вигляді.
19. Створити клас, що містить відомості про розклад руху потягів: номер потяга, назва станції, час прибуття, час відправлення.
20. Створити клас, що містить відомості про здачу студентом екзамену: номер групи, прізвище студента, предмет, оцінка.
21. Створити клас, що містить відомості про запис студента в бібліотеку: прізвище, ім’я, рік народження, дата запису до бібліотеки.

22. Створити клас, що містить відомості про номер автомобіля, марку автомобіля, рік випуску та колір.
23. Створити клас, що містить відомості про назву країни, площу країни, населення країни, назву столиці, населення столиці.
24. Створити клас, що містить відомості про розклад занять: назва предмета, № пари, день тижня, прізвище викладача.
25. Створити клас, що містить відомості про трансформатор: тип, номінальна потужність, напруга первинної обмотки, напруга вторинної обмотки, кількість фаз.

Контрольні питання

1. Що таке клас?
2. Що таке поле класу?
3. Що таке екземпляр класу?
4. Наведіть загальну форму опису класу.
5. Наведіть загальну форму опису елемента даних класу.
6. Що таке тіло класу?
7. Чи можна вкладати клас у інший клас?
8. Що таке оператор доступу до елементу класу?
9. Що таке специфікатори доступу? Наведіть приклади.

ПРАКТИЧНА РОБОТА №2.
РОБОТА З ПОЛЯМИ КЛАСУ. СПЕЦИФІКАТОРИ ДОСТУПУ

Мета

Засвоїти основні поняття організації управління доступом до даних класу. Вивчити основні принципи роботи специфікаторів доступу в мові програмування C#. Отримати досвід створення методів для організації роботи з закритими полями класу.

Стислі теоретичні відомості

У мові C #, по суті, є два типи елементів класу: відкриті і закриті (хоча насправді існує ще декілька інших різновидів). Доступ до відкритого члену можна здійснювати з коду, визначеного за межами класу. Саме цей тип – **public** – елемента класу використовувався в попередній практичній роботі. Закритий елемент класу доступний тільки методами, визначеними в самому класі. За допомогою закритих елементів і організовується управління доступом до даних класу.

Обмеження доступу до елементів класу є одним із основних принципів об'єктно-орієнтованого програмування, оскільки дозволяє виключити некоректне використання об'єкта. Виконуючи доступ до закритих даних тільки за допомогою чітко визначених методів, можна виключити присвоювання невірних значень цих даних, забезпечуючи, наприклад, перевірку діапазону представлення чисел. Тоді правильно реалізований клас утворює свого роду "чорний ящик", яким можна користуватися без втручання у внутрішній механізм його роботи.

Як уже було зазначено, управління доступом в мові C# організовується за допомогою таких специфікаторів доступу, як **public**, **private**, **protected** і **internal**. Коли елемент класу позначається специфікатором **public**, він стає доступним з будь-якого іншого коду в програмі, включаючи і методи, визначені в інших класах. Коли ж елемент класу позначається специфікатором **private**, він може бути доступний тільки іншим членам цього класу. Отже,

методи з інших класів не мають доступу до закритого елементу (**private**) даного класу.

Для того щоб стали більш зрозумілими відмінності між специфікаторами **public** і **private**, розглянемо наступний приклад програми:

```
using System;

namespace ClassElementTest
{
    class Lamp
    {
        private double power; // Закритий доступ, вказується явно
        double current; // Закритий доступ за замовчуванням
        public double NominalPower; // Відкритий доступ

        // Методи, яким доступні поля даного класу
        // Метод класу може мати доступ до закритого члену цього ж класу
        public void SetPower(double value)
        {
            if (value >= 0 && value <= NominalPower)
            {
                power = value;
                current = value / 220; // Приймемо, що I=P/U, U=220 В
            }
        }
        public double GetPower()
        {
            return power;
        }
        public double GetCurrent()
        {
            return current;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var myLamp=new Lamp();
            // Поле NominalPower класу Lamp доступне безпосередньо,
            // оскільки воно є відкритим
            myLamp.NominalPower = 100;
            // Доступ до полів power і current класу Lamp дозволений
            // тільки за допомогою його методів
            myLamp.SetPower(85.3);
            Console.WriteLine("Потужність лампи: {0} Вт, струм лампи: {1} А",
                myLamp.GetPower(), myLamp.GetCurrent());
            // Наступні види доступу до полів power і current класу Lamp не дозволяються
            // myLamp.power = -10;  // Помилка!  power - закритий член!
            // Console.WriteLine(myLamp.current);  // Помилка!  current - закритий член!
        }
    }
}
```

Як бачите, в класі *Lamp* поле *power* вказано явно як **private**, поле *current* стає **private** за замовчуванням, а поле *NominalPower* вказано як **public**. Таким чином, поля *power* і *current* недоступні безпосередньо з коду за межами даного класу, оскільки вони є закритими. Зокрема, ними не можна користуватися безпосередньо в класі *Program*. Вони доступні тільки за допомогою таких відкритих (**public**) методів, як *SetPower* () і *GetPower* (). Так, якщо видалити символи коментаря на початку рядка коду:

```
// myLamp.power = -10;  // Помилка!  power - закритий член!
```

то наведена вище програма не скомпілюється через порушення правил доступу. Але незважаючи на те, що член *power* недоступний безпосередньо за межами класу *Lamp*, вільний доступ до нього організовується за допомогою методів, визначених в класі *Lamp*, як наочно показують методи *SetPower* () і *GetPower* (). Таким чином виключається задання некоректних значень полів (у даному випадку від'ємної потужності та потужності, яка більша за номінальну).

Робоче завдання

Навчитися управляти доступом до полів класу та створювати методи для організації роботи з закритими полями класу.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання практичної роботи №1:
    - надати специфікатор доступу **private** двом, вибраним на ваш розсуд, полям даних класу;
    - створити відповідні методи для задання та зчитування значень закритих полів;
    - доповнити клас методом *ToString* () що забезпечує отримання інформації про клас у текстовому вигляді та дає змогу

використовувати спрощений запис методів *WriteLine* () для екземплярів класу. Приклад методу:

```
public override string ToString()
{
    return string.Format("Потужність лампи: {0} Вт, струм лампи: {1} А",
        power, current);
}
```

Результати використання:

```
// Старий запис методу WriteLine ()
Console.WriteLine("Потужність лампи: {0} Вт, струм лампи: {1} А",
    myLamp.GetPower(), myLamp.GetCurrent());
// Новий запис методу WriteLine ()
Console.WriteLine(myLamp);
```

- модифікувати відповідним чином метод *Main* () класу *Program*.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Що таке закритий елемент класу?
2. Що таке відкритий елемент класу?
3. Що таке специфікатор доступу?
4. Яка різниця між специфікаторами доступу **public** та **private**?
5. Чому для полів класу доцільно використовувати специфікатор доступу **private?**
6. Як забезпечити доступ до закритих полів класу з методів інших класів?
7. Чи можна для поля класу задати одночасно два специфікатори доступу?

ПРАКТИЧНА РОБОТА №3.
КОНСТРУКТОРИ КЛАСІВ

Мета

Вивчити основні принципи роботи конструкторів класів в мові програмування C#. Отримати досвід створення конструкторів для ініціалізації екземплярів класу.

Стислі теоретичні відомості

У наведених вище прикладах програм змінні екземпляра кожного об'єкта типу доводилося задавати вручну, використовуючи, зокрема, наступну послідовність операторів:

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

Такий прийом зазвичай не застосовується в професійно написаному коді C#. Крім того, він супроводжується помилками, коли можна просто забути ініціалізувати одне з полів. Існує кращий спосіб вирішити подібну задачу: скористатися конструктором.

Конструктори призначені для ініціалізації об'єктів. Вони викликаються автоматично при створенні об'єкта класу за допомогою операції **new**. Ім'я конструктора обов’язково збігається з ім'ям класу.

Нижче наведена загальна форма опису конструктора:

```
[специфікатори] ім'я_класу ([список_параметрів])
тіло_конструктора
```

Властивості конструкторів:

- конструктор не повертає значення, навіть типу **void**;
- клас може мати кілька конструкторів з різними параметрами для різних видів ініціалізації;
- якщо програміст не вказав жодного конструктора, то застосовується конструктор за замовчуванням, у якому полям

значущих типів присвоюється нуль, а полям посилальних типів - значення **null**.

Як правило, конструктор використовується для завдання початкових значень змінних екземпляра, визначених в класі, або ж для виконання будь-яких інших процедур, які потрібні для створення повністю сформованого коректного об'єкта. Для конструктора зазвичай задають специфікатор доступу **public**, оскільки конструктори, в основному, викликаються з інших класів. Список параметрів конструктора може бути порожнім, або містити один чи більше параметрів.

У всіх класів є конструктори, незалежно від того, визначаються вони явно чи ні. В C# автоматично надається конструктор, який використовується за замовчуванням і у якому всі змінні екземпляра ініціалізуються їх значеннями за замовчуванням. Для більшості типів даних значенням за замовчуванням є нульовим, для типу **bool** – значення **false**, а для типів-посилань – значення **null**. Як тільки ви визначите свій власний конструктор, то конструктор за замовчуванням більше не викликається.

Часто буває зручно задати в класі кілька конструкторів, щоб забезпечити можливість ініціалізації об'єктів різними способами. У цьому випадку використовуються параметризовані конструктори. У конструкторів параметри задаються таким же чином, як і в методах. Для цього достатньо оголосити їх в дужках після імені конструктора. Всі конструктори класу повинні мати різні сигнатури.

Після знайомства з конструкторами класів, повернемося до оператора **new**. Для класів загальна форма оператора **new** така:

```
new ім'я_класу ([список_параметрів])
```

де ім'я_класу позначає ім'я класу, що реалізується у вигляді його екземпляра. Якщо в класі не визначений його власний конструктор, то в операторі **new** буде використаний конструктор, що надається в C# за замовчуванням.

Робоче завдання

Навчитися створювати конструктори класів в мові програмування C# та отримати досвід використання конструкторів для ініціалізації екземплярів класу.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання практичної роботи №2:
    - надати специфікатор доступу **private** всім іншим полям даних класу;
    - для описаного раніше класу створити конструктор без параметрів для ініціалізації полів певними початковими значеннями (значення вибрати самостійно);
    - додатково створити два параметризованих конструктори з різними наборами параметрів для ініціалізації відповідних полів початковими значеннями;
    - модифікувати метод *Main* () класу *Program* шляхом заміни операторів присвоювання значень полів на виклики відповідних конструкторів.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Що таке конструктор класу?
2. Що таке параметризований конструктор?
3. Які специфікатори доступу можливо використовувати для конструкторів?

4. Чому для конструкторів класу доцільно використовувати специфікатор доступу **public?**
5. Як відбувається ініціалізація полів класу за допомогою конструкторів?
6. Як відбувається виклик конструктора класу?
7. Чи може клас містити більше одного конструктора?
8. Що таке конструктор за замовчуванням? Як він працює?
9. Наведіть загальний синтаксис опису конструктора класу.

ПРАКТИЧНА РОБОТА №4.
ВЛАСТИВОСТІ

Мета

Вивчити основні принципи роботи властивостей, як елементів класів в мові програмування C#. Отримати досвід створення властивостей для доступу до даних класу.

Стислі теоретичні відомості

Властивість є елементом класу, що поєднує в собі поле з методами доступу до нього. Властивості служать для організації доступу до полів класу. Як правило, властивість пов'язана із закритим полем класу і визначає методи його отримання та установки.

Властивість складається з імені і так званих аксесорів **get** і **set**. Аксесори служать для зчитування і установки значення змінної. Головна перевага властивості над спеціальними методами полягає в тому, що її ім'я може бути використано в виразах і операторах присвоювання аналогічно імені звичайної змінної. При зверненні до властивості по імені автоматично викликаються її аксесори **get** і **set**.

Нижче наведена загальна форма властивості:

```
[ атрибути ] [ специфікатори ] тип ім’я_властивості
{
[ специфікатор ] [ get код_доступу ]
[ специфікатор ] [ set код_доступу ]
}
```

де тип позначає конкретний тип властивості, наприклад **int**.

Значення специфікаторів для властивостей і методів аналогічні. Найчастіше властивості оголошуються як відкриті (зі специфікатором **public**), оскільки вони реалізують взаємодію зовнішнього коду з даними класу.

Код доступу є блоком операторів, які виконуються при отриманні (**get**) або установці (**set**) властивості.

Як тільки властивість визначено, будь-яке звернення до властивості по імені приведе до автоматичного виклику відповідного аксесора. Крім того, аксесор **set** має неявний параметр **value**, який містить значення, що присвоюється властивості.

Властивості не визначають місце в пам'яті для зберігання полів, а лише управляють доступом до полів. Сама властивість не описує поле, і тому поле повинно бути визначено незалежно від властивості (винятком з цього правила є автоматичні властивості).

Нижче наведено простий приклад програми, в якій визначається властивість *Power*, призначена для доступу до поля *_power*. В даному прикладі властивість використовується для обмеження діапазону допустимих значень потужності лампи.

```
using System;

namespace ClassElementTest
{
    class Lamp
    {
        private double _power;
        public double NominalPower;

        // Властивість для доступу до поля _power
        public double Power
        {
            get { return _power; }
            set {
                if (value>=0 && value<=NominalPower)
                {
                    _power = value;
                }
            }
        }

        // Властивість без базового поля. Призначена для розрахунку
значення струму
        public double Current
        {
```

```
            get { return _power/220; } // Приймемо, що I=P/U, U=220 В
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var myLamp = new Lamp();
            myLamp.NominalPower = 100;
            // Задаєм значення за допомогою властивості
            myLamp.Power = 83.5;
            Console.WriteLine("Потужність лампи: {0} Вт, струм лампи:
{1:n3} А",
                myLamp.Power, myLamp.Current);
        }
    }
}
```

Розглянемо наведений вище код більш докладно. У цьому коді визначається закрите поле *_power* і властивість *Power*, котра управляє доступом до поля *_power*.

Властивість *Power* вказано як **public**, а отже, вона доступна з коду за межами класу. Аксесор **get** цієї властивості просто повертає значення поля *_power*, тоді як аксесор **set** встановлює значення в поле *_power* тільки в тому випадку, якщо це значення відповідає обмеженням. Таким чином, властивість *Power* контролює значення, які можуть зберігатися в полі *_power*. У цьому, власне, і полягає основне призначення властивостей.

Починаючи з версії C# 3.0, з'явилася можливість створювати прості властивості, не вдаючись до явного визначення поля, яким управляє властивість. У цьому випадку базове поле для властивості автоматично надає компілятор. Така властивість називається автоматичною і має таку загальну форму:

```
[ атрибути ] [ специфікатори ] тип ім’я_властивості
{get; set; }
```

Зверніть увагу на те, що після позначень аксесорів **get** і **set** відразу ж ставиться крапка з комою, а тіло у них відсутнє. Такий синтаксис вказує

компілятору створити автоматично поле для зберігання значення властивості. Таке поле недоступне безпосередньо і не має імені.

