

РІВНЕНСЬКИЙ ДЕРЖАВНИЙ ГУМАНІТАРНИЙ  
УНІВЕРСИТЕТ

Кафедра вищої математики

# C# - творчість програмування

Том 1.  
Об'єктно-орієнтоване програмування.

Підручник

*Під редакцією Бичкова О.С.*

Авторський колектив:  
Іванов Є.В., Жабська Є.О, Петрівський В.Я.,  
Меркулова К.В., Шатирко А.В., Порєв Г.В.

Рівне – 2024

---

УДК 004.432.2(075.8)  
С-11

Рецензенти:

доктор технічних наук, професор Вишнівський В.В.

доктор технічних наук, професор Барабаш О.В.

доктор педагогічних наук, професор Войтович І.С.

Рекомендовано до публікації Вченою радою Рівненського  
державного гуманітарного університету, протокол № 3 від «28»  
березня 2024 р.

С# - творчість програмування. **Том 1.** Об'єктно-орієнтоване програмування: підручник \\ **Під ред. Бичкова О.С.** Авторський колектив: Іванов Є.В., Жабська Є.О, Петрівський В.Я., Меркулова К.В., Шатирко А.В., Порєв Г.В.: **Волин. обереги, 2024.**—292 с.

**ISBN 978-617-8161-12-3**

Розглянуто питання складності алгоритмів, типізації даних, подання даних у комп'ютері, машинної точності обчислень, оптимальної реалізації обчислень. Матеріал викладається мовою С#, наведено основний синтаксис мови. Розглянуто та наведено приклади коректного використання парадигм об'єктно-орієнтованого програмування. Викладання матеріалу ведеться на великій кількості прикладів. Підручник дозволить навчитися програмувати всім бажаючим, але буде особливо корисним для студентів всіх спеціальностей галузі знань «12—Інформаційні технології».

УДК 004.432.2(075.8)  
С-11

**ISBN 978-617-8161-12-3**

©Бичков О.С., 2024  
© «Волинські обереги», 2024

# Зміст

|          |                                                     |           |
|----------|-----------------------------------------------------|-----------|
| <b>1</b> | <b>Вступ</b>                                        | <b>9</b>  |
| 1.1      | Перша програма . . . . .                            | 9         |
| 1.1.1    | Контрольні запитання . . . . .                      | 13        |
| 1.2      | Об'єкти програми . . . . .                          | 14        |
| 1.2.1    | Контрольні запитання . . . . .                      | 14        |
| 1.3      | Типізація . . . . .                                 | 14        |
| 1.3.1    | Контрольні запитання . . . . .                      | 16        |
| 1.4      | Змінна . . . . .                                    | 17        |
| 1.4.1    | Контрольні запитання . . . . .                      | 18        |
| 1.5      | Ключове слово var . . . . .                         | 19        |
| 1.5.1    | Контрольні запитання . . . . .                      | 19        |
| 1.6      | Константи . . . . .                                 | 19        |
| 1.6.1    | Контрольні запитання . . . . .                      | 20        |
| 1.7      | Програмування з використанням елементів ООП . . .   | 21        |
| 1.7.1    | Контрольні запитання . . . . .                      | 26        |
| 1.8      | Атрибути private, public, protected . . . . .       | 26        |
| 1.8.1    | Контрольні запитання . . . . .                      | 27        |
| 1.9      | Конструктор . . . . .                               | 28        |
| 1.9.1    | Контрольні запитання . . . . .                      | 29        |
| 1.10     | Задачі до розділу ВСТУП . . . . .                   | 29        |
| <b>2</b> | <b>Основи алгоритмізації</b>                        | <b>31</b> |
| 2.0.1    | Контрольні запитання . . . . .                      | 33        |
| 2.1      | Класифікація внутрішніх структур алгоритмів . . . . | 33        |
| 2.1.1    | Контрольні запитання . . . . .                      | 36        |
| 2.2      | Складність алгоритмів . . . . .                     | 36        |
| 2.2.1    | Контрольні запитання . . . . .                      | 40        |
| 2.3      | Складність задач . . . . .                          | 40        |
| 2.3.1    | Контрольні запитання . . . . .                      | 47        |
| 2.4      | Задачі до розділу ОСНОВИ АЛГОРИТМІЗАЦІЇ . . .       | 48        |
| <b>3</b> | <b>Оператори мови C#</b>                            | <b>49</b> |
| 3.1      | Прості оператори . . . . .                          | 49        |
| 3.1.1    | Порожній оператор . . . . .                         | 49        |

|          |                                                                      |           |
|----------|----------------------------------------------------------------------|-----------|
| 3.1.2    | Складений оператор . . . . .                                         | 49        |
| 3.1.3    | Контрольні запитання . . . . .                                       | 50        |
| 3.2      | Умовний оператор . . . . .                                           | 50        |
| 3.2.1    | Контрольні запитання . . . . .                                       | 52        |
| 3.3      | Тернарний оператор . . . . .                                         | 53        |
| 3.3.1    | Контрольні запитання . . . . .                                       | 53        |
| 3.4      | Оператор-перемикач switch . . . . .                                  | 53        |
| 3.4.1    | Контрольні запитання . . . . .                                       | 55        |
| 3.5      | Оператор циклу for . . . . .                                         | 56        |
| 3.5.1    | Контрольні запитання . . . . .                                       | 58        |
| 3.6      | Оператор foreach . . . . .                                           | 59        |
| 3.6.1    | Контрольні запитання . . . . .                                       | 59        |
| 3.7      | Оператори do-while, while . . . . .                                  | 59        |
| 3.7.1    | Контрольні запитання . . . . .                                       | 61        |
| 3.8      | Оператор continue . . . . .                                          | 61        |
| 3.8.1    | Контрольні запитання . . . . .                                       | 61        |
| 3.9      | Оператор break . . . . .                                             | 62        |
| 3.9.1    | Контрольні запитання . . . . .                                       | 63        |
| 3.10     | Оператор goto . . . . .                                              | 63        |
| 3.10.1   | Контрольні запитання . . . . .                                       | 63        |
| 3.11     | Оператор return . . . . .                                            | 64        |
| 3.11.1   | Контрольні запитання . . . . .                                       | 64        |
| 3.12     | Задачі до розділу ОПЕРАТОРИ МОВИ C# . . . . .                        | 64        |
| <b>4</b> | <b>Булеві функції та вирази</b>                                      | <b>65</b> |
| 4.1      | Булеві функції . . . . .                                             | 65        |
| 4.1.1    | Контрольні запитання . . . . .                                       | 67        |
| 4.2      | Булеві вирази . . . . .                                              | 68        |
| 4.2.1    | Контрольні запитання . . . . .                                       | 73        |
| 4.3      | Задачі для розділу БУЛЕВІ ФУНКЦІЇ ТА ВИРАЗИ .                        | 73        |
| <b>5</b> | <b>Зображення чисел у комп'ютері</b>                                 | <b>75</b> |
| 5.1      | Системи числення . . . . .                                           | 75        |
| 5.1.1    | Контрольні запитання . . . . .                                       | 77        |
| 5.2      | Правила переведення чисел з однієї системи числення в іншу . . . . . | 78        |
| 5.2.1    | Контрольні запитання . . . . .                                       | 80        |
| 5.3      | Правило визначення точності зображення . . . . .                     | 80        |
| 5.3.1    | Контрольні запитання . . . . .                                       | 83        |
| 5.4      | Двійкова арифметика . . . . .                                        | 83        |
| 5.4.1    | Додавання двійкових чисел . . . . .                                  | 83        |
| 5.4.2    | Зображення від'ємних чисел . . . . .                                 | 84        |

---

|          |                                                                                               |            |
|----------|-----------------------------------------------------------------------------------------------|------------|
| 5.4.3    | Віднімання двійкових чисел . . . . .                                                          | 87         |
| 5.4.4    | Множення двійкових чисел . . . . .                                                            | 89         |
| 5.4.5    | Ділення двійкових чисел . . . . .                                                             | 89         |
| 5.4.6    | Контрольні запитання . . . . .                                                                | 90         |
| 5.5      | Ознака переповнення розрядної сітки при арифметичних операціях . . . . .                      | 91         |
| 5.5.1    | Контрольні запитання . . . . .                                                                | 91         |
| 5.6      | Зображення цілих чисел . . . . .                                                              | 91         |
| 5.6.1    | Контрольні запитання . . . . .                                                                | 92         |
| 5.7      | Зображення дійсних чисел . . . . .                                                            | 93         |
| 5.7.1    | Контрольні запитання . . . . .                                                                | 98         |
| 5.8      | Керування машинним зображенням чисел та особливості виконання арифметичних операцій . . . . . | 99         |
| 5.8.1    | Контрольні запитання . . . . .                                                                | 102        |
| 5.9      | Завдання для самостійної роботи . . . . .                                                     | 102        |
| <b>6</b> | <b>Класи</b>                                                                                  | <b>105</b> |
| 6.1      | Класи у C# . . . . .                                                                          | 105        |
| 6.1.1    | Поля та константи класу . . . . .                                                             | 105        |
| 6.1.2    | Модифікатор static . . . . .                                                                  | 107        |
| 6.1.3    | Контрольні запитання . . . . .                                                                | 108        |
| 6.2      | Правила доступу . . . . .                                                                     | 108        |
| 6.2.1    | Контрольні запитання . . . . .                                                                | 111        |
| 6.3      | Конструктори класів . . . . .                                                                 | 111        |
| 6.3.1    | Конструктори екземплярів класів . . . . .                                                     | 111        |
| 6.3.2    | Статичний конструктор . . . . .                                                               | 114        |
| 6.3.3    | Порядок ініціалізації полів та виклик конструкторів . . . . .                                 | 117        |
| 6.3.4    | Контрольні запитання . . . . .                                                                | 120        |
| 6.4      | Властивості . . . . .                                                                         | 121        |
| 6.4.1    | Контрольні запитання . . . . .                                                                | 125        |
| 6.5      | Індексатори . . . . .                                                                         | 125        |
| 6.5.1    | Контрольні запитання . . . . .                                                                | 129        |
| 6.6      | Інкапсуляція . . . . .                                                                        | 130        |
| 6.6.1    | Контрольні запитання . . . . .                                                                | 134        |
| 6.7      | Спадкування класів . . . . .                                                                  | 134        |
| 6.7.1    | Спадкування й проблеми доступу . . . . .                                                      | 135        |
| 6.7.2    | Припинення спадкування. . . . .                                                               | 136        |
| 6.7.3    | Контрольні запитання . . . . .                                                                | 136        |

---

|        |                                                                           |     |
|--------|---------------------------------------------------------------------------|-----|
| 6.8    | Поліморфізм . . . . .                                                     | 136 |
| 6.8.1  | Підтиповий поліморфізм: перевизначення методів . . . . .                  | 137 |
| 6.8.2  | Спеціальний (ad-hoc) поліморфізм: перевантаження операторів . . . . .     | 144 |
| 6.8.3  | Перевантаження операторів порівняння . . . . .                            | 148 |
| 6.8.4  | Контрольні запитання . . . . .                                            | 154 |
| 6.9    | Деякі особливості поліморфізму методів . . . . .                          | 154 |
| 6.9.1  | Контрольні запитання . . . . .                                            | 158 |
| 6.10   | Параметричний поліморфізм . . . . .                                       | 158 |
| 6.10.1 | Контрольні запитання . . . . .                                            | 159 |
| 6.11   | Властивості статичних класів та статичних членів класів . . . . .         | 160 |
| 6.11.1 | Контрольні запитання . . . . .                                            | 160 |
| 6.12   | Узагальнені типи та узагальнені методи . . . . .                          | 160 |
| 6.12.1 | Контрольні запитання . . . . .                                            | 164 |
| 6.13   | Перетворення типів . . . . .                                              | 164 |
| 6.13.1 | Контрольні запитання . . . . .                                            | 167 |
| 6.14   | Приховування та перевизначення . . . . .                                  | 167 |
| 6.14.1 | Контрольні запитання . . . . .                                            | 169 |
| 6.15   | Абстрактні класи, методи, властивості . . . . .                           | 169 |
| 6.15.1 | Контрольні запитання . . . . .                                            | 174 |
| 6.16   | Інтерфейси . . . . .                                                      | 174 |
| 6.16.1 | Контрольні запитання . . . . .                                            | 179 |
| 6.17   | Комбінований інтерфейс . . . . .                                          | 179 |
| 6.17.1 | Використання властивостей у інтерфейсах . . . . .                         | 182 |
| 6.17.2 | Використання індексаторів у інтерфейсах . . . . .                         | 185 |
| 6.17.3 | Контрольні запитання . . . . .                                            | 187 |
| 6.18   | Інтерфейси. Більш детально та глибоко. . . . .                            | 187 |
| 6.18.1 | Контрольні запитання . . . . .                                            | 200 |
| 6.19   | Абстрактні класи, Інтерфейси - крок до уніфікації . . . . .               | 200 |
| 6.19.1 | Контрольні запитання . . . . .                                            | 213 |
| 6.20   | Класи, абстрактні класи, інтерфейси. Спадкування чи реалізація? . . . . . | 213 |
| 6.20.1 | Контрольне запитання . . . . .                                            | 225 |
| 6.21   | Множинне спадкування чи реалізація інтерфейсів? . . . . .                 | 225 |
| 6.21.1 | Контрольні запитання . . . . .                                            | 228 |
| 6.22   | Делегати, анонімні методи й лямбда-вирази . . . . .                       | 228 |
| 6.22.1 | Делегати . . . . .                                                        | 228 |
| 6.22.2 | Анонімні методи . . . . .                                                 | 236 |
| 6.22.3 | Лямбда-вирази . . . . .                                                   | 238 |

---

|          |                                                     |            |
|----------|-----------------------------------------------------|------------|
| 6.22.4   | Контрольні запитання . . . . .                      | 239        |
| 6.23     | Задачі до розділу КЛАСИ . . . . .                   | 239        |
| 6.24     | Взаємозв'язки та взаємодія класів . . . . .         | 240        |
| 6.24.1   | Контрольні запитання . . . . .                      | 253        |
| 6.25     | Задачі до розділу КЛАСИ . . . . .                   | 253        |
| <b>7</b> | <b>Вправи</b>                                       | <b>256</b> |
| 7.1      | Вступ до програмування . . . . .                    | 256        |
| 7.2      | Алгоритми . . . . .                                 | 261        |
| 7.3      | Комплексна вправа до розділів 1-6 . . . . .         | 265        |
| 7.4      | Вправи з об'єктно-орієнтованого програмування . . . | 270        |
| 7.5      | Комплексна вправа до розділу 7 . . . . .            | 273        |
| <b>8</b> | <b>Відомості про авторів</b>                        | <b>287</b> |
|          | <b>Список літератури</b>                            | <b>290</b> |

# Передмова

Колективом авторів до вашого вивчення пропонується перший том серії **C# – творчість програмування**. Том має підназву “Об’єктно-орієнтоване програмування”. Цей том є логічним продовженням підручника [1]. У запропонованому підручнику викладено основи проектування програмного забезпечення. Матеріал базується на мові C#. Деякі розділи мають універсальний характер, їхнє засвоєння дозволить читачу надалі ефективно використовувати інші мови програмування. Це розділи, присвячені складності алгоритмів, типізації даних, поданню даних у комп’ютері (цікавий факт невиконання асоціативного закону арифметики), машинному нулю, машинній точності обчислень, програмістським умовам зупинки обчислення нескінченних сум і добутків, оптимальній реалізації обчислень, автоматній технології програмування. Наведено також алгоритми обробки інформації на різних структурах даних: простих, структурованих, усіляких видах зв’язних списків. Підручник дозволить навчитися програмувати всім, але особливо корисний для студентів усіх спеціальностей галузі знань “12 — інформаційні технології”.

Колектив авторів планує написати ще 3 томи: “Алгоритми”, “Проєктне навчання”, “Архітектура програмного забезпечення”.

Творчість - це продукт мозку людини, ремесло - це продукт рук людини. Є програмісти ремісники, а є творці. Для останніх – ця книга. А є ще наука....

О.С.Бичков

Автори висловлюють щиру подяку рецензентам за корисні поради, а також директору ТОВ «ДОК ФМ» Деркачу Володимирі Івановичу за сприяння у випуску підручника та підтримку науковців.

# 1 Вступ

## 1.1 Перша програма

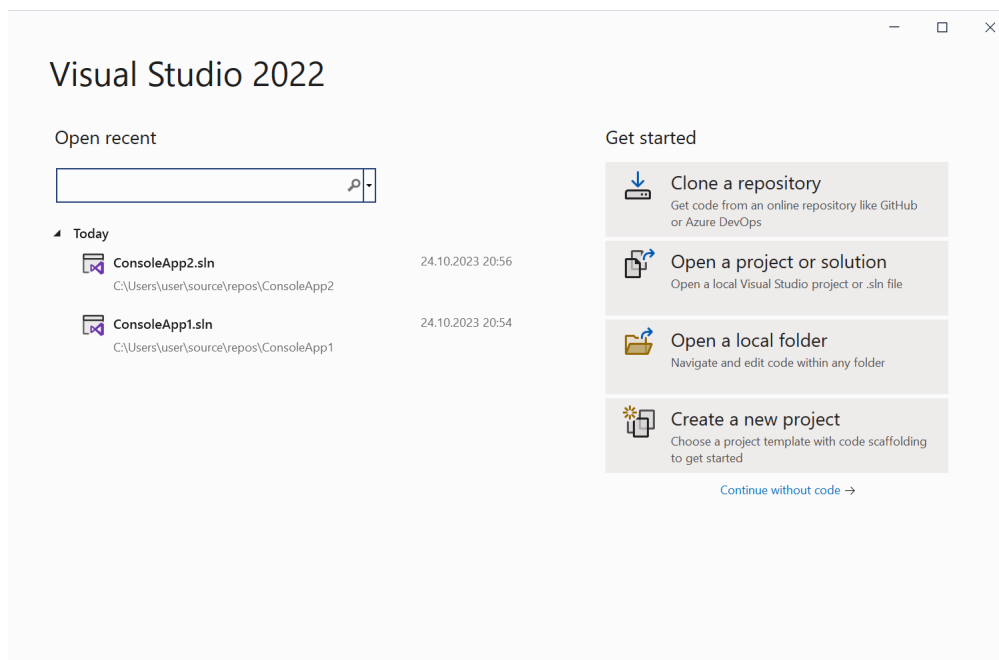
Ми пропонуємо Вам відразу приступити до програмування.

Але для початку нехай нам підкаже середовище Microsoft Visual Studio 2022.

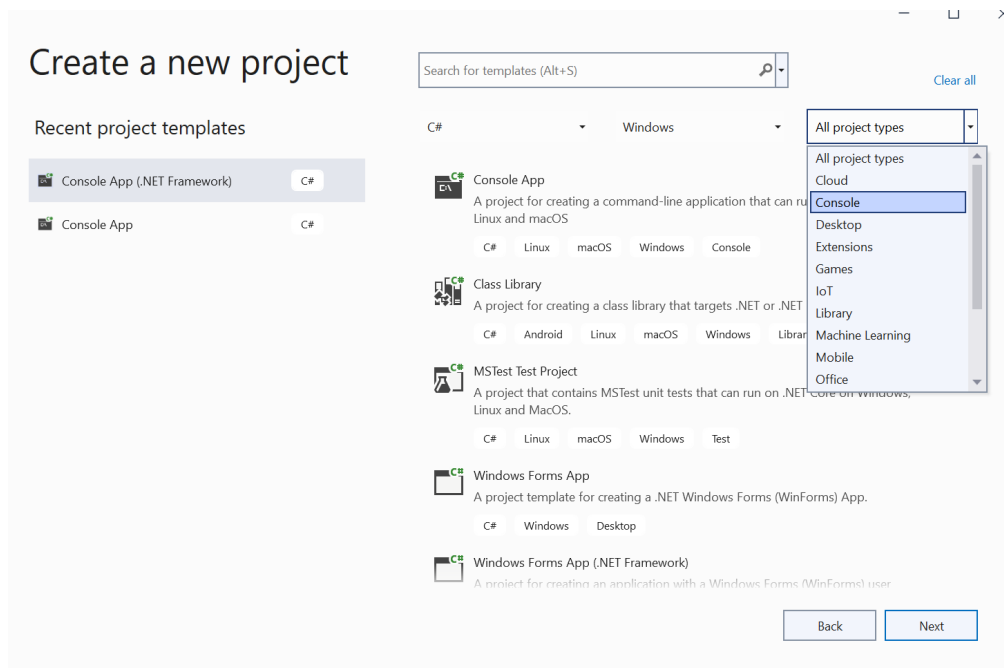
Комп'ютер працює під управлінням програми. Програми призначені для розв'язання наших задач. Тобто нам необхідно в середовищі Microsoft Visual Studio створити розв'язок – **solution**.

**Приклад 1.** Для написання програми в Microsoft Visual Studio необхідно виконати наступне:

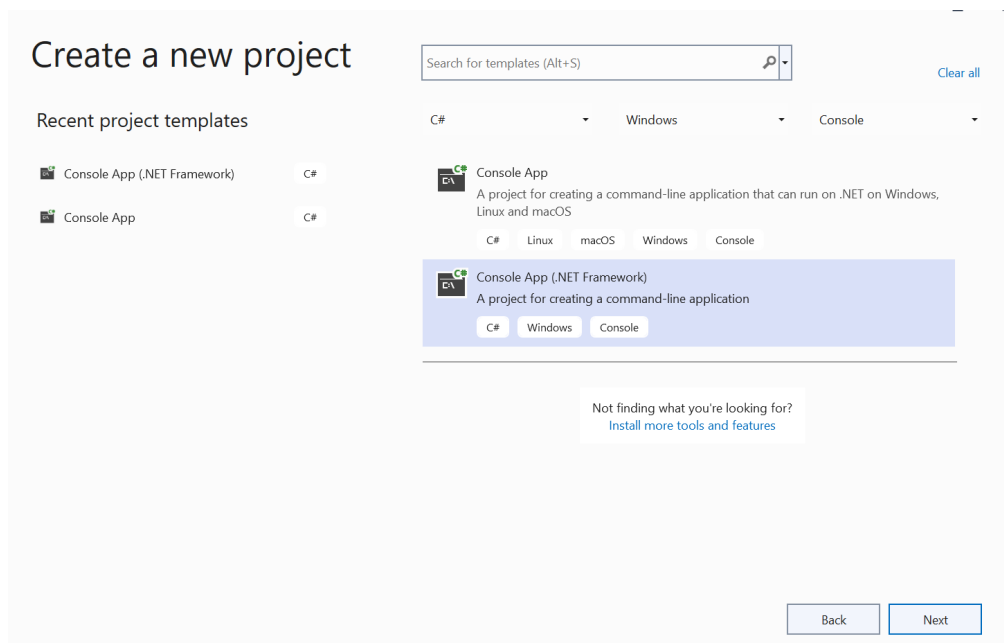
1. створити проект (Create new project);



2. Далі необхідно налаштувати проект; Наприклад, C# , Windows Console



3. Далі вибираємо Console App. (.NET Framework)



4. Після цього задаємо ім'я нашому проекту і обираємо версію .NET Framework:

## Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

Location

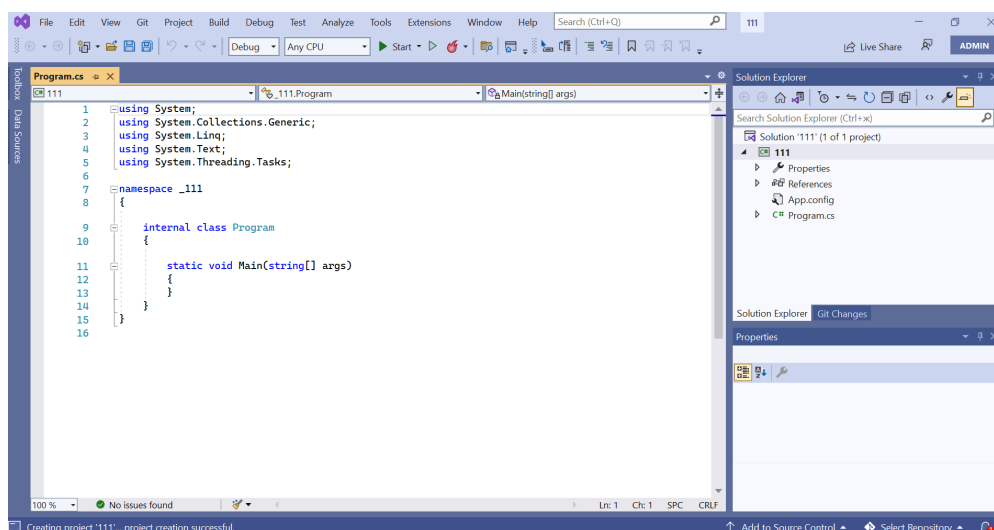
Solution name

☐ Place solution and project in the same directory

Framework

Back Create

5. У центральному вікні ми бачимо робочу область для написання програми; у правій частині – два вікна із властивостями проєкту й розв'язку. Нижче з'явиться інформаційне вікно: повідомлення про помилки, хід виконання програми і т.і. Ми їх розглянемо у наступних розділах.



6. Заготовка програми, що запропонована MS VS, це вже готовий додаток, який можна запустити, натиснувши на кнопку “Запуск”. Запуститься наша перша програма у вікні консольного додатка (швидко промайне чорний екран).

Нам треба його зупинити. Для цього введемо декілька додаткових операторів у програму.

Кожна програма – це набір операторів (команд, інструкцій). Напишемо декілька:

```
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("I will be a programmer!");
        Console.ReadKey();
    }
}
```

Запустимо програму. Ми бачимо, що в консольному вікні (консолі) надруковано фразу

I will be a programmer

Звідки вона з'явилась й чому не пропало чорне вікно? Відповідь на перше питання дає використання оператора

```
Console.WriteLine("I will be a programmer");
```

Відповідь на друге Console.ReadKey();

Слово Console означає, що ми працюємо з консоллю; слово WriteLine – що програма повинна роздрукувати текст, який записаний в круглих дужках.

Оператор Console.ReadKey() очікує на натискання будь-якої клавіші користувачем. Після натискання клавіші – додаток закривається.

Console – ім'я стандартного класу із простору імен System, що містить методи для роботи з консоллю.

WriteLine("текст") – метод класу Console, що виводить на екран заданий у лапках текст.

Readkey() – метод класу Console, що очікує натискання будь-якої клавіші клавіатури.

Розберемо програму детальніше.

Таблиця 1. Перша програма.

|                                                           |                                                                                                                                                                                                                                                                                                                   |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>namespace 111</code>                                | Простір імен 111.<br>Він дозволяє групувати проекти, що містять велику кількість класів, поєднуючи в одну групу близькі класи. Якщо над проектом працює кілька виконавців, то, як правило, кожний із них створює свій простір імен. Це дає можливість давати класам імена, не замислюючись про їхню унікальність. |
| <code>internal class Program</code>                       | Це головний клас програми, він може бути лише один.                                                                                                                                                                                                                                                               |
| <code>static void Main(string[] args)</code>              | Статична функція (метод) Main. Це головна точка входу в програму. Вона може містити якісь аргументи. Аргументи вказуються в круглих дужках.                                                                                                                                                                       |
| <code>Console.WriteLine("I will be a programmer");</code> | Цей оператор надрукує фразу I will be a programmer                                                                                                                                                                                                                                                                |
| <code>Console.ReadKey();</code>                           | Цей оператор очікує, коли буде натиснута будь-яка клавіша клавіатури. Після натискання консоль закривається.                                                                                                                                                                                                      |

### 1.1.1 Контрольні запитання

1. Яке середовище розробки використовуємо для створення програм?
2. Як створити новий проект?
3. Що таке програма?
4. Що таке namespace?
5. Що таке `static void Main(string[] args)`?
6. Яку дію виконує `Console.WriteLine()`?
7. Яку дію виконує `Console.ReadKey()`?

## 1.2 Об'єкти програми

Усі програми оперують зі своїми об'єктами. Об'єкти створюються програмістами для зберігання даних і маніпуляцій з ними.

У специфікації C# слово *object* вживається у двох змістах:

- а) тип *object*;
- б) дане, на яке посилається значення типу посилання.

У цьому підрозділі ми під об'єктами будемо розуміти змінні, константи, екземпляри класів, поля даних екземплярів класів, загалом, ті об'єкти програми, які програма обробляє.

Можна умовно розділити об'єкти на дві групи: прості й складені. Під простими об'єктами ми будемо розуміти об'єкти, значення яких належать простим типам даних, такі як числа й символи. Під складеними – структури, екземпляри класів.

Насправді, в мові C# **прості об'єкти** теж є **екземплярами класів**.

### 1.2.1 Контрольні запитання

1. З чим оперують усі програми?
2. У яких змістах вживається слово *object* у специфікації C# ?
3. Що таке прості об'єкти?
4. Що таке складені об'єкти?

## 1.3 Типізація

Типізація у мовах програмування.

Тип - це набір можливих значень, які може набувати змінна.

Наприклад, тип Ціле показує, що ми будемо працювати із значеннями -5, 1, 2, 3 тощо. Тип Булевий - значення істинна або хибність. Програміст можемо самотійно створити тип, наприклад, Нечіткий і значення тоді будуть «холодне», «гаряче», «напевно», «приблизно 10» і т.д. Типізація мови програмування визначає, чи потрібно вам задавати тип, чи мова програмування визначить це самотійно, і наскільки вільно можна з типами працювати. Будемо говорити так: у змінну можна записувати інформацію, а тип змінної описує, яка саме інформація записана в змінній і що з нею можна робити. Хоча дехто стверджуватиме, що «в змінну нічого

не записується, а значення зберігається в пам'яті». Але ми на це не зважаємо. Ось часті типи, що використовуються

- `int`- ціле число;
- `double` - дійсне число;
- `string` -рядок символів;
- `bool` -істинна/брехня.

З типізацією пов'язане також поняття анотацій типу.

Анотація типів є спеціальним синтаксисом, що дозволяє визначати тип змінної. Наприклад: `int`, `double`, `string` – це анотації цілого, дійсного та рядкового типів відповідно. Не всі мови мають обов'язкову анотацію типів. Однак анотація типів змінних дозволяє покращити підтримку вашого коду різними інструментами програмування. Ми знаємо різні види типізації. Розглянемо їх.

**Слабка та сильна.** Мова з сильною типізацією (її ще називають строгою) вимагає, щоб програмісти дотримувалися правил роботи з типами: якщо ви оголосили змінну цілого типу, то з нею треба працювати як з цілим числом. Мови зі слабкою типізацією гнучкіші: якщо ви вирішите додати число і рядок, то мова спробує зробити це. Однак результат може бути непередбачуваним. Не всі мови дозволять це зробити. І це добре.

**Статична та динамічна.** Статична типізація означає, що типи визначаються на етапі компіляції. При цьому помилки використання змінних буде видно ще до того, як програма почне виконуватись. Статичні мови перевіряють типи у програмі під час компіляції ще до запуску програми. Будь-яка програма, де використання значень порушують правила мови, вважається некоректною.