Нижче наведено приклад опису автоматичної властивості:

```
public double Voltage { get; set; }
```

Як бачите, в цьому рядку змінна явно не оголошується. Компілятор автоматично створює анонімне поле, в якому зберігається значення властивості.

На відміну від звичайних властивостей, автоматична властивість не може бути доступною тільки для читання або тільки для запису. При оголошенні цієї властивості необхідно вказувати обидва аксесори – **get** і **set**. Зробити автоматичну властивість доступною тільки для читання або тільки для запису можна оголосивши непотрібний аксесор як **private**.

Незважаючи на очевидні зручності автоматичних властивостей, їх застосування обмежується в основному тими ситуаціями, в яких не потрібно керувати установкою або отриманням значень з полів. Базове поле автоматичної властивості недоступне безпосередньо. Це означає, що на значення, яке може мати автоматична властивість, не можна накласти ніяких обмежень.

Властивостями притаманний ряд обмежень. По-перше, властивість не визначає місце для зберігання даних, і тому не може бути передана методу в якості параметра **ref** або **out**. По-друге, властивість не повинна змінювати стан базової змінної при виклику аксесора **get**. І хоча це правило не перевіряється компілятором, його порушення вважається семантичної помилкою.

За замовчуванням доступність аксесорів **set** і **get** є такою ж, як і у властивості, частиною якої вони є. Так, якщо властивість оголошується як **public**, то за замовчуванням її аксесори **set** і **get** також стають відкритими (**public**). Проте для аксесорів **set** або **get** можна вказати власний специфікатор

доступу, наприклад **private**. Доступність аксесорів, що визначається таким специфікатором, повинна бути більш обмеженою, ніж доступність властивості.

Існує ряд причин, за яких потрібно обмежити доступність аксесорів. Припустимо, що потрібно надати вільний доступ до значення властивості, але разом з тим дати можливість змінювати цю властивість тільки елементам її класу. Для цього достатньо оголосити аксесор **set** даної властивості як **private**.

Робоче завдання

Навчитися створювати властивості в мові програмування C# та отримати досвід використання властивостей для доступу до даних екземплярів класу.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання практичної роботи №3:
    - для двох полів описаного раніше класу створити властивості, що забезпечують можливість їх зчитування та зміни;
    - додатково створити дві автоматичні властивості, які описують додаткові характеристики об’єкта на ваш вибір;
    - створити властивість, доступну тільки для читання, яка розраховує певне значення характеристики об’єкта на ваш вибір;
    - модифікувати конструктори класів шляхом додавання нових властивостей та заміни операторів присвоювання значень полів на оператори присвоювання значень властивостей;
    - доповнити метод *Main* () класу *Program* операторами присвоювання значень властивостям.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Що таке властивість класу?
2. Що таке автоматична властивість?
3. Для чого призначені аксесори **set** і **get**?
4. Які специфікатори доступу можливо використовувати для властивостей?
5. Чи можна використовувати різні специфікатори доступу для аксесорів **set** і **get?**
6. Як відбувається задання та зчитування полів класу за допомогою властивостей?
7. Чи може клас містити властивості, які не пов’язані з окремими полями?
8. Наведіть загальний синтаксис опису властивості.

ПРАКТИЧНА РОБОТА №5.
КОЛЕКЦІЇ. КЛАС LIST<T>

Мета

Вивчити основні принципи роботи колекцій в мові програмування C#. Отримати досвід створення узагальнених колекцій для зберігання та роботи з екземплярами класу.

Стислі теоретичні відомості

У бібліотеках більшості сучасних об'єктно-орієнтованих мов програмування представлені стандартні класи, що реалізують основні абстрактні структури даних. Такі класи називаються колекціями, або контейнерами. Для кожного типу колекції визначені методи роботи з її елементами, які не залежать від конкретного типу даних, які зберігаються в колекції, тому один і той же вид колекції можна використовувати для зберігання даних різних типів. Використання колекцій дозволяє скоротити терміни розробки програм і підвищити їх надійність.

Кожен вид колекції підтримує свій набір операцій над даними, і швидкодія цих операцій може бути різним. Вибір виду колекції залежить від того, що потрібно робити з даними в програмі і які вимоги пред'являються до її швидкодії. Наприклад, при необхідності часто вставляти і видаляти елементи з середини послідовності слід використовувати список, а не масив, а якщо включення елементів виконується головним чином в кінець або початок послідовності - черга. Тому вивчення можливостей стандартних колекцій і їх грамотне застосування є необхідними умовами створення ефективних і професійних програм.

У бібліотеці .NET визначені стандартні класи, що реалізують більшість перерахованих раніше абстрактних структур даних. Основні простори імен, в яких описані ці класи, – **System.Collections, System.Collections.Specialized** і **System.Collections.Generic**.

Всі колекції розроблені на основі набору чітко визначених інтерфейсів. Деякі вбудовані реалізації таких інтерфейсів, в тому числі **ArrayList, Hashtable, Stack** і **Queue**, можуть застосовуватися в початковому вигляді і без будь-яких змін. Є також можливість реалізувати власну колекцію, хоча потреба в цьому виникає вкрай рідко.

У середовищі .NET Framework підтримуються п'ять типів колекцій:

- неузагальнені,
- спеціальні,
- з поразрядною організацією,
- узагальнені,
- паралельні.

Неузагальнені колекції реалізують ряд основних структур даних, включаючи динамічний масив, стек, чергу, а також словники, в яких можна зберігати пари "ключ-значення". Відносно неузагальнених колекцій важливо мати на увазі наступне: вони оперують даними типу **object**.

Таким чином, неузагальнені колекції можуть служити для зберігання даних будь-якого типу, причому в одній колекції допускається наявність різнотипних даних. Очевидно, що такі колекції не типізовані, оскільки в них зберігаються посилання на дані типу **object**. Класи і інтерфейси неузагальнених колекцій знаходяться в просторі імен **System.Collections**.

Спеціальні колекції оперують даними конкретного типу або ж роблять це якимось особливим чином. Наприклад, є спеціальні колекції для символьних рядків, а також спеціальні колекції, в яких використовується односпрямований список. Спеціальні колекції оголошуються в просторі імен **System.Collections.Specialized**.

У прикладному інтерфейсі **Collections API** визначена одна колекція з поразрядною організацією - це **BitArray**. Колекція типу **BitArray** підтримує порозрядні операції, тобто операції над окремими двійковими розрядами, наприклад, І йди виключає АБО, а отже, вона істотно відрізняється своїми

можливостями від інших типів колекцій. Колекція типу **BitArray** оголошується в просторі імен **System.Collections**.

Узагальнені колекції забезпечують узагальнену реалізацію декількох стандартних структур даних, включаючи списки, стеки, черги і словники. Такі колекції є типізованими в силу їх узагальненого характеру. Це означає, що в узагальненій колекції можуть зберігатися тільки такі елементи даних, які сумісні за типом з даної колекцією. Завдяки цьому виключається випадкова розбіжність типів. Узагальнені колекції оголошуються в просторі імен **System.Collections.Generic**.

Як правило, класи узагальнених колекцій є узагальненими еквівалентами класів неузагальнених колекцій, хоча це відповідність не є взаємно однозначною. Наприклад, в класі узагальненої колекції **LinkedList** реалізується двонаправлений список, тоді як в неузагальнених еквіваленті його не існує. У деяких випадках одні й ті ж функції існують паралельно в класах узагальнених і неузагальнених колекцій, хоча і під різними іменами. Так, узагальнений варіант класу **ArrayList** називається **List**, а узагальнений варіант класу **HashTable** – **Dictionary**.

У табл. 5.1 перераховані основні колекції, описані в просторі **System.Collections.Generic**.

Таблиця 5.1 – Колекції простору імен System.Collections.Generic

Клас	Опис
`Dictionary <Tkey, TValue>`	Зберігає пари "ключ-значення". Забезпечує такі ж функціональні можливості, як і неузагальнений клас Hashtable
`HashSet <T>`	Зберігає ряд унікальних значень, використовуючи хеш таблицю
`LinkedList <T>`	Зберігає елементи в двонаправленому списку
`List <T>`	Створює динамічний масив. Забезпечує такі ж функціональні можливості, як і неузагальнений клас ArrayList
`Queue <T>`	Створює чергу. Забезпечує такі ж функціональні можливості, як і неузагальнений клас Queue

SortedDictionary <TKey, TValue>	Створює відсортований список з пар "ключ-значення"
SortedList <TKey, TValue>	Створює відсортований список з пар "ключ-значення". Забезпечує такі ж функціональні можливості, як і неузагальнений клас SortedList
SortedSet <T>	Створює відсортовану множину
Stack <T>	Створює стек. Забезпечує такі ж функціональні можливості, як і неузагальнених клас Stack

У колекції можна зберігати не тільки об'єкти вбудованих типів. В них допускається зберігати об'єкти будь-якого типу, включаючи об'єкти класів, що визначаються користувачем.

Завдяки тому, що всі типи успадковують від класу **object**, в неузагальнених колекції можна зберігати об'єкти будь-якого типу. Для того щоб зберегти об'єкти класів, що визначаються користувачем, в типізованій колекції, доведеться скористатися класами узагальнених колекцій.

У колекції, для якої оголошено тип елементів, завдяки поліморфізму можна зберігати елементи будь-якого похідного класу, але не елементи інших типів.

Здавалося б, у порівнянні зі звичайними колекціями це обмеження, а не універсальність, однак на практиці колекції, в яких дійсно потрібно зберігати значення різних, не пов'язаних межу собою типів, майже не використовуються. Перевагою ж такого обмеження є те, що компілятор може виконати контроль типів під час компіляції, а не виконання програми, що підвищує її надійність і спрощує пошук помилок.

Використання стандартних узагальнених колекцій для зберігання і обробки даних є хорошим стилем програмування, оскільки дозволяє скоротити терміни розробки програм і підвищити їх надійність. Рекомендується ретельно вивчити властивості і методи цих класів і вибирати найбільш підходящі в залежності від розв'язуваної задачі.

Розглянемо детально клас **List<T>**. У класі **List<T>** реалізується узагальнений динамічний масив.

В класі **List<T>** визначено наступні властивості:

```
int Count {get; }
bool IsReadOnly {get; }
```

Властивість **Count** містить ряд елементів, що зберігаються в даний момент в колекції. А властивість **IsReadOnly** має логічне значення **true**, якщо колекція доступна тільки для читання. Якщо ж колекція доступна як для читання, так і для запису, то ця властивість має логічне значення **false**.

У класі **List<T>** визначається також властивість **Capacity**. Це властивість оголошується наступним чином.

```
public int Capacity {get; set; }
```

Властивість **Capacity** дозволяє встановити і отримати ємність колекції в якості динамічного масиву. Ця ємність дорівнює кількості елементів, які може містити колекція до її вимушеного розширення. Така колекція розширюється автоматично, і тому встановлювати її ємність вручну необов'язково. Але з міркувань ефективності це іноді можна зробити, якщо заздалегідь відомо кількість елементів колекції. Завдяки цьому виключаються витрати на виділення додаткової пам'яті.

У класу **List<T>** є такі конструктори.

```
public List ()
public List (IEnumerable <T> collection)
public List (int capacity)
```

Перший конструктор створює порожню колекцію класу клас List <T>з обраною за замовчуванням ємністю. Другий конструктор створює колекцію типу **List** з кількістю елементів, яка визначається розміром параметра **collection**. Третій конструктор створює колекцію типу **List**, що має початкову ємність, яка задається параметром **capacity**.

Ємність колекції, яка створюється у вигляді динамічного масиву, може збільшуватися автоматично при додаванні в неї елементів.

У класі **List<T>** визначається ряд методів, найбільш важливі з яких перераховані в табл. 5.2.

Таблиця 5.2 – Методи, визначені в класі **List <T>**

Метод	Опис
void Add (T item)	Додає елемент item в колекцію. Генерує виняток NotSupportedException, якщо колекція доступна тільки для читання
void Clear ()	Видаляє всі елементи з колекції
bool Contains (T item)	Повертає логічне значення true, якщо колекція містить елемент item, а інакше - логічне значення false
void CopyTo (T [] array, int arraylndex)	Копіює вміст колекції в масив array, починаючи з елемента, що вказується за індексом arraylndex
bool Remove (T item)	Видаляє перше входження елемента item в колекції. Повертає логічне значення true, якщо елемент item видалений. А якщо цей елемент не знайдено в колекції, то повертається логічне значення false
int IndexOf (T item)	Повертає індекс першого входження елемента item в колекції. Якщо елемент item не виявлений, то метод повертає значення -1
void Insert (int index, T item)	Вставляє в колекцію елемент item за індексом index
void RemoveAt (int index)	Видаляє з колекції елемент, розташований за індексом index
public virtual void AddRange (ICollection collection)	Додає елементи з колекції collection типу ArrayList
public virtual int BinarySearch (T item)	Виконує пошук в колекції значення, що задається параметром item. Повертає індекс знайденого елемента. Якщо шукане значення не знайдено, повертається від'ємне значення. Список повинен бути відсортований
public List <T> GetRange (int Index, int count)	Повертає частину колекції. Частина колекції починається з елемента за індексом index, і включає кількість елементів, що задається параметром count

public int IndexOf (T item)	Повертає індекс першого входження елемента item в колекції. Якщо шуканий елемент не виявлений, повертається значення -1
public void InsertRange (int index, IEnumerable <T> collection)	Вставляє елементи колекції collection починаючи з елемента за індексом index
public itemj int LastlndexOf (T	Повертає індекс останнього входження елемента item в колекції. Якщо шуканий елемент не виявлений, повертається значення -1
public void RemoveRange (int index, int count)	Видаляє частину колекції, починаючи з елемента за індексом index, і включаючи кількість елементів, яка визначається параметром count
public void Reverse ()	Повертає колекцію в зворотному порядку
public void Reverse (int index, int count)	Повертає колекцію в зворотному порядку, починаючи з елемента за індексом index, і включаючи кількість елементів, яке визначається параметром count
public void Sort ()	Сортує колекцію за зростанням
public void Sort (IComparer <T> comparer)	Сортує колекцію, використовуючи для порівняння спосіб, що задається параметром comparer. Якщо параметр comparer має пусте значення, то для порівняння використовується спосіб, який обирається за замовчуванням
public T [] ToArrayO	Повертає масив, який містить копії елементів колекції

Для циклічного звернення до елементів колекції, що представляє собою групу об'єктів, служить оператор **foreach**. Нижче наведена загальна форма оператора циклу **foreach**.

```
foreach (тип імя_змінної_циклу in ім’я_колекції)
оператор;
```

Тут тип *імя_змінної_циклу* позначає тип і ім'я змінної управління циклом, яка отримує значення наступного елемента колекції на кожному кроці виконання циклу **foreach**. А *ім’я_колекції* позначає колекцію, що циклічно переглядається. Отже, тип змінної циклу повинен відповідати типу елемента колекції. Крім того, тип може позначатися ключовим словом **var**. В цьому

випадку компілятор визначає тип змінної циклу, виходячи з типу елемента колекції.

Оператор циклу **foreach** діє таким чином. Коли цикл починається, перший елемент колекції вибирається і присвоюється змінній циклу. На кожному наступному кроці ітерації вибирається наступний елемент колекції, який зберігається в змінній циклу. Цикл завершується, коли всі елементи колекції виявляться переглянутими. Отже, оператор **foreach** циклічно опитує колекції по окремих її елементів від початку і до кінця.

Слід, однак, мати на увазі, що змінна циклу в операторі **foreach** служить тільки для читання. Це означає, що, присвоюючи цій змінній нове значення, не можна змінити вміст масиву.

Незважаючи на те що цикл **foreach** повторюється до тих пір, поки не будуть переглянуті всі елементи колекції, його можна завершити достроково, скориставшись оператором break.

Оператор **foreach** допускає циклічне звернення до масиву тільки в певному порядку: від початку і до кінця масиву, тому його застосування здається, на перший погляд, обмеженим. Але насправді це не так. У великій кількості алгоритмів, найпоширенішим з яких є алгоритм пошуку, потрібно саме такий механізм.

Розглянемо приклад програми, в якій показана робота з колекцією об’єктів класу *Lamp*.

```
using System;
using System.Collections.Generic;

namespace ClassElementTest
{
    class Lamp
    {
        private string _name;
        private double _power;

        public double Power
        {
            get { return _power; }
            set
            {
```

```
                if (value >= 0)
                {
                    _power = value;
                }
            }
        }

        public double Voltage { get; set; }

        public double Current
        {
            get { return _power / Voltage; }
        }

        public Lamp(string nameValue, double powerValue, double voltageValue)
        {
            _name = nameValue;
            Power = powerValue;
            Voltage = voltageValue;
        }

        public override string ToString()
        {
            return String.Format("Лампа {0}: потужність - {1} Вт, напруга -
{2} В, струм - {3:n3} А", _name, Power, Voltage, Current);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Створити список
            var lampList = new List<Lamp>();
            //Додати елементи в список
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));
            //Вивести список на екран використовуючи цикл for
            Console.WriteLine("Список ламп за допомогою циклу for");
            for (int i = 0; i < lampList.Count; i++)
            {
                Console.WriteLine(lampList[i]);
            }
            //Вивести список на екран використовуючи цикл for
            Console.WriteLine("Список ламп за допомогою циклу foreach");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            Console.ReadLine();
        }
    }
}
```

У розглянутій програмі є все-таки один не зовсім очевидний недолік: створену колекцію *lampList* не можна відсортувати. Справа в тому, що в класах **ArrayList і List<T>** відсутні засоби для порівняння об'єктів користувацького типу даних. Для виходу з цього становища в класі можна реалізувати інтерфейс **IComparable**, в якому визначається метод порівняння об'єктів даного класу. Якщо потрібно впорядкувати об'єкти, що зберігаються в узагальненій колекції, то для цієї мети доведеться реалізувати узагальнений варіант інтерфейсу **IComparable<T>**. У цьому інтерфейсі визначається наведена нижче узагальнена форма методу **CompareTo()**:

```
int CompareTo(T other)
```

У методі CompareTo () поточний об'єкт порівнюється з іншим об'єктом other. Для сортування об'єктів по наростаючій конкретна реалізація даного методу повинна повертати нульове значення, якщо значення порівнюваних об'єктів рівні; додатнє – якщо значення поточного об'єкта більше, ніж у об'єкту other; і від'ємне – якщо значення поточного об'єкта менше, ніж у іншого об'єкта other.

Реалізуємо метод **CompareTo()** для класу *Lamp*, виконавши порівняння ламп за потужністю. Тобто будемо вважати, що більшою є лампа у якої більша потужність.

```
using System;
using System.Collections.Generic;

namespace ClassElementTest
{
    class Lamp:IComparable<Lamp>
    {
        private string _name;
        private double _power;

        public double Power
        {
            get { return _power; }
            set
            {
                if (value >= 0)
                {
                    _power = value;
                }
```

```
            }
        }

        public double Voltage { get; set; }

        public double Current
        {
            get { return _power / Voltage; }
        }

        public Lamp(string nameValue, double powerValue, double voltageValue)
        {
            _name = nameValue;
            Power = powerValue;
            Voltage = voltageValue;
        }

        public int CompareTo(Lamp other)
        {
            if (Power==other.Power)
            {
                return 0;
            }
            else if (Power>other.Power)
            {
                return 1;
            }
            else
            {
                return -1;
            }
        }

        public override string ToString()
        {
            return String.Format("Лампа {0}: потужність - {1} Вт, напруга -
{2} В, струм - {3:n3} А",
                _name, Power, Voltage, Current);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Створити список
            var lampList = new List<Lamp>();
            //Додати елементи в список
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));
            //Список до сортування
            Console.WriteLine("Список ламп до сортування");
            foreach (var lamp in lampList)
```

```
            {
                Console.WriteLine(lamp);
            }
            //Відсортуємо список
            lampList.Sort();
            Console.WriteLine("Відсортований список ламп");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            Console.ReadLine();
        }
    }
}
```

Робоче завдання

Навчитися створювати колекції в мові програмування C# та отримати досвід використання колекцій для зберігання та роботи з сукупністю екземплярів класів.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання практичної роботи №4:
    - у методі *Main* () класу *Program* створити колекцію типу **List<T>** із екземплярами описаного класу;
    - додати в колекцію 5-10 нових елементів використовуючи методи **Add()** та **Insert()**;
    - вивести значення елементів колекції на екран за допомогою одного з циклів **for, while, do/while** на вибір;
    - вивести значення елементів колекції на екран за допомогою циклу **foreach**;
    - доповнити клас реалізацією інтерфейсу **IComparable<T>** шляхом створення методу **CompareTo()** з порівнянням по одній із властивостей на вибір;

- у методі *Main* () класу *Program* виконати сортування створеної колекції та вивести результат на екран.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Які структури даних ви знаєте?
2. Які операції можна виконувати над списками?
3. Опишіть структуру даних стек.
4. Опишіть структуру даних черга.
5. Що таке колекція?
6. Які типи колекцій реалізовано в мові C#?
7. Що таке неузагальнені колекції?
8. Що таке узагальнені колекції?
9. Перерахуйте найважливіші колекції, описані в просторі System.Collections.Generic.
10. Які типи даних можна зберігати в узагальнених колекціях?
11. Які основні властивості визначені в класі **List<T>**?
12. Опишіть основні методи, визначені в класі **List<T>**.
13. Опишіть роботу оператора **foreach.**
14. Наведіть синтаксис оператора **foreach.**
15. Як забезпечити можливість сортування колекції з типом даних, що визначений користувачем?

ПРАКТИЧНА РОБОТА №6.
ФАЙЛОВЕ ВВЕДЕННЯ ТА ВИВЕДЕННЯ. СЕРІАЛІЗАЦІЯ ОБ'ЄКТІВ

Мета

Вивчити основні принципи роботи з файлами та серіалізації об'єктів в мові програмування C#. Отримати досвід збереження окремих екземплярів класу та колекцій і з допомогою бінарного форматера та у файлах XML.

Стислі теоретичні відомості

У середовищі .NET Framework передбачені класи для організації введення-виведення в файли. На рівні операційної системи файли складаються з байтів, тому існують відповідні методи для введення і виведення байтів в файли. Крім того, існують спеціальні операції символьного введення-виведення в файл, які застосовуються при обробці тексту.

Для створення байтового потоку, пов'язаного з файлом, служить клас **FileStream**. Клас **FileStream** визначено в просторі імен **System.IO**. Тому на початку будь-якої програми, що його використовує клас **FileStream**, додається такий рядок коду:

```
using System.IO;
```

Для відкриття файлу створюється об'єкт класу **FileStream**. В цьому класі визначено кілька конструкторів. Нижче наведено найпоширеніший серед них:

```
FileStream (string шлях, FileMode режим)
```

де *шлях* позначає ім'я файлу, включаючи повний шлях до нього; а *режим* порядок відкриття файлу, де вказується одне зі значень, що перерахуванні табл. 6.1. Як правило, цей конструктор відкриває файл для доступу з метою читання або запису. Винятком з цього правила є відкриття файлу в режимі **FileMode.Append**, коли файл стає доступним тільки для запису.

Таблиця 6.1 Значення режимів відкриття файлів FileMode

Значення	Опис
FileMode.Append	**Додає дані в кінець файлу**
FileMode.Create	**Створює новий вихідний файл. Існуючий файл з таким же ім'ям буде знищено**
FileMode.CreateNew	**Створює новий вихідний файл. Файл з таким же ім'ям не повинен існувати**
FileMode.Open	**Відкриває існуючий файл**
FileMode.OpenOrCreate	**Відкриває файл, якщо він існує. В іншому випадку створює новий файл**
FileMode.Truncate	**Відкриває існуючий файл, але скорочує його довжину до нуля**

Якщо спроба відкрити файл виявляється невдалою, то генерується помилка:

- якщо файл не можна відкрити тому що він не існує, генерується помилка **FileNotFoundException**;
- якщо файл не можна відкрити через помилки введення-виведення, генерується помилка **IOException**.

До числа інших помилок, які можуть бути згенеровані при відкритті файлу, відносяться такі: **ArgumentNullException** (вказано порожнє ім'я файлу), **ArgumentException** (вказано невірне ім'я файлу), **ArgumentOutOfRangeException** (вказана невірний режим), **SeaurityException** (у користувача немає прав доступу до файлу), **PathTooLongException** (занадто довге ім'я файлу або шлях до нього), **NotSupportedException** (в імені файлу вказано пристрій, який не підтримується), а також **DirectoryNotFoundException** (вказано невірний каталог).

По завершенні роботи з файлом його слід закрити, викликавши метод **Close()**.

У класі **FileStream** визначені два методу для читання байтів з файлу: **ReadByte()** і **Read()**. Так, для читання одного байта з файлу використовується метод **ReadByte()**, загальна форма якого наведена нижче:

```
int ReadByte()
```

Коли цей метод викликається, з файлу зчитується один байт, який потім повертається у вигляді цілого значення.

Для читання блоку байтів з файлу служить метод **Read()**, загальна форма якого виглядає так.

```
int Read(byte [] array, int offset, int count)
```

У методі **Read()** робиться спроба зчитати кількість *count* байтів в масив *array*, починаючи з елемента *array*[*offset*]. Метод повертає кількість байтів, успішно зчитаних з файлу. Якщо ж виникає помилка введення-виведення, то генерується виключення **IOException**.

Для запису байта в файл служить метод **WriteByte()**. Нижче наведена його найпростіша форма.

```
void WriteByte(byte value)
```

Цей метод виконує запис в файл байта, що позначається параметром *value*.

Для запису в файл цілого масиву байтів може бути викликаний метод **Write()**. Нижче наведена його загальна форма.

```
void Write(byte [] array, int offset, int count)
```

У методі **Write()** робиться спроба записати в файл кількість *count* байтів з масиву *array*, починаючи з елемента *array*[*offset*]. Метод повертає кількість байтів, успішно записаних в файл.

По завершенні виведення в файл, його слід закрити за допомогою методу **Close()**.

Незважаючи на те, що файли часто обробляються побайтово, для цього можна скористатися також символьними потоками. Перевага символьних потоків полягає в тому, що вони оперують символами безпосередньо в Unicode. Якщо потрібно зберегти текст, то для цього найкраще підійдуть саме символьні потоки. В цілому, для виконання операцій символьного введення-виведення в файли використовуються класи **StreamReader** та **StreamWriter**.

Для створення символьного потоку виведення досить укласти об'єкт класу **Stream**, наприклад **FileStream**, в оболонку класу **StreamWriter**. У класі **StreamWriter** визначено кілька конструкторів. Нижче наведено найпоширеніший серед них:

```
StreamWriter(Stream потік)
```

де *потік* позначає ім'я відкритого потоку. Цей конструктор генерує виняток **ArgumentException**, якщо потік не відкритий для виведення, а також виключення **ArgumentNullException**, якщо потік виявляється порожнім.

У деяких випадках зручніше відкривати файл засобами самого класу **StreamWriter**. Для цього служить один з наступних конструкторів:

```
StreamWriter(string шлях)
StreamWriter(string шлях, bool append)
```

де *шлях* – це ім'я файлу, включаючи повний шлях до нього. Якщо в другій формі цього конструктора значення параметра *append* рівне **true**, то дані приєднуються в кінець існуючого файлу. В іншому випадку ці дані перезаписують вміст зазначеного файлу. Але незалежно від форми конструктора файл створюється, якщо він не існує.

Нижче наведено простий приклад програми виведення на диск опису об’єкту класу *Lamp* у вигляді текстових рядків, як зберігаються в файлі test.txt. Для символьного виводу в файл в цій програмі використовується об'єкт класу FileStream, укладений в оболонку класу StreamWriter.

```
using System;
using System.IO;

namespace ClassElementTest
{
    class Lamp
    {  ...  }

    class Program
    {
        static void Main(string[] args)
        {
            var sampleLamp = new Lamp("СЛ-21", 15, 220);
            var fout=new StreamWriter("text.txt");
            fout.WriteLine(sampleLamp);
            fout.Close();
        }
    }
}
```

Для створення символьного потоку введення досить укласти байтовий потік в оболонку класу **StreamReader**. У класі **StreamReader** визначено кілька конструкторів. Найбільш часто використовується конструктор:

```
StreamReader(Stream потік)
```

де *потік* позначає ім'я відкритого потоку. Цей конструктор генерує виняток **ArgumentNullException**, якщо потік виявляється порожнім, а також виключення **ArgumentException**, якщо потік не відкритий для введення.

Іноді файл простіше відкрити, використовуючи безпосередньо клас **StreamReader**, аналогічно класу **StreamWriter**. Для цієї мети служить наступний конструктор:

```
StreamReader(string шлях)
```

де *шлях* – це ім'я файлу, включаючи повний шлях до нього. Вказаний файл повинен існувати. В іншому випадку генерується виняток **FileNotFoundException**. Якщо шлях виявляється порожнім, то генерується виключення **ArgumentNullException**.

Нижче наведено простий приклад програми зчитування з диску текстових рядків, як зберігаються в файлі *text.txt*.