Наприклад, більшість мов із статичною типізацією відхилить вираз `"a" + 15`. Компілятор знає, що `"a"`— це рядок, а `15` — це ціле число, і оператор `«+»` працює тільки тоді, коли лівий та правий операнд оператора відносяться до одного типу. Так що не потрібно запускати програму, щоб зрозуміти, що програма некоректна. У мовах із динамічною типізацією типи визначаються під час виконання програми. Так, у такій мові в одній і тій же змінній можуть зберігатись різні значення у різні моменти виконання програми.

**Явна та неявна.** Часто можна зустріти поділ на явну та неявну типізацію. Коли типізація неявна, тип визначається під час

виконання програми у момент, коли їй присвоюється значення. Явна типізація означає, що тип змінної визначається при оголошенні змінної. Наприклад, в C# змінна оголошується так:

```
int b;
```

У статично типізованих мовах типізація майже завжди явна, а у динамічно типізованих — неявна. Додавання анотацій типів зазвичай не змінює того, що відбувається в порівнянні з тим, що сталося б без неї, але текст програми стає більш читабельним. Висновки, які ми можемо зробити, вивчивши типізацію.

- Сильна типізація обмежує свободу програміста, проте поведінка програми передбачувана.
- Слабка типізація дає більше свободи, але з нею треба бути більш акуратнішим.
- Статична типізація додає роботи під час написання коду: багато потрібно прописувати вручну.
- Динамічна типізація простіша в написанні, але погіршує розуміння коду і можуть з'явитись помилки, які складно передбачити, а тому і знайти.
- Програма, написана мовою зі статичною типізацією, зазвичай працює швидше, ніж програма, що написана мовою з динамічною типізацією.

### 1.3.1 Контрольні запитання

1. Що таке тип?
2. Що таке строга типізація?
3. Що таке слабка типізація?
4. Що таке статична типізація?
5. Що таке динамічна типізація?
6. Що таке явна типізація?
7. Що таке неявна типізація?

## 1.4 Змінна

Змінну (синтаксично) можна розглядати як позначення об'єкта в тексті програми. Семантично, змінна – це місце зберігання даних в пам'яті комп'ютера. Нехай навіть такого складного, як екземпляр класу. У змінну можна записувати дані й зчитувати. Дані, що зберігаються в змінній, називаються значенням змінної.

Власне кажучи, при звертанні до змінної під час виконання програми відбувається звернення (за адресою) до деякої ділянки пам'яті, у якій зберігається її значення.

Синтаксично кожна змінна повинна бути певного типу даних. Семантично кожний об'єкт повинен бути певного класу. Усі типи мови C# діляться на *вбудовані* й *ті, що розроблені користувачем*.

Тип даних визначає:

1. множину значень, яку можуть набувати величини цього типу;
2. операції, які можна застосовувати до величин цього типу;
3. внутрішнє зображення даних у пам'яті комп'ютера.

Слід чітко розуміти, що для того, щоб використовувати змінну, її спочатку потрібно об'явити й ініціалізувати, тобто присвоїти початкове значення.

Розглянемо фрагмент простої програми:

```
class Program
{
    static void Main(string[] args)
    {
        int a;
        a = 5;
        int b, c;
        bool d;
        d = true;
        long e = 10;
        float f = 5.5f;
        char g = 'g';
    }
}
```

Розберемо її детальніше:

|                              |                                                                                                                              |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>int a;</code>          | оголошуємо змінну <code>a</code> типу <code>int</code>                                                                       |
| <code>a = 5;</code>          | записуємо в змінну <code>a</code> число <code>5</code> (ініціалізуємо змінну)                                                |
| <code>int b, c;</code>       | оголосити можна відразу декілька змінних через кому                                                                          |
| <code>bool d;</code>         | оголошуємо змінну <code>d</code> типу <code>bool</code>                                                                      |
| <code>d = true;</code>       | записуємо в змінну <code>d</code> значення <code>true</code> (істина)                                                        |
| <code>long e = 10;</code>    | при оголошенні змінної можна відразу ж задавати їй значення                                                                  |
| <code>float f = 5.5f;</code> | щоб записати число із плаваючою точкою (типу <code>float</code> ), необхідно після значення додавати суфікс <code>f</code> . |
| <code>char g = 'g';</code>   | оголошуємо символьну змінну <code>c</code> із початковим значенням <code>g</code>                                            |

При використанні змінної, у яку не було записано значення, компілятор видасть помилку

"Use of unassigned local variable [variablename]"

Наприклад,

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int a;
        Console.WriteLine(a); // compilation error
    }
}
```

### 1.4.1 Контрольні запитання

1. Що таке змінна?
2. Що таке значення змінної?
3. Де зберігається значення змінної?
4. На які типи даних діляться всі типи у мові C# ?
5. Що визначає тип даних?

## 1.5 Ключове слово var

Ключове слово `var` дозволяє створювати змінні без явної вказівки типу даних. Тип даних такої змінної визначає компілятор у місці її ініціалізації

```
static void Main(string[] args)
{
    var number=5; //змінна number буде типу int
    var text = "I_will_be_a_programmer"; // змінна text буде типу string
    var number2 = 0.5; // змінна number2 буде типу double
}
```

Це означає, що в змінну, що оголошена як **var**, не можна записати дані типу, несумісного з типом, заданим для неї компілятором:

```
static void Main(string[] args)
{
    var number = 5;
    number = "I_will_be_a_programmer"; // помилка, number ВЖЕ визначена
    як int
}
```

### 1.5.1 Контрольні запитання

1. Що виконує ключове слово `var`?
2. Коли визначається тип змінної, оголошеної як `var`?
3. Чи можна в змінну, оголошену як `var`, записати дані іншого типу після ініціалізації?

## 1.6 Константи

Ще програми працюють із такими об'єктами, як константи.

**Константа** – це іменоване значення, яке стане відомим під час компіляції програми.

Оголошення константи подібно звичайному оголошенню змінної, але перед ним необхідно додати ключове слово `const`:

```
static void Main(string[] args)
{
    const int months = 12;
    months = 13;
}
```

|                                     |                                                                              |
|-------------------------------------|------------------------------------------------------------------------------|
| <code>const int months = 12;</code> | оголошення константи                                                         |
| <code>months = 13;</code>           | помилка компіляції<br>(оскільки не можна<br>присвоїти значення<br>константі) |

При оголошенні константи необхідно обов'язково проініціалізувати її певним значенням.

Константи також роблять ваш код більш читабельним.

```
static void Main(string[] args)
{
    const int group_Students = 15;
    const int course_Group = 4;
    int all_Students = group_Students * course_Group;
}
```

Наведений вихідний код читається й розуміється краще, ніж такий:

```
static void Main(string[] args)
{
    int all_Students = 15 * 4;
}
```

Значення можуть бути задані в тексті програми за допомогою числових і строкових літералів:

```
static void Main(string[] args)
{
    Console.WriteLine(100);           // 100 – це числовий літерал
    Console.WriteLine("Student");     // рядок "Student" – строковий літерал
}
```

Як ми бачимо, строкові літерали беруться в лапки.

### 1.6.1 Контрольні запитання

1. Що таке константа?
2. Яке ключове слово використовується при оголошенні констант?
3. Назвіть правило оголошення констант?

## 1.7 Програмування з використанням елементів ООП

Уся логіка роботи програми при розв’язку конкретної задачі мовою С# описується усередині класів. Ми вже мали можливість познайомитися з головним класом будь-якої програми Program.

Розберемося з поняттям клас і надалі навчимося їх правильно описувати.

Навколишній світ розбитий на групи, множини: тварини й рослини, студенти й викладачі, гравці й тренери тощо.

**Клас** – це тип даних. З його допомогою описуються характеристики, властивості й можливі дії виділеної групи об’єктів. Клас дає загальний опис об’єктів, указує “на що вони схожі”.

Клас може описувати людину, студента, автомобіль, будинок, ліфт і т.д. Описавши клас, ми можемо створити його **екземпляр** – конкретного представника класу.

**Приклад 2.** Розглянемо класи “Людина” і “Студент”.

**Клас** “Людина” має фіксовані поля-опису: Прізвище, Ім’я, По батькові, Зріст, Вага, Дата народження; **властивості**: Вік.

**Клас** “Студент” має фіксовані поля-опису: Прізвище, Ім’я, По батькові, Зріст, Вага, Дата народження, Факультет, Група, Курс; **властивості**: може не мати.

Бачимо, що деякі поля повторюються, тобто при описі класу “Студент” можна використовувати дані із класу “Людей”. Це називається **спадкуванням**.

Студент із прізвищем **Зачишигріва**, який вчиться на **першому** курсі в **другій** групі (і має всі інші конкретні свої поля-опису) – це конкретний екземпляр (об’єкт) класу “Студент”.

Розглянемо ще приклади.

**Приклад 3.** Нехай клас — це проєкт ліфта. Саме проєкт, креслення, опис технічних характеристик.

У ліфта є висота, ширина, довжина — це **властивості**. Він уміє переміщатися — нагору/униз, це **методи**.

Ліфти, що встановлені в наших будинках, — це конкретні екземпляри класу “Ліфт”, у яких є особистий номер, розміри. Також є **інтерфейс** — це кнопки керування ліфтом.

Для опису ліфта треба задати:

- поля для опису обладнання ліфту;

- методи для опису як він повинен переміщатися – нагору / униз;
- інтерфейс для опису взаємодії людини й ліфта.

За проєктом, кресленням ми можемо виготовити багато ліфтів. Можемо, наприклад, міняти їм колір дверей (атрибут `public`, **властивості читання/запису**), але не можемо міняти швидкість, яка постійна (атрибут `public`, властивості лише **читання**). У нього є багато інших внутрішніх властивостей, які нам невідомі, вони є схованими (атрибут `private`), але завдяки їм ліфт працює.

**Приклад 4.** Клас Автомат “Калашникова” (`class Mashinegan`).

Автомат має магазин, прицільну рамку, багнет-ніж, спусковий гачок, інші деталі, властиві всім автоматам. Також він має різні режими стрілянини — одиночні й черга. Залежно від поточного режиму стрілянини (читай стану), автомат по-різному стріляє: одиночними пострілами або чергою. Таким чином, при **однаковому** натисканні на спусковий гачок автомат має різну поведінку. Під час стрілянини кількість патронів у магазині зменшується (значення поля “магазин” змінюється).

Для опису автомата треба задати:

- поля для опису будови автомата;
- методи для опису, як повинен автомат поводитися, наприклад, стріляти в різних режимах;
- інтерфейс для опису взаємодії стрільця й автомата.

Без ООП ми, для імітації роботи автомата, викликали щось на зразок функції `Fire(ammocount, burstmode)`, де `ammocount` — кількість набоїв у магазині, `burstmode` — режим стрілянини.

А в ООП уся інформація зберігається в об’єкті й ми викликаємо просто метод `Fire()`, який входить в опис класу `Mashinegan`.

**Приклад 5.** Клас “Автомобіль”.

В автомобіля є **поля**, **властивості** і є **функції**, які він може виконувати. До полів можна віднести колеса, двигун, кузов, сидіння і т.д. Серед властивостей можна виділити, наприклад, витрату палива, колір кузова, об’єм двигуна, тип кузова, тип коробки передач, кількість посадкових місць і т.д. Основні функції автомобіля — їхати вперед/назад, повернути ліворуч/праворуч і т.д.

Автомобіль УАЗ 3160 — належить до класу легкових автомобілів; `Actyon Sports` — теж належить до класу легкових авто. Різниця між

ними істотна, але насправді вони мають багато спільного – колір кузова, витрата пального і т.і. Тому ці дві різні машини виділяємо в один клас – клас легкових авто. Ці дві машини є двома різними екземплярами одного класу.

Автомобіль “Богдан” – має ті ж формальні характеристики, що попередні авто. Але є й додаткові функції й властивості: відкривання дверей за натисканням кнопки, перевезення пасажирів, місця для пасажирів, що стоять, місця для пасажирів, що сидять.

Аналогічно можна описати особливості вантажних автомобілів. Для опису автомобіля треба задати:

- поля для опису будова автомобіля;
- методи для опису як повинен автомобіль поводитись, наприклад, їхати вперед/назад, повертати ліворуч/праворуч, перемикає коробку передач і т.і.;
- інтерфейс для опису взаємодії водія й автомобіля: натискання на педаль гальма, газу, зчеплення; поворот керма.

Отже, клас є лише загальним описом, схемою екземпляра. Клас може містити поля, методи, властивості й події. **Поля** визначають стан, а **методи** – поведінку конкретного об’єкту. Формально, опис класу має вигляд:

```
class Myfirstclass                                // назва класу
{
    public field_type field_name;                  // поле
    public void method_name()                     // метод
    {
        //Оператори для роботи з field_name;
    }
}
```

Розглянемо докладніше приклад із класом “Ліфт”. Опишемо його формально.

```
class Elevator
{
}
```

Такий запис говорить про те, що ми будемо працювати із класом Elevator. Повний опис класу буде записаний у фігурних дужках. Як ми вже відзначали, програми працюють із об’єктами – екземплярами класів. У нашому випадку екземпляр класу Elevator – це конкретний ліфт у конкретному будинку.

Таким чином, нам треба дати команду на оголошення такого екземпляру. Це здійснюється за допомогою задання імені екземпляра

й вказівки імені класу (типу), до якого даний екземпляр буде належати. Тобто маємо:

```
Elevator MyElevator;
```

Ми оголосили, що в програмі будемо працювати з екземпляром MyElevator типу (або класу) Elevator.

Тепер потрібно такий екземпляр створити. Для цього запишемо такий рядок:

```
MyElevator = new Elevator();
```

В такий спосіб у пам'яті комп'ютера за допомогою методу new виділяється пам'ять під зберігання даних об'єкта MyElevator.

Розширимо наші знання про реальний об'єкт Ліфт. Ліфт має висоту, ширину, довжину й вантажопідйомність. Проєктований Ліфт може їздити нагору/униз. Опишемо це в класі Elevator. Розміри ліфта задамо цілими числами, а напрямок руху будемо друкувати на екрані словами.

```
class Elevator
{
    public int height;
    public int width;
    public int length;
    public int capacity;
}
```

Тепер можна створити екземпляр класу Elevator (конкретний ліфт) із заданими розмірами.

```
Elevator MyElevator;
MyElevator = new Elevator();
MyElevator.height = 2;
MyElevator.width = 3;
MyElevator.length = 1;
MyElevator.capacity = 800;
```

Записуючи MyElevator.height ми говоримо комп'ютеру, що прагнемо саме в екземплярі MyElevator присвоїти значення 2 полю height. Адже в одній програмі може бути кілька екземплярів класу Elevator.

Наприклад,

```
Elevator HisElevator, MyElevator, N10Elevator;
```

Конкретний ліфт може рухатися, тому нам треба описати елементи цього руху. Тобто необхідно вказати комп'ютеру які кроки (інструкції) він повинен виконувати, якщо ми прагнемо, щоб ліфт рухався нагору або вниз.

Нехай для імітації руху нагору/униз на екран комп'ютера буде на екран виводитись відповідне слово “нагору” або “униз”.

Оформимо це у вигляді методу класу.

```
void Up()
{
    Console.WriteLine("Up");
}
```

Запис `Console.WriteLine` говорить комп'ютеру про те, що необхідно викликати метод `WriteLine` зі стандартного класу `Console` для друку слова `Up` на екран комп'ютера.

Слово `void` говорить про те, що після закінчення роботи методу більше нічого робити не треба, `void` означає – нічого, порожнеча.

Щоб запустити ліфт нагору, необхідно просто викликати метод `Up()`, тобто записати в головному методі `Main` рядок

```
Up();
```

Відповідно, для імітації руху вниз необхідно описати такий метод:

```
void Down();
{
    Console.WriteLine("Down");
}
```

Отже, маємо таку програму:

```
using System;
namespace ExampleElevator
{
    class Elevator
    {
        public int height;
        public int width;
        public int length;
        public int capacity;
        public void Up()
        {
            Console.WriteLine("Up");
        }
        public void Down()
        {
            Console.WriteLine("Down");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Elevator MyElevator;
            MyElevator = new Elevator();
            MyElevator.height = 2;
            MyElevator.width = 3;
            MyElevator.length = 1;
            MyElevator.capacity = 800;
            MyElevator.Up();
            MyElevator.Down();
        }
    }
}
```

```
}  
}
```

Атрибут `public` означає, що метод, описаний в одному класі, може викликатися в інших класах. Якщо ми не прагнемо давати такої можливості, то слід використовувати атрибут `private`.

Отже, ми розібралися, із чого складаються класи, навчилися їх описувати. Так само ми вміємо створювати екземпляри класів і виконувати задані дії з їхніми полями.

Ще раз хочеться підкреслити той факт, що сучасне програмування вимагає вміння оперувати із класами і їх екземплярами, а мова `C#` це об'єктно-орієнтований мова програмування, у чому ми ще не раз переконаємося.

### 1.7.1 Контрольні запитання

1. Що таке клас?
2. Що таке екземпляр класу?
3. Назвіть складові класу.
4. Що таке поля?
5. Що таке методи?
6. Наведіть приклад оголошення класу?
7. Якою мовою є мова `C#` ?

## 1.8 Атрибути `private`, `public`, `protected`

Якщо ж ми в попередньому параграфі використовували атрибут доступу `public`, то має сенс розглянути питання доступності полів більш докладно. Екземпляри одного класу можуть бути полями в іншому класі. У нашому випадку в класі `Program` оголошується й використовується екземпляр класу `Elevator`.

Ми в класі `Program` маємо можливість працювати із полями, що оголошені в класі `Elevator`. Ми змінюємо значення, можемо друкувати їх. Також ми використовуємо методи, що оголошені в класі `Elevator`. Для цього ці поля і методи оголошені з атрибутом `public`. Але інколи має сенс заборонити доступ до полів і методів класу, зробити їх закритими або мовою `C#` – `private`.

Пояснимо це більш детально. Якщо ми застосовуємо для поля атрибут *public*, то екземпляри будь-якого класу можуть використовувати це поле або метод. У випадку використання *private* — лише екземпляри даного класу можуть звертатися до елементів класу. Тобто поля і методи захищені від несанкціонованого доступу до них з боку «чужих» екземплярів. Це підвищує загальну надійність програмного забезпечення. Залишився атрибут *protected*. Цей атрибут у ООП має особливе значення. З одного боку він робить поля і методи захищеними, а з іншого дає можливість екземплярам інших, але спеціальних класів, використовувати елементи класу з атрибутом *protected*. Ці класи називаються нащадками. Їх ми будемо вивчати в наступних розділах. Припустимо, ми не хочемо, щоб можна було змінювати вантажопідйомність ліфта, тоді необхідно в класі *Elevator* записати так:

```
class Elevator
{
    public int heigth;
    public int width;
    public int length;
    private int capacity=800;
public void Up();
{
    Console.WriteLine("Up");
}
public void Down();
{
    Console.WriteLine("Down");
}
}
```

І тоді, якщо в класі *Program* запишемо

```
MyElevator.capacity = 800;
```

то з'явиться помилка:

### 1.8.1 Контрольні запитання

1. Опишіть атрибут *public*.
2. Опишіть атрибут *protected*.
3. Опишіть атрибут *private*.

## 1.9 Конструктор

Ми вже говорили, що екземпляр класу необхідно спочатку оголосити, а потім утворити. У процедурі утворення екземплярів бере участь спеціальний метод, який називається конструктор. У нашому класі Elevator ми не бачимо опису методу-конструктора. Але він все одно присутній. Можна сказати, що він описаний «неявно». Якщо програміст не написав конструктор, то система доповнює цей опис самостійно. Але програміст може і сам зробити це. Конструктор призначений для «конструювання» екземпляру класу, тобто надання початкових значень полям. Сконструємо конкретний ліфт, тобто екземпляр класу з наперед заданими параметрами. І припустимо, що ці параметри будуть використовуватись для всіх екземплярів, тобто усі ліфти будуть однакові. Опишемо такий конструктор:

```
class Elevator
{
    public int height;
    public int width;
    public int length;
    public int capacity;
    void Elevator()
    {
        height = 2;
        width = 3;
        length = 1;
        capacity = 800;
    }
    public void Up();
    {
        Console.WriteLine("Up");
    }
    public void Down();
    {
        Console.WriteLine("Down");
    }
}
```

Тепер якщо ми в класі Program запишемо

```
Elevator MyElevator = new Elevator();
```

то буде утворений новий екземпляр класу Elevator з іменем MyElevator і поля цього екземпляру будуть мати значення, що задаються конструктором Elevator().

Припустимо, що ми хочемо для різних екземплярів класу надавати полям різні початкові значення. Це можна зробити за допомогою конструктора з параметрами. Саме параметри і будуть задавати ці значення. У такому разі конструктор буде мати такий вигляд:

```
void Elevator(int h, int w, int l, int c)
{
    height = h;
    width = w;
    length = l;
    capacity = c;
}
```

У нашому випадку всі значення розмірів ліфтів і вантажопідйомності цілі числа, тому в круглих дужках ми записали імена параметрів і вказали типи їх значень як `int`. Тепер при створенні екземпляра класу можна викликати конструктор `Elevator` і вказувати необхідні розміри ліфтів:

```
Elevator MyElevator = new Elevator(2,4,6,1800);
```

Або знову використовувати конструктор без параметрів, якщо необхідно.

### 1.9.1 Контрольні запитання

1. Що таке конструктор?
2. Наведіть приклад конструктора.
3. Які є конструктори?

## 1.10 Задачі до розділу ВСТУП

1. Опишіть клас гелікоптер, що містить як приватні, так і публічні поля. Приватні поля ініціалізуються у конструкторі. Передбачте взаємодію між полями класу за допомогою методів. Передбачте використання констант.
2. Опишіть клас приватне підприємство, що містить як приватні, так і публічні поля. Приватні поля ініціалізуються у конструкторі. Передбачте взаємодію між полями класу за допомогою методів. Передбачте використання констант.
3. Опишіть клас магазин, що містить як приватні, так і публічні поля. Приватні поля ініціалізуються у конструкторі. Передбачте взаємодію між полями класу за допомогою методів. Передбачте використання констант.

4. Опишіть клас бібліотека, що містить як приватні, так і публічні поля. Приватні поля ініціалізуються у конструкторі. Передбачте взаємодію між полями класу за допомогою методів. Передбачте використання констант.
5. Опишіть клас будинок, що містить як приватні, так і публічні поля. Приватні поля ініціалізуються у конструкторі. Передбачте взаємодію між полями класу за допомогою методів. Передбачте використання констант.

## 2 Основи алгоритмізації

Метою програмування є опис процесів обробки даних.

Введемо деякі означення.

**Означення 2.1.** *Дані – це зображення фактів та ідей у формалізованому вигляді, придатному для передавання й переробки в деякому процесі.*

**Означення 2.2.** *Інформація – це вміст, який надається даним при їхньому зображенні.*

**Означення 2.3.** *Обробка даних – це виконання систематичної послідовності дій з даними.*

**Означення 2.4.** *Сукупність носіїв даних, які використовуються за будь-якої обробки даних, називатимемо інформаційним середовищем.*

**Означення 2.5.** *Набір даних, що містяться в будь-який момент в інформаційному середовищі, називатимемо станом цього середовища.*

**Означення 2.6.** *Процес обробки даних можна визначити як послідовність змін станів деякого інформаційного середовища.*

Описати процес – означає визначити послідовність станів заданого інформаційного середовища. Якщо ми хочемо, щоб за заданим описом необхідний процес породжувався автоматично на комп'ютері, то необхідно, щоб опис був формалізованим. Такий опис називається програмою.

Основною у процесі програмування є розробка алгоритму – один із найскладніших етапів розв'язання задач з використанням комп'ютера. Візьмемо за основу уявлення про алгоритм як опис деякого обчислювального процесу. Визначальною особливістю обчислювального процесу є можливість розчленувати його на окремі дискретні дії. Отже, для написання алгоритму нам належить лише по пунктах їх перелічити. Алгоритм – це деяке правило перетворення

інформації, застосування якого до заданої (початкової) інформації приводить до результату – нової інформації.

Кожне правило алгоритму містить точний опис деякої елементарної дії з перетворення інформації і вказівки на інструкцію, яку необхідно виконати.

Велика увага у теорії алгоритмів приділяється методам задання (описи, конструювання) алгоритмів – форматам запису.

Загальноприйнято оперувати мовою блок–схем. Інший різновид мови задання алгоритмів – це діаграми Насси – Шнейдермана. Для спрощення ми використовуватимемо словесний опис. Однак навіть таке спрощення надалі дозволить нам швидко перейти до написання програм.

Алгоритми мають такі властивості:

- **Елементарність.** Кожна команда з набору містить указівку виконати деяку елементарну дію, що однозначно розуміється й точно виконується.
- **Визначеність.** Виконання алгоритму строго визначене. Це означає, що на кожному кроці треба не лише точно виконати команду, але й однозначно визначити наступну. Тому повторне виконання алгоритму для одних і тих самих початкових даних точно повторює перше його виконання.
- **Масовість.** Алгоритми зазвичай описують хід розв’язання не однієї–єдиної задачі, а цілого класу однотипних задач.
- **Результативність.** Виконання будь-якого алгоритму має закінчитися через скінченну кількість кроків із деяким результатом. Однак кількість кроків алгоритму, який розв’язує деяку задачу, заздалегідь невідома й може бути дуже великою. Тому властивість результативності конкретного алгоритму часто необхідно спеціально доводити, щоб результат був досяжним.

Алгоритми в процесі роботи для отримання результату обробляють деякі дані, тобто мають вхід (вхідні дані) і вихід (результат). Вхідні дані потрібні не для кожного алгоритму.

Крім вхідних і вихідних, зазвичай алгоритм передбачає тимчасове формування проміжних даних (ще їх називають допоміжними). Тому ще однією характеристикою алгоритму є обсяг пам’яті, яку він використовує.

При конструюванні алгоритму слід строго визначити кожен його крок, передбачивши будь-які можливі стани процесу й відповідні інструкції для їхньої обробки. Лише такий алгоритм гарантує однозначне отримання необхідного результату та класифікується як детермінований. Щодо такого алгоритму можна стверджувати: його неодноразове застосування до однакових вхідних даних завжди приводить до одного й того самого результату. Стохастичні алгоритми ми не розглядатимемо.

### 2.0.1 Контрольні запитання

1. Що таке дані?
2. Що таке інформація?
3. Що таке обробка даних?
4. Що таке інформаційне середовище?
5. Що таке стан інформаційного середовища?
6. Що означає «описати процес»?
7. Які є методи задання алгоритмів?
8. Назвіть властивості алгоритмів.

## 2.1 Класифікація внутрішніх структур алгоритмів

Основними алгоритмічними структурами є: лінійна, розгалуження та цикл.

**Лінійні структури.** Прості задачі мають лінійний алгоритм розв'язання. Це означає, що він не містить перевірок умов і повторень.

Розглянемо алгоритм такої задачі: за даними довжинами катетів  $a$ ,  $b$  обчислити площу прямокутного трикутника.

*Алгоритм:*

Вхід: Числа  $a$ ,  $b$ .

1. Ввести  $a$ ,  $b$ .

2.  $S = ab/2$ .
3. Вивести значення  $S$ .
4. Кінець.

Вихід: Значення площі  $S$ .

Для перевірки працездатності алгоритму необхідно задати значення вхідних змінних, обчислити кінцевий результат за алгоритмом і порівняти із результатом ручного розрахунку.

**Структура–розгалуження.** Досить часто та або інша дія має виконуватись залежно від значення логічного виразу, який є умовою. У такому разі використовується розгалуження.

Розглянемо алгоритм розв’язання квадратного рівняння

$$ax^2 + bx + c = 0.$$

*Алгоритм:*

Вхід: Коефіцієнти  $a$ ,  $b$  і  $c$  із рівняння  $ax^2 + bx + c = 0$

1. Обчислити  $D = b^2 - 4ac$ .
2. Якщо  $D < 0$ , то (відповісти “Розв’язків немає”; Перейти до 6).
3. Якщо  $D = 0$ , то (обчислити  $x = -b/(2a)$ ); Перейти до 6.
4. Обчислити  $x_1 = (-b + \sqrt{D})/(2a)$ .
5. Обчислити  $x_2 = (-b - \sqrt{D})/(2a)$ .
6. Закінчити роботу.

Вихід: “Розв’язків немає” або корінь  $x$ , або корені  $x_1$ ,  $x_2$ .

У цьому алгоритмі використовується команда вигляду

Якщо  $\langle \text{умова} \rangle$  то  $\langle \text{послідовність команд} \rangle$ .

Виконуючи цю команду, необхідно перевірити істинність умови. Якщо умову виконано, то здійснюємо перехід до виконання першої із послідовності команди, що стоїть після слова “то”, і потім переходимо до виконання наступних лінійних команд. Якщо умова помилкова, то пропускаємо  $\langle \text{послідовність команд} \rangle$  і переходимо до виконання наступної команди. Такі команди називають умовними, або розгалуженнями.

При тестуванні алгоритмів із розгалуженнями необхідно підбирати такі початкові дані, щоб можна було перевірити всі гілки.

**Циклічні структури.** Якщо будь-які операції необхідно виконати кілька разів, то їх не переписують кожного разу заново, а організують цикл. Розглянемо алгоритм задачі: підрахувати кількість цифр у записі натурального числа  $n$ . Ідея розв'язання: із заданого числа вибирати із молодшого розряду цифру за цифрою, доки число не вичерпається, тобто не стане рівним нулю; кожну цифру враховувати, тобто підраховувати їх кількість.

*Алгоритм:*

Вхід: Число  $n$ .

1. Ввести число  $n$ .
2.  $K = 0$  {готуємо лічильник кількості цифр}.
3. Якщо  $n = 0$ , то перехід до п.7.
4.  $K = K + 1$ .
5.  $n = n \text{ div } 10$ .
6. Перехід до п.3.
7. Друк .
8. Кінець.

Вихід: Кількість цифр .

Слід зважати на те, що за конкретної реалізації цього алгоритму у середовищі конкретної мови програмування можна використовувати цикли як з перед-, так і з пост-умовою.

Розглянемо алгоритм Евкліда обчислення найбільшого спільного дільника цілих додатних чисел  $A$  та  $B$ : НСД( $A, B$ ).

*Алгоритм:*

Вхід:  $A, B$ .

1. Обчислити  $U = A$ .
2. Обчислити  $V = B$ .
3. Поки  $U \neq V$  виконувати 4:
4. Якщо  $U < V$ , то Обчислити  $V = V - U$ ,  
Інакше Обчислити  $U = U - V$ .
5. Обчислити  $D = U$ .

Вихід:  $D$  – найбільший спільний дільник  $A$  і  $B$ .

У цьому алгоритмі використано команду повторення, яка має вигляд

Поки  $\langle \text{Умова} \rangle$  виконувати  $\langle \text{Команда} \rangle$ .

Виконуючи команду, спочатку перевіряємо істинність “Умови”. Якщо “Умова” істинна, то виконуємо “Команда”, указану після слова “виконувати”, і повторюємо перевірку “Умови”. Виконання “Команда” й перевірка “Умови” повторюються, доки “Умова” істинна. Якщо “Умова” хибна, то переходимо до виконання команди, наступної за командою повторення. У цьому самому алгоритмі використано ще один різновид команди розгалуження – команду вигляду

Якщо  $\langle \text{Умова} \rangle$ , то  $\langle \text{Команда1} \rangle$  інакше  $\langle \text{Команда2} \rangle$

Виконуючи її, спочатку перевіряємо “Умову”. Якщо “Умова” виконана, то виконується “Команда1”, інакше – “Команда2”. Далі переходимо до наступної команди. Зауважимо, що команда повторення, як і команди розгалуження, містить у собі інші команди.

### 2.1.1 Контрольні запитання

1. Назвіть основні алгоритмічні структури.
2. Опишіть лінійні структури.
3. Опишіть структуру-розгалуження.
4. Опишіть циклічну структуру.

## 2.2 Складність алгоритмів

Природно, що перед розробкою програми слід придумати, як розв’язати поставлену задачу. При розробці алгоритму необхідно, перш за все, приділити увагу часу його роботи й обсягу пам’яті, яку необхідно витратити для зберігання й обробки даних. Дані поділяють на вхідну інформацію (вхідне слово), ті, що вимагають проміжного зберігання, і вихідну інформацію (вихідне слово). Не всі дані вимагають одночасного зберігання й використання, а тому можна планувати роботу з ефективного їх використання.

Для розв’язання задачі можна написати багато різних алгоритмів. Чим вони відрізняються, який краще використовувати? Як визначити поняття “краще”? Такі питання виникають (або мають виникати) у професійного програміста.

Для початку замінімо поняття “краще” на “ефективніше” й далі його використовуватимемо. Ефективність програми є дуже важливою її характеристикою. Вона має дві складові: розмір і час.

Розмір вимірюється обсягом пам'яті, що необхідна для виконання програми. Іноді обсяг необхідної пам'яті є домінуючим фактором в оцінці ефективності програми. Проте останніми роками ця складова ефективності поступово втрачає своє значення.

Часова ефективність програми визначається часом, необхідним для її виконання. Вона залежить як від конкретної реалізації алгоритму, так і від власне вибраного алгоритму для розв'язання поставленої задачі.

Отже, складність обчислення, тобто роботи алгоритму на певному вхідному слові, визначається ресурсами, а ресурс, необхідний для конкретного обчислення, – це число. Знаходженням та оцінкою такого числа ми й займемося.

Основною властивістю алгоритму є виконання поставленої задачі за скінченну кількість кроків. Швидкість, з якою алгоритм виконується на конкретному пристрої, може істотно залежати від набору вхідних даних. При цьому швидкий у середньому алгоритм здатний давати збої в окремих “поганих” випадках. І, якщо задача має напевно розв'язуватися за певний час роботи процесора, то ми віддамо перевагу алгоритму, повільнішому в середньому, проте надійному в гірших ситуаціях. Саме уміння передбачати погані ситуації і відрізняє кваліфікованого алгоритміста.

Будь-який крок алгоритму реалізується деякою кількістю машинних операцій. Деталізація алгоритму має бути такою, щоб на кожному окремому кроці не потрібне було його подальше алгоритмічне опрацювання. Тут можливі лише дві ситуації: або фіксований час виконання такого кроку визначено деяким набором простих, без циклів, команд мови програмування, або йдеться про ускладнений крок, для якого відповідний аналіз уже проводився і результати відомі.

Тепер перейдемо до головного питання, яке можна поставити будь-якому програмісту: припинить його програма своє виконання або ж працюватиме до нескінченності? За теоремою, доведеною Тьюрінгом у 30–ті рр. XX ст., не існує алгоритму, що дає відповідь на це питання для довільної програми, тобто проблема зупинки алгоритмічно не розв'язна. Проте у багатьох практичних випадках достатньо легко довести існування алгоритмів, які розв'язують поставлену задачу. Тому нас цікавить не просто існування алгоритмів для нашої задачі, а й їх ефективність. Причому з усіх складових

ефективності ми звертатимемо особливу увагу на час (швидкодію) роботи алгоритму. Справедливою, хоч і з деякими обмовками, є така формула:

загальний час роботи = кількість операцій алгоритму \* час на 1 операцію.

Поліпшення обох членів у правій частині формули здійснюється незалежно. Грубо кажучи, програми відповідають за перший співмножник, а процесори – за другий.

Розглянемо деякі приклади.

**Алгоритм обміну значень двох змінних** цілого типу –  $a$  і  $b$  – реалізується в загальному випадку за три кроки, незалежно від того, до якого типу простих даних він застосовується:

$temp = a$

$a = b$

$b = temp$

Якщо застосуємо для обміну значеннями алгоритм, що використовує арифметичні операції, то дістанемо таку послідовність дій:

$a = a + b$

$b = a - b$

$a = a - b$

**Знайти суму натуральних чисел від 1 до заданого  $n$ .** Якщо скористатися відомою формулою для суми арифметичної прогресії, то для обчислення також знадобляться лише три кроки: додавання, множення й ділення. Якщо ж реалізувати обчислювальний процес як циклічний (цикл із параметром), і керувальна змінна пробіжить значення від 1 до  $n$ , то доведеться виконати  $n$  кроків алгоритму.

Сумнівів, який з алгоритмів ефективніший, здається, не виникає.

Для алгоритмів, подібних до щойно розглянутого ( $n$  кроків для обробки  $n$  вхідних значень), кількість кроків є функцією від кількості елементів, що обробляються, –  $g(n)$ . Не для кожного алгоритму таку функцію можна легко обчислити.

Для оцінки швидкості зростання досліджуваної функції порівняємо її з функцією, дія якої вже добре відома. При дослідженні поведінки функцій  $g(n)$  натурального  $n$  розглядатимемо такі функції  $g(n)$ , що зростають не швидше  $f(n)$ , тобто існує така пара додатних значень  $M$  і  $n_0$ , що  $|g(n)| \leq M|f(n)|$  для  $n \geq n_0$ .

Таку залежність позначають у математиці символом  $O$ :  $g(n) = O(f(n))$ . Іноді символ  $O$  використовують у рівняннях ліворуч і праворуч від знаку рівності, наприклад  $f(n, O(r_1(n)), \dots, O(r_k(n))) = g(n, O(s_1(n)), \dots, O(s_m(n)))$ . Такі рівняння слід розуміти таким чином: для кожного кортежу функцій  $(p_1, \dots, p_k)$ , такого, що  $p_1(n) = O(r_1(n)), \dots, p_k(n) = O(r_k(n))$  існує кортеж функцій  $(q_1, \dots, q_m)$ , такий, що  $q_1(n) = O(s_1(n)), \dots, q_m(n) = O(s_m(n))$  і для всіх  $n$  має місце  $f(n, p_1(n), \dots, p_k(n)) = g(n, q_1(n), \dots, q_m(n))$ . Слід зауважити, що у рівняннях, що включають символ  $O$  знак рівності має сенс односторонньої рівності: з  $O(g(n)) = O(f(n))$  в загальному випадку не випливає, що  $O(f(n)) = O(g(n))$ .

Укажемо кілька важливих властивостей  $O$ -оцінювання:

1.  $f(n) = O(f(n))$ ;
2.  $cO(f(n)) = O(f(n))$ , де  $c$  – деяка константа;
3.  $O(f(n)) + O(f(n)) = O(f(n))$ ;
4.  $O(O(f(n))) = O(f(n))$ ;
5.  $O(f(n))O(g(n)) = O(f(n)g(n))$ ;
6.  $O(cf) = O(f)$ , де  $c$  – деяка константа;
7.  $O(f + g)$  дорівнює домінанті  $O(f)$  і  $O(g)$ .

Можна сказати, що ефективність алгоритму безпосереднього підсумовування  $n$  елементів відповідає лінійній складності, оскільки його швидкодія, тобто кількість кроків, згідно із властивістю 1 становить  $O(n)$ .

Розглянемо таку задачу: для набору з  $n$  попарно нерівних відрізків підрахувати кількість усіх трійок, з яких можна отримати не вироджені трикутники.

Треба перевірити  $n(n-1)(n-2)$  варіантів, що відповідає кубічній складності  $O(n^3)$ .

У загальному випадку, якщо ефективність алгоритму визначається багаточленом фіксованого порядку  $k$  від обсягу вхідних даних  $n$ , то вона задовольняється оцінкою  $O(n^k)$ , не зважаючи, згідно з властивістю 2, на старший коефіцієнт і решту членів полінома.

$O$ -оцінювання дозволяє не використовувати конкретний обчислювальний пристрій для аналізу складності.

Нехай є деяка віртуальна машина  $M$ , яка обчислює функцію  $f$ , і визначені вхідні дані:  $x$  – деяке двійкове слово. Визначимо функцію

$T(M, x)$  як кількість операцій, що потрібні машині  $M$  для роботи на вхідному слові  $x$ . Справедлива така теорема, яка є наслідком з так званої теореми Блюма про прискорення (1967).

**Теорема 2.1.** *Існує така всюди визначена на двійкових словах обчислювана функція  $f$ , що будь-яку машину  $M$ , яка її обчислює, можна прискорити так: існує інша машина  $M'$  що також обчислює  $f$ , для якої виконується нерівність  $t_{M'}(n) \leq \log_2 t_M(n)$  для майже всіх  $n$  (тобто всіх натуральних чисел за виключенням елементів деякої скінченної множини натуральних чисел), де  $t_M(n) = \max_{|x| \leq n} T(M, x)$ ,  $|x|$  позначає довжину слова  $x$ .*

За більш вільного формулювання скажімо так: існує задача, для якої існує багато алгоритмів вирішення, але неможливо придумати найшвидший алгоритм, оскільки для кожного алгоритму вирішення існує ще швидший алгоритм, який її вирішує. Слід зауважити, що така властивість має місце не для кожної задачі.

### 2.2.1 Контрольні запитання

1. Що таке складність алгоритмів?
2. Опишіть алгоритм обміну значень двох змінних.
3. Опишіть пошук суми натуральних чисел від 1 до заданого  $n$ .

## 2.3 Складність задач

Поняття складності алгоритму пов'язане з поняттям складності задачі. Із теореми Блюма можна зробити висновок, що визначати складність задачі через складність найкращого алгоритму для її розв'язання – не найвдаліший спосіб.

Нехай необхідно побудувати алгоритм для розв'язання такого класу задач: обчислити значення виразу  $a_n x^n + \dots + a_1 x + a_0$  у точці  $x = b$ , де  $a_i \in Z$ ,  $b \in Z$ , а  $Z$  – множина цілих чисел. Наведемо алгоритм для даного класу задач.

Вхідні дані:  $(a_n, a_{n-1}, \dots, a_0)$ , де  $a_i \in Z$ ,  $b \in Z$ ,  $(a_n, a_{n-1}, \dots, a_0)$  – вектор з  $n + 1$  цілих чисел,  $b$  – ціле число.

Результат:  $r = f(b)$ , де  $f(b) = a_n b^n + \dots + a_1 b + a_0$ .

Змінні:  $i$ ,  $x$ ,  $r$  – цілого типу.

*Алгоритм:*

1. Покласти  $i$  рівним  $n$ ,  $r$  рівним 0,  $b$  рівним  $b$ .
2. Піднести до степеня  $i$ .
3. Помножити на степінь.
4. Покласти  $s$  рівній сумі  $r$  і добутку  $a_i$  на степінь.
5. Якщо  $i = 0$ , то  $r$  – результат (стоп). Інакше покласти  $i = i - 1$ , перейти до кроку 2.

Послідовність обчислень за алгоритмом описує вираз

$$a_n x^n + \dots + a_1 x + a_0.$$

Існує також інший алгоритм для розв'язання задач цього класу.

Вхідні дані: ті самі, що й у попередній задачі.

Результат: такий самий.

Змінні:  $r$ ,  $x$ ,  $i$  – цілого типу.

*Алгоритм:*

1. Покласти  $i$  рівним  $n$ ,  $x$  рівним  $b$ .
2. Покласти  $r$  рівним  $a_i$ .
3. Помножити  $r$  на  $x$ .
4. Покласти  $r$  рівним добутку.
5. Покласти  $i$  рівним  $i - 1$ .
6. Покласти  $r$  рівним  $r + a_i$ .
7. Якщо  $i = 0$ , то  $r$  – результат, інакше перейти до кроку 3.

Послідовність обчислень за цим алгоритмом можна пояснити виразом

$$(\dots(((a_n x + a_{n-1} x + a_{n-2})x + a_{n-3})x + \dots + a_1)x + a_0.$$

Такий метод обчислення значення полінома у точці називається схемою Горнера.

Виразимо складність дії піднесення до степеня  $i$  як  $i - 1$  операцій множення. Тоді складність другого алгоритму (за схемою Горнера) у вигляді кількості операцій додавання та множення дорівнюватиме  $2n$ , а для прямого алгоритму становитиме

$$\sum_{i=1}^n (i+1) + n = \frac{n+1}{2}(n+2) + n = \frac{n^2 + 3n + 2}{2} + n > \frac{n^2}{2} + 2n > 2n.$$

Таким чином, другий алгоритм ефективніший за перший.

Під складністю задачі розумітимемо час, необхідний для обчислення функції, за допомогою якої знаходиться її розв'язок. Під складністю розумітимемо складність алгоритму у найгіршому випадку. Оскільки не можна побудувати для кожної функції найкращої машини, що її обчислює, то введемо поняття класу складності.

**Означення 2.7.** Клас складності містить функції  $f$ , для яких існує машина  $M$ , що обчислює  $f$ , така, що функція  $T(M, x)$  за часом обмежена функцією  $t(n)$  із точністю до мультиплікативної константи, тобто

$$O(t(n)) = \{f \mid \exists M : (M \text{ обчислює } f \ \& \ (t_M(n) = O(t(n))))\},$$

де  $t(n)$  – порядок класу.

Класом задач складності  $t(n)$  назвемо сукупність задач, які розв'язуються за час порядку  $t(n)$ .

$O$ -оцінки виражають відносну швидкість алгоритму залежно від початкових даних. Розглянемо класи  $O$ -складності алгоритмів. Нехай  $N$  – кількість оброблюваних даних (табл. 2.1).

Розглянемо прийоми проведення аналізу алгоритмів і визначення їх складності. Тимчасова складність алгоритму може бути поражена, виходячи з аналізу його керувальних структур. Ми пам'ятаємо, що до керувальних структур належать лінійні та умовні вирази, цикли (табл. 2.2).

Якщо немає рекурсії й циклів, то всі керувальні структури можуть бути зведені до структур константної складності. Отже, і весь алгоритм також характеризується константною складністю. Тому визначення складності алгоритму в основному зводиться до аналізу циклів і рекурсивних викликів.

Розглянемо алгоритм обробки елементів масиву.

*Алгоритм:*

1. Для  $i = 1$  до  $N$  виконувати
2. початок
3. {Простий вираз}
4. кінець;

Складність цього алгоритму становить  $O(N)$ , оскільки тіло циклу виконується  $N$  разів, а складність тіла циклу дорівнює  $O(1)$ .

| Часова складність        | Опис програм                                                                                                                                                                                                                 |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $O(1)$                   | Більшість операцій у програмі виконується або один раз, або не більш ніж фіксовану кількість разів. Будь-який алгоритм, що завжди вимагає (незалежно від обсягу даних) одного й того самого часу, має константну складність. |
| $O(N)$                   | Час роботи програми обмежений лінійною функцією від $N$ . Властиво алгоритмам, які виконують над кожним елементом вхідних даних фіксовану кількість операцій.                                                                |
| $O(N^2), O(N^3), O(N^k)$ | Поліноміальна складність. $O(N^2)$ – квадратична складність, $O(N^3)$ – кубічна складність. Час роботи програми обмежений поліноміальною функцією.                                                                           |
| $O(\log(N))$             | Час роботи програми логарифмічний. Зустрічається зазвичай у програмах, які ділять велику задачу на маленькі й розв’язують одну з них.                                                                                        |
| $O(N \log(N))$           | Такий час роботи мають алгоритми, які ділять велику задачу на маленькі, а потім, розв’язавши їх, сполучають розв’язки.                                                                                                       |
| $O(2^N)$                 | Експоненціальна складність. Такі алгоритми найчастіше виникають унаслідок підходу, що має назву “метод грубої сили”.                                                                                                         |

Табл. 2.1: Часова складність

Якщо один цикл вкладено в інший та обидва мають однакову кількість повторень, то вся конструкція характеризується квадратичною складністю.

Розглянемо алгоритм.

*Алгоритм:*

1. Для  $i = 1$  до  $N$  виконувати
2. Для  $j = 1$  до  $N$  виконувати

| Керовальна структура                | Складність                                                                                                                   |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| Простий вираз                       | $O(1)$                                                                                                                       |
| Лінійний вираз $S_1, \dots, S_n$    | Домінанта $O(C_1), O(C_2), \dots, O(C_n)$ , де $C_1, C_2, \dots, C_n$ – складності обчислення виразів $S_1, S_2, \dots, S_n$ |
| Якщо <умова> то дія1 Інакше дія2    | Домінанта $O(C_1), O(C_2), O(C_3)$ , де $C_1, C_2, C_3$ – складність обчислень дій та умови, відповідно                      |
| Цикл із $n$ повтореннями тіла $S_1$ | $O(n \cdot C_1)$ , де $C_1$ – складність обчислення $S_1$                                                                    |

Табл. 2.2: Складність керувальних структур

3. початок

4. {Простий вираз}

5. кінець;

Його складність становить  $O(N^2)$ .

Оцінимо складність алгоритму, що має назву “Трійки Піфагора”. Суть задачі: знайти всі трійки натуральних чисел  $(x, y, z)$ , що задовольняють рівняння  $x^2 + y^2 = z^2$ . Піфагор знайшов формули, які в сучасному формалізмі можна записати як

$$x = 2n + 1, y = 2n(n + 1), z = 2n^2 + 2n + 1,$$

де  $n$  – ціле число. Наприклад:

Трикутники із такими сторонами є прямокутними. Перед аналізом складності програми, яка знаходить трійки Піфагора, зазначимо, що існують два способи аналізу складності: висхідний (від внутрішніх керувальних структур до зовнішніх) і нисхідний (від зовнішніх – до внутрішніх). Ми користуватимемося висхідним. Розглянемо алгоритм.

*Алгоритм:*

Вхід:  $n$ .

1.  $small = 1$

2. Поки  $small < n$  виконувати  
Початок

| <i>A</i> | <i>B</i> | <i>C</i> |
|----------|----------|----------|
| 3        | 4        | 5        |
| 5        | 12       | 13       |
| 8        | 15       | 17       |
| 7        | 24       | 25       |
| 20       | 21       | 29       |
| 12       | 35       | 37       |
| 9        | 40       | 41       |
| 28       | 45       | 53       |
| 11       | 60       | 61       |

Табл. 2.3: Трійки Піфагора

3.  $next = small$
4. Поки  $next \leq n$  виконуват  
Початок
5.  $last = next$
6. Поки  $last < n$  виконувати  
Початок
7. Якщо  $last \leq 2 * small$  і  $next \leq 2 * small$  і  $last * last = small * small + next * next$
8. то  
друк( $small$ ) друк( $next$ ) друк( $last$ )
9.  $last = last + 1$  кінець циклу  $last$
10.  $next = next + 1$  кінець циклу  $next$
11.  $small = small + 1$  кінець циклу  $small$
12. Кінець

Вихід: роздруковані значення  $small$ ,  $next$ ,  $last$ .

Легко перевірити, що складність алгоритму становить  $O(N^3)$ .

Проаналізувавши програму, можна дійти висновку, що основну оцінку складності забезпечують вкладені цикли. Тому для зменшення складності, перш за все, можна спробувати зменшити глибину вкладеності циклів. Далі слід розглянути можливість скорочення

кількості операторів у циклах з найбільшою глибиною вкладеності. Однак ці рекомендації не завжди можна виконати.

Оцінимо алгоритм бінарного пошуку ключа в упорядкованому масиві. Суть алгоритму: розглядаємо серединний елемент масиву й перевіряємо відповідність ключа до його значення. Якщо не вдається знайти відповідність, то порівнюємо ключ і значення серединного елемента, а потім переміщаємося в нижню половину списку елементів масиву, якщо ключ менше, і верхню – якщо більше. В обраній половині знову шукаємо середину та знову порівнюємо із ключем. Якщо не виходить, то знову ділимо надвоє поточний інтервал.

Розглянемо детальніше алгоритм. Вхід:  $low$ ,  $high$  – змінні, що задають нижню й верхню межі масиву,  $key$  – шукане число.

1. поки  $low \leq high$  виконувати
2. початок
3.  $mid = (low + high)/2$ ;
4.  $data = a[mid]$ ;
5. якщо  $key = data$
6. тоді початок  $search = mid$ ; закінчити роботу кінець
7. інакше якщо  $key < data$
8. то  $high = mid - 1$
9. інакше  $low = mid + 1$ ;
10. кінець;
11.  $search = -1$ ;

Перша ітерація циклу має справу з усім списком елементів. Кожна подальша ітерація ділить список навпіл. Наприклад, розмірами списків для послідовних кроків алгоритму є  $n$ ,  $n/2$ ,  $n/2^2$ , ...  $n/2^m$ . Урешті-решт знайдеться таке ціле  $m$ , що  $n/2^m < 2$  або  $n < 2^{m+1}$ . Оскільки  $m$  – перше ціле, для якого  $n/2^m < 2$ , то має бути правильним  $n/2^m - 1 \geq 2$  або  $2^m \leq n$ . Із цього випливає, що  $2 \leq n < 2^m + 1$ .

Візьмемо логарифм кожної частини нерівності й дістанемо  $m \leq \log_2(n) = x < m + 1$ . Значення  $m$  – найбільше ціле, що менше або дорівнює  $x$ . Отже, складність дорівнює  $O(n \log_2 n)$ .

Знайдемо вираз для часу, що необхідний програмі для обробки масиву з  $N$  елементами як функцію від  $N$ . Зазвичай нас цікавить

середній випадок – очікуваний час роботи програми на типових вхідних даних, і гірший – очікуваний час роботи програми на найгірших вхідних даних.

Через труднощі, пов’язані з проведенням аналізу часової складності алгоритму в середньому, часто доводиться задовольнятися оцінками для гіршого й кращого випадків. Ці оцінки, по суті, визначають нижню й верхню межі складності в середньому. У такому разі оцінка, одержана для найгіршого випадку, може служити хорошою апроксимацією складності в середньому.

Основним недоліком  $O$ -оцінювання є його надмірна грубість. Якщо алгоритм  $A$  виконує деяку задачу за  $0.001 \cdot N$ , тоді як для її розв’язання за допомогою алгоритму  $B$  потрібно  $1000 \cdot N$ , то  $B$  у мільйон разів швидше за  $A$ . Однак  $A$  та  $B$  мають одну й ту саму складність  $O(N)$ .

Можна умовно розділити задачі на прості, які мають поліноміальний час розв’язання (наприклад, розв’язання системи лінійних алгебраїчних рівнянь у раціональних числах); складні – задачі, які не розв’язуються за поліноміальний час, або алгоритм розв’язання за поліноміальний час не знайдений. Крім цього, існують принципово алгоритмічно нерозв’язні задачі.

Завершимо розгляд основ теорії складності тезою великого вченого.

**Теза Колмогорова.** Проблеми, які не можуть бути розв’язані без великого перебору, залишаються за межами можливостей машини на скільки завгодно високому ступені розвитку техніки й культури.

### 2.3.1 Контрольні запитання

1. Пояснити, що таке алгоритм.
2. Навести властивості алгоритму.
3. Навести класифікацію внутрішніх структур алгоритму.
4. Що таке складність задачі, алгоритму?
5. Навести тезу Колмогорова.
6. Як обчислювати складність задачі, алгоритму?

## 2.4 Задачі до розділу ОСНОВИ АЛГОРИТМІЗАЦІЇ

1. Напишіть програму пошуку елемента у масиві, обчисліть складність розробленого алгоритму.
2. Напишіть програму сортування масиву, обчисліть складність розробленого алгоритму.
3. Напишіть програму обчислення коренів квадратного рівняння, обчисліть складність розробленого алгоритму.
4. Напишіть програму визначення коефіцієнтів біному Ньютона, обчисліть складність розробленого алгоритму.
5. Напишіть програму заміни символів без використання спеціальних функцій, обчисліть складність розробленого алгоритму.

## 3 Оператори мови C#

У цьому розділі розглядаються оператори управління програмою в C# . Є такі категорії операторів:

- Оператори вибору до яких відносяться if та switch;
- Оператори циклування, які складаються з for, while, do while, та foreach;
- Оператори переходів, до яких відносяться break, continue, goto, return, та throw. Оператор throw відноситься до операторів обробки виняткових ситуацій і буде розглянутий пізніше.
- Порожній оператор
- Складений оператор

### 3.1 Прості оператори

#### 3.1.1 Порожній оператор

*Синтаксис:* ;

Використовується у випадку, коли за синтаксисом вимагається наявність оператора. Наприклад:

```
for (;;);
```

Остання крапка з комою – це порожній оператор. Він необхідний синтаксично, оскільки в операторі for після закриваючої круглої дужки обов'язково має стояти якийсь оператор (зазвичай складений, що містить кілька інших операторів, які задають тіло циклу). Оскільки в даному випадку відсутні будь-які оператори, то необхідно поставити крапку з комою – порожній оператор.

#### 3.1.2 Складений оператор

*Синтаксис:* { }

Він відіграє величезну роль при написанні програм, дозволяє синтаксично об'єднувати кілька операторів до одного цілого, що вважається одним оператором. В одному складеному може бути кілька простих операторів присвоювання:

```
{
    a = 4;
    b = 5;
    c = 6;
}
```

Складений оператор може містити оголошення та визначення. Як буде детальніше розглянуто далі, визначені змінні мають локальну область дії, тобто є доступними лише в межах даного складеного оператора. Складений оператор може містити інші складені оператори. Наприклад:

```
{
    int i = 5;
    j++;
    {
        float i = 4;
        i += 5;
    }
}
```

#### 3.1.3 Контрольні запитання

1. Що таке простий оператор?
2. Наведіть приклад використання простого оператора.
3. Що таке складений оператор?
4. Наведіть приклад використання складених операторів.

## 3.2 Умовний оператор

Умовний оператор відіграє ключову роль при описі алгоритмів, що мають структуру типу розгалуження.

*Синтаксис:*

```
if (<умовний вираз>) <оператор1> [else <оператор2>]
```

Умовний вираз повинен повертати значення true або false. Інші значення не допускаються. Оператор діє так: якщо <умовний вираз> виявиться істинним значенням (true), то виконується <оператор1>. В іншому випадку, за наявності необов'язкової конструкції [else <оператор2>], виконується <оператор2>. Наприклад:

```
if (a > 5) i++;  
else j++;
```

Далі наведено приклад програми, яка визначає, чи є введене число парним.