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        var fin = new StreamReader(@"D:\text.txt");
        while (!fin.EndOfStream)
        {
            var s = fin.ReadLine();
            Console.WriteLine(s);
        }
        fin.Close();
    }
}
```

Зверніть увагу на те, як в цій програмі визначається кінець файлу. Доступна для читання властивість **EndOfStream** має логічне значення **true**, коли досягається кінець потоку, в іншому випадку – логічне значення **false**. Отже, властивість **EndOfStream** можна використовувати для відстеження кінця файлу.

У середовищі .NET Framework також визначено клас **File**, який є корисним для роботи з файлами, оскільки він містить статичні методи, що виконують типові операції над файлами. Зокрема, в класі **File** є методи для копіювання та переміщення, шифрування і розшифрування, видалення файлів, а також для отримання і завдання інформації про файлах, включаючи відомості про їхнє існування, часу створення, останнього доступу і різні атрибути файлів (тільки для читання, прихованих і ін.). Крім того, в класі **File** є ряд зручних методів для читання з файлів і записи в них, відкриття файлу і отримання посилання типу **FileStream** на нього.

Ряд методів для роботи з файлами визначено також в класі **FileInfo**. Цей клас відрізняється від класу **File** дуже важливою перевагою: для операцій над файлами він надає методи екземпляра і властивості, а не статичні методи. Тому для виконання декількох операцій над одним і тим же файлом краще скористатися класом **FileInfo**.

У C# є можливість зберігати не тільки дані примітивних типів, але і об'єкти. Термін серіалізація описує процес збереження (і, можливо, передачі) стану об'єкта в потоці (наприклад, файловому потоці і потоці в пам'яті). Послідовність даних, що зберігається містить всю інформацію, необхідну для реконструкції (або десеріалізації) стану об'єкта з метою подальшого використання. Застосовуючи цю технологію, дуже просто зберігати великі обсяги даних (в різних форматах) з мінімальними зусиллями. У багатьох випадках збереження даних програми з використанням серіалізації виливається в код меншого обсягу, ніж застосування класів для читання/запису з простору імен **System.IO**.

Щоб зробити об'єкт доступним для серіалізації, необхідно позначити клас (або структуру) атрибутом **[Serializable]**. Атрибути – це додаткові відомості про клас, які зберігаються в його метаданих. Якщо існують поля, які не повинні (або не можуть) брати участь в серіалізації, можна помітити такі поля атрибутом **[NonSerialized]**.

Об'єкти можна зберігати в одному з таких форматів: бінарному, SOAP або у вигляді XML-файла. Перераховані можливості представлені такими класами: **BinaryFormatter, SoapFormatter, XmlSerializer.**

Тип **BinaryFormatter** серіалізує стан об'єкта в потік, використовуючи компактний бінарний формат. Цей тип визначений в просторі імен **System.Runtime.Serialization.Formatters.Binary**. Таким чином, щоб отримати доступ до цього типу, необхідно вказати наступну директиву **using**:

```
using System.Runtime.Serialization.Formatters.Binary;
```

Тип SoapFormatter зберігає стан об'єкта у вигляді повідомлення SOAP. Цей тип визначений в просторі імен **System.Runtime.Serialization.Formatters.Soap**, що знаходиться в окремій збірці. Тому для форматування об'єктів в повідомлення SOAP необхідно спочатку встановити посилання на

System.Runtime.Serialization.Formatters.Soap.dll, використовуючи діалогове вікно *Add Reference* (Додати посилання) в Visual Studio і потім вказати наступну директиву **using**:

```
using System.Runtime.Serialization.Formatters.Soap;
```

Для збереження об'єктів в документі XML передбачений тип **XmlSerializer**. Щоб використовувати цей тип, потрібно вказати директиву **using**:

```
using System.Xml.Serialization;
```

Двома ключовими методами типу **BinaryFormatter** є: **Serialize()** – зберігає об'єкт в зазначений потік у вигляді послідовності байтів; **Deserialize()** – перетворює збережену послідовність байтів в об'єкт.

Припустимо, що після створення екземпляра класу і модифікації деяких даних стану потрібно зберегти цей екземпляр у файлі *.dat. Почати слід з створення самого файлу *.dat. Для цього можна створити екземпляр типу **System.IO.FileStream**. Потім потрібно буде створити екземпляр **BinaryFormatter** і передати йому **FileStream** і об'єкт для збереження.

Для прикладу розглянемо додаток в якому реалізована серіалізація об’єктів типу Lamp.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ClassElementTest
{
    [Serializable]
    class Lamp
    {   ...   }
    class Program
    {
        static void Main(string[] args)
        {
            var sampleLamp = new Lamp("СЛ-21", 15, 220);
            var binFormat = new BinaryFormatter();
            var fStream = new FileStream("Lamp.dat", FileMode.Create);
```

```
            binFormat.Serialize(fStream, sampleLamp);
            fStream.Close();
        }
    }
}
```

Після виконання програми можна переглянути вміст файлу *Lamp.dat* в папці *bin\Debug* поточного проекту.

Тепер припустимо, що необхідно прочитати збережений об'єкт із двійкового файлу назад в змінну. Після відкриття файлу *Lamp.dat* (за допомогою методу **File.OpenRead()**) можна викликати метод **Deserialize()** класу **BinaryFormatter**. **Deserialize()** повертає об'єкт загального типу **System.Object**, тому необхідно застосувати явне приведення, як показано нижче:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ClassElementTest
{
    [Serializable]
    class Lamp
    {  ...  }
    class Program
    {
        static void Main(string[] args)
        {
            var dataFile = File.OpenRead(@"D:\Lamp.dat");
            var binFormat = new BinaryFormatter();
            var sampleLamp = (Lamp)binFormat.Deserialize(dataFile);
            Console.WriteLine(sampleLamp);
        }
    }
}
```

Серіалізация об'єктів з використанням XmlSerializer може використовуватися для збереження стану заданого об'єкта у вигляді чистої XML-розмітки. XML-документ складається із тексту, і придатний до читання людиною. Розглянемо наступний код:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

```
using System.Xml.Serialization;

namespace ClassElementTest
{
    [Serializable]
    public class Lamp
    {   ...   }
    class Program
    {
        static void Main(string[] args)
        {
            {
                var sampleLamp = new Lamp("СЛ-21", 15, 220);
                var xmlFormat = new XmlSerializer(typeof (Lamp));
                var xmlFile = new FileStream("Lamp.xml", FileMode.Create);
                xmlFormat.Serialize(xmlFile, sampleLamp);
                xmlFile.Close();
            }
            {
                var xmlFile = File.OpenRead(@"D:\Lamp.xml");
                var xmlFormat = new XmlSerializer(typeof(Lamp));
                var sampleLamp = (Lamp)xmlFormat.Deserialize(xmlFile);
                Console.WriteLine(sampleLamp);
            }
        }
    }
}
```

Ключова відмінність, від прикладу з **BinaryFormatter** полягає в тому, що тип **XmlSerializer** вимагає вказівки інформації про клас, який необхідно серіалізувати. У створеному файлі XML знаходяться показані нижче дані XML:

```
<?xml version="1.0"?>
<Lamp xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <Power>15</Power>
  <Voltage>220</Voltage>
</Lamp>
```

Клас **XmlSerializer** вимагає, щоб всі серіалізовані типи об'єктів підтримували стандартний конструктор (без параметрів).

Особливістю сереалізації колекцій об'єктів, є те, що більшість типів з просторів імен **System.Collections** і **System.Collections.Generic** вже позначені

атрибутом **[Serializable]**. Таким чином, щоб зберегти сукупність об'єктів, можна просто додати їх в колекцію (таку як звичайний масив, **ArrayList** або **List<T>**) і серіалізувати даний об'єкт в бажаний потік (файл).

Робоче завдання

Навчитися виконувати введення та виведення даних з файлів в мові програмування C# та отримати досвід використання серіалізації окремих об’єктів та колекцій.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання практичної роботи №5:
    - надати створеному класу специфікатор **public** та атрибут **[Serializable]**;
    - створити один екземпляр класу та вивести текстову інформацію про нього у файл. Відкрити файл у текстовому редакторі та перевірити наявність інформації. Навести вміст файлу у протоколі виконання роботи;
    - створити один екземпляр класу та виконати його серіалізацію у бінарному форматі у файл. Знайти файл на диску та перевірити наявність даних;
    - зчитати серіалізований об’єкт в бінарному форматі з файлу у нову змінну;
    - виконати серіалізацію раніше створеної колекції об’єктів у файл формату XML Відкрити файл у текстовому редакторі та перевірити наявність інформації. Навести вміст файлу у протоколі виконання роботи.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Що таке файл?
2. Який клас служить для байтового файлового введення/виведення?
3. Який клас служить для символьного файлового введення/виведення?
4. Як відкрити файл для зчитування?
5. Як відкрити файл для запису?
6. Які є режими відкриття файлу?
7. Які помилки можливі при роботі з файлами?
8. Як зчитати байти або символи з файлу?
9. Як записати байти або символи у файл?
10. Як визначається кінець файлу?
11. Для чого призначені класи **File** та **FileInfo**?
12. Що таке серіалізація?
13. Які види серіалізації доступні у мові C#?
14. Яка послідовність дій виконується для серіалізації об’єкта?
15. Яка послідовність дій виконується для десеріалізації об’єкта?
16. Як виконується серіалізація та десеріалізація колекцій?

ПРАКТИЧНА РОБОТА №7. РОЗРОБКА ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ WPF. ЧАСТИНА 1. СТВОРЕННЯ ПРОСТОГО ВІКОННОГО ДОДАТКУ.

Мета

Познайомитися з основними можливостями розробки графічного інтерфейсу користувача з використанням технології WPF. Отримати досвід створення простих віконних додатків. Вивчити базові елементи мови розмітки XAML.

Стислі теоретичні відомості

Windows Presentation Foundation (WPF) система для побудови додатків Windows з візуально привабливими можливостями взаємодії з користувачем, графічна (презентаційна) підсистема у складі .NET Framework.

WPF надає засоби для створення візуального інтерфейсу, включаючи мову XAML (Extensible Application Markup Language), елементи управління, прив'язку даних, макети, двомірну і тривимірну графіку, анімацію, стилі, шаблони, документи, текст, мультимедіа і оформлення.

Побудова інтерфейсів користувача для WPF-додатків здійснюється із застосуванням мови розширеної розмітки додатків XAML. XAML-документ містить розмітку, що описує зовнішній вигляд і поведінку вікна або сторінки додатка, а пов'язані з ним файли коду C# – логіку програми. Мова XAML забезпечує поділ процесу дизайну додатки (графічної частини) і розробки бізнес-логіки (програмного коду) між дизайнерами і розробниками.

XAML базується на мові розширеної розмітки XML (Extensible Markup Language) і його синтаксис визначаються наступними правилами:

- кожен елемент XAML-документа відображається на деякий екземпляр класу .NET. Ім'я такого елемента в точності відповідає імені класу. Наприклад, елемент <*Button*> служить інструкцією для побудови об'єкта класу **Button**;

- елементи XAML можна вкладати один в одного. вкладення елементів розмітки зазвичай відображає вкладеність елементів інтерфейсу;
- властивості класу визначаються за допомогою атрибутів або за допомогою вкладених дескрипторів зі спеціальним синтаксисом.

XAML пропонує дуже просту і ясну схему визначення різних елементів і їх властивостей. Кожен елемент, як і будь-який елемент XML, повинен мати відкритий і закритий тег, як у випадку з елементом *Window*:

```
<Window> </ Window>
```

Або елемент може мати скорочену форму з закриваючим слешем в кінці, наприклад:

```
<Window />
```

Кожен елемент в XAML відповідає певному класу C#, а властивості цього класу відповідають атрибутам елемента *Button*.

Наприклад, створимо просте вікно з кнопкою. Для цього, при створенні нового проекту потрібно у лівій частині діалогового вікна вибрати пункт Visual C#, в правій – пункт WPF Application (рис. 7.1).

Після натискання кнопки "ОК" буде сформований шаблон проекту. При цьому згенерується наступний XAML-документ:

```
<Window x:Class="WPFTest_1.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:WPFTest_1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

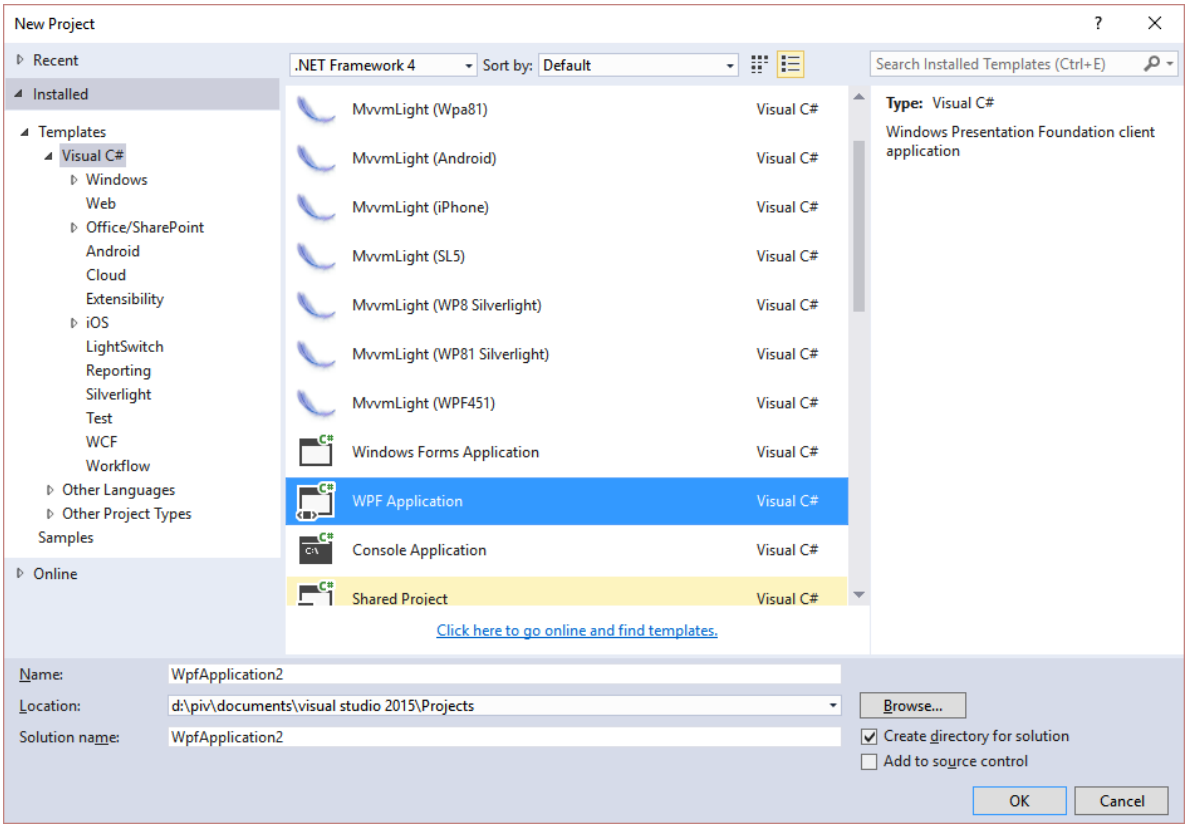

Рис. 7.1 – Створення проекту

Дизайнер проекту наведено на рис. 7.2.