```
using System;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        int a;  
        Console.WriteLine("Enter_a_number:");  
        a = Convert.ToInt32(Console.ReadLine());  
        if (a % 2 == 0)  
        {  
            Console.WriteLine("Number_" + a + "_is_even");  
        }  
        else  
        {  
            Console.WriteLine("Number_" + a + "_is_odd");  
        } Console.ReadKey();  
    }  
}
```

Допускаються вкладені оператори if:

```
if (i > j) i++;  
else  
    if (i == j) j++;  
    else Console.WriteLine("the_end");
```

У такому випадку оператор else пов'язується з найближчим оператором if, у якого відсутня конструкція else. Останній фрагмент програми еквівалентний наступному:

```
if (i > j) i++;  
else  
{  
    if (i == j) j++;  
    else Console.WriteLine("the_end");  
}
```

Використання складеного оператора в такому випадку робить програму більш читабельною.

Розглянемо наступний приклад:

```
if (i < 0)  
    Console.WriteLine("less_than_zero");  
else  
    if (i == 0)  
        Console.WriteLine("zero");  
    else  
        if (i==1)  
            Console.WriteLine("one");  
        else  
            Console.WriteLine("greater_than_one");
```

Таку програмну конструкцію називають сходами(ladder) і записують у більш читабельному вигляді:

```
if (i < 0)
    Console.WriteLine("less_than_zero");
else if (i == 0)
    Console.WriteLine("zero");
else if (i == 1)
    Console.WriteLine("greater_than_zero");
else
    Console.WriteLine("greater_than_one");
```

Або в загальному випадку:

```
if (умова)
{ оператори }
else if (умова)
{ оператори }
else if (умова)
{ оператори }
.....
else
{ оператори }
```

У деяких мовах програмування є спеціальний оператор розгалуження для запису сходинок, але в C# для цього використовується вкладений оператор if.

Зауважимо, що на відміну від інших мов програмування, C# не містить ключового слова then. Ті, хто вивчає C# , уже знаючи мову Pascal, часто роблять помилки, наприклад у такому фрагменті програми:

```
if (x > 2)
then x = 2;
```

Умови в операторах if (а також while) мають братися в круглі дужки. Тому в наступному рядку буде помилка:

```
if x < 1 x = 1;
```

#### 3.2.1 Контрольні запитання

1. Що таке умовний оператор?
2. Наведіть синтаксис умовного оператора.
3. Наведіть приклад умовного оператора.

## 3.3 Тернарний оператор

Цим оператором можна замінювати прості за складністю оператори if-else. Тернарний оператор має таку структуру:

<логічний вираз> ? <вираз1> : <вираз2>

Спочатку обчислюється логічний вираз. Якщо він істинний, то обчислюється вираз1, в іншому випадку – обчислюється вираз2.

Як приклад наведемо програму перевірки числа на парність:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int a;
        Console.WriteLine("Enter a number:");
        a = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine(a % 2 == 0 ? "Even" : "Odd");
        Console.ReadKey();
    }
}
```

У наступному прикладі програма шукає максимум з двох чисел:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int a, b, max;
        Console.WriteLine("Enter the 1st number:");
        a = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter the 2nd number:");
        b = Convert.ToInt32(Console.ReadLine());
        max = a > b ? a : b;
        Console.WriteLine("Maximum: {0}", max);
    }
}
```

### 3.3.1 Контрольні запитання

1. Що таке тернарний оператор?
2. Синтаксис тернарного оператора.
3. Наведіть приклад тернарного оператора.

## 3.4 Оператор-перемикач switch

*Синтаксис:*

```
switch (<вираз>)  
{  
[case <константний вираз>: <оператор> break;]  
...  
[case <константний вираз>: <оператор> break;]  
[default: <оператор> break;]  
}
```

Вираз switch повинен бути цілого типу, такого як char, byte, int, і т.п., типу перерахування, або типу символьного рядка. Таким чином, вирази з плаваючою точкою, наприклад, не допускаються. Часто вираз switch є простою змінною. Константи в case повинні бути типу, сумісного з виразом. Крім того не може бути випадків, коли константи в case співпадають між собою. Обчислюється <вираз>, потім виконується оператор, що йде за умовою case, для якої значення константного виразу збігається зі значенням початкового <виразу>. Оператор break здійснює вихід із switch. Наприклад:

```
char c = ' ';  
int i = 0, j = 0, k = 0;  
switch (c)  
{  
    case 'a': i++; Console.WriteLine(i); break;  
    case 'b': j++; break;  
    default: k++; break;  
}
```

Залежно від значення c виконуються ті оператори, що визначаються умовою case, тобто якщо c == 'a', то виконується

```
i++; Console.WriteLine(i); break;
```

якщо c == 'b', то виконується j++; break; інакше виконується k++;

Послідовність в default виконується, якщо жоден константний вираз не збігається з виразом в switch. Частина default може бути і відсутня. Якщо її немає, то ніяких дій не виконуються у випадку якщо жоден константний вираз не збігається з виразом в switch.

В мові C# кожний case та default повинен завершатися оператором break.

Допускається, щоб кілька міток case відповідали одному фрагменту коду. Наприклад:

```
switch (x)  
{  
    case 1:  
    case 2:  
    case 3:  
        Console.WriteLine("case_1,_2,_3");  
        break;
```

```
        case 5:
            Console.WriteLine("case_5");
            break;
        default:
            Console.WriteLine("default");
            break;
    }
```

У цьому прикладі, якщо значення змінної  $x \in 1, 2$  або  $3$ , то виконується блок

```
Console.WriteLine("case_1,_2,_3");
break;
```

Оператор `switch` може бути вкладеним. Наприклад:

```
switch (x)
{
    case 1:
        Console.WriteLine("First_case_of_the_outer_switch");
        switch (y)
        {
            case 1:
                Console.WriteLine("First_case_of_the_nested_switch");
                break;
            case 2:
                Console.WriteLine("Second_case_of_the_inner_switch");
                break;
        } // end of inner switch
        break;
    case 2:
        Console.WriteLine("Second_case_of_the_outer_switch");
        break;
}
```

У частині `case` можна здійснювати прямі переходи за допомогою не тільки `break`, а й таких опцій:

- `goto case x` (перехід до наступного `case`);
- `goto default` (перехід до опції `default`);
- інші опції переходів: `return`, `throw`, `continue`, якщо `switch` вкладений в цикл, або `goto label`.

### 3.4.1 Контрольні запитання

1. Що таке оператор-перемикач `switch`?
2. Синтаксис оператора `switch`.
3. Які типи даних можуть використовуватися у виразі оператора `switch`?

4. Назвіть ключові слова у конструкції switch.
5. Назвіть оператори переходу.

## 3.5 Оператор циклу for

Оператор for використовується для запису алгоритмів із циклічною структурою і несе в мові C# особливе навантаження.

*Синтаксис:*

```
for([<початковий вираз>;<умовний вираз>;<вираз-приріст>])  
    <оператор>
```

де <початковий вираз>, <умовний вираз> та <вираз-приріст> будемо називати параметрами циклу.

Оператор діє так. Спочатку обчислюється <початковий вираз>, потім перевіряється <умовний вираз>, у випадку його істинності виконується <оператор>, а після цього – <вираз-приріст>. Однак <початковий вираз>, <умовний вираз>, <вираз-приріст> можуть бути відсутніми. Обов'язковою є лише наявність <оператора> (синтаксично мають бути наявні відповідні крапки з комою). Якщо умовного виразу немає, то значення його вважається істинним. Тому мінімальний оператор for має вигляд for (;;);.

Зазвичай <початковий вираз> на практиці використовується для початкової ініціалізації змінних, <умовний вираз> визначає умову виконання оператора, що задає тіло циклу, а <вираз-приріст> задає умову зміни змінних, що використовуються в циклі, наприклад, як лічильники. Нехай нам необхідно кілька разів надрукувати повідомлення “Hello, world”. Для цього використаємо оператор for:

```
for (i = 0; i < 10; i++)  
    Console.WriteLine("Hello, _world");
```

Відповідне повідомлення надрукується 10 разів.

Змінну i можна оголосити безпосередньо в циклі:

```
for (int i = 0; i < 10; i++)  
    Console.WriteLine("Hello, _world");
```

Тоді сфера видимості змінної i буде обмежена циклом.

Приріст може бути як позитивним, так і негативним. Наприклад, треба надрукувати числа від 200 до 100:

```
for (int i = 200; i >= 100; i--)  
    Console.WriteLine(i);
```

Наступний приклад обчислює перші 10 чисел Фібоначчі.

Нагадування. Перші два числа рівні 1. Поточне число дорівнює сумі двох попередніх.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
        {
            Console.WriteLine(prevFib);
            int newFib = prevFib + curFib;
            prevFib = curFib;
            curFib = newFib;
        }
    }
}
```

Можна використовувати декілька змінних в початковому виразі та прирості. У такому разі окремі вирази розділяються комою.

```
for (int i = 200, j = 10; i + j < 1000; i += 20, j -= 5)
    Console.WriteLine(i);
```

Можна використовувати і більше двох змінних, але на практиці це не рекомендується робити, тому що такі цикли становляться важкими для розуміння, а читабельність програм значно погіршується.

Розглянемо таку задачу. Нехай необхідно надрукувати рядок довжиною `WIDTH`, у якому спочатку міститься `x0` символів '-', потім `w` символів '\*', і до кінця рядка – знову символи '-':

Це можна запрограмувати так:

```
for (x = 0; x < x0; ++x) Console.Write('-');
for (; x < x0 + w; x++) Console.Write('*');
for (; x < WIDTH; ++x) Console.Write('-');
Console.WriteLine();
```

або

```
for (x = 0; x < WIDTH; x++)
    Console.Write(x < x0 ? '-' : x < x0 + w ? '*' : '-');
Console.WriteLine();
```

Допоміжні змінні, які не мають змісту (наприклад, лічильник повторення циклу, що не використовується в самому тілі циклу), прийнято позначати однією літерою, наприклад `i`, `j`. Можна навіть написати так:

```
int _;
for (_ = 0; _ < 10; _++)
    Console.WriteLine("{0}", _);
```

(підкреслювання в ідентифікаторах - рівноправна літера).

Якщо залишити умовний вираз пустим, то отримаємо нескінченний цикл:

```
for (;;)
{
    оператори
}
```

У наведеному прикладі всі три параметри циклу є пустими. Такі цикли можуть використовуватися, наприклад, для програми, яка очікує якісь команди користувача і потім виконує їх. Насправді такі цикли не працюють нескінченно довго і в тілі циклу є оператори для його завершення.

Ще одним випадком є цикли, які не містять тіла циклу. Наприклад:

```
for (x = 0; x < 100; s += x, x++) ;
Console.WriteLine(s);
```

Тілом циклу є пустий оператор ; . У таких циклах корисна робота виконується в параметрах. Цикл можна написати більш коротко

```
for (x = 0; x < 100; s += x++) ;
```

#### 3.5.1 Контрольні запитання

1. Що таке оператор for?
2. Синтаксис оператора for.
3. Сфера видимості лічильника циклу.
4. Скільки змінних можна використовувати в початковому виразі та прирості?
5. Які параметри циклу for можуть бути порожніми?
6. Чи може бути тіло циклу порожнім?
7. Наведіть приклад використання циклу for.

## 3.6 Оператор foreach

Цикл `foreach` використовується для ітерації по елементах колекції. В якості колекції можуть використовуватися масиви, рядки, або інші сукупності об'єктів, які будуть розглянуті пізніше.

*Синтаксис:*

```
foreach (<тип><змінна> in <колекція>) <оператори>
```

Наприклад:

```
foreach (char x in "abcd")  
    Console.WriteLine(x);
```

Змінна `x` “пробігає” по символах рядку і виводить їх на консоль. Точніше кажучи, коли цикл починається, то `x` приймає значення першого елемента рядка, а саме літери ‘а’. На кожному наступному кроці циклу, значення змінної змінюється на наступний символ. Коли всі символи вичерпані, то цикл завершується.

Важливо знати, що значення змінної `x` не можна міняти.

### 3.6.1 Контрольні запитання

1. Що таке оператор `foreach`?
2. Синтаксис оператору `foreach`.
3. Чи може змінна оператору `foreach` змінювати своє значення?

## 3.7 Оператори do-while, while

Поряд з оператором `for` для запису циклічних алгоритмів використовуються також оператори `do-while` та `while`.

*Синтаксис:*

```
while (<вираз>) <оператор>
```

Дія: поки істинний <вираз>, виконується <оператор>.

*Синтаксис:*

```
do <оператор> while (<вираз>)
```

Дія: виконується <оператор>, поки істинний <вираз>.

Для уявлення про відмінність у цих операторах розглянемо два простих фрагменти програм:

```
int i = 0;
while (i > 0)
{
    i--;
    Console.WriteLine("{0}", i);
}
```

```
int i = 0;
do
{
    i--;
    Console.WriteLine("{0}", i);
}
while (i > 0);
```

Перший фрагмент нічого не виводить, а другий – виводить значення  $-1$ . Дійсно, у другому фрагменті спочатку виконується тіло оператора, а вже потім перевіряється умовний вираз. Якщо він хибний, то оператор більше не виконується.

Цикл `do-while` виконується принаймні один раз, тоді як цикл `while` може не виконуватися жодного разу.

При використанні циклів не варто забувати ініціалізувати змінні, що виступають як лічильники. Розглянемо фрагмент програми, у якій деяке повідомлення має друкуватись 10 разів:

```
using System;
class Program
{
    static void Main()
    {
        int i = 0;
        while (i < 10)
        {
            Console.WriteLine("{0}-th_time", i + 1);
            i++;
        }
        while (i < 10)
        {
            Console.WriteLine("{0}-th_time", i * i + 1);
            i++;
        }
    }
}
```

Тут змінна `i` не проініціалізована перед виконанням другого циклу й містить не  $0$ , а інше значення. Цикл може виконуватись не  $10$ , а довільну кількість разів (у тому числі й  $0$ ). Тому у відповідному рядку необхідно написати `int i=0;`

У цьому фрагменті програми було б ще краще використовувати цикл `for`, у якому всі операції над індексом циклу зібрані в заголовку циклу:

```
for (i=0; i<10; i++) Console.WriteLine(...);
```

Крім того треба звернути увагу на те, що в таких циклах більша вірогідність допустити помилку та отримати нескінченний цикл.

### 3.7.1 Контрольні запитання

1. Що таке оператор `do-while`?
2. Що таке оператор `while`?
3. Відмінність між `do-while` та `while`.
4. Синтаксис оператора `do-while`.
5. Синтаксис оператора `while`.
6. Приклад оператора `do-while`.
7. Приклад оператора `while`.

## 3.8 Оператор *continue*

*Синтаксис:* `continue`;

Передає керування на наступну ітерацію в циклах. Нагадаємо, що в циклах `while` та `do` наступна ітерація починається з обчислення умовного виразу, а в операторі `for` – виразу-приросту (а потім умовного виразу). Розглянемо фрагмент програми:

```
i = 10;
while (i > 0)
{
    i--;
    if (i % 2 == 0) continue;
    Console.WriteLine(i);
}
```

У цьому фрагменті надруковані непарні значення змінної `i` від 9 до 1. Якщо значення змінної `i` виявляється парним, то виконується оператор `continue`, який перериває виконання циклу, і виконується наступна ітерація.

### 3.8.1 Контрольні запитання

1. Що таке оператор `continue`?

2. Синтаксис оператора continue.
3. Коли використовується оператор continue?
4. Приклад використання оператора continue.

## 3.9 Оператор break

*Синтаксис:* break;

Перериває виконання операторів циклу та switch. Як зазначалося раніше, для того, щоб забезпечити виконання лише одного оператора – case або default в операторі switch, необхідно в кінці кожної групи операторів case ставити break.

Розглянемо приклад використання оператора в циклі:

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 20; j++)
    {
        if (j % 5 == 4) break;
    }
```

При виконанні умови  $j \% 5 == 4$  відбувається переривання виконання внутрішнього циклу. При цьому зовнішній цикл за  $i$  продовжує виконуватись.

Якщо оператор switch знаходиться всередині циклу, то break, який стоїть в case, діє тільки в межах оператора switch і не здійснює вихід із цикла.

У циклі може стояти кілька операторів break, але велика кількість break робить код програми погано зрозумілим.

Наведемо приклад програми, яка знаходить перше число, що більше 10 і кратне 7. Коли ми знайшли таке число, немає сенсу далі виконувати дії і ми виходимо з циклу, використовуючи оператор break:

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int i;
        for (i = 10; ; i++)
        {
            if (i % 7 == 0)
                break;
        }
        Console.WriteLine("i={0}", i);
        Console.ReadKey();
    }
}
```

### 3.9.1 Контрольні запитання

1. Що таке оператор break?
2. Синтаксис оператору break.
3. Коли використовується оператор break?
4. Приклад використання оператора break.

## 3.10 Оператор goto

*Синтаксис:*

goto <мітка>

...

<мітка>:<оператор>

Передає керування на мітку. Міткою може бути будь-який ідентифікатор. Якщо потрібно вийти з кількох циклів за великого рівня вкладеності (більше 2), єдина можливість – це оператор goto. Можна ввійти за допомогою goto у блок, тіло циклу, оператор switch. Наприклад:

```
int i, j, k;
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        for (k = 0; k < 100; k++)
            if (error(i, j, k)) goto exit;
exit: ;
```

У цьому фрагменті у випадку, коли значення деякого методу error виявиться ненульовим, переривається виконання всіх циклів, і керування передається на мітку exit, за якою стоїть порожній оператор.

### 3.10.1 Контрольні запитання

1. Що таке оператор goto?
2. Синтаксис оператора goto.
3. Коли використовується оператор goto?
4. Приклад використання оператора goto.

## 3.11 Оператор return

*Синтаксис:* return <константний вираз>

Повертає керування у викличний метод. Розглянемо метод Main:

```
static int Main()  
{  
    return 0;  
}
```

Метод Main може повертати значення типу int. Тому в тілі цього методу міститься оператор return. Загалом, будь-який метод, що має тип результату, відмінний від void, має містити оператор return. Значення, що повертається оператором return, на практиці часто використовується для видачі певної інформації. Наприклад, якщо метод відпрацював нормально, то можна повернути нуль.

### 3.11.1 Контрольні запитання

1. Що таке оператор return?
2. Синтаксис оператора return.
3. Коли використовується оператор return?
4. Приклад використання оператора return.

## 3.12 Задачі до розділу ОПЕРАТОРИ МОВИ C#

1. Напишіть програму обчислення суми ряду цілих чисел з використанням різних типів циклів.
2. Напишіть програму визначення парності та непарності числа з використанням різних операторів вибору.
3. Напишіть програму довільного циклічного процесу з використанням операторів break та continue.

# 4 Булеві функції та вирази

## 4.1 Булеві функції

Булеві функції – це функції однієї або кількох змінних, значення і кожен з аргументів яких можуть приймати лише одне з двох значень: 0 або 1. Значення 1 може трактуватися “істина”, а значення 0 – як “хибність”. При такій інтерпретації булаві функції можуть використовуватися для визначення істинності певного висловлювання через істинність інших висловлювань.

Розглянемо приклад. Якщо  $A$ ,  $B$  – висловлювання (речення, які можна охарактеризувати як істинні або хибні, наприклад “число 1 менше за число 2” або “число 1 більше за число 2”), то висловлювання “ $A$  та  $B$ ” є істинним тоді і тільки тоді, коли висловлювання  $A$  істинне та висловлювання  $B$  істинне. Нехай  $x$  – значення, яке рівне 0, якщо  $A$  істинне, або 1, якщо  $A$  хибне. Аналогічно, нехай  $y$  – значення, яке рівне 0, якщо  $B$  істинне, або 1, якщо  $B$  хибне, та нехай  $z$  – значення, яке рівне 0, якщо висловлювання “ $A$  та  $B$ ” істинне, або 1, якщо висловлювання “ $A$  та  $B$ ” хибне. Тоді значення  $z$  можна однозначно визначити, знаючи значення  $x$  та  $y$ , а саме:

- якщо  $x = 0$  і  $y = 0$ , то  $z = 0$ ;
- якщо  $x = 0$  і  $y = 1$ , то  $z = 0$ ;
- якщо  $x = 1$  і  $y = 0$ , то  $z = 0$ ;
- якщо  $x = 1$  і  $y = 1$ , то  $z = 1$ .

Вказані 4 випадки вичерпують усі можливі набори значень  $x$  та  $y$ , тому наведені правила визначають функцію від двох аргументів, які можуть приймати значення 0 або 1, яка сама може приймати значення 0 або 1. Цю функцію називають булевою операцією I (кон’юнкцією) і часто позначають як  $\wedge$ . Таким чином,  $z = x \wedge y$  (символ операції  $\wedge$  ставлять між її аргументами, що є інфіксною формою запису, яка характерна і для запису арифметичних операцій). Дана операція має властивості  $0 \wedge 0 = 0$ ,  $0 \wedge 1 = 0$ ,  $1 \wedge 0 = 0$ ,  $1 \wedge 1 = 1$ . Фактично, значення  $x \wedge y$  можна виразити, застосувавши арифметичну операцію множення:  $x \wedge y = x \cdot y$  (при  $x \in \{0, 1\}$ ).

та  $y \in \{0, 1\}$ ). Проте відмінність арифметичної операції множення від операції  $\wedge$  полягає в тому, що арифметичне множення визначне для будь-яких цілих (раціональних, дійсних) чисел, а булева операція  $\wedge$  – лише для елементів множини  $\{0, 1\}$  і крім того, вона завжди приймає значення лише в цій множині. Слід зауважити, що існують і інші способи виразити кон'юнкцію через числові операції, наприклад, її можна виразити через операцію взяття мінімуму двох чисел таким чином:  $x \wedge y = \min x, y$  при  $x \in \{0, 1\}$  та  $y \in \{0, 1\}$ .

Розглянемо інший приклад. Якщо  $A$  – висловлювання, то висловлювання “не  $A$ ” (заперечення  $A$ ) є істинним тоді і тільки тоді, коли висловлювання  $A$  хибне. Нехай  $x$  – значення, яке рівне 0, якщо  $A$  істинне, або 1, якщо  $A$  хибне. Аналогічно, нехай  $y$  – значення, яке рівне 0, якщо висловлювання “не  $A$ ” істинне, або 1, якщо висловлювання “не  $A$ ” хибне. Тоді значення  $y$  можна однозначно визначити, знаючи значення  $x$ , а саме:

- якщо  $x = 0$ , то  $y = 1$ ;
- якщо  $x = 1$ , то  $y = 0$ .

Наведені правила визначають функцію від одного аргументу, який може приймати значення 0 або 1, яка сама може приймати значення 0 або 1. Цю функцію називають *булевою операцією заперечення* і часто позначають горизонтальною рисою згори над її аргументом:  $z = \bar{x}$ . Дана операція має властивості  $\bar{\bar{0}} = 0$ ,  $\bar{\bar{1}} = 1$ . Значення  $\bar{x}$  можна виразити, застосувавши до  $x$  арифметичну операцію віднімання від одиниці:  $\bar{x} = 1 - x$  (при  $x \in \{0, 1\}$ ). Відмінність арифметичної операції від булевої аналогічна поясненій вище відмінності  $\wedge$  на арифметичного множення.

Математичне визначення булевої функції наведено нижче.

**Означення 4.1** ([?]). *Булевою функцією  $n$  змінних (де  $n \geq 1$  – натуральне число) називається функція, яка відображає множини  $\{0, 1\}^n$  (декартовий степінь множини  $\{0, 1\}$ , що складається з усіх можливих  $n$ -ок елементів множини  $\{0, 1\}$ ) в множину  $\{0, 1\}$ .*

Прикладом булевої функції 2 змінних є булева операція  $\wedge$ .

Булеву функцію  $n$  змінних, яка приймає значення 1 для всіх можливих наборів значень її аргументів будемо позначати як  $\mathbf{1}_n$ , а булеву функцію  $n$  змінних, яка приймає значення 0 для всіх можливих наборів значень її аргументів будемо позначати як  $\mathbf{0}_n$ . Наприклад,  $\mathbf{0}_1(0) = 0$ ,  $\mathbf{0}_1(1) = 0$ ,  $\mathbf{1}_2(0, 0) = 1$ ,  $\mathbf{1}_2(0, 1) = 1$ ,  $\mathbf{1}_2(1, 0) = 1$ ,  $\mathbf{1}_2(1, 1) = 1$ .

| $x_1$ | $x_2$ | ... | $x_n$ | $f(x_1, x_2, \dots, x_n)$ |
|-------|-------|-----|-------|---------------------------|
| 0     | 0     | ... | 0     | $f(0, 0, \dots, 0)$       |
| 0     | 0     | ... | 1     | $f(0, 0, \dots, 1)$       |
| ...   | ...   | ... | ...   |                           |
| 1     | 1     | ... | 1     | $f(1, 1, \dots, 1)$       |

Табл. 4.1: Таблиця істинності булевої функції.

В силу скінченності множини  $\{0, 1\}^n$ , будь-яку булеву функцію  $n$  змінних можна задати табличним способом, вказавши її значення на кожному можливому наборі значень аргументів (табл. 4.1).

Таблицю такого типу називають *таблицею істинності* булевої функції  $f$ . В такій таблиці рядки мають пробігати по одному разу усі можливі комбінації значень аргументів функції (щоб функція була однозначно визначеною). У загальному випадку, множина  $\{0, 1\}^n$  (область визначення булевої функції  $n$  змінних містить) містить

$$|\{0, 1\}^n| = |\{0, 1\}|^n = 2^n$$

елементів, тому в таблиці істинності функції  $n$  змінних має бути  $2^n$  рядків.

Наприклад, таблиця істинності булевої операції заперечення (табл. 4.2) містить 2 рядки, а таблиця істинності операції  $\wedge$  (табл. 4.3) містить 4 рядки.

| $x_1$ | $\bar{x}$ |
|-------|-----------|
| 0     | 1         |
| 1     | 0         |

Табл. 4.2: Таблиця істинності заперечення.

Слід зауважити, що розмір таблиці необхідний для задання булевої функції  $n$  аргументів швидко росте зі збільшенням значення  $n$  (наприклад, 32 рядки при  $n = 5$ , 1024 рядки при  $n = 10$ ).

### 4.1.1 Контрольні запитання

1. Що таке булева функція?

| $x_1$ | $x_2$ | $x_1 \wedge x_2$ |
|-------|-------|------------------|
| 0     | 0     | 0                |
| 0     | 1     | 0                |
| 1     | 0     | 0                |
| 1     | 1     | 1                |

Табл. 4.3: Таблиця істинності кон'юнкції.

2. Яких значення приймає булева функція?
3. Що таке кон'юнкція?
4. Що таке диз'юнкція?
5. Що таке логічне заперечення?
6. Що таке таблиця істинності?

## 4.2 Булеві вирази

Існують інші способи задання булевих функцій, досить поширеним з яких є *булеві вирази*. Припустимо, що є перелік деяких відомих (базових) булевих функцій, для яких зафіксовані позначення. Тоді через них шляхом підстановки можна виразити нові булеві функції. Наприклад, за допомогою операції  $\wedge$  можна виразити таку булеву функцію 3 аргументів:  $f(x_1, x_2, x_3) = (x_1 \wedge x_2) \wedge x_3$  ( $\wedge$  застосовується до результату  $\wedge$  від перших двох аргументів та до третього аргументу). З того, що  $x \wedge y = x \cdot y$  при  $x \in 0, 1$  та  $y \in 0, 1$  випливає, що для операції  $\wedge$  виконується закон асоціативності (так само як для арифметичної операції множення):  $(x_1 \wedge x_2) \wedge x_3 = x_1 \wedge (x_2 \wedge x_3)$ . Тому дужки у правій частині визначення функції  $f$  можна опустити:  $f(x_1, x_2, x_3) = x_1 \wedge x_2 \wedge x_3$ . Значення  $f(x_1, x_2, x_3)$  можна трактувати як “ $x_1$  та  $x_2$  та  $x_3$ ”.

Отже, через деякі базові булеві функції можна виражати інші булеві функції. Але виникає питання: які функції слід узяти за базові, щоб через них можна було виразити будь-яку булеву функцію? Виявляється, що булевої операції заперечення, кон'юнкції, та констант 0, 1 достатньо, щоб виразити будь-яку булеву функцію. В загальному випадку, систему (базових) булевих функцій, через

які можна виразити будь-яку булеву функцію, називають *функціонально повною*. Таким чином, заперечення і кон'юнкція складають функціонально повну систему, але слід зауважити, що це єдина функціонально повна система (існує нескінченно багато інших систем булевих функцій, які мають цю властивість).

Наприклад, *булева операція АБО (диз'юнкцію)* має два аргументи, позначається символом  $\vee$  і задається таблицею істинності, наведеною нижче (таб. 4.4).

| $x_1$ | $x_2$ | $x_1 \vee x_2$ |
|-------|-------|----------------|
| 0     | 0     | 0              |
| 0     | 1     | 1              |
| 1     | 0     | 1              |
| 1     | 1     | 1              |

Табл. 4.4: Таблиця істинності диз'юнкції.

Диз'юнкцію можна виразити через заперечення і кон'юнкцію таким чином:  $x_1 \vee x_2 = \overline{\overline{x_1} \wedge \overline{x_2}}$ . Справедливість цього виразу можна перевірити за допомогою таблиць істинності диз'юнкції, кон'юнкції та заперечення. Цей вираз пов'язаний з такою логічною інтерпретацією: якщо  $A$  – висловлювання, істинність якого визначається  $x_1$ , а  $B$  – висловлювання, істинність якого визначається  $x_2$ , то висловлювання “ $A$  або  $B$ ” істинне тоді і тільки тоді, коли істинне заперечення висловлювання “не  $A$  та не  $B$ ”. Слід зауважити, що як і кон'юнкцію, диз'юнкцію не складно виразити через числові операції:  $x \vee y = \max x, y = x + y - xy$  при  $x \in \{0, 1\}$  та  $y \in \{0, 1\}$ .

Хоча диз'юнкція і виражається через заперечення і кон'юнкцію, її, зазвичай, включають до числа базових булевих операцій, через які виражають інші булеві функції. Це є підставою визначення поняття булевого виразу, наведеного нижче.

**Означення 4.2 ([?]).** Нехай дано скінчений перелік різних символів – імен змінних  $x_1, \dots, x_n$ . Поняття булевого виразу від змінних  $x_1, \dots, x_n$  визначається індуктивним чином:

(1) константи 0, 1 та змінні  $x_1, \dots, x_n$  є булевими виразами від змінних  $x_1, \dots, x_n$ ;

(2) якщо  $E$  та  $E'$  – булеві вирази від змінних  $x_1, \dots, x_n$ , то вирази  $(E \vee E')$ ,  $(E \wedge E')$ , та  $\overline{E}$  є булевими виразами від змінних  $x_1, \dots, x_n$ ;

(3) кожний булевий вираз можна отримати за допомогою скінченного числа застосувань правил (1) та (2).

Наприклад,  $x_1 \vee x_2$ ,  $(x_1 \vee 0) \vee 1$ ,  $(x_1 \vee x_2) \vee \overline{(x_3 \wedge x_4)}$  є булевими виразами (від змінних  $x_1, x_2, x_3, x_4$ ).

Для позначення довільного булевого виразу від  $x_1, x_2, \dots, x_n$  використовують позначення  $\phi(x_1, x_2, \dots, x_n)$ .

**Означення 4.3** ([?]). Для заданого булевого виразу  $\phi(x_1, x_2, \dots, x_n)$ , булева функція, зображена  $\phi$  – це булева функція  $f_\phi : \{0, 1\}^n \rightarrow \{0, 1\}$ , така, що для кожного кортежу  $(b_1, b_2, \dots, b_n) \in \{0, 1\}^n$ ,  $f_\phi(b_1, b_2, \dots, b_n)$  є значенням, отриманим шляхом підстановки  $b_1$  замість  $x_1$ ,  $b_2$  замість  $x_2, \dots, b_n$  замість  $x_n$  у виразі  $\phi(x_1, x_2, \dots, x_n)$  та спрощенні отриманого виразу до значення 0 або 1 за допомогою таких правил:  $0 \vee 0 = 0$ ,  $0 \vee 1 = 1$ ,  $1 \vee 0 = 1$ ,  $1 \vee 1 = 1$ ,  $0 \wedge 0 = 0$ ,  $0 \wedge 1 = 0$ ,  $1 \wedge 0 = 0$ ,  $1 \wedge 1 = 1$ ,  $\bar{0}$ ,  $\bar{1}$ .

Легко бачити, що кожний булевий вираз зображає єдину булеву функцію. Проте поняття булевого виразу відрізняється від поняття булевої функції: булевий вираз є лише символічним записом, тому 0 та  $x_1 \wedge \bar{x}$  – різні булеві вирази від змінної  $x_1$ . Водночас вони зображають одну і ту ж булеву функцію  $\mathbf{0}_1$  (для функції суттєвими є лише відповідність між значеннями аргументів та результатом).

**Означення 4** [1]. Два булевих вирази називаються еквівалентними, якщо вони зображають одну і ту ж булеву функцію.

В цілому, для будь-якої булевої функції існує нескінченно багато різних булевих виразів, які її зображають. Тому задача встановлення, чи рівні функції, зображені двома даними і різними булевими виразами, не є простою. Для її вирішення можна побудувати таблиці істинності функцій, зображених цими виразами і порівняти значення у відповідних рядках (справді, це те ж саме, що обчислити і порівняти значення обох функцій, користуючись їх виразами для всіх допустимих значень аргументів). Інший спосіб вирішення цієї задачі полягає у використанні властивостей булевих операцій диз'юнкції, кон'юнкції та заперечення для виконання еквівалентних перетворень виразів. Якщо за допомогою таких перетворень обидва вихідні вирази будуть приведені до одного і того ж виразу, то можна зробити висновок, що вони зображають одну і ту ж функцію (є еквівалентними).

Основні властивості операцій кон'юнкції, диз'юнкції, заперечення:

При програмуванні мовою C# булеві вирази можна реалізовувати за допомогою булевих операцій `&&` (кон'юнкція), `||` (диз'юнкція)

$$x \vee 0 = x, x \wedge 1 = x$$

$$x \vee 1 = 1, x \wedge 0 = 0$$

$$x \vee y = y \vee x, x \wedge y = y \wedge x \text{ (комутативність)}$$

$$(x \vee y) \vee z = x \vee (y \vee z), (x \wedge y) \wedge z = x \wedge (y \wedge z) \text{ (асоціативність)}$$

$$x \vee x = x, x \wedge x = x \text{ (ідемпотентність)}$$

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \text{ (дистрибутивність)}$$

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \text{ (дистрибутивність)}$$

$$x \vee (x \wedge y) = x, x \wedge (x \vee y) = x \text{ (поглинання)}$$

$$x \vee \bar{x} = 1, x \wedge \bar{x} = 0$$

$$\bar{\bar{x}} = x \text{ (інволюція)}$$

$$\overline{x \vee y} = \bar{x} \wedge \bar{y}, \overline{x \wedge y} = \bar{x} \vee \bar{y} \text{ (закони Де-Моргана)}$$

Табл. 4.5: Властивості булевих операцій

та ! (заперечення). Проте замість числових значень 0 і 1 аргументами і значеннями булевих операцій в C# є спеціальні значення false, true, які належать типу даних bool.