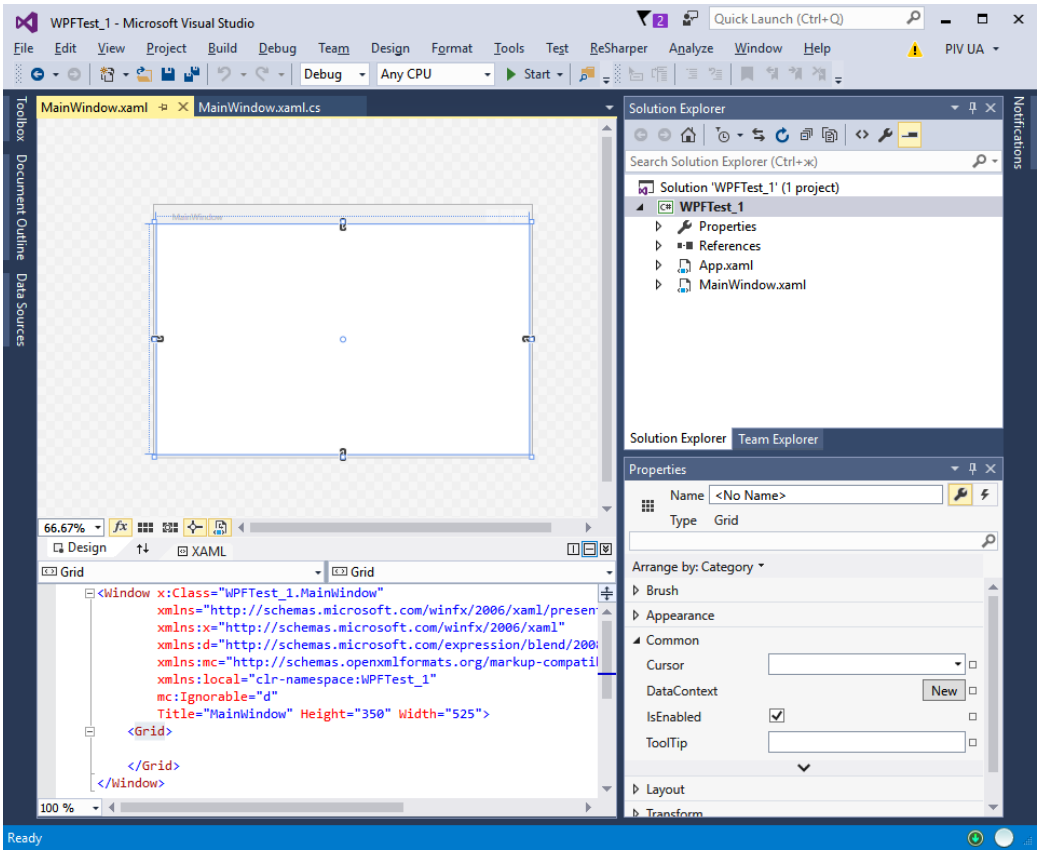

Рис. 7.2 – Дизайнер проекту

Спочатку йде елемент самого вищого рівня – *Window*, потім йде вкладений елемент *Grid* – контейнер для інших елементів, і в ньому вже визначимо елемент *Button*, який представляє кнопку. Для створення кнопки додаємо такий код між тегами <*Grid*> </*Grid*>:

```
<Button x:Name = "firstButton" Width = "300" Height = "70"
Content = "Натисни мене" />
```

Для кнопки ми можемо визначити властивості у вигляді атрибутів. Тут визначені атрибути *x:Name* (ім'я кнопки), *Width*, *Height* і *Content*. Подібним чином ми можемо визначити і інші атрибути, які нам потрібні.

При створенні нового проекту WPF на додаток до створюваного файлу *MainWindow.xaml* створюється також файл відокремленого коду *MainWindow.xaml.cs*, де має міститися логіка програми, пов'язана з розміткою. Файли XAML дозволяють нам визначити інтерфейс вікна, але для створення логіки програми все одно доведеться користуватися кодом C #.

У програмі часто потрібно звернутися до якогось елементу управління. Для цього треба визначити у елемента в XAML властивість *Name*.

Ще однією точкою взаємодії між XAML і C# є події. За допомогою атрибутів в XAML ми можемо поставити події, які будуть пов'язані з обробниками в коді C#.

Додаймо до попереднього проекту можливість реагувати на натискання кнопки. Для цього розмістимо на формі три текстових поля і надаймо їм імена (*Name*) *X*, *Y*, *Result*. Отримаємо такий код XAML:

```
<Window x:Class="WPFTest_1.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:WPFTest_1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
```

```
    <Grid>
        <TextBox x:Name = "X" Width = "150" Height = "30"
VerticalAlignment = "Top" Margin = "20" Text="0" />
        <TextBox x:Name = "Y" Width = "150" Height = "30"
VerticalAlignment = "Top" Margin = "55" Text="0" />
        <TextBox x:Name = "Result" Width = "150" Height = "30"
VerticalAlignment = "Top" Margin = "90" />
        <Button x:Name = "firstButton" Width = "300" Height = "70"
Content = "Натисни мене" Click="firstButton_Click" />
    </Grid>
</Window>
```

Атрибут *Margin* у цьому випадку задає відступ від верхньої межі вікна. Атрибут кнопки

```
     Click="firstButton_Click"
```

задає назву обробника її натискання. В обробнику задаються дії, які потрібно виконати при натискання кнопки.

Змінимо файл коду, додавши в нього обробник натискання кнопки:

```
using System.Windows;

namespace WPFTest_1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void firstButton_Click(object sender, RoutedEventArgs
e)
        {
            var a = double.Parse(X.Text);
            var b = double.Parse(Y.Text);
            var res = a*a + b*b;
            Result.Text = res.ToString();
        }
    }
}
```

Визначивши імена елементів в XAML, потім ми можемо до них звертатися в коді C#:

```
Result.Text = res.ToString();
```

У обробнику натиснення кнопки просто обчислюємо суму квадратів двох чисел в перших полях і виводимо результат у третє текстове поле.

Розміщення різних елементів інтерфейсу виконується за допомогою компонування (layout). Завдяки компонуванню ми можемо зручним чином налаштувати елементи інтерфейсу та позиціонувати їх. Наприклад, елементи компоновки в WPF дозволяють при ресайзі – стисканні або розтягуванні – дуже зручно автоматично масштабувати елементи.

У WPF компоновка здійснюється за допомогою спеціальних контейнерів. Фреймворк надає нам такі контейнери: *Grid, UniformGrid, StackPanel, WrapPanel, DockPanel і Canvas*. Різні контейнери можуть містити в собі інші контейнери.

Grid це найбільш потужний і часто використовуваний контейнер, що нагадує звичайну таблицю. Він містить стовпці і рядки, кількість яких задає розробник. Для визначення рядків використовується властивість *RowDefinitions*, а для визначення стовпців – властивість *ColumnDefinitions*:

```
<Grid.RowDefinitions>
   <RowDefinition> </ RowDefinition>
   <RowDefinition> </ RowDefinition>
   <RowDefinition> </ RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
   <ColumnDefinition> </ ColumnDefinition>
   <ColumnDefinition> </ ColumnDefinition>
   <ColumnDefinition> </ ColumnDefinition>
</Grid.ColumnDefinitions>
```

Кожен рядок задається за допомогою вкладеного елемента *RowDefinition*. При цьому задавати додаткову інформацію необов'язково. Кожен стовпець задається за допомогою вкладеного елемента *ColumnDefinition*.

Щоб задати позицію елемента управління з прив'язкою до певної комірки *Grid*, в розмітці елемента потрібно прописати значення властивостей *Grid.Column* і *Grid.Row*, тим самим вказуючи, в якому стовпці і рядку буде знаходитися елемент.

```
<Button Grid.Column = "0" Grid.Row = "0" Content =
"Рядок 0 Стовпець 0" />
<Button Grid.Column = "2" Grid.Row = "2" Content =
"Рядок 2 Стовпець 2" />
```

Є кілька варіантів настройки розмірів комірок Grid.

Автоматичні розміри. Тут стовпець або рядок займає те місце, яке їм потрібно

```
<ColumnDefinition Width = "Auto" />
<RowDefinition Height = "Auto" />
```

Абсолютні розміри. В даному випадку висота і ширина вказуються в одиницях, незалежних від пристрою:

```
<ColumnDefinition Width = "150" />
<RowDefinition Height = "150" />
```

Також абсолютні розміри можна задати в пікселях, дюймах, сантиметрах або точках: пікселі (px), дюйми (in), сантиметри (cm), точки (pt). наприклад,

```
<ColumnDefinition Width = "1 in" />
<RowDefinition Height = "10 px" />
```

Пропорційні розміри. Наприклад, нижче задаються два стовпці, другий з яких має ширину в чверть від ширини першого:

```
<ColumnDefinition Width = "*" />
<ColumnDefinition Width = "0.25*" />
```

Якщо рядок або стовпець має висоту, рівну *, то даний рядок або стовпчик буде займати все місце, що залишилося. Якщо у нас є кілька рядків або стовпців, висота яких дорівнює *, то все доступне місце ділиться порівну між усіма такими термінами і стовпцями. Використання коефіцієнтів (0.25*) дозволяє зменшити або збільшити виділене місце на даний коефіцієнт. При цьому всі коефіцієнти складаються (коефіцієнт * аналогічний 1*) і потім весь простір ділиться на суму коефіцієнтів.

Щоб взаємодіяти з користувачем, отримувати від користувача введення з клавіатури або миші і використовувати введені дані в програмі, використовуються елементи управління.

Всі елементи управління можуть бути умовно розділені на кілька підгруп:

- Елементи керування вмістом, наприклад кнопки (*Button*), мітки (*Label*).
- Спеціальні контейнери, які містять інші елементи, але на відміну від елементів *Grid* або *Canvas* не є контейнерами компонування – ScrollViewer, GroupBox.
- Декоратори, чиє призначення створення певного фону навколо вкладених елементів, наприклад, *Border* або *Viewbox*.
- Елементи управління списками, наприклад, *ListBox, ComboBox*.
- Текстові елементи управління, наприклад, *TextBox, RichTextBox*.
- Елементи, засновані на діапазонах значень, наприклад, *ProgressBar, Slider*.
- Елементи для роботи з датами, наприклад, *DatePicker* і *Calendar*.

- Інші елементи управління, які не увійшли в попередні підгрупи, наприклад, *Image*.

Розглянемо деякі з основних властивостей, які є характерними практично для усіх елементів управління.

Name. За визначеним іменем можна звертатися до елементу як в коді програми, так і в XAML розмітці. Приклад використання вже був показаний вище.

Visibility. Ця властивість встановлює параметри видимості елемента і може приймати одне з трьох значень:

- *Visible* – елемент видно і він бере участь в компонуванні.
- *Collapsed* елемент не видно і він не бере участь в компонуванні.
- *Hidden* елемент е видно, але при цьому бере участь в компонуванні.

Властивості шрифтів:

- *FontFamily* визначає сімейство шрифту (наприклад, *Arial, Verdana* та т. д.)
- *FontSize* визначає висоту шрифту
- *FontStyle* визначає нахил шрифту, приймає одне з трьох значень - *Normal, Italic, Oblique*.
- *FontWeight* визначає товщину шрифту і приймає ряд значень, як *Black, Bold* та ін.
- *FontStretch* визначає, як буде розтягувати або стискати текст, наприклад, значення *Condensed* стискає текст, а *Expanded* розтягує.

Кольори фону і шрифту. Властивості *Background* і *Foreground* задають відповідно колір фону і тексту елемента управління.

Найпростіший спосіб завдання кольору в коді XAML:

```
Background = "DarkRed"
```

Як значення властивість *Background* (*Foreground*) може приймати запис у вигляді шістнадцяткового значення в форматі #rrggbb, де rr - червона складова, bb - зелена складова, а bb - синя. Або можна використовувати назви кольорів безпосередньо: *Red*, *Green*, *LightGray*.

Робоче завдання

Навчитися створювати прості додатки з графічним інтерфейсом, використовуючи технологію WPF.

Хід роботи

1. Створити новий проект типу WPF Application.
2. Додати в проект новий файл коду і скопіювати до нього клас що був розроблений під час виконання практичної роботи №6. Зверніть увагу, що копіювати потрібно тільки сам клас, а не код методу *Main()*!
3. Додати в основне вікно текстові поля (*Text*), які будуть відповідати полям класу. Біля них розмістити пояснення (опис) у елементах типу *Label*. Розміщення та вирівнювання всіх елементів виконати з використанням рядків і стовпчиків контейнера Grid.
4. Додати багаторядкове текстове поле для виведення результату (*TextBlock*).
5. Кожному елементу присвоїти відповідне ім’я.
6. Для текстових полів, що призначені для введення та виведення інформації, задати різні властивості шрифтів на вибір.
7. Додати кнопку. Для кнопки запрограмувати обробник натискання, в якому виконати такі дії:

- за допомогою конструктора створити новий екземпляр класу, описаного в попередніх практичних заняттях. Значення полів взяти з відповідних текстових елементів.
- Вивести текстовий опис створеного об’єкта у текстове поле для результату.

8. У протоколі виконання практичного заняття навести код C# та код розмітки XAML, а також зображення вікна запущеної програми.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Для чого призначена технологія WPF?
2. Що таке XAML?
3. Як створити новий проект додатку з графічним інтерфейсом?
4. Опишіть основні правила синтаксису XAML.
5. Як додати новий елемент інтерфейсу на вікно додатку?
6. Як задати певні атрибути елементів?
7. Які основні атрибути ви знаєте?
8. Як звернутися до певного елемента інтерфейсу із коду C#?
9. Що таке обробник події для елемента інтерфейсу?
10. Як задати певний обробник натискання кнопки?
11. Що таке компоновка?
12. Для чого призначений контейнер Grid?
13. Опишіть роботу контейнера Grid.
14. Які елементи інтерфейсу можуть бути використані для введення та виведення певної інформації?
15. Які групи елементів інтерфейсу можна виділити в середовищі WPF?

ПРАКТИЧНА РОБОТА №8. РОЗРОБКА ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ WPF. ЧАСТИНА 2. ДІАЛОГОВІ ВІКНА, МЕНЮ ТА ПАНЕЛІ

Мета

Познайомитися з розширеними можливостями розробки графічного інтерфейсу користувача з використанням технології WPF. Отримати досвід використання діалогових вікон для відкриття та збереження файлів. Навчитися створювати головне меню програми.

Стислі теоретичні відомості

У WPF доступні для використання декілька стандартних діалогових вікон, таких як **OpenFileDialog** і **SaveFileDialog**. Робота з будь-яким з цих діалогових вікон зводиться до створення об'єкта відповідного класу і викликом методу ShowDialog (). В класах визначені різні члени, що дозволяють встановлювати фільтри файлів і шляхи до каталогів і отримувати доступ до вибраних користувачем файлів.