При реалізації булевих виразів у C# слід звернути увагу на таку особливість: якщо під час виконання програми трапляється операція &&, то спочатку обчислюється перший аргумент цієї операції, і якщо він має значення false, то другий не обчислюється, а результатом операції є false (оскільки результатом кон'юнкції хибного значення з будь-яким значенням є хибне значення). В іншому випадку, другий аргумент обчислюється і його значення стає значенням операції. Аналогічно, якщо перший аргумент операції || має значення true, то другий не обчислюється, а результатом операції є true. Інакше, обчислюється другий аргумент і його значення стає значенням операції. Через зазначену особливість у наведеному нижче умовному операторі не буде відбуватися спроба ділення на нуль, якщо змінна x (цілого типу) має значення нуль:

```
if (x!=0 && y/x>1) { }
```

Водночас, виконання наведеного нижче оператора призведе до спроби ділення на нуль (помилки під час виконання програми), якщо змінна x (цілого типу) має значення нуль:

```
if (y/x>1 && x!=0) { }
```

Таким чином, у цілому операції && та || в мові C# не можна вважати комутативними, на відміну від математичних операцій

кон'юнкції та диз'юнкції. Проте їх можна вважати комутативним, якщо вони застосовуються до змінних типу `bool`, а не до складних виразів. Наприклад, якщо визначені булеві змінні:

```
bool b1 = true;
bool b2 = false;
```

то значення виразів `b1 && b2` і `b2 && b1` визначені і однакові.

У тексті програми операція `&&` має більш високий пріоритет, ніж `||`, а операція заперечення `!` має більш високий пріоритет, ніж `&&`. Це дозволяє записувати булеві вирази, не використовуючи зайвих дужок.

Іноді виникає необхідність створювати програми, які працюють з великою або наперед невідомою кількістю булевих змінних. У таких випадках можна скористатись масивом булевих змінних.

Розглянемо фрагмент програми на мові `C#`, яка перебирає усі комбінації значень  $n$  булевих змінних, що є елементами масиву (який можна використати при формуванні таблиці істинності деякої булевої функції  $n$  змінних):

```
using System;
class Program
{
    static void Main(string[] args)
    {
        int n = 3;
        // визначимо масив з n булевих змінних
        // зауважимо, що спочатку усі x[i] матимуть значення false
        bool[] x = new bool[n];
        // перебір усіх 2N комбінацій значень елементів масиву x
        while (true)
        {
            // виведемо на екран поточну комбінацію,
            // позначаючи false як 0 і true як 1
            for (int i = 0; i < n; i++)
                Console.Write(x[i] ? '1' : '0');
            Console.WriteLine();
            // сформуємо чергову комбінацію
            // для цього будемо розглядати x як зображення
            // n-цифрового двійкового числа
            // (x[i]=true, якщо i-та цифра числа = 1, інакше x[i]=false)
            // якщо поточна комбінація – зображення числа N,
            // то наступною комбінацією зробимо зображенням числа N+1
            // для додавання одиниці використаємо простий варіант
            // алгоритму додавання в стовпчик
            bool carry = true; // індикатор переносу
            // цикл по цифрах, починаючи з останньої (молодший розряд)
            for (int i = n - 1; i >= 0; i--)
            {
                if (x[i] && carry)
                    x[i] = false;
                else
                {

```

```

        x[i] = x[i] || carry;
        carry = false;
    }
}
// якщо маємо переніс після циклу, це означає,
// що сума не може бути зображена N-цифровим двійковим
// числом, тобто більша за  $2^N - 1$ ;
// оскільки на кожній ітерації циклу ми додавали 1
// починаючи з масиву x, що зображає число нуль, то це
// означає, що ми вже перебрали  $2^N$  комбінацій
// і перебір необхідно завершити
if (carry) break;
}
}
}

```

Наприклад, при  $n = 3$  програма виводить на екран таку послідовність з  $2^3 = 8$  рядків:

```

000
001
010
011
100
101
110
111

```

### 4.2.1 Контрольні запитання

1. Що таке булеві вирази?
2. Пріоритетність операторів у булевих виразах.
3. Основні властивості операцій кон'юнкції, диз'юнкції, заперечення.

## 4.3 Задачі для розділу БУЛЕВІ ФУНКЦІЇ ТА ВИРАЗИ

1. Написати програму, яка виводить на консоль таблиці істинності булевих функцій, зображених такими булевими виразами:
  - a)  $(x_1 \vee \overline{x_2}) \wedge x_3$
  - b)  $((x_1 \vee (x_3 \vee x_3)) \vee x_4) \vee x_5$
  - c)  $\overline{((x_1 \vee x_3) \wedge (x_3 \vee x_4))} \wedge x_5$ .

2. Написати програму, яка отримує з клавіатури цілі числа  $a_1$ ,  $a_2$ ,  $a_3$  і виводить на екран число 1, якщо принаймні два числа серед  $a_1$ ,  $a_2$ ,  $a_3$  є додатними, і виводить 0 в іншому випадку.
3. Написати програму, яка за заданою таблицею істинності булевої функції трьох змінних (введеною з клавіатури) знаходить і виводить на екран один з можливих булевих виразів, який зображає функцію, що має цю таблицю істинності.

# 5 Зображення чисел у комп'ютері

Різнноманітні дані, з якими можуть працювати програми, певним чином подаються у пам'яті комп'ютера. Знання цього подання буває важливим для уникнення і знаходження помилок у програмах. Розглянемо таку програму:

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("{0}", 1 / 2.0 f - 0.5 f);
        Console.WriteLine("{0}", 1 / 5.0 f - 0.2 f);
    }
}
```

Здавалося б, програма має вивести на екран два нулі. Перевіримо. Запустивши програму, побачимо на екрані

0  
0

Точним є лише перше значення. Для розуміння причини цього явища необхідно розібратися із зображенням чисел усередині комп'ютера.

## 5.1 Системи числення

Числа, якими ми звикли користуватися, називаються десятковими, тому арифметика дій з ними також називається десятковою. Кожне число можна скласти з набору цифр, що містить 10 символів – цифр. Якщо припустити, що алфавітом є множина  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , а число – це слово, то зрозуміло, що назва “десяткове” походить від кількості літер алфавіту.

Візьмемо, наприклад, число 358. Цей запис означає, що в числі три сотні, п'ять десятків і вісім одиниць. Отже, можна записати таку рівність:  $358 = 300 + 50 + 8 = 3 * 10^2 + 5 * 10^1 + 8 * 10^0$ .

Цифри, які утворюють наше число, множаться на послідовні степені числа 10. Вони збільшуються на десять у степені, який на

одиницю менше порядкового номера цифри, при нумерації справа наліво. Таким чином, кожне написане нами число може бути зображене у вигляді спеціального розкладу за степенями числа 10, а для запису використовується набір з десяти цифр.

Позначимо  $i$ -ту цифру числа через  $a_i$ . Тоді число можна записати у вигляді  $a_n a_{n-1} \dots a_2 a_1$ . Запис числа можна зобразити так:

$$a_n a_{n-1} \dots a_2 a_1 = a_n 10^{n-1} + a_{n-1} 10^{n-2} + \dots + a_2 10^1 + a_1 10^0 \quad (5.1)$$

де  $a_i$  – символ з алфавіту 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Число 10 є основою утворення числа, яка називається *основою системи числення*, а сама система числення називається *десятьковою*.

Під системою числення розуміється спосіб зображення будь-якого числа за допомогою деякого алфавіту символів, що називаються *цифрами*.

Наочність зображення чисел і порівняльна простота виконання арифметичних операцій характерні для позиційних систем числення.

Система числення називається *позиційною*, якщо цифра має різні значення, залежно від того, яку позицію вона займає в послідовності цифр, що зображують число. Прикладом *непозиційної* системи числення є римська система зображення чисел.

Кількість  $p$  символів алфавіту, що використовуються в позиційній системі числення, називається її *основою*.

Загалом у системі з основою  $p$  будь-яке число  $X$  можна зобразити у вигляді полінома з основою  $p$ :

$$X = A_r p^r + A_{r-1} p^{r-1} + \dots + A_0 p^0 + A_{-1} p^{-1} + \dots$$

тоді саме число у вигляді послідовності літер алфавіту записується як

$$X = A_r A_{r-1} \dots A_1 A_0 . A_{-1} A_{-2} \dots$$

У цій послідовності крапка відокремлює цілу частину числа від дробової, тобто коефіцієнти при додатніх степенях основи, включаючи нуль, – від коефіцієнтів при від'ємних степенях.

Технічно реалізувати десяткову систему числення на комп'ютері дуже складно: для збереження десяткової цифри потрібен пристрій з десятима стійкими станами.

У комп'ютерах застосовують позиційні системи числення з недесятковою основою: двійкову, вісімкову, шістнадцяткову.

Технічно досить просто реалізувати двійковий елемент: є сигнал – немає сигналу. Крім того, у двійковій системі найпростіше реалізуються всі арифметичні операції: наприклад, двійкова таблиця множення складається лише із чотирьох рядків:

$$\begin{aligned}0 \times 0 &= 0 \\0 \times 1 &= 0 \\1 \times 0 &= 0 \\1 \times 1 &= 1\end{aligned}$$

Вивчати принципи роботи комп'ютера без двійкової системи числення неможливо.

Для двійкової системи числення формулу 5.1 можна записати у вигляді

$$X = A_m 2^m + A_{m-1} 2^{m-1} + \dots + A_1 2^1 + A_0 2^0 + A_{-1} 2^{-1} + \dots,$$

де  $A_i$  – або 0, або 1.

У вісімковій системі застосовуються вісім цифр: 0, 1, 2, 3, 4, 5, 6, 7. Для неї формула (5.1) має вигляд

$$X = A_m 8^m + A_{m-1} 8^{m-1} + \dots + A_1 8^1 + A_0 8^0 + A_{-1} 8^{-1} + \dots,$$

де коефіцієнти  $A_i$  набувають значень від 0 до 7.

Аналогічне зображення справедливе і для шістнадцяткової системи. Алфавіт складається з 16 літер: десяти цифр і шести літер – А, В, С, D, Е, F, що позначають числа 10, 11, 12, 13, 14, 15, відповідно.

У табл. 5.1 подано відповідність чисел у різних системах числення.

Розглянемо процес зображення чисел у різних системах числення.

### 5.1.1 Контрольні запитання

1. Що таке система числення?
2. Які Вам відомі системи числення?
3. Загальна формула запису числа у системі числення.

| Десяткова | Двійкова | Вісімкова | Шістнадцяткова |
|-----------|----------|-----------|----------------|
| 00        | 0000     | 00        | 0              |
| 01        | 0001     | 01        | 1              |
| 02        | 0010     | 02        | 2              |
| 03        | 0011     | 03        | 3              |
| 04        | 0100     | 04        | 4              |
| 05        | 0101     | 05        | 5              |
| 06        | 0110     | 06        | 6              |
| 07        | 0111     | 07        | 7              |
| 08        | 1000     | 10        | 8              |
| 09        | 1001     | 11        | 9              |
| 10        | 1010     | 12        | A              |
| 11        | 1011     | 13        | B              |
| 12        | 1100     | 14        | C              |
| 13        | 1101     | 15        | D              |
| 14        | 1110     | 16        | E              |
| 15        | 1111     | 17        | F              |

Табл. 5.1: Системи числення.

## 5.2 Правила переведення чисел з однієї системи числення в іншу

Для переходу від двійкової системи до десяткової зручно скористатися підсумовуванням за степенями двійки. Наприклад:

$$(10100)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 16 + 4 = (20)_{10}.$$

Загалом, при некратних основах, переведення додатних чисел із системи з основою  $p$  у систему з основою  $s$  зводиться до виконання такого алгоритму:

- поділ цілої та дробової частин числа в системі з основою  $p$ ;
- послідовне цілочисельне ділення із залишком цілої частини й цілих часток, що утворюються, на основу нової системи  $s$  числення доти, доки частка не стане менше цієї основи;



розряду записується 1. Потім розглядається степінь  $n - 2$  і т. д. Процес продовжується, доки у залишку не виявиться нуль або не буде отримана необхідна кількість значущих цифр числа. Для дробових чисел крапка ставиться після двійки в нульовому степені, перед від'ємними степенями. Наприклад, переведемо число 10.5:

$$2^3 = 8 < 10.5 - \text{записуємо } 1 \text{ у старший розряд і } 10.5 - 8 = 2.5;$$

$$2^2 = 4 > 2.5 - \text{записуємо в наступний розряд } 0;$$

$$2^1 = 2 < 2.5 - \text{записуємо в наступний розряд } 1 \text{ і } 2.5 - 2 = 0.5;$$

$$2^0 = 1 > 0.5 - \text{записуємо в наступний розряд } 0;$$

$$2^{-1} = 0.5 - \text{записуємо в наступний розряд } 1 \text{ і } 0.5 - 0.5 = 0.$$

### 5.2.1 Контрольні запитання

1. Алгоритм переведення додатних чисел із системи з основою  $r$  у систему з основою  $s$ .

## 5.3 Правило визначення точності зображення

Припустимо, що задано десяткове число, яке у вигляді десяткового дробу зображується з точністю до  $N$  знаків. Для того, щоб відповідне двійкове число було тієї самої точності, у ньому необхідно записати  $M$  знаків так, щоб  $2^M > 10^N$ .

Запишемо алгоритм перетворення. Вхідні дані алгоритму: через  $A$  позначатимемо вихідний правильний десятковий дріб, записаний у десятковій формі. Нехай цей дріб містить  $N$  знаків.

*Алгоритм:*

1. Визначимо кількість необхідних двійкових знаків  $M$  з нерівності  $2^M > 10^N$ .
2. Здійснюємо переведення цілої частини числа.
3. Цикл обчислення цифр двійкового зображення дробової частини числа. Номер цифри після десяткової крапки будемо позначати символом  $K$ .

3.1.  $K = 1$ .

3.2. Якщо  $2^{-K} > A$ , то в запис двійкового числа на  $K$ -те місце записуємо нуль, інакше у запис двійкового числа на  $K$ -те місце записуємо 1 і  $A = A - 2^{-K}$

3.3.  $K = K + 1$ .

3.4. Якщо  $K > M$ , то робота алгоритму завершена, інакше переходимо до пункту 2.

Вихідні дані: запис числа у двійковому зображенні.

Нехай необхідно перевести число 5.401 у двійкову систему числення. Ціла частина зображується двійковим числом 101. Дробову частину розкладемо за степенями двійки. Нам необхідно одержати стільки знаків  $M$  двійкового зображення, щоб  $10^3$  було менше  $2^M$ , тобто  $1000 < 2^M$ . Тоді у двійковому розкладанні має бути не менше десяти знаків, оскільки  $2^9 = 512$  і тільки  $2^{10} = 1024$ .

Наведемо послідовність виконання алгоритму.

Крок 1.  $2^{-2} = 0.25$ ;  $0.401 - 0.25 = 0.151$ .

Крок 2. Тепер необхідно степенем двійки зобразити 0.151:

$$2^{-3} = 0.125; 0.151 - 0.125 = 0.026.$$

Крок 3. Найближчий до цього числа 0.026 степінь двійки

$$2^{-6} = 0.015625; 0.026 - 0.015625 = 0.010375.$$

Тепер наше уточнене двійкове число становить 0.011001. Після коми вже є шість знаків, але цього поки що недостатньо, тому виконуємо ще один крок.

Крок 4. Працюємо з числом 0.010375. Найближчий до цього числа степінь двійки  $2^{-7} = 0.0078125$ ;

$$0.010375 - 0.0078125 = 0.0025625$$

Крок 5. Працюємо з числом 0.0025625. Найближчий до цього числа степінь двійки  $2^{-9} = 0.001953125$ ;

$$0.0025625 - 0.001953125 = 0.000609375.$$

Останній залишок менше  $2^{-10}$ , і якби ми бажали продовжувати наближення до вихідного числа, то нам би знадобилося  $2^{-11}$ , але це вже перевершує необхідну точність, отже, розрахунки можна припинити й записати остаточне двійкове зображення дробової частини:

$$0.401_{(10)} = 0.011001101_{(2)}$$

Остаточного маємо  $5.401_{(10)} = 101.011001101_{(2)}$ .

Розглянемо правила перетворення систем числення з кратними основами на прикладі двійкової, вісімкової та шістнадцяткової систем. Основами вісімкової та шістнадцяткової систем є цілі степені числа 2:

$$8 = 2^3, 16 = 2^4.$$

На цьому факті ґрунтується нижченаведений підхід.

Для переведення вісімкового чи шістнадцяткового числа у двійкову форму досить замінити кожен цифру цього числа відповідним три- або чотирирозрядним двійковим числом, відкидаючи незначущі нулі в старших розрядах.

Переведемо числа  $513.7_{(8)}$  та  $3E7.B_{(16)}$  у двійкову систему числення:

$$(513.7)_{(8)} = (101001011.111)_2$$

$$(3E7.B)_{(16)} = (1111100111.1011)_2$$

Перехід від двійкової до вісімкової (шістнадцяткової) систем здійснюють у такий спосіб: рухаючись від десяткової крапки ліворуч і праворуч, розбивають двійкове число на групи по три розряди для вісімкової і по чотири – для шістнадцяткової систем, доповнюючи до необхідної кількості (три чи чотири) нулями крайні ліву та праву групи. Потім кожен групу замінюють відповідною вісімковою чи шістнадцятковою цифрами. Наприклад, переведемо двійкове число  $10011101.110$  у вісімкове:

$$010\ 011\ 101.\ 110 = (235.6)_8,$$

а потім у шістнадцяткове:

$$1001\ 1101.\ 1100 = (9D.C)_{16}.$$

Розглянемо інші приклади переведення числа в десяткову систему числення:

$$\begin{aligned} 1000001_{(2)} &= 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + \\ &+ 0 \times 2^1 + 1 \times 2^0 = 64 + 1 = 65_{(10)}. \end{aligned}$$

$$1000011111.0101_{(2)} = 1 \times 2^9 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 +$$

$$+1 \times 2^0 + 1 \times 2^{-2} + 1 \times 2^{-4} = 512 + 16 + 8 + 4 + 2 + 1 + 0.25 + 0.0625 = \\ = 543.3125_{(10)}.$$

$$1216.04_{(8)} = 1 \times 8^3 + 2 \times 8^2 + 1 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-2} = \\ = 512 + 128 + 8 + 6 + 0.0625 = 654.0625_{(10)}.$$

$$29A.5_{(16)} = 2 \times 16^2 + 9 \times 16^1 + 10 \times 16^0 + 5 \times 16^{-1} = \\ = 512 + 144 + 10 + 0.3125 = 656.3125_{(10)}.$$

### 5.3.1 Контрольні запитання

1. Опишіть алгоритм визначення точності числа.

## 5.4 Двійкова арифметика

До арифметичних належать операції додавання, віднімання, множення й ділення. Як вони виконуються з двійковими числами? Наведемо двійкові операції і розглянемо детально їхнє виконання (табл. 5.2).

| Двійкове додавання                                        | Двійкове віднімання | Двійкове множення |
|-----------------------------------------------------------|---------------------|-------------------|
| $0 + 0 = 0$                                               | $0 - 0 = 0$         | $0 * 0 = 0$       |
| $0 + 1 = 1$                                               | $1 - 0 = 1$         | $0 * 1 = 0$       |
| $1 + 0 = 1$                                               | $1 - 1 = 0$         | $1 * 0 = 0$       |
| $1 + 1 = 0$ та одиниця<br>перенесення в<br>старший розряд | $10 - 1 = 1$        | $1 * 1 = 0$       |

Табл. 5.2: Двійкова арифметика

### 5.4.1 Додавання двійкових чисел

Операція додавання виконується порозрядно зліва направо, як і при додаванні десяткових чисел. При розрядному переповненні додаємо один старший розряд.

Розглянемо приклади.

1. Обчислити  $10011 + 10001$ .

Маємо:

$$\begin{array}{rcccccc} & + & 1 & 0 & 0 & 1 & 1 \\ & & 1 & 0 & 0 & 0 & 1 \\ \hline & & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$$

Розглянемо виконання операції додавання порозрядно. Нумеруємо розряди зліва направо.

*Перший розряд:*  $1 + 1 = 2$ . Одержали число, більше ніж 1, тому записуємо 0, а 1 переносимо у старший розряд.

*Другий розряд:*  $1 + 0 + 1$  (перенесення)  $= 2$ . Записуємо 0, а 1 переносимо у старший розряд.

*Третій розряд:*  $0 + 0 + 1$  (перенесення)  $= 1$ . Записуємо 1.

*Четвертий розряд:*  $0 + 0 = 0$ . Записуємо 0.

*П'ятий розряд:*  $1 + 1 = 2$ . Записуємо 0 у п'ятий розряд і 1 – у шостий.

Таким чином, одержали число 100100. Разом

$$10011 + 10001 = 100100.$$

2. Обчислити  $1010.01 + 1011.10$

Маємо:

$$\begin{array}{rccccccccc} & + & 1 & 0 & 1 & 0 & . & 0 & 1 \\ & & 1 & 0 & 1 & 1 & . & 1 & 0 \\ \hline & & 1 & 0 & 1 & 0 & 1 & . & 1 & 1 \end{array}$$

### 5.4.2 Зображення від'ємних чисел

Для спрощення виконання арифметичних операцій застосовують спеціальні коди для зображення чисел. Операція віднімання (алгебраїчного додавання) зводиться до арифметичного додавання кодів. Для зображення цілих чисел зі знаком у комп'ютері застосовують прямий, зворотний і додатковий коди.

Загальна ідея побудови кодів така. Код трактується як число без знака, а діапазон чисел, що зображуються кодами без знака, розбивається на два піддіапазони. Перший з них зображує додатні числа, а другий – від'ємні. Коди формуються так, щоб значення старшого розряду вказувало на знак числа, що зображується.

**Прямий код.** Прямий код двійкового числа  $A$ , що зображується в  $n$ -розрядній сітці, визначається як  $A$ , якщо  $A \geq 0$ ;  $a + |A|$ , якщо  $A < 0$ , де  $a$  – величина, що дорівнює вазі старшого розряду сітки: для дробів  $a = 1$ , а для цілих  $a = 2^{n-1}$ . Діапазон чисел, що зображуються прямим кодом, становить  $0 \leq |A| \leq a$ .

Ознакою зображення додатних чисел є наявність нуля в старшому розряді. Наявність у ньому 1 говорить про те, що число від'ємне.

Таким чином, прямий код двійкового числа збігається за зображенням із записом самого числа, але в розряді знака стоїть 0 чи 1 відповідно для додатних чи від'ємних чисел. Наприклад, для чотирирозрядної сітки маємо

0101 (5)

1101 (−5)

Додавання у прямому коді чисел, що мають однакові знаки, здійснюється досить просто. Числа додаються, і сумі присвоюється знак доданків. Однак для операції додавання чисел з різними знаками прямий код незручний. У цьому випадку доводиться визначати більше за модулем число, здійснювати віднімання, а потім різниці присвоювати знак більшого за модулем числа. Операція віднімання зводиться до операції арифметичного додавання за допомогою оберненого й додаткового кодів.

**Обернений код** Обернений код двійкового числа  $A$ , що зображується в  $n$ -розрядній сітці, визначається як  $I = A$ , якщо  $A \geq 0$ ;  $I = b - |A|$ , якщо  $A < 0$ , де  $b$  – величина найбільшого числа без знака, що розміщується в  $n$ -розрядній сітці: для дробових чисел  $b = 2 - 2^{-(n-1)}$ , а для цілих  $b = 2^n - 1$ . Діапазон чисел, що зображуються зворотним кодом, такий самий, як і для чисел прямого коду:  $0 \leq |I| \leq a$ . За означенням, зворотний код від'ємного числа є доповненням модуля вихідного числа до найбільшого числа без знака, що міститься в розрядній сітці. У зв'язку з цим одержання зворотного коду двійкового від'ємного числа зводиться до одержання інверсії  $n$ -розрядного коду абсолютної величини цього числа. Оскільки модуль чисел, що зображуються в  $n$ -розрядній сітці,  $|I| < a$ , то в старшому (знаковому) розряді зворотного коду в додатних чисел буде 0, а у від'ємних – 1. У цифрових розрядах зворотного двійкового коду зображується або модуль числа (для додатних чисел), або його інверсія (для від'ємних).

Розглянемо операцію алгебраїчного додавання:

$$\begin{array}{r}
 + \quad 0101 \quad (5) \\
 \quad 1101 \quad (-2) \\
 \hline
 \quad 10010 \\
 \quad \quad 1 \\
 \hline
 \quad 0011 \quad (3)
 \end{array}$$

Перенесення в знаковому розряді при використанні зворотного коду має додаватися у молодший розряд суми. Таке перенесення називається *круговим*, або *циклічним*.

При виконанні розрахунків на машині може виникнути як додатний, так і від'ємний нуль. Додатний нуль має вигляд:

$(+0) = 00000...00$  – для прямого зображення;

$(+0) = 00000...00$  – для зворотного зображення.

Від'ємний нуль має вигляд:

$(-0) = 100000...00$  – для прямого зображення;

$(-0) = 111111...11$  – для зворотного зображення.

Щоб уникнути виникнення двох нулів, застосовують додатковий код.

**Додатковий код.** Перейдемо тепер до питання зображення від'ємних чисел. Для визначеності розглянемо зображення чисел, у якому будь-яке число займає рівно вісім бітів. Із запису у двійковій системі числення рівності  $(-1) + 1 = 0$  легко знайти, який вигляд повинне мати невідоме нам поки що двійкове зображення xxxxxxxx числа  $-1$ :

$$\text{xxxxxxx} + 00000001 = 00000000.$$

Зрозуміло, що на місці символів xxxxxxxx має бути число 11111111. Правильним результатом додавання при цьому варто було б вважати 100000000, а не 00000000, але в нас є восьмибітове зображення і, оскільки результат має вміститися в байт, то старша одиниця відкидається.

Отже, число  $-1$  кодується як 11111111. Далі вже зовсім просто: для одержання  $-2$  потрібно  $-1$  зменшити на одиницю, що дасть 11111110; число  $-3$  зображується як 11111101 і т. д.

Від'ємні числа завжди містять у двійковому зображенні одиницю в найстаршому розряді, який називають знаковим. Абсолютна величина числа, що кодується, отримується як двійкове доповнення інших бітів (нулі потрібно замінити на одиниці, і навпаки), збільшене на одиницю. Таке зображення називається додатковим кодом.

Додатковий код двійкового числа  $A$ , що зображується в  $n$ -розрядній сітці, визначається як  $I = A$ , якщо  $A \geq 0$ ;  $I = c - |A|$ , якщо  $A < 0$ , де  $c$  – величина, що дорівнює вазі розряду, який іде за старшим розрядом використовуваної розрядної сітки: для дробів  $c = 2^0$ , для цілих чисел  $c = 2^n$ . Для додатних і від'ємних чисел діапазони значень, що зображуються, різні. Для додатних чисел, як і для прямого коду,  $0 \leq I < a$ , а для від'ємних  $0 < |I| \leq a$ . Із визначення додаткового коду випливає, що старший (знаковий) розряд коду додатного числа дорівнює 0, а від'ємного становить  $-1$ . Додатковий код від'ємного числа зручно одержувати через зворотний

код.

Додатковий код від'ємного числа може бути отриманий зі зворотного додаванням 1 до молодшого розряду зворотного коду. Наприклад, для чотирирозрядної сітки

0101 (5)

1011 (−5)

Для переведення числа в додатковий код можна рекомендувати й такий алгоритм:

1. У двійковому зображенні абсолютної величини числа всі одиниці замінити нулями, і навпаки.
2. До отриманого результату додати одиницю.

Застосуємо наведений алгоритм до числа −5. Нехай нас цікавить стандартне 16-розрядне значення. Спочатку зобразимо модуль числа у двійковому вигляді, доповнивши його нулями ліворуч до необхідної кількості розрядів. Тоді десяткове число 5 перетвориться на 0000 0000 0000 0101. Зробимо заміну нулів і одиниць; одержимо 1111 1111 1111 1010. Додамо одиницю й дістанемо відповідь: 1111 1111 1111 1011. При визначенні величини числа знаковий розряд не використовується, тому числа зі знаком мають максимальне значення, що складається із 15 двійкових розрядів. Властивостями зображення чисел у додатковому коді є:

1. Повторне застосування розглянутих вище правил переведення в додатковий код повертає вихідне число.
2. Додавання вихідного числа з його додатковим кодом завжди дає нульовий результат, тому що  $X + (-X) = 0$ . Якщо узагальнити цю властивість, можна стверджувати, що замість віднімання завжди можна скористатися додаванням зменшуваного та додаткового коду від'ємника.

### 5.4.3 Віднімання двійкових чисел

Спочатку розглянемо операцію віднімання для простого двійкового зображення. Щораз, коли цифра в розряді від'ємника більша за цифру в тому самому розряді зменшуваного, запозичується 1 з наступного, старшого розряду. Ця запозичена одиниця дорівнює двом одиницям поточного розряду.

Розглянемо приклади виконання операції віднімання.

1. Потрібно від одного двійкового числа відняти інше:

$$\begin{array}{r}
 \phantom{0} 1 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \\
 \phantom{0} \phantom{0} 1 \phantom{0} 1 \phantom{0} 0 \\
 \hline
 \phantom{0} \phantom{0} 1 \phantom{0} 1 \phantom{0} 1
 \end{array}$$

Перший розряд:  $1 - 0 = 1$ . Записуємо 1.

Другий розряд:  $0 - 1$ . Запозичуємо одиницю в старшому розряді. Одиниця зі старшого розряду переходить у молодший як дві одиниці, тому при вирахуванні одержимо 1.

Третій розряд: одиницю цього розряду ми запозичували, тому зараз у розряді 0. Запозичуємо одиницю зі старшого розряду. Знову одержуємо 1.

2. Виконати  $10100.11 - 1010.00$ :

$$\begin{array}{r}
 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} 0 \phantom{.} \phantom{0} 1 \phantom{0} 1 \\
 \phantom{0} \phantom{0} \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{.} \phantom{0} 0 \phantom{0} \\
 \hline
 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{.} \phantom{0} 1 \phantom{0} 1
 \end{array}$$

Розглянемо віднімання через додатковий код. Його можна замінити на додавання, якщо від'ємник замінити на число в додатковому коді. Маємо

$$a - b = a + add(-b) = a + \bar{b} + 1,$$

де  $add()$  позначена функція одержання додаткового коду,  $\bar{b}$  – число, отримане інверсією числа  $b$ . Запишемо це:

$$\begin{array}{r}
 + \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} (5) \\
 \phantom{0} 1 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} (-3) \\
 \hline
 \phantom{0} 1 \phantom{0} 0 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} (2)
 \end{array}$$

При алгебраїчному додаванні в додатковому коді перенесення зі старшого розряду відкидається. Тому результатом буде число 10. Результат алгебраїчного додавання доданків, зображених у додатковому коді, отримується в тому самому коді, що й доданки. Щоб змінити знак числа, необхідно:

- у прямому коді змінити знаковий розряд числа на протилежний;
- у зворотному коді інвертувати код;
- у додатковому коді інвертувати код, додати 1 у молодший розряд.

#### 5.4.4 Множення двійкових чисел

Операція множення у двійковій системі числення цікава своєю реалізацією. Для того, щоб помножити двійкове число на 2 (десятькова двійка – це 10 у двійковій системі), достатньо до числа, що множитись, праворуч приписати один нуль.

Розглянемо кілька прикладів.

1. Виконати  $10101 * 10 = 101010$ .

Ми знаємо, що будь-яке двійкове число розкладається за степенями двійки, тому множення у двійковій системі числення зводиться до множення на 10 (тобто на десяткову двійку), отже, множення – це низка послідовних зсувів. Загальне правило таке: як і для десяткових чисел, множення двійкових виконується порозрядно. І для кожного розряду другого множника до першого множника дописується один нуль праворуч.

2. Помножити два числа: 1011 і 101.

Це множення можна звести до суми трьох порозрядних множень:

$$1011 * 1 + 1011 * 0 + 1011 * 100 = 1011 + 101100 = 110111.$$

До стовпчика те саме можна записати як

$$\begin{array}{cccccc} & * & 1 & 0 & 1 & 1 \\ & & & 1 & 0 & 1 \\ \hline & & 1 & 0 & 1 & 1 \\ & 0 & 0 & 0 & 0 & \\ 1 & 0 & 1 & 1 & & \\ \hline 1 & 1 & 0 & 1 & 1 & 1 \end{array}$$

### 5.4.5 Ділення двійкових чисел

Нехай нам необхідно розділити два десяткових числа, наприклад 234 на 7. У діленому спочатку виділяємо праворуч від старшого розряду таку кількість цифр, щоб число, яке вийшло, було мінімальним і, у той самий час, більше дільника. У даному прикладі необхідне нам число – це 23. Потім ділимо отримане число на дільник із залишком. Описану операцію повторюємо, доки залишок не виявиться менше дільника. Тоді число під ризкою – це частка, а останній залишок – залишок операції.

Операція ділення двійкових чисел виконується так само. Розглянемо приклад: виконати операцію ділення  $10010011 : 101$ :

$$\begin{array}{ccccccccc|ccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & 1 & 0 & 1 \end{array}$$

Рухаючись від старшого розряду, шукаємо перше число, яке буде більше дільника. Це чотирирозрядне число 1001. Тепер необхідно знайти цілочисельний результат ділення виділеного числа на дільник. І тут ми знову виграємо порівняно з десятковою системою. Справа в тому, що число, яке підбирається, обов'язково є цифрою, а цифри в нас тільки дві. Оскільки 1001 явно більше 101, то тут усе зрозуміло – це 1. Виконаємо крок операції:

$$\begin{array}{r} \phantom{-} 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1 \mid 1\ 0\ 1 \\ \phantom{-} \phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\ 1\ 0\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \end{array}$$

Залишок від виконаної операції становить 100. Це менше ніж 101, тому, щоб виконати другий крок, необхідно дописати до 100 наступну цифру – цифру 0. Дістанемо:

$$\begin{array}{r} \phantom{-} 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1 \mid 1\ 0\ 1 \\ \phantom{-} \phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\ 1\ 0\ 0\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \end{array}$$

1000 більше 101, тому на другому кроці ми знову допишемо в частку цифру 1 та одержимо

$$\begin{array}{r} \phantom{-} 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1 \mid 1\ 0\ 1 \\ \phantom{-} \phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\ 1\ 0\ 0\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\phantom{1}\ 1\ 1\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \end{array}$$

Після кількох ітерацій маємо

$$\begin{array}{r} \phantom{-} 1\ 0\ 0\ 1\ 0\ 0\ 1\ 1 \mid 1\ 0\ 1 \\ \phantom{-} \phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\ 1\ 0\ 0\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\phantom{1}\ 1\ 1\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\phantom{1}\phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\phantom{1}\phantom{1}\phantom{1}\ 1\ 1\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\phantom{1}\phantom{1}\phantom{1}\phantom{1}\ 1\ 0\ 1 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \\ \phantom{-} \phantom{1}\phantom{1}\phantom{1}\phantom{1}\phantom{1}\phantom{1}\phantom{1}\ 1\ 0 \phantom{0\ 0\ 1\ 1} \phantom{\mid} \phantom{1\ 0\ 1} \end{array}$$

Отриманий залишок 10 менше 101, але в нас закінчилися цифри в діленому, тому 10 – це остаточний залишок, а 11101 – шукана частка.

## 5.4.6 Контрольні запитання

1. Як відбувається додавання двійкових чисел?

2. Як відбувається множення двійкових чисел?
3. Як відбувається віднімання двійкових чисел?
4. Як відбувається ділення двійкових чисел?
5. Як зображуються від'ємні числа у двійковій системі числення?

## 5.5 Ознака переповнення розрядної сітки при арифметичних операціях

При алгебраїчному додаванні двох чисел, що містяться в розрядній сітці, може виникнути переповнення, тобто утворитися сума, що вимагає для свого зображення на один цифровий розряд більше порівняно з розрядною сіткою доданків. Сформулюємо правило (ознаку) для виявлення переповнення розрядної сітки.

При алгебраїчному додаванні двох двійкових чисел з використанням додаткового (зворотного) коду для їхнього зображення ознакою переповнення розрядної сітки є перенесення в знаковий розряд суми за відсутності перенесення з нього (додатне переповнення). Ознакою від'ємного переповнення є перенесення зі знакового розряду суми за відсутності перенесення в нього.

Якщо й у знаковий, і зі знакового розряду суми є чи немає перенесення, то переповнення відсутнє. При додатному переповненні результат операції додатний, при від'ємному – від'ємний. Розглянемо додатне та від'ємне переповнення:

а)  $0\ 101 + 0\ 101 = 1\ 010$

б)  $1\ 011 + 1\ 011 = 10110$

Приклад а) демонструє додатне переповнення, б) – від'ємне.

### 5.5.1 Контрольні запитання

1. Назвіть ознаку переповнення розрядної сітки при арифметичних операціях.

## 5.6 Зображення цілих чисел

Пригадаємо, що в мові C# існують вбудовані типи `sbyte`, `byte`, `short`, `int`, `long` для зображення цілих чисел зі знаком. Також існують вбу-

довані типи `byte`, `ushort`, `uint`, `ulong` для зображення беззнакових цілих чисел. У пам'яті комп'ютера дані зберігаються в послідовно розташованих байтах. Для даних різних типів використовується різна кількість байтів (табл. 5.3).

| Назва типу          | Обсяг    | Діапазон значень                    |
|---------------------|----------|-------------------------------------|
| <code>sbyte</code>  | 1 байт   | від -128 до 127                     |
| <code>short</code>  | 2 байти  | від -32 768 до 32 767               |
| <code>int</code>    | 4 байти  | від -2 147 483 648 до 2 147 483 647 |
| <code>long</code>   | 8 байтів | від $-2^{63}$ до $2^{63} - 1$       |
| <code>byte</code>   | 1 байт   | від 0 до 255                        |
| <code>ushort</code> | 2 байти  | від 0 до 65 535                     |
| <code>uint</code>   | 4 байти  | від 0 до 4 294 967 295              |
| <code>ulong</code>  | 8 байтів | від 0 до $2^{64} - 1$               |

Табл. 5.3: Цілі типи даних в C# .

Ми бачимо, що діапазон можливих значень цілих типів істотно залежить від кількості байтів, необхідних для внутрішнього зображення.

За загальноприйнятими правилами, при зображенні чисел зі знаком крайній лівий біт старшого байта 0 використовується для зображення знака плюс і 1 – для мінуса.

Легко бачити, що при цьому найменшим від'ємним числом, що належить типу `byte`, є число -128 – двійкове зображення 10000000, а найбільшим – число 127 (зображення 01111111). Зверніть увагу: якщо до числа 01111111 додати одиницю, то вийде 10000000, що означає:  $127 + 1 = -128$ . Отже, множину елементів типу `byte` можна зобразити у вигляді згорнутого в кільце відрізка  $[-128; 127]$ .

Для елементів множини цілих чисел, що зображується типом `byte`, не виконується властивість цілих чисел  $x + 1 > x$ . Нерозуміння цього факту може призвести до вкрай неприємних наслідків.

Машинні значення діапазону цілих типів наведені в табл. 5.4.

### 5.6.1 Контрольні запитання

1. Які є типи даних цілих чисел у мові C# ?
2. Назвіть обсяг пам'яті для кожного типу цілого числа.

| Назва типу | Машинне зображення                                                             |
|------------|--------------------------------------------------------------------------------|
| sbyte      | 10000000 ...<br>01111111                                                       |
| short      | 10000000 00000000 ...<br>01111111 01111111                                     |
| int        | 10000000 00000000 00000000 00000000 ...<br>01111111 11111111 11111111 01111111 |
| byte       | 00000000 ...<br>11111111                                                       |
| ushort     | 00000000 00000000 ...<br>11111111 11111111                                     |
| uint       | 00000000 00000000 00000000 00000000 ...<br>11111111 11111111 11111111 11111111 |

Табл. 5.4: Машинне зображення значень цілих типів.

3. Назвіть діапазон значень для кожного типу цілого числа.

## 5.7 Зображення дійсних чисел

На відміну від зображення цілих чисел, що відрізняється абсолютною точністю, з дійсними числами не все так просто. Для зображення дійсних чисел у мові C# використовуються типи `float`, `double`. Тип `float` займає 4 байти. З них 1 байт приділяється для знака, 8 бітів – для характеристики і 23 біти – для мантиси. Старший біт мантиси завжди дорівнює 1, тому він не заповнюється, у зв'язку з чим діапазон значень змінної із плаваючою точкою приблизно дорівнює від  $3.4\text{E}-38$  до  $3.4\text{E}+38$  (див. табл. 5.5).

| Тип                 | Обсяг (байт) | Розмір мантиси, біт | Розмір порядку, біт |
|---------------------|--------------|---------------------|---------------------|
| <code>float</code>  | 4            | 23                  | 8                   |
| <code>double</code> | 8            | 52                  | 11                  |

Табл. 5.5: Типи дійсних чисел.

Тип `double` займає 8 байтів пам'яті. Його формат аналогічний

формату float. Біти пам'яті розподіляються так: 1 біт для знака, 11 – для характеристики і 52 – для мантиси. Із врахуванням випущеного старшого біта мантиси діапазон значень становить від  $1.7E-308$  до  $1.7E+308$ . У класичній математиці дійсні числа мають властивість щільності: між будь-якими двома різними числами завжди знайдеться відмінне від них третє. На комп'ютерне зображення дана властивість не поширюється. Розглянемо два типи зображення дійсних чисел:

а) з порядком:

|            |         |              |         |
|------------|---------|--------------|---------|
| знак числа | порядок | знак порядку | мантиса |
|------------|---------|--------------|---------|

б) з характеристикою:

|            |                |         |
|------------|----------------|---------|
| знак числа | характеристика | мантиса |
|------------|----------------|---------|

Введення характеристики не вимагає виділення одного біта для знака порядку і спрощує виконання операцій порівняння ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ) та арифметичних операцій над дійсними числами.

Найрозповсюдженішим способом реалізації дійсних чисел у комп'ютері є використання зображення з характеристикою, що отримується з порядку додаванням такого зсуву, щоб характеристика була позитивною. Нехай, наприклад, на характеристику виділяється 8 бітів. У 8 розрядах вміщуються двійкові числа від 00000000 до 11111111 (від 0 до 255 – у десятковій системі). Усього маємо 256 значень. Розумно ці значення порівну розподілити між від'ємними й додатними значеннями порядку числа: від  $-127$  до  $128$ . Мінімальному значенню порядку має відповідати нульове значення характеристики. Отже, отримуємо зміщення, що дорівнює 127. Для обчислення істинного порядку числа можна використовувати формулу  $x = 2^{n-1} + k + p$ , де  $n$  – кількість бітів, відведених для характеристики;  $p$  – порядок числа;  $k$  – поправочний коефіцієнт фірми IBM, рівний  $-1$  (табл. 5.6).

| Тип    | Характеристика       | Кількість бітів на характеристику |
|--------|----------------------|-----------------------------------|
| float  | $x = 2^7 + p - 1$    | 8                                 |
| double | $x = 2^{10} + p - 1$ | 11                                |

Табл. 5.6: Характеристика.

Для простоти розглянемо тип `float`, оскільки він найкоротший. Зображення інших типів відрізняються від нього тільки кількісно. Зауважимо, що в процесорах Intel байти у багатобайтових значеннях переставляються так, що молодший іде першим, а старший – останнім.

У мантисі зберігається двійкове ціле число. Щоб одержати істинне значення мантиси, до неї треба умовно додати ліворуч одиницю з крапкою. Таким чином, маючи 23 двійкових розряди, ми запишемо числа з точністю до 24 двійкових розрядів. Це пов'язано з тим, що використовується нормалізоване зображення дійсного числа. Нормалізація означає, що мантиса (для двійкового зображення), крім випадку, коли вона дорівнює 0, має знаходитися в інтервалі  $2^{-1} \leq M < 1$ .

Наведений метод нормалізації є класичним, коли її результат зображується у вигляді правильного дробу, тобто з одиницею після крапки й нулем у цілій частині числа. Тоді ненульова мантиса будь-якого числа з плаваючою точкою має починатися з двійкової одиниці. Ця одиниця враховується, однак не записується в мантису. Її часто називають *прихованою одиницею*.

У комп'ютерах на базі процесорів Intel нормалізована мантиса містить старший біт ліворуч від крапки. Іншими словами, нормалізована мантиса належить інтервалу  $1 \leq M < 2$ . У пам'яті машини для даних типу `float`, `double` цей біт не зберігається, він прихований і використовується для збільшення порядку. Для додатних і від'ємних чисел нормалізована мантиса в пам'яті зображена в прямому коді.

Старший біт у зображенні чисел у форматі з плаваючою точкою є знаковим. Зазвичай нуль позначає додатне число, а одиниця – від'ємне.

Якщо всі біти характеристики дорівнюють одиниці, а мантиси – нулю, то ми одержуємо комбінацію, відому як INF (від англійського Infinity – нескінченність). Ця комбінація використовується тоді, коли результат обчислень перевищує максимально припустиме форматом число. Залежно від значення знакового біта, нескінченність може бути додатною чи від'ємною. Якщо ж при такому порядку в мантисі принаймні один біт не дорівнює нулю, то така комбінація називається NAN (Not A Number – не число). Спроби використання комбінацій NAN чи INF приводять до помилки часу виконання.

Із врахуванням усіх цих правил комбінація, коли й у мантисі, і в порядку всі біти дорівнюють нулю, дає  $0^0$ . З математичного погля-

ду ця комбінація безглузда, але розробники формату домовилися вважати її нулем.

Тип `double` подібний типу `float`, різниця полягає лише в кількості розрядів і в тому, яке значення характеристики приймається за нуль. У `double` ми маємо 11 розрядів для характеристики, а за нуль приймається значення 1023.

Позначимо нормалізоване машинне зображення числа із плаваючою точкою  $f$ :

$$f = \pm M \times 2^p$$

де  $M$  – мантиса ( $1 \leq |M| < 2$ ),  $2$  – основа системи числення,  $p$  – цілочисельна характеристика. Також позначимо  $\varepsilon_0 = 2^{p_{\min}}$ ,  $\varepsilon_\infty = 2^{p_{\max}}$ , де  $p_{\min}$ ,  $p_{\max}$  – константи, що задають діапазон зміни характеристики. Тоді можна записати, що  $\varepsilon_0 \leq |f| < \varepsilon_\infty$ . Кількість розрядів, які відводяться під мантису, обмежує відносну точність зображення чисел у машині. Як характеристику цієї точності використовують величину  $\varepsilon_1$ , визначену як найменше машинне число, результат додавання якого до 1 буде машинним числом більше 1. Нагадаємо, що для дійсних типів `float`, `double` кількості бітів, що відводяться під характеристику, дорівнюють 8, 11. При цьому зсув дорівнює відповідно 127 і 1023. Мантиса має сховану одиницю. Ці дані нам знадобляться для подальших обчислень. Величина порядку  $p = 1 \dots 1$  відведена під спеціальні (нечислові) значення. Таким чином, найменші за модулем додатне й від'ємне числа мають вигляд (трикрапка зазначає, що подібне зображення має місце для двох розглянутих дійсних типів)

|   |        |       |
|---|--------|-------|
| 0 | 0...01 | 0...0 |
| 1 | 0...01 | 0...0 |

Відповідно,  $\varepsilon_0$  для дійсних типів `float`, `double` дорівнює  $2^{-126}$ ,  $2^{-1022}$ .

Праворуч від  $\varepsilon_0$  розташовується множина точок, що йдуть одна за одною з кроком  $2^{p_{\min}-n}$  де  $n$  – кількість бітів для зображення мантиси конкретного дійсного типу. Чим правіше числа, тим більше буде крок між ними. Наприкінці діапазону зображення відстань між двома сусідніми точками досягає  $2^{p_{\max}-n}$ . При цьому максимальні за модулем додатне й від'ємне числа мають вигляд

|   |        |       |
|---|--------|-------|
| 0 | 1...10 | 1...1 |
| 1 | 1...10 | 1...1 |

Відповідно,  $\varepsilon_\infty$  дорівнює  $2^{-128}$ ,  $2^{-1024}$ .

Знайдемо значення  $\varepsilon_1$ . Зображення числа 1 має вигляд

|        |   |            |       |
|--------|---|------------|-------|
| float  | 0 | 01111111   | 0...0 |
| double | 0 | 0111111111 | 0...0 |

а найближче до 1 зверху машинне число –

|        |   |            |        |
|--------|---|------------|--------|
| float  | 0 | 01111111   | 0...01 |
| double | 0 | 0111111111 | 0...01 |

Відповідно,  $\varepsilon_1$  дорівнює  $2^{-24}$ ,  $2^{-53}$  та  $2^{-64}$ .

Важливою характеристикою комп'ютера є співвідношення  $\varepsilon_0, \varepsilon_1, \varepsilon_\infty$ .

**Означення 5.1.** *Характеристики точності називаються збалансованими, якщо виконуються нерівності  $\varepsilon_0 < \varepsilon_1^2$ ,  $\varepsilon_\infty > \varepsilon_1^{-2}$ . Якщо ці умови не виконані, то при реалізації обчислювальних задач необхідно використовувати спеціальні прийоми.*

Описані особливості машинного зображення дійсних чисел приводять до того, що не для всіх його елементів правильні такі співвідношення:

1.  $x + 1 > x$ ;
2.  $x + x$  існує;
3.  $a + (b + c) = (a + b) + c$ ;
4.  $a + b + c = c + b + a$ .

Наведемо кілька прикладів.

1. Показати, що  $a + (b + c) \neq (a + b) + c$ .

```
using System;
class Program
{
    static void Main()
    {
        double x, y, z;
        x = 1.0e-16;
        y = 1.0 + (x + x);
        z = (1.0 + x) + x;
        if (y == z)
            Console.WriteLine("!!Yes!!");
        else
            Console.WriteLine("!!No!!");
        Console.WriteLine("y={0:F16}", y);
        Console.WriteLine("z={0:F16}", z);
    }
}
```

2. Написати програму для розв'язання квадратного рівняння для значень коефіцієнтів:  $a = 0.2e-45$ ,  $b = c = 1$ .

1

```
using System;
class Program
{
    static void Main()
    {
        double a, b, c, d, x1, x2;
        a = 0.2e-45;
        b = 1.0;
        c = 1.0;
        d = b * b - 4 * a * c;
        if (d > 0)
        {
            x1 = (-b + Math.Sqrt(d)) / (2 * a);
            x2 = (-b - Math.Sqrt(d)) / (2 * a);
            Console.WriteLine("x1={0:E}\nx2={1:E}", x1, x2);
            Console.WriteLine("a*x1^2+b*x1+c={0:E}", a * x1 * x1 + b
                * x1 + c);
            Console.WriteLine("a*x2^2+b*x2+c={0:E}", a * x2 * x2 + b
                * x2 + c);
        }
        else
        {
            if (d == 0)
            {
                x1 = -b / (2 * a);
                Console.WriteLine("x1={0:E}", x1);
            }
            else
            {
                Console.WriteLine("no_roots");
            }
        }
    }
}
```

При завершенні роботи програми ми одержимо, що  $x_1 = 0.00e+0$ ,  $x_2 = 5.0e+45$ . Підставивши ці значення до квадратного рівняння, у результаті дістанемо 1.

### 5.7.1 Контрольні запитання

1. Які є типи даних дійсних чисел у мові C# ?
2. Назвіть обсяг пам'яті для кожного типу дійсного числа.
3. Назвіть діапазон значень для кожного типу дійсного числа.
4. Що таке мантиса?
5. Які є типи зображень дійсних чисел?
6. Опишіть типи зображень дійсних чисел.
7. У якому випадку отримаємо inf?

## 5.8 Керування машинним зображенням чисел та особливості виконання арифметичних операцій

Співпроцесор майже завжди виконує всі операції над дійсними числами у спеціальному 80-бітному форматі. Він має спеціальний керуючий двобайтовий регістр. Установлення окремих бітів регістра диктує співпроцесору ту чи іншу поведінку. Його біти відповідають за те, як будуть округлятися числа, як співпроцесор розуміє нескінченність. Розглянемо тільки два біти регістра – восьмий і дев'ятий. Саме вони визначають, як оброблятимуться числа всередині співпроцесора.

Якщо восьмий біт містить одиницю (так встановлено за умовчанням), то 10 байтів внутрішніх регістрів співпроцесора використовуватимуться повністю. Якщо ж цей біт дорівнює нулю, то все визначається значенням дев'ятого біта. Якщо він дорівнює одиниці, то використовуються 53 розряди мантиси (інші завжди дорівнюють нулю). Якщо ж цей біт дорівнює нулю – то лише 24 розряди мантиси. Це збільшує швидкість обчислень, але зменшує точність. Інакше кажучи, точність роботи співпроцесора може бути знижена до типу *double* чи навіть *float*. Однак це стосується тільки мантиси, порядок завжди буде містити 15 бітів.

Сучасні співпроцесори обробляють числа з такою швидкістю, що навряд чи в кого-небудь може виникнути необхідність у прискоренні за рахунок точності. Проте це треба знати.

Ми вже зазначали, що властивість повноти не виконується при машинному зображенні дійсних чисел, оскільки не кожен правильний дріб може бути зображений у вигляді скінченного. Наприклад,  $1/9 = 0.11111\dots$ ,  $1/3 = 0.333333\dots$  і т. д. При роботі з такими числами використовується не точне, а наближене значення. Це потрібно враховувати при написанні програм.

Двійкові дробі теж можуть бути нескінченними. До того ж, не кожне число, що виражається скінченим десятковим дробом, може бути також зображено скінченим двійковим. Наприклад, число  $1/5$  зображується скінченим десятковим дробом як 0.2, але воно не може бути виражене скінченим двійковим дробом. Такі дробі завжди періодичні, оскільки раціональне число зображується скінченим або періодичним дробом у системі числення з будь-якою основою.

Розглянемо кілька прикладів некоректного використання дій-

сних типів.

### 1. Приклад появи зайвих цифр.

```
using System;
class Program
{
    static void Main()
    {
        float r;
        r = 0.1f;
        Console.WriteLine("{0:E8}", r);
    }
}
```

Після виконання програми ми побачимо 1.00000001e-001. Точність типу float – 7–8 десяткових розрядів, так що результат правильний. Однак звідки взяли інші цифри ?

Виявляється, число 0.1 не зображується у вигляді скінченного двійкового дробу, воно дорівнює 0.0(0011). І цей нескінченний двійковий дріб округляється на 24 знаках; ми одержуємо не 0.1, а деяке наближене число. Коли експериментуємо з різними числами, можна помітити, що точно зображуються ті числа, що виражаються у вигляді  $m/2^n$ , де  $m, n$  – деякі цілі числа.

### 2. Приклад незрозумілого результату виконання програми.

```
using System;
class Program
{
    static void Main()
    {
        float r;
        r = (float)0.1;
        if (r == 0.1)
            Console.WriteLine("equal");
        else
            Console.WriteLine("not_equal");
    }
}
```

Після запуску програми ми отримаємо “not equal”. Змінна  $r$  при присвоюванні одержує значення 0.100000001, тому що значення 0.1 має тип double, а потім, при присвоюванні, відбувається округлення до типу float. Ліва і права частини рівності в умові спочатку перетворюються на тип double, і лише потім здійснюється порівняння. Число, що зберігається в змінній  $r$  (уже не 0.1!), зображується у вигляді скінченного двійкового дробу. При перетворенні цього числа в double молодші, надлишкові, порівняно з типом float, розряди мантиси, просто заповнюються нулями. Ми одержуємо те саме число, тільки записане у форматі double.

Над числом 0.1 із правої частини не відбувається проміжного

перетворення в float і воно матиме нескінченне зображення у двійковому коді, а деякі з молодших розрядів мантиси міститимуть ненульові значення. Тому порівнюють близькі, але не рівні числа.

Якщо замінити число 0.1 на 0.5, то одержимо “equal”, тому що 0.5 – це скінченний двійковий дріб. При прямому зведенні його до типу float у молодших розрядах будуть нулі. Такі самі нулі виявляються в цих розрядах при перетворенні числа 0.5 типу float у тип double. Тому в результаті ми порівнюємо два однакових числа.

3. Розглянемо програму:

```
using System;
class Program
{
    static void Main()
    {
        float r1;
        double r2;
        r1 = (float)0.1;
        r2 = 0.1;
        if (r1 == r2)
            Console.WriteLine("equal");
        else
            Console.WriteLine("not_equal");
    }
}
```

Одержимо той самий результат, тобто “not equal”, із тих самих причин.

4. Розглянемо програму:

```
using System;
class Program
{
    static void Main()
    {
        float r;
        int i;
        r = 1;
        for (i = 1; i <= 10; i++)
            r -= 0.1f;
        Console.WriteLine("r={0}", r);
    }
}
```

Після виконання програми на екрані з’явиться  $-1.490116\text{E-}008$ , тому що число 0.1 не може бути передане точно у жодному із дійсних типів. При цьому постійно відбуваються округлення, завдяки яким ми одержуємо в результаті не нуль, а майже нуль.

Коли ми маємо справу з обчисленнями з обмеженою точністю, виникає парадокс. Наприклад, ми маємо точність до трьох значущих цифр. Додамо до числа 1.00 число  $1.00 \cdot 10^{-4}$ . Ми мали б

одержати 1.0001. Однак у нас обмежена точність, тому ми змушені округляти до трьох значущих цифр. У результаті виходить 1.00. Іншими словами, до деякого числа ми додаємо інше, яке більше нуля, і, через обмежену точність, одержуємо те саме число.

**Означення 5.2.** *Найменше додатне число, при додаванні якого до одиниці результат не дорівнює одиниці, називається машинним епсілоном.*

Написати програму для обчислення машинного епсілона.

```
using System;
class Program
{
    static void Main()
    {
        double r;
        r = 1;
        while (1 + r / 2 > 1) r = r / 2;
        Console.WriteLine("{0:E20}", r);
    }
}
```

У результаті буде отримане число 2.22044604925031310000E-016.

### 5.8.1 Контрольні запитання

1. У якому форматі співпроцесор виконує операції на дійсними числами?
2. У яких випадках виникають «зайві цифри»?
3. Правила порівняння дійсних чисел.

## 5.9 Завдання для самостійної роботи

1. Перевести десяткові числа у двійкові, вісімкові та шістнадцяткові:
  - a) 38;
  - b) 17.0162;
  - c) 0.313;
  - d) 7.127;
  - e) 13.0015.

2. Перевести шістнадцяткові числа в десяткові, двійкові, вісімкові:
  - a) AC2F. F;
  - b) FF.F;
  - c) A3B.12.
3. Перевести двійкове число у вісімкове та шістнадцяткове:
  - a) 10011011.1101;
  - b) 101011100.001101;
  - c) 1110001.010101.
4. Перевести число з десяткової у двійкову систему числення.
5. Перевести число із двійкової в десяткову систему числення.
6. Обчислити вирази:

Адитивні операції:

  - a)  $11001 + 101$ ;
  - b)  $11001 + 11001$ ;
  - c)  $1001 - 111$ ;
  - d)  $10011 + 101$ ;
  - e)  $11011 - 1111$ ;
  - f)  $11111 + 10011$ .

Мультиплікативні операції:

  - a)  $1101 / 110$ ;
  - b)  $1010 / 10$ ;
  - c)  $1011 / 11$ ;
  - d)  $101011 / 101$ ;
  - e)  $10010 / 1001$ ;
  - f)  $1101 * 1110$ ;
  - g)  $1010 * 110$ ;
  - h)  $1011 * 11$ ;
  - i)  $101011 * 1101$ ;
  - j)  $10010 * 1001$ .

7. Надрукувати число  $x$ , таке, що  $x + 1 = x$ .
8. Надрукувати ціле число  $x$ , таке, що  $x + 1 < x$ .

# 6 Класи

## 6.1 Класи у C#

У першому розділі ми вже почали вивчати таке поняття, як клас. Тепер розглянемо його більш детально.

Клас — це логічна структура, що дозволяє створювати користувацькі типи. Клас визначає дані й поведінку екземпляра цього типу.

Формально, опис класу має вигляд:

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

Модифікаторів доступу для класів є два: - public та internal.

Модифікатор доступу public означає, що доступ до класу можливий з будь-якого місця своєї збірки або з іншої збірки, на яку є посилання. Модифікатор доступу internal – доступ до класу можливий тільки зі збірки, у якій він оголошений.

Класи в C# можуть містити такі члени:

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

Усі члени класу, як і сам клас, мають свій рівень доступу.

Поступово ми вивчимо всі члени класу. Почнемо з полів.

### 6.1.1 Поля та константи класу

**Поле** – це змінна, що оголошена усередині класу (або структури). Синтаксично поля є звичайними змінними (об'єктами) мови. Їхній опис задовольняє звичайним правилам оголошення змінних.

Змістовно поля формують ту саму абстракцію даних, яку описує клас. Нагадаємо, що, коли створюється новий екземпляр класу (у стеку), то це є набір полів класу. Два різні екземпляри одного класу мають однаковий набір полів, але відрізняються значеннями, що зберігаються в цих полях. Як заведено, поля оголошуються з модифікаторами доступу `private` або `protected`, щоб заборонити прямий доступ до них.

Для отримання доступу до полів слід використовувати властивості або методи.

Приклад оголошення полів у класі (з першого розділу):

```
class Elevator
{
    public int Height;
    public int Width;
    public int Length;
    private int Capacity;
}
```

У цьому прикладі деякі поля мають модифікатор `public`. За його допомогою ми даємо доступ до цих полів з будь-якого місця програми.

## Константні поля

Членом класу може бути константа – іменоване незмінне значення. Оголошення константи подібне оголошенню поля класу, але воно включає атрибут `const`. Константа-член класу аналогічна тому поняттю константи, що ми розглядали у розділі 1. Особливістю констант є те, що в момент компіляції їх значення повинні бути визначені.