Розглянемо такий приклад:

```
using System.Windows;
using System.IO;
using Microsoft.Win32;

namespace WPFTest_1
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void SaveBtn_Click(object sender, RoutedEventArgs e)
        {
            var saveDlg = new SaveFileDialog();
            saveDlg.Filter = "Text Files | * .txt";
            if (true == saveDlg.ShowDialog())
            {
                // Зберегти дані з TextBlock в зазначеному файлі
                File.WriteAllText(saveDlg.FileName, text.Text);

            }
```

```
        }

        private void OpenBtn_Click(object sender, RoutedEventArgs e)
        {
            // Створити діалогове вікно відкриття файлу, що відображає
тільки текстові файли
            var openDlg = new OpenFileDialog();
            openDlg.Filter = "Text Files | * .txt";
            if (true == openDlg.ShowDialog())
            {
                // Завантажити вміст вибраного файлу
                string dataFromFile =
File.ReadAllText(openDlg.FileName);
                // Показати рядок в TextBlock
                text.Text = dataFromFile;
            }
        }
    }
}
```

```
<Window x:Class="WPFTest_1.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:WPFTest_1"
        mc:Ignorable="d"
        Title="Window1" Height="300" Width="500">
    <DockPanel>
        <StackPanel>
            <Button x:Name="OpenBtn" Content="Відкрити" Height="50"
Click="OpenBtn_Click"></Button>
            <Button x:Name="SaveBtn" Content="Зберегти" Height="50"
Click="SaveBtn_Click"/>
            <TextBox x:Name="text" TextWrapping="Wrap"
AcceptsReturn="True"></TextBox>
        </StackPanel>
    </DockPanel>
</Window>
```

Як бачите, для роботи додатку необхыдно імпортувати простори імен **System.IO** і **Microsoft.Win32** в файл коду.

У головному вікні програми створено дві кнопки і текстове поле. При натисканні на кнопку «Відкрити» відкривається діалогове вікно вибору файлу. Вміст вибраного файлу відображається у текстовому полі. При натисканні кнопки «Зберегти» вміст текстового поля записується у текстовий файл.

Створення меню в WPF відбувається за допомогою класу **Menu**, який підтримує колекцію об'єктів **MenuItem**. При побудові меню в XAML кожен **MenuItem** може обробляти різні події, з яких найважливішою є **Click**, що виникає, коли користувач вибирає елементи.

Додамо до попереднього прикладу наступну розмітку в контекст елемента **DockPanel**:

```
<Window x:Class="WPFTest_1.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:WPFTest_1"
        mc:Ignorable="d"
        Title="Window1" Height="300" Width="500">
    <DockPanel>
        <Menu DockPanel.Dock="Top" HorizontalAlignment = "Left" Background =
"White" BorderBrush = "Black">
            <MenuItem Header = "_Файл">
                <MenuItem Header = "_Відкрити" Click="OpenBtn_Click" />
                <MenuItem Header = "_Зберегти" Click="SaveBtn_Click" />
                <Separator />
                <MenuItem Header = "_Вихід" Click="MenuItem_Click"/>
            </MenuItem>
            <MenuItem Header = "_Довідка">
                <MenuItem x:Name="AboutMenuItem" Header = "_Про програму"
Click="AboutMenuItem_Click" />
            </MenuItem>
        </Menu>
        <StackPanel DockPanel.Dock="Bottom">
            <Button x:Name="OpenBtn" Content="Відкрити" Height="50"
Click="OpenBtn_Click"></Button>
            <Button x:Name="SaveBtn" Content="Зберегти" Height="50"
Click="SaveBtn_Click"/>
            <TextBox x:Name="text" TextWrapping="Wrap"
AcceptsReturn="True"></TextBox>
        </StackPanel>
    </DockPanel>
</Window>
```

Зверніть увагу, що меню стикуватися до верхньої частини **DockPanel**.

```
DockPanel.Dock="Top"
```

Крім того, елемент **Separator** використовується для додавання в систему меню тонкої горизонтальної лінії безпосередньо перед пунктом *Вихід*. Значення **Header** для кожного **MenuItem** містять символ підкреслення

(наприклад, *_Вихід*). Так вказується символ, який стане підкресленим, коли користувач натисне клавішу <Alt> (для введення клавіатурного скорочення).

Після побудови системи меню необхідно реалізувати різні обробники подій. Перш за все, знадобиться обробник пункту меню *Файл - Вихід*, який просто закриває вікно, що, в свою чергу, призводить до завершення програми, оскільки це вікно самого вищого рівня.

Нижче показаний код доданих обробників подій:

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
     Close();
}
private void AboutMenuItem_Click(object sender, RoutedEventArgs e)
{
     MessageBox.Show("Приклад програми");
}
```

Панелі інструментів створюються за допомогою класу **ToolBar,** і зазвичай забезпечують альтернативний спосіб активізації пунктів меню. Перемістимо раніше додаеі кнопки «Відкрити» та «Зберегти» на панель інструментів. Для цього модифікуємо код розмітки головного вікна таким чином:

```
<DockPanel>
        <Menu DockPanel.Dock="Top" HorizontalAlignment = "Left" Background =
"White" BorderBrush = "Black">
            <MenuItem Header = "_Файл">
                <MenuItem Header = "_Відкрити" Click="OpenBtn_Click" />
                <MenuItem Header = "_Зберегти" Click="SaveBtn_Click" />
                <Separator />
                <MenuItem Header = "_Вихід" Click="MenuItem_Click"/>
            </MenuItem>
            <MenuItem Header = "_Довідка">
                <MenuItem x:Name="AboutMenuItem" Header = "_Про програму"
                  Click="AboutMenuItem_Click" />
            </MenuItem>
        </Menu>
        <ToolBar DockPanel.Dock="Top">
            <Button x:Name="OpenBtn" Content="Відкрити"
                  Click="OpenBtn_Click"></Button>
            <Button x:Name="SaveBtn" Content="Зберегти"
                  Click="SaveBtn_Click"/>
        </ToolBar>
        <TextBox x:Name="text" TextWrapping="Wrap"
AcceptsReturn="True"></TextBox>
</DockPanel>
```

Елемент управління **TabControl** відображає набір пов'язаних елементів управління WPF у декільеох вкладках представлених елементами **TabItem**. Два елементи типу вкладок надаються автоматично. Для додавання додаткових вкладок потрібно просто натиснути правою кнопкою миші на вузлі **TabControl** у вікні *Document Outline* і вибрати в контекстному меню пункт *Add Tabltem*; можна також натиснути правою кнопкою миші на самому елементі **TabControl** в візуальному конструкторі і вибрати аналогічний пункт меню. Кожен елемент управління **TabItem** містить властивість **Header**, що відповідає за заголовок вкладки. При виборі вкладки для редагування ця вкладка стає активною, і її вміст можна компонувати, перетягуючи елементи управління з панелі інструментів.

Додамо до раніше створеної програми дві вкладки. У першій буде відображатися текстове поле для редагування, а у другому інформація про текст: кількість символів та кількість слів.

```
<Window x:Class="WPFTest_1.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WPFTest_1"
        mc:Ignorable="d"
        Title="Window1" Height="300" Width="500">
    <DockPanel>
        <Menu DockPanel.Dock="Top" HorizontalAlignment = "Left" Background = "White"
BorderBrush = "Black">
            <MenuItem Header = "_Файл">
                <MenuItem Header = "_Відкрити" Click="OpenBtn_Click" />
                <MenuItem Header = "_Зберегти" Click="SaveBtn_Click" />
                <Separator />
                <MenuItem Header = "_Вихід" Click="MenuItem_Click"/>
            </MenuItem>
            <MenuItem Header = "_Довідка">
                <MenuItem x:Name="AboutMenuItem" Header = "_Про програму"
Click="AboutMenuItem_Click" />
            </MenuItem>
        </Menu>
        <ToolBar DockPanel.Dock="Top">
            <Button x:Name="OpenBtn" Content="Відкрити" Click="OpenBtn_Click"></Button>
            <Button x:Name="SaveBtn" Content="Зберегти" Click="SaveBtn_Click"/>
        </ToolBar>
        <TabControl>
            <TabItem Header="Текст">
                <TextBox x:Name="text" TextWrapping="Wrap"
AcceptsReturn="True"></TextBox>
            </TabItem>
            <TabItem Header="Статистика">
                <StackPanel>
                    <Label x:Name="LettersCount" ></Label>
```

```
                    <Label x:Name="WordCount" ></Label>
                </StackPanel>
            </TabItem>
        </TabControl>
    </DockPanel>
</Window>
```

```
        private void OpenBtn_Click(object sender, RoutedEventArgs e)
        {
            var openDlg = new OpenFileDialog();
            openDlg.Filter = "Text Files | * .txt";
            if (true == openDlg.ShowDialog())
            {
                // Завантажити вміст вибраного файлу
                string dataFromFile = File.ReadAllText(openDlg.FileName);
                // Показати рядок в TextBlock
                text.Text = dataFromFile;
                LettersCount.Content = "Кількість символів:\t"+dataFromFile.Length;
                WordCount.Content = "Кількість слів:\t\t" +
                                    dataFromFile.Split(new char[] {' ', '\r', '\n'},
                                        StringSplitOptions.RemoveEmptyEntries).Length;
            }
        }
```

Зображення вікна остаточного варіанту програми показане на рис. 8.1.

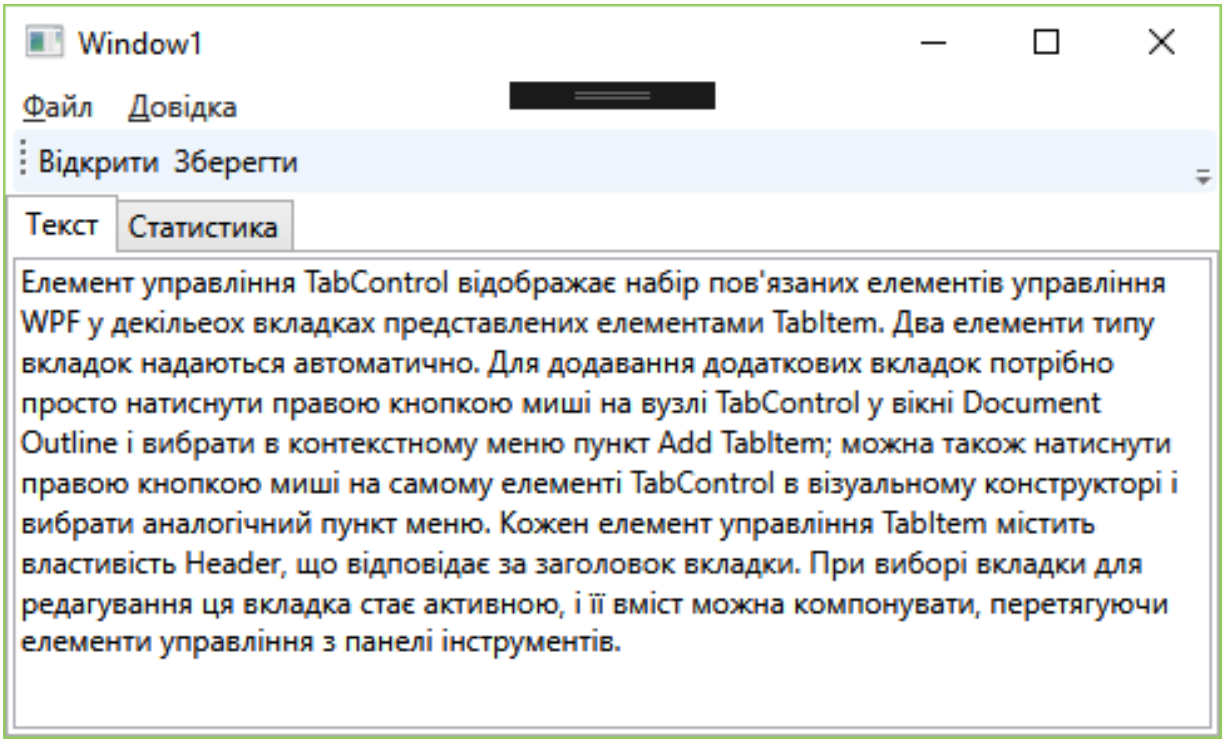

Рис. 8.1 – Зображення вікна програми

Робоче завдання

Навчитися використовувати розширені можливості створення додатків з графічним інтерфейсом, використовуючи технологію WPF.

Хід роботи

1. Модифікувати програму створену під час виконання практичної роботи № 7 додавши у головне вікно меню та панель інструментів.
2. Забезпечити можливість збереження та відкриття файлів з об’єктами класу, що містять серіалізовані дані у бінарному форматі.
3. Забезпечити можливість збереження файлів з описами об’єтів у текстовому форматі.
4. За допомогою елемента управління **TabControl** додати в головне вікно дві вкладки що містять окремо дані полів об’єкту і текстовий опис об’єкту.
5. У протоколі виконання практичного заняття навести код C# та код розмітки XAML, а також зображення вікна запущеної програми.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Опишіть призначення та основні прийоми роботи з класом **OpenFileDialog.**
2. Опишіть призначення та основні прийоми роботи з класом **SaveFileDialog.**
3. Як створити головне меню у вікні програми?
4. Як створити панель інструментів?
5. Для чого призначений елемент управління **TabControl?**
6. Як додати нові вкладки до елемента управління **TabControl?**

ПРАКТИЧНА РОБОТА №9. РОЗРОБКА ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ WPF. ЧАСТИНА 3. ЕЛЕМЕНТ DATAGRID

Мета

Познайомитися з особливостями використання елемента **DataGrid** під час розробки графічного інтерфейсу користувача з використанням технології WPF. Отримати досвід відображення та редагування колекцій об’єктів за допомогою елемента **DataGrid**.

Стислі теоретичні відомості

DataGrid – гнучкий елемент управління для відображення даних у вигляді таблиці з декількома стовпцями, що допускає сортування, редагування та інші дії.

Основні типи стовпців, що підтримуються елементом **DataGrid**:

- **DataGridTextColumn** – для відображення рядків; в звичайному режимі використовується елемент **TextBlock**, а в режимі редагування – елемент **TextBox**;
- **DataGridHyperlinkColumn** – представляє звичайний текст в вигляді гіперпосилання;
- **DataGridCheckBoxColumn** – для відображення логічних значень; використовується елемент **CheckBox**, який у позначеному стані відповідає значенню **true**, а в скинутому – значенню **false**.
- **DataGridComboBoxColumn** – для відображення перерахувань; в звичайному режимі використовується елемент **TextBlock**, а в режимі редагування - елемент **ComboBox**, що містить можливі значення.
- **DataGridTemplateColumn** – дозволяє задати довільні шаблони для відображення значення в звичайному режимі і в режимі

редагування. Робиться це за допомогою властивостей **CellTemplate** і **CellEditingTemplate**.

Елемент **DataGrid** автоматично підтримує зміну порядку і розміру стовпців і сортування за стовпцями, але будь-яку можливість можна відключити, встановивши значення **false** для таких властивостей: **CanUserReorderColumns**, **CanUserResizeColumns**, **CanllserResizeRows** і **CanUserSortColumns**. Властивості **GridLinesVisibility** і **HeadersVisibility** дозволяють відключити показ ліній сітки і заголовків відповідно.

Якщо об'єкти, які відображаються в елементі **DataGrid**, задаються за допомогою властивості **ItemsSource**, то **DataGrid** намагається автоматично згенерувати стовпці, що відповідають властивостям класу об’єктів. У такому випадку для відображення рядків вибирається стовпець типу **DataGridTextColumn**, для відображення URI – стовпець типу **DataGridHyperlinkColumn**, для відображення логічних величин – стовпець типу **DataGridCheckBoxColumn**, а для відображення перерахувань – стовпець типу **DataGridComboBoxColumn**.

Розглянемо приклад програми, в якій за допомогою елемента **DataGrid** буде відображатися колекція об’єктів класу *Lamp*, що описаний у попередніх практичних роботах. Розмітка вікна надзвичайно проста і містить тільки порожній елемент **DataGrid**:

```
    <Grid>
        <DataGrid x:Name="lampGrid">

        </DataGrid>
    </Grid>
```

Встановимо властивість ItemsSource елемента з назвою `lampGrid` в наступному коді:

```
        public Window2()
        {
            InitializeComponent();
```

```
            var lampList = new List<Lamp>();
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));

            lampGrid.ItemsSource = lampList;
        }
```

Єдиний візуальний недолік автоматичної генерації стовпців – це назви в заголовках, які збігаються з іменами відповідних властивостей. Результат показаний на рис. 9.1.

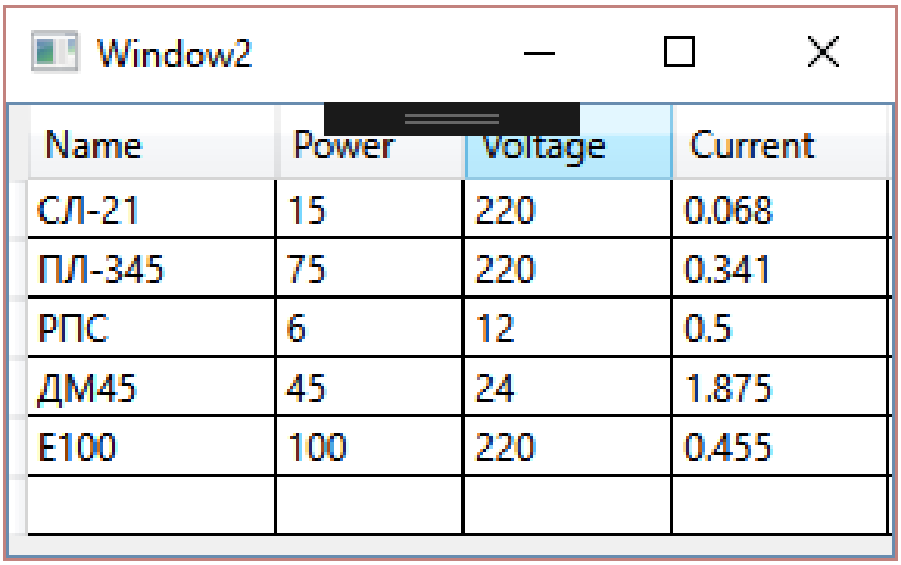

Рис. 9.1 – Елемент **DataGrid** з автоматично згенерованими стовпцями

DataGrid на рис. 9.1 автоматично підтримує редагування всіх полів кожного елемента. Варто клацнути по будь-якій клітинці, і вона автоматично перетворюється в поле введення **TextBox**. Результат будь-якого завершеного редагування відображається в колекції **ItemsSource**.

Якщо в елементі **DataGrid** присутні явно визначені стовпці, то автоматично згенеровані стовпці додаються після них. Окремі автоматично згенеровані стовпці можна налаштувати або видалити, обробивши подію **AutoGeneratingGolumn**, яке виникає один раз для кожного стовпчика. Після генерації всіх стовпців один раз виникає подія **AutoGeneratedColumns**.

При використанні властивості **ItemsSource** автоматично підтримується редагування даних в окремих об'єктах. Оскільки колекція **ItemsSource** допускає також додавання і видалення об'єктів, то автоматично підтримуються і ці операції. В сітці **DataGrid** внизу присутній порожній рядок, в який можна додати дані. У класі **DataGrid** визначені методи і команди для таких типових дій, як початок редагування (клавіша F2), скасування редагування (клавіша Esc), збереження результатів редагування (клавіша Enter) і видалення рядка (клавіша Delete).

Для запобігання редагування можна присвоїти властивості **IsReadOnly** значення **true**, а щоб заборонити додавання або видалення рядків, потрібно присвоїти значення **false** властивості **CanUserAddRows** або **CanUserDeleteRows** відповідно.

Щоб скасувати автоматичну генерацію стовпців, потрібно присвоїти властивості **AutoGenerateColumns** значення **false**.

Елемент **DataGrid** підтримує кілька моделей вибору за допомогою двох властивостей – **SelectionMode** і **SelectionUnit**. Властивості **SelectionMode** можна присвоїти значення **Single** – тоді дозволено вибирати тільки один об'єкт, або значення **Extended** – в цьому випадку можна вибирати кілька об'єктів (це стандартний режим). Визначення слова «об'єкт» залежить від значення властивості **SelectionUnit**:

- **Cell** – дозволено вибирати тільки окремі клітинки;
- **FullRow** – дозволено вибирати тільки цілі рядки;
- **CellOrRowHeader** – дозволено вибирати і те і інше (для вибору всього рядку слід клацнути по його заголовку).

У режимі вибору декількох об'єктів натискання клавіші Shift дозволяє вибирати сусідні об'єкти, а, натискання клавіші Ctrl – довільно розташовані об'єкти.

При виборі рядків генерується подія **Selected**, а властивість **SelectedItems** містить колекцію обраних об'єктів. При виборі окремих клітинок генерується подія **SelectedCellChanged**, а властивість **SelectedCells** містить інформацію про відповідні рядки і стовпці.

Клас **DataGrid** підтримує і інші дії, наприклад, взаємодію з буфером обміну та можливість «заморожувати» стовпці.

Взаємодія з буфером обміну. Налаштувати, які саме дані копіюються з **DataGrid** в буфер обміну (наприклад, при натисканні Ctrl + C після вибору об'єктів), дозволяє властивість **ClipboardCopyMode**. Вона може приймати наступні значення:

- **ExcludeHeader** – не включати заголовки стовпців в текст, що копіюється. Це режим за замовчуванням;
- **IncludeHeader** – включати заголовки стовпців в текст, що копіюється;
- **None** – нічого не копіювати в буфер обміну.

Заморожування стовпців. Елемент **DataGrid** дозволяє заморозити будь-яке число стовпців. Це означає, що вони не будуть висунуті за межі області елемента при горизонтальній прокрутці. Приблизно так само заморозка стовпців працює в Microsoft Excel. Щоб заморозити один або декілька стовпців, досить присвоїти властивості **FrozenColumnCount** будь-яке значення, більше 0.

Робоче завдання

Навчитися використовувати елемент **DataGrid** для відображення та редагування колекцій об’єктів.

Хід роботи

1. Модифікувати програму створену під час виконання практичної роботи № 8 додавши в елемент управління **TabControl** нову вкладку що містить елемент **DataGrid**.
2. Створити та заповнити список об’єктів класу, описаного в попередніх роботах.
3. Зв’язати список з елементом **DataGrid** за допомогою властивості **ItemsSource**.
4. Перевірити можливість редагування значень властивостей об’єктів, а також додавання та видалення об’єктів.
5. Доповнити головне меню програми пунктами, що забезпечують можливість зберігати список у форматі XML.
6. У протоколі виконання практичного заняття навести код C# та код розмітки XAML, а також зображення вікна запущеної програми.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Опишіть призначення та основні можливості елемента **DataGrid.**
2. Які основні типи стовпців підтримуються елементом **DataGrid**?
3. Для чого призначена властивість **ItemsSource**?
4. Як відбувається автоматична генерація стовпців?
5. Які обмеження є в автоматичної генерації стовпців?
6. Як відбувається додавання та видалення елементів таблиці?
7. Як відбувається вибір рядків та комірок таблиці**?**
8. Як скопіювати вміст комірок чи рядків у буфер обміну**?**
9. Як «заморозити» декілька стовпців таблиці?

ПРАКТИЧНА РОБОТА №10. РОЗРОБКА ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ WPF. ЧАСТИНА 4. ПОБУДОВА ГРАФІКІВ ЗА ДОПОМОГОЮ БІБЛІОТЕКИ КЛАСІВ OXYPLOT

Мета

Познайомитися з особливостями використання сторонніх бібліотек класів у додатках написаних мовою програмування C#. Отримати досвід відображення графіків за допомогою бібліотеки класів OxyPlot.

Стислі теоретичні відомості

В додатках написаних мовою C# є можливість використовувати сторонні класи зібрані у скомпільовані бібліотеки класів.

Існує дуже велика кількість готових бібліотек з відкритим вихідним кодом, що написані програмістами зі всього світу, і які призначені для вирішення найрізноманітніших задач. Ці бібліотеки вільно поширюються в мережі Internet і найзручнішим засобом для їх використання є пакетний менеджер NuGet.

Для використання класів необхідно перш за все додати у проект посилання на збірку, яка містить переносну бібліотеку класів. Якщо бібліотеку додавати через менеджер NuGet то практично всі дії виконуються автоматично. Для цього можна, наприклад, в браузері рішень клацнути по пункту «References» правою кнопкою по назві проекту і вибрати пункт меню «Manage NuGet Packages...».

Далі потрібно в пошуковому полі ввести ім’я бібліотеки або ключові слова для пошуку і з отриманих результатів вибрати бажаний. Додавання бібліотеки відбувається після натискання кнопки «Install».

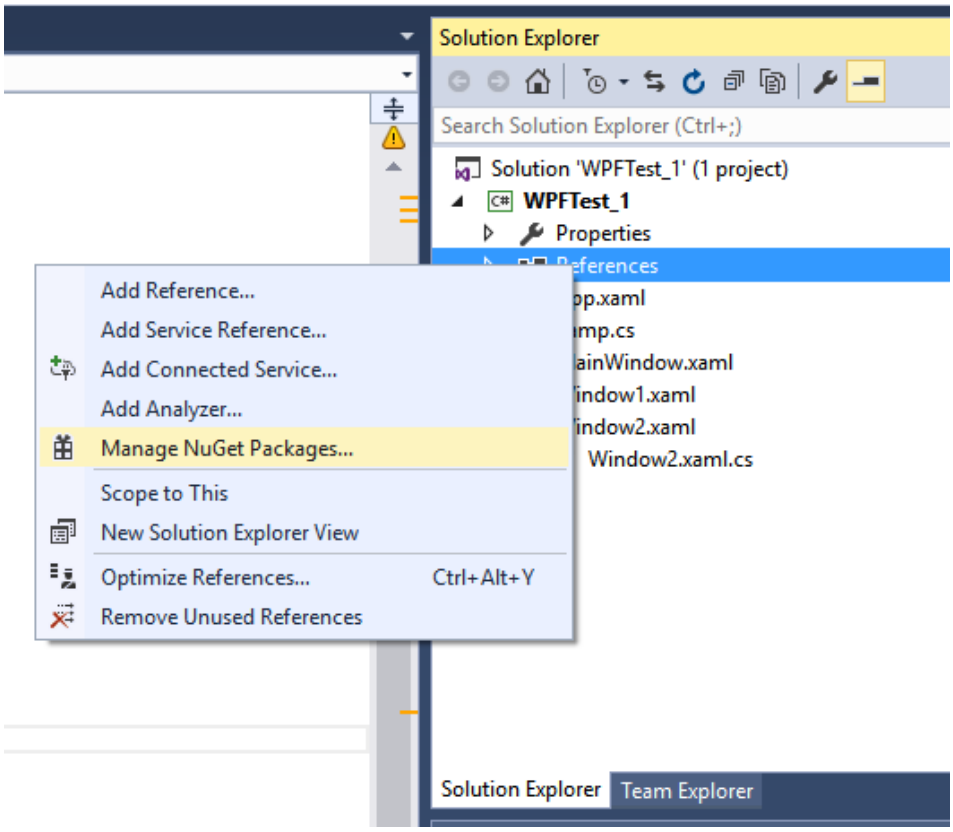

Рис. 10.1 – Додавання бібліотеки класів через менеджер NuGet

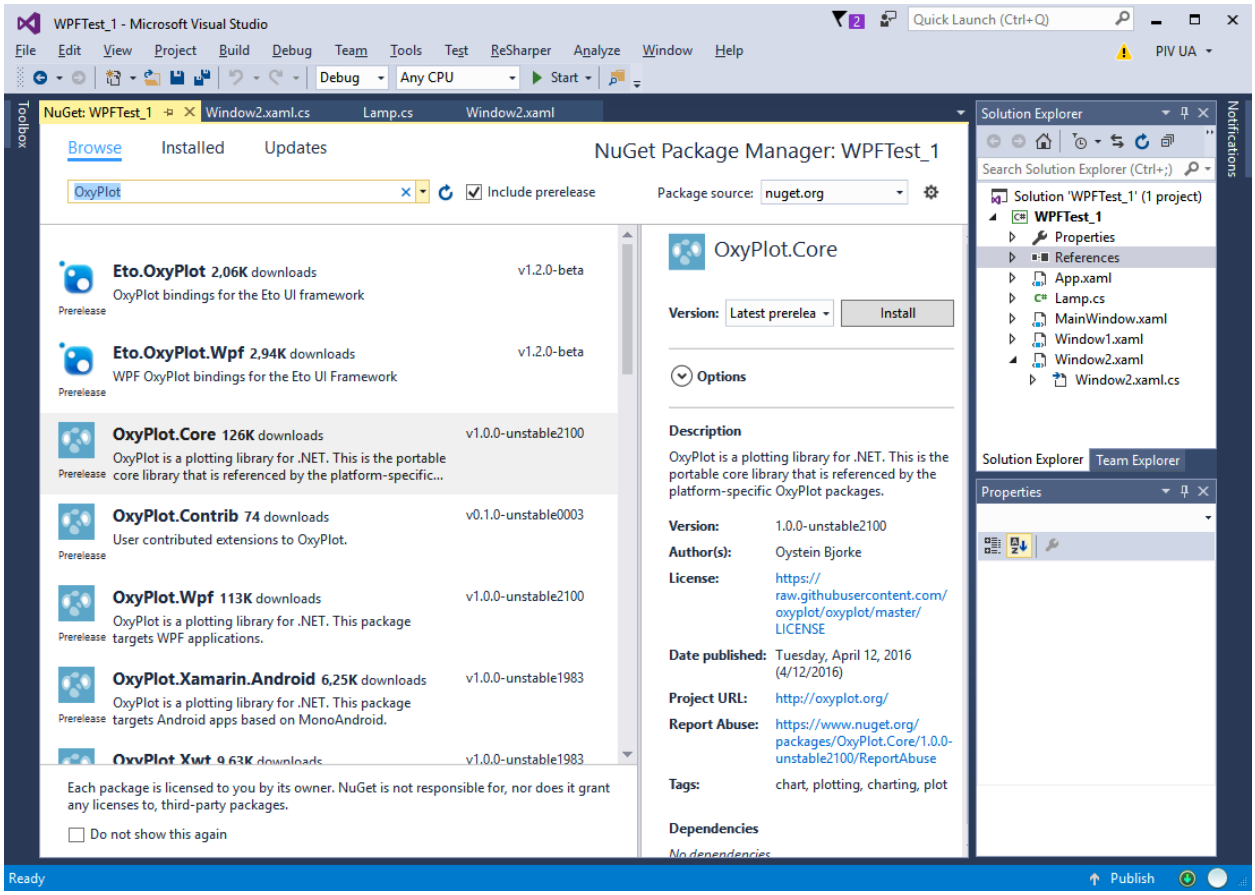

Рис. 10.2 – Пошук бібліотеки класів через менеджер NuGet

Для додавання бібліотеки вручну, спершу викличемо контекстне меню на пункті «References», і виберемо в ньому «Add Reference…», як показано на малюнку нижче.

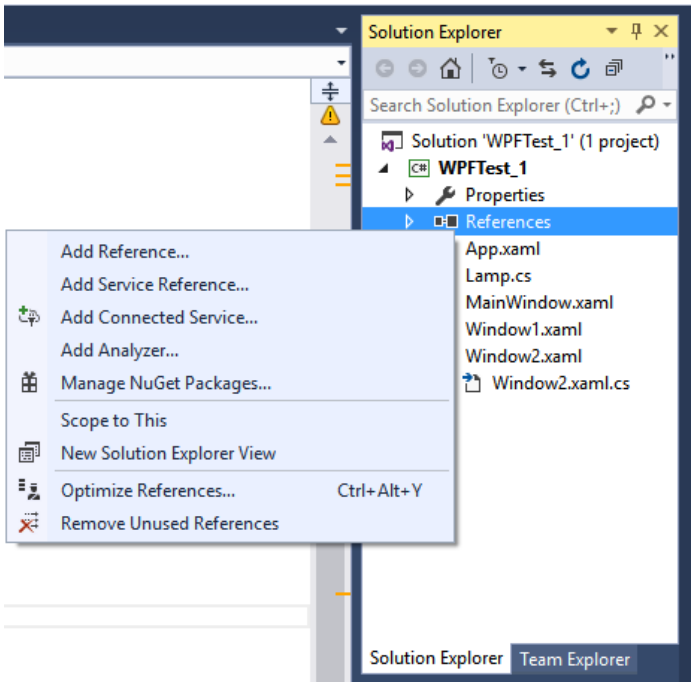

Рис. 10.3 – Додавання бібліотеки класів

У вікні в лівій області вибираємо пункт «Browse» і в низу вікна, натискаємо на кнопку «Browse…», як показано на малюнку нижче.

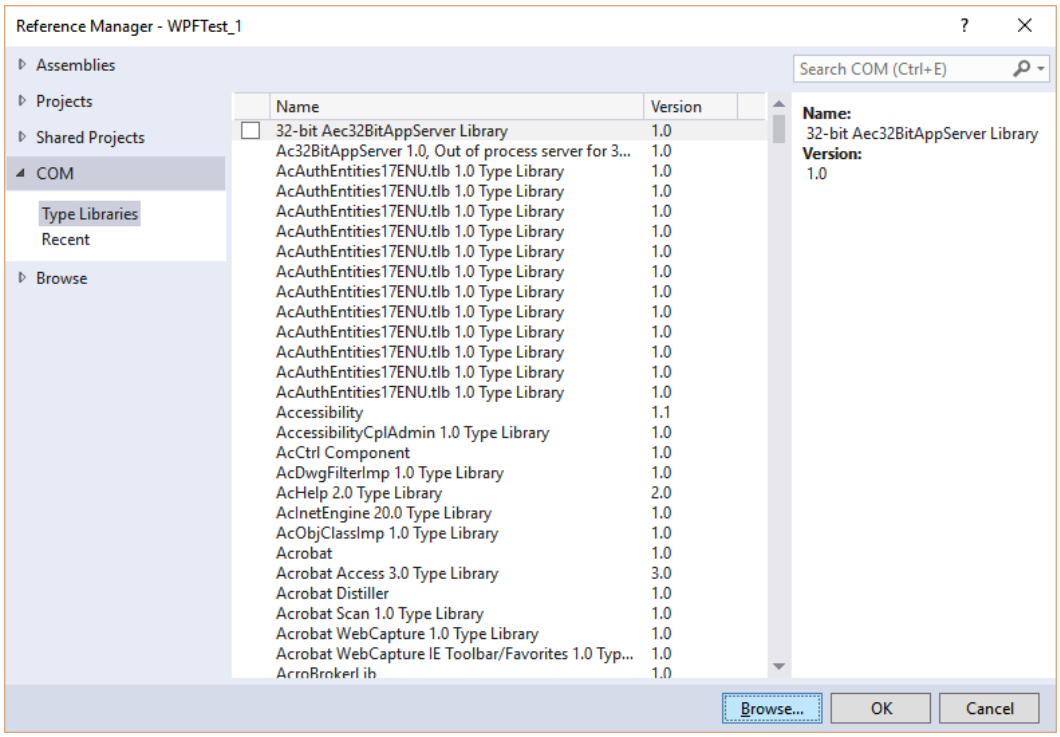

Рис. 10.3 – Додавання бібліотеки класів

У вікні потрібно перейти в папку, в яке лежить бібліотека (DLL), вибрати цю бібліотеку і натиснути на кнопку «Додати». Після чого, натиснути на кнопку «ОК» в попередньому вікні. В результаті, список посилань проекту, буде поповнено ще однією бібліотекою. Тепер, ми можемо використовувати в програмі класи з підключеної бібліотеки. Проте перед цим потрібно підключити простір імен. Для цього, додаємо рядок