```
class Elevator
{
    public int Height;
    public int Width;
    public const int Length;
    private const int Capacity=800;
}
```

У даному прикладі оголошено дві константи, але лише константу `Capacity` оголошено коректно. Для константи `Length` не задано значення.

До констант можна звертатися, не утворюючи екземпляр класу, тобто лише за допомогою імені класу. Наприклад, щоб звернутися до константи `Capacity` в класі `Program` можна записати просто `Elevator.Capacity`.

Існує специфічний різновид полів – поля для читання. Їх можна ініціалізувати лише один раз – у місці оголошення або у конструкторі класу. Ініціалізація такого поля відбувається під час виконання програми. Після цього значення поля може бути лише прочитано, але не змінено. Це дає змогу нам утворювати незмінні елементи програми більш гнучко, ніж за допомогою констант, тобто для різних екземплярів класів можуть бути різні значення незмінних полів.

```
class Elevator
{
    public int Height;
    public int Width;
    public int Length;
    public readonly int Capacity;

    public Elevator(int H, int W, int L, int C)
    {
        Height = H;
        Width = W;
        Length = L;
        Capacity = C;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Elevator MyElevator = new Elevator(2,4,6,1800);
        Elevator HisElevator = new Elevator(2,4,6,800);
        HisElevator.Capacity = 600; // неприпустимо!
    }
}
```

### 6.1.2 Модифікатор static

Ми вже бачили, що, щоб використовувати члени класу, необхідно було утворити екземпляр класу. Однак використовуючи модифікатор `static`, ми можемо оголосити член класу, що існує і до якого можна звернутися, незалежно від екземплярів цього класу. Наприклад, розглянемо такий клас:

```
using System;

namespace ConsoleApplication1
{
    class Circle
    {
        public static double Radius;
        public static double Pi = 3.14159265;
        public double Square(double R)
```

```
        {
            return Pi * R * R;
        }
    }

    class Program
    {
        static void Main(string [] args)
        {
            double TotalSquare;
            // доступ до Radius можливий через використання static
            Circle.Radius = 3.67;
            // доступ до Pi можливий
            double Length = 2 * Circle.Pi * Circle.Radius;

        }
    }
}
```

### 6.1.3 Контрольні запитання

1. Що таке клас?
2. Які є модифікатори доступу для класів?
3. Які члени може містити клас?
4. Що таке поля для читання?
5. Використання модифікатору доступу static.

## 6.2 Правила доступу

Для того щоб краще зрозуміти права доступу, необхідно ввести поняття збірки (англ. assembly).

Збірка – це (скомпільована) колекція типів та ресурсів у вигляді .exe або .dll файлу (файлів). Збірка надає можливість управління версіями, повторного використання коду, області дії та політики безпеки.

Модифікатори доступу дозволяють задати допустиму області видимості для класів і членів класу. Тобто задати той контекст, в якому можна використовувати задане поле, метод або клас.

Модифікаторів доступу для класів, що не є членами інших типів, є два: public та internal. Для таких класів модифікатор доступу public означає, що доступ до класу можливий з будь-якого місця своєї збірки або з іншої збірки, на яку є посилання; модифікатор

доступу `internal` – доступ до класу можливий тільки зі збірки, у якій він оголошений.

Для членів класу може бути п'ять видів доступу:

- `public` – доступ не обмежений;
- `protected` – доступ до члена можливий тільки усередині класу, або в класі-спадкоємці (при спадкуванні); при цьому доступ у класі-спадкоємці до нестатичного члена з таким доступом має відбуватися через екземпляр класу-спадкоємця, або екземпляр типу, побудованого на основі класу-спадкоємця;
- `internal` – доступ до члена можливий тільки зі збірки, у якій він оголошений;
- `private` – доступ до члена можливий тільки усередині класу;
- `protected internal` – доступ до члена можливий зі збірки, або із класу-спадкоємця в іншій збірці; при доступі до нестатичного члена з класу-спадкоємця діють обмеження, вказані вище для рівня доступу `protected`.

Слід зауважити, що у випадку, коли ви не вказуєте модифікатор доступу, за замовчуванням для членів класу встановлюється рівень доступу `private`, а для типів, що не є членами інших типів, – `internal`. Для членів типів, відмінних від класів, діють окремі правила визначення рівня доступу за замовчуванням.

Розглянемо такий приклад.

```
public class Example
{
    int A;    // оголошено зміну цілого типу, як private
    private int B; // поле доступно лише із цього класу
    protected int C; // поле доступно із цього класу та класів нащадків
    internal int D; // поле доступно з будь-якого місця збірки
    protected internal int E; // доступно з будь-якого місця збірки й з класів
        нащадків
    public int F; // доступно з будь-якого місця збірки, а також з інших збірок

    private void WriteConsoleF()
    {
        Console.WriteLine("Поле_F_має_значення_={0}", F);
    }

    public void WriteConsoleA()
    {
        Console.WriteLine("Поле_A_має_значення_={0}", A);
    }

    internal void WriteConsoleB()
    {
        Console.WriteLine("Поле_B_має_значення_={0}", B);
    }

    protected void WriteConsoleE()
    {
```

```
        Console.WriteLine("Поле_E_має_значення_{0}", E);  
    }  
}
```

Розглянемо, як буде виконуватись доступ до полів та як будуть викликатись методи. Напишемо головну програму, або клас. Будемо звертатись до поля, використовуючи ім'я test екземпляру класу Example. Маємо:

```
class Program  
{  
    static void Main(string[] args)  
    {  
        Example test = new Example();  
        test.a = 4; //Помилка, отримати доступ не можна  
        test.b = 3; // Помилка, отримати доступ не можна  
        test.c = 1; /* Помилка, отримати доступ неможна через те, що клас Program не є нащадком класу Example */  
        test.d = 5; // поле d доступно з будь-якого місця збірки. Помилки немає.  
        test.e = 8; // поле e доступно з будь-якого місця збірки. Помилки немає.  
        test.f = 8; // поле f доступно з будь-якого місця  
  
        // тестування доступу до методів  
        test.WriteConsoleF(); /* Помилка. Немає доступу через те, що метод WriteConsoleF оголошено, як private в класі Example, а метод використовується в класі Program */  
  
        // Оскільки цей метод оголошений як protected, а клас Program не є нащадком класу State  
        test.WriteConsoleE(); /* Помилка. Немає доступу через те, що метод WriteConsole оголошено, як protected в класі Example, а клас Program не є нащадком класу Example */  
  
        test.WriteConsoleA(); // коректне використання методу. Метод оголошено як public  
  
        test.WriteConsoleB(); // коректне використання методу. Метод оголошено як internal  
  
        Console.ReadKey();  
    }  
}
```

Модифікатор `internal` дозволяє надати доступ до класу іншим класам, не гарантуючи при цьому доступ абсолютно всім класам. Клас, що позначений як `internal`, є доступним всім класам, що входять у збірку. Застосування модифікатора `internal` до полів класу корисно, коли необхідно організувати взаємодію між декількома класами.

Слід зауважити, що член класу є доступним, якщо доступ дозволений його власним модифікатором й модифікатором класу, у якому він оголошений.

Розглянемо простий приклад.

```
internal class MyClass
{
    public void PublicMethod()
    {

    }

    internal void InternalMethod ()
    {

    }

    protected void ProtectedMethod ()
    {

    }
}
```

Бачимо, що ми не маємо доступу до методу PublicMethod() з інших збірок, через те що клас оголошений як internal.

### 6.2.1 Контрольні запитання

1. Що таке збірка?
2. Назвіть та охарактеризуйте модифікатори доступу.
3. Які модифікатори за замовчуванням встановлюються для членів класу та для інших типів?

## 6.3 Конструктори класів

### 6.3.1 Конструктори екземплярів класів

У першому розділі ми почали вже вивчати роботу конструктора. Тепер розглянемо його роботу більш детально.

Оголошення конструктора має такий вигляд:

```
public class_name ([arguments])
{
    // тіло конструктора
}
```

При оголошенні конструктора не потрібно вказувати тип результату. Оголошення класу може містити більше одного конструктора екземпляра класу при умові, що всі конструктори розрізняються за кількістю та/або типами аргументів. Якщо оголошення класу

не містить жодного конструктора, то компілятор автоматично забезпечить класу неявний конструктор екземпляра класу за замовчуванням без параметрів і з тілом вигляду {} (який має у випадку неабстрактних класів рівень доступу public). Якщо оголошення класу містить принаймні один конструктор, то компілятор не створюватиме жодного неявного конструктора для класу. Аналогічно випадку інших членів класу, якщо в оголошенні конструктора не вказаний модифікатор доступу, то за замовчуванням рівень доступу для конструктора встановлюється як private, що означає, що його не можна використати поза межами класу (або вкладених типів) для створення екземплярів цього класу.

Розглянемо такий конструктор для класу Elevator:

```
class Elevator
{
    public int Height;
    public int Width;
    public int Length;
    public int Capacity;

    public Elevator(int Height, int Width, int Length, int
        Capacity)
    {
        this.Height = Height;
        this.Width = Width;
        this.Length = Length;
        this.Capacity = Capacity;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Elevator MyElevator = new Elevator(2, 4, 6, 1800); // виклик
        конструкторів
        Elevator HisElevator = new Elevator(2, 4, 6, 800); // та
        ініціалізація полів
        HisElevator.Capacity = 600; // зміна значення поля Capacity
    }
}
```

У даному прикладі ми використовували ключове слово this. У випадку, коли імена аргументів конструктора співпадають з іменами полів, ключове слово this дає доступ до екземпляра класу, який в поточний момент утворюється. Якщо імена різні, то this можна не використовувати.

Якщо в класі записаний один чи декілька конструкторів, то ми вже не зможемо утворити об'єкт за допомогою конструктора за замовчуванням:

```
class Elevator
{
    ...

    public Elevator(int Height, int Width, int Length, int
        Capacity)
    {
        ...
    }
}

class Program
{
    static void Main(string[] args)
    {
        Elevator MyElevator = new Elevator() // помилка!
    }
}
```

Слід зазначити, що в документації Microsoft “конструктором за замовчуванням” називається також конструктор без параметрів. Конструктор без параметрів можна також використовувати для надання початкових значень полям екземпляра.

У C# не можна в класі описати два конструктори з однаковою кількістю і типами аргументів через те, що оператор `new` в такому випадку не знав би, який конструктор викликати.

Припустимо, що Вам необхідно кожному ліфту присвоїти унікальний ідентифікаційний номер. Введемо поле `Number` і розглянемо, що може виникнути при ініціалізації цього поля в різних випадках. Нехай необхідно мати конструктор, який автоматично збільшує поточний ідентифікаційний номер ліфта й ще один конструктор, який задає цей номер “вручну”. Для такої ситуації краще використовувати “зчеплені” конструктори:

```
class Elevator
{
    static public int IdNumber=0;
    public int Height;
    public int Width;
    public int Length;
    public int Capacity;
    public int Number;

    public Elevator()
        : this(0,0,0,0)
    {
    }
    public Elevator(int Height, int Width, int Length, int
        Capacity)
        : this(Height, Width, Length, Capacity, ++IdNumber)
    {
    }
}
```

```
public Elevator(int Height, int Width, int Length, int
    Capacity, int Number)
{
    this.Height = Height;
    this.Width = Width;
    this.Length = Length;
    this.Capacity = Capacity;
    this.Number = Number;
}

class Program
{
    static void Main(string[] args)
    {
        Elevator FirstElevator = new Elevator();
        Console.WriteLine(FirstElevator.Number);
        Elevator SecondElevator = new Elevator(1,2,3,4);
        Console.WriteLine(SecondElevator.Number);
        Elevator ThirdElevator = new Elevator(11, 22, 33, 44, 10)
            ;
        Console.WriteLine(ThirdElevator.Number);

        Console.ReadKey();
    }
}
```

Після виконання програма роздрукує значення 1, 2, 10 у стовпчик. При виклику першого конструктора без параметрів `Elevator FirstElevator = new Elevator()` буде викликаний конструктор з чотирма аргументами за допомогою `this(0,0,0,0)`. Цей конструктор отримає у ролі фактичних аргументів чотири нулі. Потім конструктор з чотирма параметрами викличе конструктор з п'ятьма. І п'ятим фактичним аргументом буде збільшене на одиницю значення поля `IdNumber`, тобто 1. Нагадаємо, що це поле мало значення нуль. І вже конструктор з п'ятьма фактичними аргументами утворить екземпляр класу та надасть полям значення 0, 0, 0, 0, 1 відповідно.

Так само програма буде працювати при створенні другого екземпляра класу `Elevator SecondElevator = new Elevator()`. Але вже буде передано не нулі, а значення 1, 2, 3, 4. При виклику конструктора для створення третього екземпляру класу, останнім фактичним аргументом передається число 10 і тому поле `Number` прийме саме це значення.

### 6.3.2 Статичний конструктор

У C# передбачено можливість написання **статичних конструкторів класу**. Цей конструктор викликається один раз, а не при кожному створенні екземплярів класу.

Статичний конструктор не можна викликати безпосередньо. Якщо звичайний конструктор викликається в момент створення екземпляра класу, то статичний – викликається автоматично перед створенням першого екземпляра класу або перед використанням посилення на які-небудь статичні члени за виключенням констант. Виконуюча система .NET не може підказати програмістові, у який саме момент часу буде викликаний статичний конструктор, тому не слід розміщувати в ньому код, який повинен бути виконаний у конкретний момент часу (наприклад, у залежності від значень інших змінних).

Такий конструктор буває корисний для виконання яких-небудь підготовчих дій, які повинні бути виконані лише один раз. У статичному конструкторі можна задати початкові значення для статичних членів класу.

Якщо статичний конструктор ініціює виключення, середовище виконання не викликає його в другий раз, і клас залишається неініціалізованим. Ще необхідно відзначити, що статичний конструктор не повинен мати модифікатори доступу й параметрів.

Синтаксис статичного конструктора такий:

```
class class_name
{
    static class_name()
    {
        // код ініціалізації
    }

    // решта визначення класу
}
```

Присвоювання значень статичним полям відбувається в порядку їх оголошення. Конструктор статичний або екземплярний не обов'язково має ініціалізувати усі поля, до яких має доступ. При створенні екземпляра класу усі поля приймають, відповідно до типу, початкове нульове значення.

Відзначимо також, що допускається визначати в класі одночасно статичний конструктор і конструктор екземпляра без параметрів. Хоча їх списки параметрів ідентичні, конфлікту не виникає, тому що статичний конструктор виконується до конструктора екземпляра.

Якщо у вас є більше одного класу зі статичним конструктором, то

невідомо, який зі статичних конструкторів буде викликаний першим. Тому ви не повинні розміщувати в статичному конструкторі код, виконання якого залежить від результатів роботи інших статичних конструкторів.

Розглянемо такий приклад:

```
using System;

namespace StaticConstructor
{
    public class TestStaticConstructor
    {
        private int y = Inity();
        private int x = Initx();
        public static int a = InitA0();
        public static int b = Initb0();

        static TestStaticConstructor()
        {
            Console.WriteLine("_Call_static_TestStaticConstructor()");
            ;
            Console.WriteLine("a_" + a);
            Console.WriteLine("b_" + b);
        }
        private static int Initx()
        {
            Console.WriteLine("Initialization_X");
            return 1;
        }
        private static int Inity()
        {
            Console.WriteLine("Initialization_Y");
            return 2;
        }
        private static int InitA0()
        {
            Console.WriteLine("Initialization_static_A0");
            return 3;
        }
        private static int Initb0()
        {
            Console.WriteLine("Initialization_static_B0");
            return 4;
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            TestStaticConstructor a = new TestStaticConstructor();
        }
    }
}
```

У результаті роботи програми буде надруковано на екрані:

Initialization static A0

Initialization static B0

Call static TestStaticConstructor()

a= 3

b= 4 Initialization Y Initialization X

Для продовження натисніть будь-яку клавішу

Розберемо цей приклад детальніше. Статичний конструктор TestStaticConstructor() викликається перед створенням екземпляра класу. Однак ініціалізація статичних полів виконується перед виконанням тіла цього конструктора. Це гарантує правильну ініціалізацію полів перед тим, як відбудеться звертання до них з тіла статичного конструктора.

У цьому прикладі статичний конструктор буде виконуватися перед тим, як програма звернеться до поля або методу класу, або перед створенням екземпляра цього класу.

Якщо в програму додати нестатичний конструктор, то можна побачити, що при створенні екземпляра класу буде використовуватися саме він.

### 6.3.3 Порядок ініціалізації полів та виклик конструкторів

При першому створенні екземпляра класу необхідно користуватись таким порядком ініціалізації та виклику конструкторів (випадок наслідування класів розберемо пізніше):

1. Статичні поля класу набувають початкового значення нуль;

Наприклад:

```
static int i;
```

значення поля i буде 0.

2. Запускаються ініціалізатори статичних полів в тому порядку, в якому вони записані у програмі;

Наприклад:

```
static int i=j+1; // на момент виконання j=0 (п.1)
```

```
static int j=100;
```

```
static int k=j+1; // на момент виконання j=100
```

після виконання змінні будуть мати такі значення:

i=1;

j=100;

k=101;

За наявності статичного конструктора ініціалізатори статичних полів запускаються безпосередньо перед запуском конструктора. Якщо його немає, то в будь-який час до першого звернення до цих полів.

3. Запускається статичний конструктор класу.
4. Поля екземпляра класу набувають початкового значення нуль.
5. Запускаються ініціалізатори нестатичних полів в тому порядку, в якому вони записані у програмі;

Наприклад:

```
public int i=j+1; // на момент виконання j=0 (п.4)
```

```
public int j=100;
```

```
public int k=j+1; // на момент виконання j=100
```

після виконання змінні будуть мати такі значення:

i=1;

j=100;

k=101;

6. Запускається нестатичний конструктор екземпляра класу.

Для демонстрації цих правил розглянемо такий приклад:

```
using System;
```

```
namespace TestStaticNonStaticInitialization
{
    public class TestStaticNonStaticInitialization
    {
        public TestStaticNonStaticInitialization()
        {
            Console.WriteLine("Call_NonStatic_Constructor");
            Console.WriteLine("x=" + x);
            Console.WriteLine("y=" + y);
        }
        static TestStaticNonStaticInitialization()
        {
            Console.WriteLine("Call_Static_Constructor");
        }
    }
}
```

```

    }
    static int InitValue(string message)
    {
        Console.WriteLine(message);
        return 100;
    }
    public static void StaticMethod()
    {
        Console.WriteLine("Call_Static_Method");
    }
    private int y = InitValue("Initialization_of_NonStatic_value_
        y");
    private int x = InitValue("Initialization_of_NonStatic_value
        x");
    public static int a = InitValue("Initialization_of_Static_
        value_a");
    public static int b = InitValue("Initialization_of_Static_
        value_b");
}
class Program
{
    static void Main(string [] arg)
    {
        Console.WriteLine("Main");
        TestStaticNonStaticInitialization.StaticMethod();
        Console.WriteLine("Create_instance_#1");
        TestStaticNonStaticInitialization N1incance = new
            TestStaticNonStaticInitialization();
        Console.WriteLine("Create_instance_#2");
        TestStaticNonStaticInitialization N2incance = new
            TestStaticNonStaticInitialization();
    }
}
}

```

Після запуску програми на екрані буде надруковано:

Main

Initialization of Static value a

Initialization of Static value b

Call Static Constructor

Call Static Method

Create instance #1

Initialization of NonStatic value y

Initialization of NonStatic value x

Call NonStatic Constructor

x=100

y=100

Create instance #2

Initialization of NonStatic value y

Initialization of NonStatic value x

Call NonStatic Constructor

x=100

y=100

Тобто, ми бачимо, що:

1. ініціалізація статичних полів виконується до команд статичного конструктора;
2. через те що клас має статичний конструктор, статична ініціалізація виконується безпосередньо при першому зверненні до даного класу, тобто при виклику статичного методу `TestStaticNonStaticInitialization.StaticMethod()`;
3. ініціалізатори статичних полів і статичний конструктор запускаються до статичного методу;
4. при створенні екземпляра класу нестатичним конструктором спочатку ініціалізуються поля, потім виконуються інші команди конструктора.

Якщо видалити нестатичний конструктор, то можна побачити, що ініціалізація статичних полів не буде виконуватись безпосередньо до звернення до них (для демонстрації цього в текст програми додано зміну `work1`):

```
static void Main(string [] arg)
{
    int work1;

    Console.WriteLine("Main");

    Console.WriteLine("Create_instance_#1");

    work1 = TestStaticNonStaticInitialization.b;

    Console.WriteLine("work1=" + work1);

    TestStaticNonStaticInitialization N1instance = new
        TestStaticNonStaticInitialization();

    Console.WriteLine("Create_instance_#2");

    TestStaticNonStaticInitialization N2instance = new
        TestStaticNonStaticInitialization();

}
```

### 6.3.4 Контрольні запитання

1. Що таке конструктор?

2. Оголошення конструктора.
3. Ключове слово `this`.
4. Що таке статичний конструктор?
5. Коли викликається статичний конструктор?
6. Які функції виконує статичний конструктор?
7. Назвіть порядок ініціалізації полів.
8. Назвіть порядок виклику конструкторів.

## 6.4 Властивості

Ми вже навчилися використовувати для доступу до полів класу методи. За допомогою методів ми будемо логіку оброблення даних у програмі. Але в мові C# передбачені спеціальні методи, які називаються **властивості**. Вони призначені для читання значення поля або для встановлення відповідного значення.

У простому випадку синтаксис властивостей має такий вигляд:

[модифікатор доступу] тип назва

```
{
    get
    {
        return ім'я_поля;
    }
    set
    {
        ім'я_поля = value;
    }
}
```

Визначення властивості містить блоки `get` і `set`. У блоці `get` ми зчитуємо значення поля, а в блоці `set` встановлюємо необхідне значення, яке позначається ключовим словом `value`.

```
class Elevator
{
    private int number;
    public int Height;
    public int Width;
    public int Length;
    public readonly int Capacity;
```

```
void Elevator(string S, int H, int W, int L, int C)
{
    Number = 1205;
    Height = H;
    Width = W;
    Length = L;
    Capacity = C;
}

public int Number
{
    get
    {
        return number;
    }
    set
    {
        number = value;
    }
}
```

Тепер ми можемо записати в методі Main встановлення та зчитування значення у поле number:

```
// після виклику конструктора поле number буде мати значення 1398
Elevator MyElevator = new Elevator(1398,2,4,6,1800);

MyElevator.Number = 1208; /* викликається блок set. Значення 1208
    передається у властивість через value */
```

Властивості рекомендується використовувати, коли необхідно при встановленні значення полю виконувати додаткові дії. Наприклад, перевіряти дані на коректність. У даному прикладі значення поля number не може бути від'ємним. В такому разі можна записати:

```
public int Number
{
    get
    {
        return number;
    }
    set
    {
        if (value >= 0) // перевіряємо значення на коректність
        {
            number = value;
        }
        else
        {
            Console.WriteLine("Incorrect_input");
        }
    }
}
```

Необов'язково використовувати у властивості обидва блоки get і set. Наприклад, можна опустити блок зчитування значення. Тобто

можна зробити поле із властивістю "лише запис". Це записується таким чином:

```
public int Number
{
    set
    {
        if (value > 0) // перевіряємо значення на коректність
        {
            number = value;
        }
        else
        {
            Console.WriteLine("Incorrect_input");
        }
    }
}
```

Аналогічно можна не писати блок set, тобто зробити поле із властивістю "лише читання".

У наведеному прикладі ми використовували модифікатор доступу public. Він впливає на обидва блоки. Але якщо ми хочемо для блоків встановити різні типи доступів, то це допускається мовою C#.

```
public int Number
{
    get
    {
        return number;
    }
    private set
    {
        number = value;
    }
}
```

Тепер встановлення властивості є допустимим лише в класі.

Також припустимо використовувати так звані автоматичні властивості.

```
class Elevator
{
    public int Number {get; set;}
    public int Height {get; set;}
    public int Width {get; set;}
    public int Length {get; set;}
    public readonly int Capacity;

    void Elevator(string S, int H, int W, int L, int C)
    {
        Number = 1205;
        Height = H;
        Width = W;
        Length = L;
    }
}
```

```
        Capacity = C;
    }
}
```

При їх використанні необхідний код програми дописує компілятор. Так запис

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

можна вважати еквівалентним такому:

```
int field_for_Number_property;
public int Number
{
    get
    {
        return field_for_Number_property;
    }

    set
    {
        field_for_Number_property = value;
    }
}
```

Однак слід зауважити, що автоматична властивість також не може бути доступною лише для читання або для запису. Для того щоб зробити одну з відповідних двох форм доступу недоступною поза межами класу, можна використовувати модифікатор `private`.

Для надання початкових значень властивостям можна використовувати ініціалізатори так само, як і при ініціалізації полів. Синтаксис ініціалізації такий самий. Покажемо це на прикладі:

```
using System;

namespace InitPropTest
{
    class Elevator
    {
        public int Number { get; set; }
        public int Height { get; set; }
        public int Width { get; set; }
        public int Length { get; set; }
        public int Capacity { get; set; }
    }

    class Program
    {

        static void Main(string[] args)
        {
            Elevator MyElevator = new Elevator {Number = 1208, Height =
                2, Width = 3, Length = 2, Capacity = 800};
            Console.WriteLine("Number=" + MyElevator.Number + "Height
                =" +
                MyElevator.Height + "Width=" + MyElevator.Width +
```

```

        "_Length_" + MyElevator.Length + "_Capacity_" +
        MyElevator.Capacity);
    Console.ReadKey();
    }
}
}

```

У даному прикладі значення властивостей задаються за допомогою ініціалізатора об'єкта.

Властивості не визначають місце в пам'яті комп'ютера для зберігання даних, тому не можуть використовуватись як параметр методу `ref` чи `out`.

### 6.4.1 Контрольні запитання

1. Що таке властивість?
2. Синтаксис властивостей.
3. Блоки `set` та `get`.
4. Чи може властивість не мати блоків `set` та/або `get`?
5. Автоматичні властивості.

## 6.5 Індексатори

Розглянемо таку задачу. Нехай необхідно на табло виводити напис про номер поверху при натисканні на кнопку в ліфті. Яким чином можна в класі `Elevator` реалізувати цю функціональність? Найпростіше рішення – додати як член класу поле масив

```
public int[] Batton;
```

і для звертання до виводу цього напису використовувати

```
MyElevator.Batton[i];
```

де `i` номер поверху. Однак `C#` пропонує більш елегантний і зрозумілий розв'язок – індексатори. Індексатори дозволяють розглядати екземпляр класу як масив. Для визначення індексатора застосовується такий синтаксис:

```

атрибути
[] модифікатори[] тип this [ списокформальнихпараметрів ]
{ set get }

```

де:

- атрибути — додаткова інформація про індексатор, що дозволяє дати спеціальне ім'я індексатору, щоб його можна було використовувати з коду на інших мовах, що не підтримують індексатори;
- модифікатори — модифікатори доступу; до індексатора можна застосовувати майже всі модифікатори C#, за виключенням `static`;
- список формальних параметрів — указує параметри, за допомогою яких здійснюється індексація. Він передається в `get` і `set`, які використовуються в індексаторі так само, як у властивостях, `get` застосовується для значення, яке необхідно повернути, а `set` — для зміни цього значення.

Слід зазначити, що індексатори розрізняються за кількістю й типами формальних параметрів, але не мають імен (на відміну від властивостей).

Наша програма, яка розв'язує задачу виглядає так:

```
using System;

namespace TestIndexer
{
    class ElevatorWithIndexer
    {
        public int height;
        public int width;
        public int length;
        public int capacity;
        private string[] floor;
        private uint AmmountOfFloor;

        public ElevatorWithIndexer(uint Count)
        {
            // опис конструктора
            floor = new string[Count];
            AmmountOfFloor = Count;
            height = 2;
            width = 3;
            length = 1;
            capacity = 800;
        }

        public uint Size
        {
            // отримання розміру масиву
            get
            {
                return AmmountOfFloor;
            }
        }

        public string this[uint index]
        {
            // опис індексатора
            get
```

```

        {
            if (index > -1 && index < Size)
                return floor[index]; // отримання елементу
            else return "Wrong_number_of_floor";
        }
        set
        {
            floor[index] = value; // встановлення значення
        }
    }

    public void Up()
    {
        Console.WriteLine("Up");
    }
    public void Down()
    {
        Console.WriteLine("Down");
    }
}

class Program
{
    static void Main(string[] args)
    {
        ElevatorWithIndexer MyElevator = new ElevatorWithIndexer
            (3);
        MyElevator[0] = "first";
        MyElevator[1] = "second";
        MyElevator[2] = "third";
        for (uint i = 0; i < MyElevator.Size; i++)
        {
            Console.WriteLine("Number_of_floor_is:{0}",
                MyElevator[i]);
        }
        Console.ReadKey();
    }
}

```

У класі `ElevatorWithIndexer` ми оголосили масив рядків `private string[] floor` для зберігання назви поверху й поле `AmountOfFloor` для зберігання кількості поверхів у будинку. Для оголошення масивів також можна використовувати клас `ArrayList` бібліотеки класів `.NET Framework`, який призначений для зберігання сукупності об'єктів. Конструктор `public ElevatorWithIndexer(int Count)` має параметр `int Count` для створення масиву рядків `floor`. У класі також з'явилася властивість `public int Size`. Вона призначена для повернення поточного розміру масиву, коли це знадобиться програмі.

Розглянемо докладніше опис і роботу індексатора.

```

public string this[int index]
{
    get
        // опис індексатора

```

```
{
    if (index > -1 && index < Size)
        return floor[index];          // код для одержання значення елемента
    else return "Wrong_number_of_floor";
}
set
{
    floor[index] = value;              // код для задання елементу
}
}
```

Він повертає в ролі результату своєї роботи рядок і має модифікатор `public`, тому що повинен бути доступний поза класом. Нагадаємо, що екземпляр класу виступає в ролі масиву. Індексатор дозволяє звертатися до елементу як `MyElevator[i]` (або `ElevatorWithIndexer[i]`, у випадку, якщо індексатор оголошений як `static`). В описі індексатора використовується параметр `int index`. Цей параметр задає номер поточного елемента масиву, який індексатор повинен повернути або задати. Тепер ми в програмі можемо для виводу назви номера поверху використовувати простий синтаксис звертання до елемента масиву.