```
using <назва_простору_імен>;
```

в кінець блоку директив **using**, який розташований в самому початку основного файлу проекту.

Використання бібліотек класів розглянемо на прикладі бібліотеки для побудов графіків **OxyPlot**. **OxyPlot** містить багато різних типів графіків, осей, рядів тощо. Всі побудовані графіки можуть бути легко експортовані у різні формати файлів, такі як PNG, PDF і SVG.

Для створення графіку у вікні додатку потрібно додати простір імен і відповідний компонент WPF, який називається **PlotView** (виділено напівжирним шрифтом):

```
<Window x:Class="WPFTest_1.Window3"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:local="clr-namespace:WPFTest_1"

        xmlns:oxy=http://oxyplot.org/wpf

        mc:Ignorable="d"
        Title="Window3" Height="300" Width="300">
    <Grid>
        <oxy:PlotView x:Name="LampPlot" />
    </Grid>
</Window>
```

У коді програми потрібно створити об’єкт класу **PlotModel** та присвоїти його властивості **Model** компонента **PlotView**:

```
      var LampModel = new PlotModel();
      LampPlot.Model = LampModel;
```

Подальша робота ведеться з об’єктом класу **PlotModel** через відповідні властивості та методи. Наприклад, назву графіку можна задати через властивість **Title**. Ряди даних зберігаються в колекції **Series** і можуть бути додані через метод **Add**(). Для прикладу, на основі раніше створеного списку побудуємо залежність струму від потужності для різних типів ламп:

```
      var LampModel = new PlotModel();
      LampPlot.Model = LampModel;

      LampModel.Title = "Залежність струму від потужності для різних
типів ламп";

      var lampSeries = new LineSeries();
      foreach (var lamp in lampList)
      {
       lampSeries.Points.Add(new DataPoint(lamp.Power, lamp.Current));
      }
      LampModel.Series.Add(lampSeries);
```

Результат зображено на рис. 10.4.

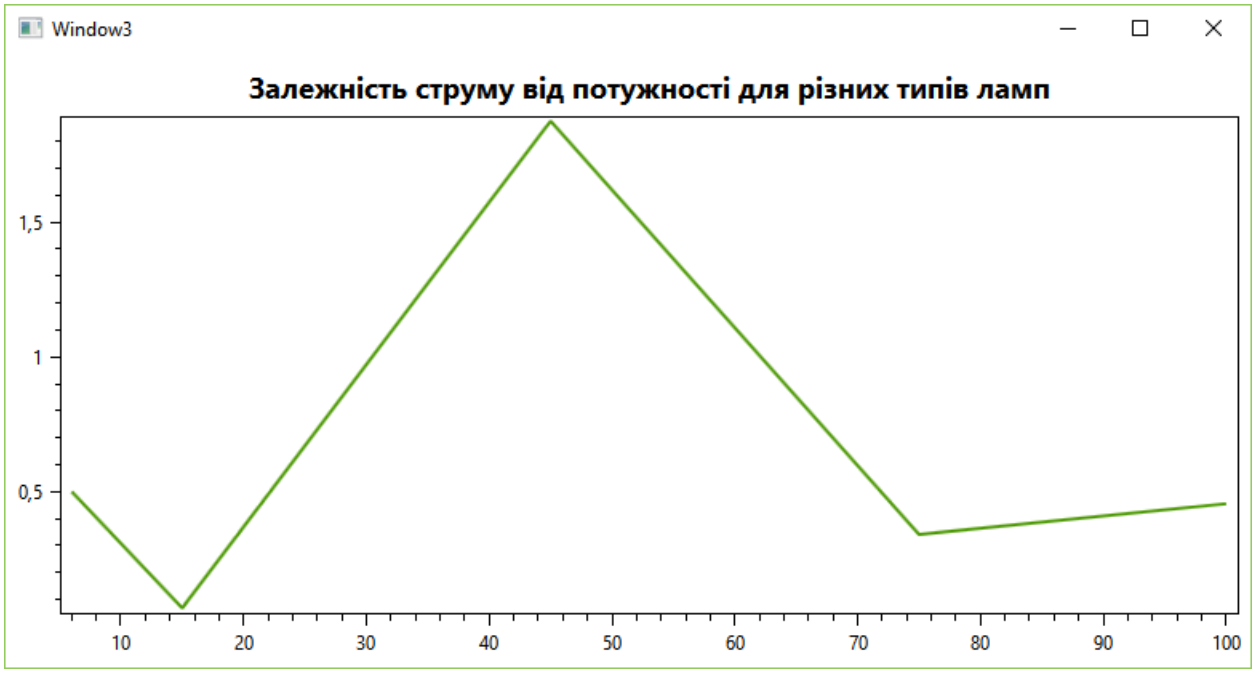

Рис. 10.4 – Зображення вікна з графіком

Характеристики осей можуть бути задані через властивість **Axes** класу **PlotModel.** Додамо підписи осей:

```
            LampModel.Axes.Add(new LinearAxis
            {
                Position = AxisPosition.Bottom,
                Minimum = 0,
                Maximum = 120,
                Title = "Рном, Вт"
            });
            LampModel.Axes.Add(new LinearAxis
            {
                Position = AxisPosition.Left,
                Minimum = 0,
                Maximum = 3,
                Title = "I, A"
            });
```

Результат зображено на рис. 10.5.

Рис. 10.5 – Зображення вікна з графіком

Зберегти зображення графіка у файл можна за допомогою методу класу **PlotView**:

```
SaveBitmap( назва_файлу , ширина , висота , колір );
```

Наприклад:

```
LampPlot.SaveBitmap("plot1.png",700,300,OxyColors.Transparent);
```

Робоче завдання

Навчитися використовувати бібліотеку класів **OxyPlot** для побудови графіків.

Хід роботи

1. Модифікувати програму створену під час виконання практичної роботи № 9 додавши в проект посилання на бібліотеку **OxyPlot.**
2. Додати в елемент управління **TabControl** нову вкладку що містить елемент **PlotView**.
3. Створити об’єкт класу **PlotModel** та присвоїти його властивості **Model** компонента **PlotView**.
4. Створити новий ряд даних, який представлятиме залежність між двома властивостями на вибір створеного у попередніх роботах класу.
5. Додати створений ряд даних на графік.
6. Задати властивості осей графіка.
7. Створити новий ряд даних з довільними даними на вибір і додати його на графік.
8. На вибір змінити декілька властивостей рядів та осей, таких як колір, шрифт, розмір, товщина тощо.
6. Доповнити головне меню програми пунктами, що забезпечують можливість зберігати графік у файл формату PNG.
9. У протоколі виконання практичного заняття навести код C# та код розмітки XAML, а також зображення вікна запущеної програми.

Індивідуальне завдання

Відповідно до завдання практичної роботи №1.

Контрольні питання

1. Що таке бібліотека класів?
2. Як додати посилання на бібліотеку у проект?
3. Як використовувати класи, що містяться в бібліотеці?

4. Для чого призначена бібліотека класів **OxyPlot**?
5. Як додати графік на вікно додатку?
6. Як створити новий ряд даних?
7. Які властивості ряду можна змінити**?**
8. Як задати властивості осей графіка**?**
9. Як зберегти графік у файл?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Голуб Б.М. C#. Концепція та синтаксис. Навч. посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.
2. Троелсен Э. Язык программирования C# 5.0 и платформа .NET 4.5, 6-е изд.: Пер. с англ. – М.: ООО “И.Д. Вильямс”, 2013. – 1312 с.
3. Шилдт Г. C# 3.0. Полное руководство / Пер. с англ. – М.: Диалектика-Вильямс, 2009. – 992 с.
4. Котов, О.М. Язык C#: краткое описание и введение в технологии программирования: учебное пособие / О.М. Котов. – Екатеринбург: Изд-во Урал. ун-та, 2014. – 208 с.
5. Уотсон К. Microsoft Visual C# 2008. Базовый курс / К. Уотсон, К. Нейгел, Я.Х. Педерсен, Дж. Д. Рид, М. Скиннер, Э. Уайт. / Пер. с англ. – М.: Диалектика-Вильямс, 2009. – 1216 с.

Додаткова література:

1. Павловская Т.А. C#. Программирование на языке высокого уровня: Учебник для вузов. – СПб.: Питер, 2014. – 432 с.
2. Нейгел К., Ивьен Б., Глинн Д. C# 4.0 и платформа .NET 4 для профессионалов / Пер. с англ. – К.: Диалектика, 2011. – 1440 с.
3. Рихтер Дж. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. / Пер. с англ. – СПб: Питер, М: Русская Редакция, 2007. – 656 с.
4. Агапов В.П. Основы программирования на языке C#: учебное пособие / В. П. Агапов. – Москва: МГСУ, 2012. – 128 с.
5. Петцольдт Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. /Пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002. – 576 с.
6. Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М. Объектно-ориентированное программирование на C#: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский (Приволжский) федеральный университет, 2012. – 134 с.
7. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.
8. Культин Н. Б. C# в задачах и примерах. – СПб.: БХВ-Петербург, 2007. – 240 с.
9. Ватсон К. C# / К. Ватсон, М. Беллиназо, О. Корне, Д. Эспиноза, 3. Гринфосс и др. / Пер. с англ. яз. – М.: Изд. «Лори». – 2005, 862 с.
10. Бокс Д., Селлз К. Основы платформы .NET, том 1. Общеязыковая исполняющая среда.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 288 с.
11. Либерти Дж., Программирование на C#. Создание .NET-приложений. Изд. 2-е. / Пер. с англ. – Спб.: Изд. «Символ», 2003. – 688 с.

Звіт про виконання практичної роботи з дисципліни
«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ»

Студент: П.І.Б.	**Група:** ХХ-ХХ	**ст. 88**
Практична робота № Х	**Варіант:** ХХ	

Тема

Мета

Індивідуальне завдання

Текст програми

Результати виконання програми

Тест № 1

Вхідні дані

…….

Вихідні дані

…….

…….

Тест № N

Вхідні дані

…….

Вихідні дані

…….

Дата виконання:_____ **Дата захисту:**_____

Оцінка:_____

Викладач:_____ **Підпис:**_____