```
for (int i = 0; i < MyElevator.Size; i++)
{
    Console.WriteLine("Number_of_floor_is:{0}", MyElevator[i]);
}
```

Аналогічним чином визначаються багатовимірні індексатори:

```
public string this[int index1, int index2]
{
    // опис індексатору
    get
    {
        // код для одержання елементу
    }
    set
    {
        // код для задання елементу
    }
}
```

Як індекси не обов'язково використовувати лише цілі числа. Можна також використовувати рядки або символи. Також слід зауважити, що індексатори можуть бути перевантажені в окремому класі або структурі. Тобто, якщо є необхідність звертатись з використанням числового індексу або символьного значення, можна визначити декілька індексаторів. Наприклад, описати ще один індексатор

```
public string this[char index]
{
    // опис індексатора
    get
```

```
    {
        if (index >= '0' && index < Size)
            return floor[index]; // отримання елементу
        else return "Wrong_number_of_floor";
    }
    set
    {
        floor[index] = value; // встановлення значення
    }
}
```

та використовувати для індексації символи:

```
class Program
{
    static void Main(string[] args)
    {
        ElevatorWithIndexer MyElevator = new ElevatorWithIndexer('3');
        MyElevator['0'] = "first";
        MyElevator['1'] = "second";
        MyElevator['2'] = "third";
        for (char i = '0'; i < MyElevator.Size; i++)
        {
            Console.WriteLine("Number_of_floor_is:{0}", MyElevator[i]);
        }
        Console.ReadKey();
    }
}
```

При описі індексаторів можна використовувати ключове слово `param`:

```
public IEnumerable<string> this[params char[] keys]
{
    get { return keys.Select(key => floor[key]).AsEnumerable(); }
}
```

Тоді ми можемо використовувати такий вираз:

```
MyElevator['0', '1', '2'];
```

Припустимо, що ми будемо використовувати обидва індексатори. Тоді якщо передавати масив або кілька аргументів як параметри індексатору, то отримаємо `IEnumerable` як результат. Якщо передавати один аргумент, то отримаємо одне значення.

### 6.5.1 Контрольні запитання

1. Що таке індексатор?
2. Синтаксис індексаторів.
3. Які типи даних можна використати в якості індексів?

## 6.6 Інкапсуляція

Інкапсуляція – це контроль доступу до внутрішніх даних та процедур екземплярів. Завжди необхідно намагатися використовувати базову концепцію: ніколи не визначати поля даних з відкритим доступом. Зазвичай таке відбувається, коли визначається невеликий клас і розробнику або ліньки як слід обміркувати його, або він просто поспішає. Необхідно зробити так, щоб користувачі екземпляра спілкувалися з ним лише керованим чином. Зазвичай, це означає організацію взаємодії з екземпляром лише через методи цього екземпляра (або властивості, які є викликами методів). Таким чином, екземпляр розглядаються як "чорний ящик". За допомогою інкапсуляції можна спроектувати такий дизайн, що гарантує, що внутрішній стан екземпляра ніколи не буде порушений. Проста ілюстрація до сказаного: створимо фіктивний екземпляр, що є прямокутником.

```
class MyRectangle
{
public uint width;
public uint height;
}
```

Тут ви бачите найпримітивніший приклад класу прямокутника. Нас цікавить лише ширина та довжина сторін прямокутника. Зрозуміло, що корисний клас прямокутника, призначений для використання в графічному механізмі, також повинен мати координати початкової точки, але для простоти прикладу обмежимося шириною і довжиною. Тому два поля – `width` та `height` – оголошені загальнодоступними. Можливо, це викликано поспіхом під час проектування цього базового маленького класу. Тепер припустимо, що згодом цей маленький клас застосовувався у багатьох місцях програми. Не варто забувати, що цей клас прямокутника в тому вигляді, як він є, не надто корисний, і тому було вирішено зробити його кориснішим. Припустимо, існує клієнтський код, який використовує цей клас прямокутника і якому потрібно обчислити площу прямокутника. Принципи об'єктно-орієнтованого програмування підказують нам, що найкращий спосіб зробити - це дозволити екземплярам `MyRectangle` самотійно повідомляти клієнту значення площі. Тому давайте так і вчинимо:

```
class MyRectangle
{
public uint width;
public uint height;
public uint GetArea()
```

```
{  
return width*height;  
}  
}
```

Як ви бачите, до класу доданий новий член - метод `GetArea`. При виклику його, `MyRectangle` обчислить площу і поверне результат. Тепер є базовий клас прямокутника, який має одну допоміжну функцію, яка покликана трохи полегшити життя користувачам, якщо вони забажають дізнатися площу прямокутника. Але припустімо, що виникла причина попередньо обчислити площу прямокутника, щоб щоразу при виклику методу `GetArea` не довелося обчислювати її заново. Можливо, так потрібно зробити тому, що відомо, що `GetArea` буде викликатися багаторазово з одним і тим же екземпляром протягом усього часу його існування. Тоді клас `MyRectangle` виглядає приблизно так:

```
class MyRectangle  
{  
public uint width;  
public uint height;  
public uint area;  
public uint GetArea()  
{  
    return area;  
}  
}
```

Придивившись уважніше, ви помітите помилки. Мислення стандартного програміста таке: усі поля загальнодоступні. Це дозволяє користувачу екземплярів класу `MyRectangle` мати прямий доступ до його полів. Який сенс у методу `GetArea`, якщо користувач може безпосередньо звернутися до поля `area`? Так, можливо, варто було б зробити поле `area` приватним? Таким чином, для отримання площі прямокутника клієнти змушені були викликати `GetArea`. Це, безперечно, крок у правильному напрямку, як ілюструє такий код:

```
class MyRectangle  
{  
public uint width;  
public uint height;  
private uint area;  
public uint GetArea()  
{  
    if (area == 0 )  
    {  
        area=width*height;  
    }  
    return area;  
}  
}
```

Поле `area` зроблено приватним, змушуючи споживача викликати метод `GetArea`, щоб отримати площу прямокутника. Однак під час роботи над прикладом стало ясно, що в якийсь момент має бути обчислена площа прямокутника. Через лінощі було вирішено перевіряти значення поля `area` перед поверненням, і якщо воно дорівнює 0, це означає, що воно має бути попередньо обчислене. Це — груба спроба оптимізації, але тепер площа обчислюється лише за необхідності. Припустимо, що користувач екземпляра прямокутника ніколи не забажає дізнатися про його площу. Тоді, за умови використання попереднього коду, йому ніколи не доведеться витратити час на обчислення цієї площі. Зрозуміло, у цьому дещо надуманому прикладі така оптимізація буде дуже незначною. Але якщо ви трохи замислитесь, то, напевно, уявите собі приклад, який істотно виграє від застосування відкладеного обчислення. Подумайте про доступ до бази даних мережі з повільним доступом, коли під час виконання потрібні лише кілька полів таблиці. Або для того ж об'єкта доступу до бази даних може виявитися занадто дорогою операція підрахунку рядків у таблиці. Тому за потреби необхідно застосовувати описаний вище підхід. У класі прямокутника все ще є кричуща проблема. Оскільки поля `width` та `height` є загальнодоступними, що станеться, якщо користувач змінить одне із значень після виклику `GetArea` на екземплярі? І тут буде отримано найгірший приклад неузгодженості нутроців об'єкта. Цілісність стану об'єкта буде порушена. Це безперечно погана ситуація. Таким чином, у коді, як і раніше, присутня помилка. Поля `width` і `height` також мають бути зроблені приватними:

```
class MyRectangle
{
private uint width;
private uint height;
private uint area;
public uint Width
{
Get

{
returnwidth;
}
set{
width=value;
ComputeArea();
}
public uint Height
{
get
{
returnheight;
}
```

```
}  
Set  
{  
    height=value;  
    ComputeArea();  
}  
public uint Area  
{  
    Get  
    {  
        return area;  
    }  
    private void ComputeArea()  
    {  
        area=width*height;  
    }  
}
```

Ця версія `MyRectangle` виглядає набагато краще. Після того, як поля `width` і `height` були зроблені приватними, стало ясно, що споживачеві цих об'єктів потрібен якийсь спосіб встановлення та отримання значень ширини та висоти. І тут знадобляться методи-властивості `C#`.

Тепер зміни внутрішнього стану будуть оброблятися в тілі методу, а методи, що викликаються при цьому, відносяться до набору спеціально іменованих методів класу. Тепер з'явилася можливість суворішого контролю доступу до полів екземпляра. І разом із цим контролем прийшов більш високий ступінь інкапсуляції.

Можна ефективно керувати внутрішнім станом екземпляра, тому він ніколи не стане неузгодженим. Гарантувати цілісність стану екземпляра, коли зовнішні сутності мають доступом до внутрішньому стану “з чорного ходу”, не можна. У даному прикладі точно відомо, коли змінюються поля `width` і `height`, тому він може вжити необхідних дій для обчислення нової площі. Отже, ми бачимо, що незначний обсяг додаткової роботи із забезпечення інкапсуляції згодом багаторазово відшкодується.

Одна з найчудовіших властивостей інкапсуляції, яку слід добре запам'ятати, полягає в тому, що при використанні властивостей внутрішній стан екземпляра може змінюватися для підтримки іншого алгоритму, що не зачепить його користувачів. Іншими словами, інтерфейс, видимий користувачу (також званий контрактом), не змінюється. Наприклад, у фінальній реалізації класу `MyRectangle` – це площа.

У цьому полягає міць інкапсуляції. Коли змінюється внутрішня реалізація екземпляра, а користувачі, що його використовують, можуть не змінюватися, — значить, інкапсуляція працює так, як повинна.

### 6.6.1 Контрольні запитання

1. Що таке інкапсуляція?
2. Завдання інкапсуляції.
3. Приклад використання інкапсуляції.

## 6.7 Спадкування класів

Спадкування – це здатність передавати особисті ознаки та особливості нащадкам.

Спадкування є одним з механізмів, що забезпечують повторне використання коду в об'єктно-орієнтованих мовах програмування.

При спадкуванні, члени базового (батьківського) класу, такі, як поля, властивості й методи, успадковуються у класі-нащадку. На додаток до них у класі-нащадку можуть визначатися НОВІ властивості й методи і можуть перевизначатися успадковані методи. Таким чином, спадкування дозволяє створювати нові класи, які повторно використовують, розширюють та/або змінюють поведінку інших класів.

Клас, члени якого успадковуються, називається базовим класом, а клас, який успадковує ці члени, називається похідним класом.

Похідний клас успадковує всі члени базового класу, за винятком конструкторів. Припускається ситуація, коли базовий клас сам є похідним від іншого класу.

Розглянемо класичний приклад. Нехай необхідно задати тип, який описує людину надаючи їй ім'я та прізвище. Це можна реалізувати за допомогою такого класу:

```
class Person
{
    private string FirstName;
    private string LastName;
    public string FirstN
    {
        get { return FirstName; }
        set { FirstName = value; }
    }

    public string LastN
    {
        get { return LastName; }
        set { LastName = value; }
    }
}
```

```
    }

    public void Display()
    {
        Console.WriteLine(FirstN + "_" + LastN);
    }
}
```

Тепер опишемо співробітника компанії. Додатково до прізвища й імені цей опис повинен включати ще назву відділу, зарплатню та посаду. Це буде виглядати так:

```
class Employee : Person
{
    string Department;
    string Salary;
    string Position
}
```

Після двокрапки ми записали ім'я базового класу `Person`. Клас `Employee` спадкує всі поля, властивості, методи, що є в класі `Person`.

### 6.7.1 Спадкування й проблеми доступу

Як ми вже говорили, клас нащадок успадковує від базового класу все, що він має. Але скористатися в похідному класі можна не всім. Наприклад у нашому класі `Person` є `private` поле `FirstName`. Тому екземпляр класу `Employee` не може мати прямий доступ до цих даних.

Однак для доступу можна скористатися `public` властивістю `FirstN` для надання значення або його зчитування. Ще одним способом доступу до захищених членів класу є використання модифікатора доступу – `protected`. Цей модифікатор забезпечує вільний доступ до членів базового класу тільки для прямих або непрямих нащадків класу (при умові, що доступ до нестатичних `protected`-членів базового класу з класу-нащадка відбувається через екземпляр класу-нащадка, або типу, побудованого на основі класу-нащадка, наприклад, нащадка цього класу-нащадка).

```
class Person
{
    protected string FirstName;
    protected string LastName;
    public string FirstN
    {
        get { return FirstName; }
        set { FirstName = value; }
    }
    public string LastN
    {
```

```
        get { return LastName; }
        set { LastName = value; }
    }
    public void Display()
    {
        Console.WriteLine(FirstN + "_" + LastN)
    }
}
```

Захищені члени базового класу доступні для всіх прямих і непрямих нащадків даного класу.

### 6.7.2 Припинення спадкування.

Принцип спадкування допускає необмежену глибину спадкування. Похідний клас, що є спадкоємцем базового класу, може у свою чергу так само виявитися в ролі базового класу.

Якщо при розробці класу виникла ситуація, при якій подальше доповнення чи модифікації можливостей класу є недоцільним, клас може бути закритий для наступного спадкування. Для того щоб припинити спадкування, достатньо перед оголошенням класу написати ключове слово `sealed`.

Наприклад, можна припинити спадкування на класі `Employee` написавши так:

```
sealed class Employee : Person
```

Але вже буде помилкою оголосити такий клас:

```
sealed class Administrator: Employee
```

### 6.7.3 Контрольні запитання

1. Що таке спадкування?
2. Які члени успадковує дочірній клас?
3. Модифікатори доступу при наслідуванні.
4. Приклад спадкування.
5. Закриття класу для наслідування.

## 6.8 Поліморфізм

Тепер ми підійшли до такого поняття як поліморфізм (від грец. різноманітність форм).

Під поліморфізмом будемо розуміти здатність приймати різні форми.

### 6.8.1 Підтиповий поліморфізм: перевизначення методів

Повертаючись до нашого прикладу з ліфтом, можна сказати, що реалізація натискання на клавішу вибору поверху для руху ліфта може бути різною: ліфт піднімається/опускається за допомогою мотора й тросів або діє антигравітаційний двигун.

При вивченні підтипового поліморфізму ми будемо виділяти два аспекти. Обидва аспекти мають відношення до спадкування:

Перший – при виконанні програми, екземпляр похідного класу розглядається як екземпляр базового класу, наприклад в аргументах методу, колекції, тобто приймає іншу форму.

Другий – похідний клас перевизначає (override) віртуальний (virtual) метод базового класу. Тобто похідний клас надає власне визначення й реалізацію методу.

Можна сказати, що якщо перший аспект має автоматичну реалізацію, то для реалізації другого необхідно зробити деякі дії.

По-перше, треба створити ієрархію класів, спадкувавши усі класи від одного базового. По-друге, необхідно через однаковий виклик методу базового класу застосувати для обробки даних відповідний метод похідного класу.

З точки зору системи типів C#, похідний клас вважається підтипом його базового класу, отже у C# поліморфізм і спадкування є пов'язаними.

Для вивчення поліморфізму у програмуванні треба ввести ще кілька таких понять, як: віртуальний, перевизначення методу.

Віртуальний метод – це метод, який може бути перевизначений у класі-нащадку. Такий метод може мати свою реалізацію в базовому класі.

Перевизначення методу – це зміна реалізації методу.

Розглянемо класичний приклад. Нехай необхідно для різних геометричних фігур (Коло, Прямокутник, Трикутник) друкувати їхню назву.

Створимо базовий клас Figure, з якого будуть спадковуватися похідні класи для рисування відповідних фігур.

```
public class Figure
{
    // поля класу
```

```
// віртуальний метод для перевизначення
public virtual void Draw()
{
    Console.WriteLine("Фігура");
}
}
```

Тепер створимо похідні класи для рисування:

```
class Circle : Figure
{
    public override void Draw()
    {
        // друк назви фігури Коло
        base.Draw();
        Console.WriteLine("Коло");
    }
}
class Rectangle : Figure
{
    public override void Draw()
    {
        // друк назви фігури Прямокутник
        base.Draw();
        Console.WriteLine("Прямокутник");
    }
}
class Triangle : Figure
{
    public override void Draw()
    {
        // друк назви фігури Трикутник
        base.Draw();
        Console.WriteLine("Трикутник");
    }
}
```

Продемонструємо в програмі обидва аспекти поліморфізму. Створимо список екземплярів класу Figure і занесемо в нього фігури: Коло, Прямокутник, Трикутник. При заповненні списку figures очікується екземпляр класу Figure. Однак, ми передаємо в нього послідовно екземпляри класів Circle, Rectangle, Circle. При цьому відбувається неявне перетворення з типу похідного класу в тип базового класу. У даному фрагменті програми проявляється перший аспект поліморфізму.

```
List<Figure> figures = new List<Figure>();
    shapes.Add(new Rectangle());
    shapes.Add(new Triangle());
    shapes.Add(new Circle());
```

У наступному фрагменті ми демонструємо другий аспект, що пов'язаний із застосуванням у виклику перевизначеного методу похідного класу:

```
foreach (Figure s in shapes)
{
    s.Draw();
}
Console.ReadKey();
}
```

Для кожного екземпляра колекції буде викликаний свій метод за допомогою сигнатури базового. Остаточна програма прийме вигляд:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication7
{
    public class Figure
    {
        // поля класу

        // виртуальний метод для переопределения
        public virtual void Draw()
        {
            Console.WriteLine("Фигура");
        }
    }

    class Circle : Figure
    {
        public override void Draw()
        {
            // печать название фигуры
            base.Draw();
            Console.WriteLine("Круг");
        }
    }

    class Rectangle : Figure
    {
        public override void Draw()
        {
            // печать название фигуры
            base.Draw();
            Console.WriteLine("Прямоугольник");
        }
    }

    class Triangle : Figure
    {
        public override void Draw()
        {
            // печать название фигуры
            base.Draw();
            Console.WriteLine("Треугольник");
        }
    }
}
```

```
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<Figure> shapes = new List<Figure>();
            shapes.Add(new Rectangle());
            shapes.Add(new Triangle());
            shapes.Add(new Circle());

            foreach (Figure s in shapes)
            {
                s.Draw();
            }
            Console.ReadKey();
        }
    }
}
```

Щоб не було плутанини в термінології, розглянемо ще один приклад, що демонструє поліморфізм (перевизначення) і перевантаження методів.

```
using System;
namespace Polymorphism2
{
    class A
    {
        public virtual string DemoPoly(string a)
        {
            return "Base_class._Display_string";
        }
        public virtual string DemoPoly(double a)
        {
            return "Base_class._Display_double";
        }
    }
    class B : A
    {
        public override string DemoPoly(string a)
        {
            return "Inharitance_class._Demo_Poly_string";
        }
        public override sealed string DemoPoly(double a)
        {
            return "Inharitance_class._Demo_Poly_double";
        }
    }
    class Program
    {
        public static void Main(string[] args)
        {
            A a = new A();
            B b = new B();
            Console.WriteLine(a.DemoPoly("test"));
        }
    }
}
```

```

        Console.WriteLine(b.DemoPoly("test"));
        Console.WriteLine(a.DemoPoly(3.14));
        Console.WriteLine(b.DemoPoly(3.14));
        Console.ReadKey();
    }
}

```

У даному прикладі в базовому класі А два методи Demopoly(string a) і Demopoly(double a), які перевантажені (overload) у даному класі - у них різна сигнатура. У похідному від А класі В є ті ж методи, які перевизначені (override). Додатково в класі В метод Demopoly(double a) позначений як запечатаний sealed. Це зроблено для того, щоб припинити віртуальне спадкування. Але слід зауважити, що запечатані методи можна перевизначити у похідному класі, використовуючи ключове слово new:

```

class C : B
{
    public new string Demopoly(double a)
    {
        return "Inheritance_class._Demo_Poly_double";
    }
}

```

Зазвичай, якщо метод Demopoly() викликається за допомогою екземпляра класу С, то викликається саме метод класу С. У випадку, коли для доступу до екземпляра класу С використовується екземпляр класу А чи В, то буде викликаний метод Demopoly() з класу В:

```

using System;
namespace Polymorphism3
{
    class A
    {
        public virtual string Demopoly(string a)
        {
            return "Base_class._Display_string";
        }
        public virtual string Demopoly(double a)
        {
            return "Base_class._Display_double_from_A";
        }
    }
    class B : A
    {
        public override string Demopoly(string a)
        {
            return "Inheritance_class._Demo_Poly_string";
        }
        public override sealed string Demopoly(double a)
        {
            return "Inheritance_class._Demo_Poly_double_from_B";
        }
    }
}

```

```
    }  
}  
class C : B  
{  
    public new string Demopoly(double a)  
    {  
        return "Inharitance_class._Demo_Poly_double_from_C";  
    }  
}  
class Program  
{  
    public static void Main(string[] args)  
    {  
  
        C c = new C();  
        A a = (A)c;  
        B b = (B)c;  
        Console.WriteLine(a.Demopoly(3.14)); // надруковано  
        Inharitance class. Demo Poly double from B  
        Console.WriteLine(b.Demopoly(3.14)); // надруковано  
        Inharitance class. Demo Poly double from B  
        Console.WriteLine(c.Demopoly(3.14)); // надруковано  
        Inharitance class. Demo Poly double from C  
        Console.ReadKey();  
    }  
}
```

Не слід плутати поліморфізм із сховуванням (hide) методу. Якщо необхідно, щоб метод похідного класу мав таке саме ім'я як і метод базового класу, але не брав участь у віртуалізації, то можна за допомогою ключового слова `new` сховати реалізацію методу базового класу:

```
using System;  
  
namespace HideTest  
{  
    class A  
    {  
        public void Test()  
        {  
            Console.WriteLine("test_from_Class_A");  
        }  
    }  
    class B: A  
    {  
        public new void Test()  
        {  
            Console.WriteLine("test_from_Class_B");  
        }  
    }  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            B test2 = new B();  
        }  
    }  
}
```

```

        test2.Test();           // test from Class B
        A test1 = (A)test2;
        test1.Test();          // test from Class A
        Console.ReadKey();
    }
}

```

У даному прикладі виклик методу `Test()` в операторі `test2.Test()` не має відношення до поліморфізму. Сховування методів використовується у випадку, коли, наприклад, у бібліотеці є клас `A` і від нього спадкується клас `B`, в якому описаний метод `Test()`. Припустимо, що виходить нова версія класу `A`, але вже цей клас теж містить в собі зовсім інший метод `Test()`. Тоді, щоб не було плутанини, слід метод `Test()` в класі `B` описати як `new`.

Розглянемо ще випадки використання поліморфізму. У наступному прикладі клас `Employee` має весь функціонал класу `Person`. Але нехай нам необхідно для співробітника друкувати не прізвище, а назву відділу. Таке припустимо, але необхідно дещо дописати в описі. Для цього в класі `Person` указати, що цей метод буде перевантажуватися у нащадку. Це робиться за допомогою модифікатора `virtual`. В похідному класі відповідний метод, позначається модифікатором `override`.

```

class Person
{
    ...
    public virtual void Display()
    {
        Console.WriteLine(FirstN + "_" + LastN);
    }
}

class Employee : Person
{
    ...
    public override void Display()
    {
        Console.WriteLine(FirstN + "_" + LastN + "department" +
            department);
    }
}

```

Залишилось питання, а що робити з полями базового класу, що спадкуються, у класі нащадку? Для цих полів в конструкторі класу нащадка достатньо указати на конструктор базового класу за допомогою ключового слова `base`.

```

class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

```

```
public Person(string NameLast, string NameFirst)
{
    // конструктор базового класу
    FirstName = NameFirst;
    LastName = NameLast;
}

public virtual void Display()
{
    Console.WriteLine(FirstName + "_" + LastName);
}
}

class Employee : Person
{
    public string Department { get; set; }
    public string Salary { get; set; }
    public string Position { get; set; }

    public Employee(string NameFirst, string NameLast, string Dept,
        string Sal, string Pos) : base(NameFirst, NameLast)
    {
        // конструктор класу нащадка ініціалізує
        // поля базового класу (NameFirst, NameLast)
        // за допомогою base

        Department = dept;
        Salary = Sal;
        Position = Pos;
    }

    public override void Display()
    {
        base.Display(); // викликаємо метод базового класу
        Console.WriteLine("Department:_" + Department);
    }
}
```

Як ми бачимо, можна використовувати ключове слово `base` не лише для конструктора. В нашому прикладі ми в похідному класі викликаємо метод базового класу.

### 6.8.2 Спеціальний (ad-hoc) поліморфізм: перевантаження операторів

У C# існує ряд операторів для роботи із вбудованими типами даних. Це оператори «+», «-», «!», «==», «!=» й інші. Так, бінарний оператор «+» виконує операцію додавання над чисельними типами даних. Але ми цей оператор використовували над рядками для виконання операції конкатенації. Це відбувається завдяки тому, що оператор «+» перевантажений у класі `String`.

Для перевантаження операторів використовуються `public static`

методи члени класів із ключовим словом `operator`. Ми будемо перевантажувати унарні й бінарні оператори. Унарні оператори – це оператори, які мають один операнд, бінарні – два.

До унарних операторів, які можна перевантажувати, належать: «+», «-», «!», « », «++», «-», «true», «false».

До бінарних операторів, які можна перевантажувати, належать: «+», «-», «\*», «/», «

Оператори «==», «!=», «<», «>», «<=», «>=» теж можна, але тільки парами – якщо перевантажується «==», то необхідно перевантажувати «!=» й навпаки.

Інші оператори перевантажувати не можна, але можна певні оператори обчислити за допомогою операторів, що допускають перевантаження. Синтаксис перевантаження операторів такий: Перевантаження унарного оператора:

```
public static тип[] operator оператортип[]([ операнду] операнд[])
{
    //нова реалізація оператора
}
```

Перевантаження бінарного оператора:

```
public static тип[] operator оператортип[]([ операнду1] операнд[1], тип[
    операнду2] операнд[2])
{
    // нова реалізація оператора
}
```

Модифікатори `public` и `static` обов'язкові. Замість `[оператор]` може бути будь-який оператор, який можна перевантажити.

Ознакою того, що робиться саме перевантаження оператора, служить ключове слово `operator`, за яким повинен розташовуватися знак оператора, що перевантажується. У якості операнду при перевантаженні унарного оператора може виступати лише об'єкт того ж класу, що й клас, для якого ми робимо перевантаження. При перевантаженні бінарного оператора хоча б один із двох операндів повинен бути того ж класу, для якого робиться це перевантаження. Слід зазначити, що параметри можна передавати в метод для реалізації відповідної операції лише за значенням, `ref` і `out` не слід використовувати. У C# операції скороченого присвоювання автоматично виконуються при перевантаженні відповідних бінарних операцій. Тобто, якщо в класі вже перевантажені операції `+` и `-`, ми можемо використовувати `i +=`, `i -=`.

Розглянемо простий приклад. Нехай необхідно працювати із точками на площині. Точка на площині має дві координати (x,y).

Визначимо бінарну операцію «+», що застосується до двох точок таким чином:

$$(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2);$$

Аналогічно визначається бінарна операція «-»:

$$(x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2);$$

Визначимо унарні операції:

$$(x, y)++ \text{ це } (x+1, y+1)$$

$$(x, y)- \text{ це } (x-1, y-1)$$

$$-(x, y) \text{ це } (-x, -y).$$

Розглянемо програму, що демонструє перевантаження операторів.

```
using System;

class Point2D
{
    public int x;
    public int y;

    public Point2D()
    {
        x = 0;
        y = 0;
    }

    public Point2D (int x, int y)
    { this.x = x; this.y = y; }

    // повертає (нову) точку, кожна з координат якої більше за відповідну
    // координату точки-аргументу на одиницю
    public static Point2D operator ++ (Point2D par)
    {
        return new Point2D(par.x+1, par.y+1);
    }

    // повертає (нову) точку, кожна з координат є сумою
    // відповідних координат двох точок аргументів
    public static Point2D operator +(Point2D par1, Point2D par2)
    {
        return new Point2D(par1.x + par2.x, par1.y + par2.y);
    }

    // повертає (нову) точку, кожна з координат якої менше за відповідну
```

```

// координату точки-аргументу на одиницю
public static Point2D operator -- (Point2D par)
{
    return new Point2D(par.x-1, par.y-1);
}

// повертає (нову) точку, кожна з координат є різницею
// відповідних координат двох точок аргументів
public static Point2D operator -(Point2D par1, Point2D par2)
{
    return new Point2D(par1.x - par2.x, par1.y - par2.y);
}
public void Display(Point2D key)
{
    Console.WriteLine("x={0},y={1}", key.x, key.y);
}

static void Main(string[] args)
{
    Point2D p1 = new Point2D();
    Point2D p2 = new Point2D();
    Point2D p3 = new Point2D();
    Point2D p4 = new Point2D();

    p1 = p2++; // Постфіксна.
    p1.Display(p1); // (0,0)
    p2.Display(p2); // (1,1)
    p3=++p4; // Префіксна.
    p3.Display(p3); // (1,1)
    p4.Display(p4); // (1,1)
    p1 = p4 - p3;
    p1.Display(p1); // (0,0)
}
}

```

При перевантаженні бінарних операцій не обов'язково передавати їй операнди однакового типу. Але важливо пам'ятати, що якщо необхідно передавати операнди в будь-якому порядку, то необхідно реалізовувати обидві версії методу.

З перевантаженням бінарних операцій питань не повинно виникати. Розберемо детальніше перевантаження унарних операцій.

Нехай є дві змінні типу Point2D.

```

Point2D a = new Point2D(0,0);
Point2D b = new Point2D(0,0);

```

Обчислимо `a = b++`;

Цей запис матиме таку семантику:

a) Point2D temp = b;

Змінна `temp` класу `Point2D` буде вказувати на ту саму ділянку пам'яті, що й `b`, тобто значення `temp -> (x,y)` і `b -> (x,y)` співпадають.

б) `b = operator++(b);`

Змінна `b` буде вказувати на НОВУ ділянку пам'яті, де зберігається значення на 1 більше, `(x+1,y+1)` ніж вихідне значення `(x,y)`. Змінна продовжує `temp` вказувати на вихідне значення `(x,y)`.

в) `a = temp;`

Змінна `y` стане вказувати на ту ж саму ділянку пам'яті, що й `temp`, тобто на `(x,y)`. Тоді, якщо `b` вказувало на значення `(0,0)`, то після виконання

`a = b++;`

`a` буде вказувати на ту ж саму ділянку, що зберігає `(0,0)`. Тобто при перевантаженні повністю збережено семантику стандартних унарних операцій.

### 6.8.3 Перевантаження операторів порівняння

Тепер розглянемо як перевантажувати оператори порівняння. У мові `C#` усі класи спадкуються від класу `Object`, тому поряд із стандартними операторами нам необхідно також перевантажити два методи цього класу `Equals` і `GetHashCode`. Метод `Equals` дозволяє порівнювати два об'єкти, а метод `GetHashCode` обчислює хеш-код об'єкту. Якщо метод `Equals` вважає, що два об'єкти рівні, то метод `GetHashCode` повинен згенерувати для них однакові значення. Нагадаємо, що деякі оператори необхідно перевантажувати парами: `==` і `!=`, `<` і `<=`, `>` і `>=`. Покажемо, як перевантажити оператори порівняння для нашого прикладу точок на площині. Будемо вважати, що точки співпадають, якщо їхні координати рівні, одна точка менше/більше за іншу, якщо відповідні координати менше/більше.

```
public static bool operator ==( Point2D a, Point2D b)
{
    if (a.x == b.x && a.y == b.y)
        return true;
    else return false;
}
public static bool operator !=( Point2D a, Point2D b)
```

```

{
    if (a.x != b.x || a.y != b.y)
        return true;
    else return false;
}
public override bool Equals(object o)
{
    if(o.GetType() != GetType())
        return false;
    Point2D t = (Point2D) o;
    return (x == t.x && y == t.y);
}
public override int GetHashCode ()
{
    return 0;
}

```

Розглянемо ще один приклад для закріплення. Давайте напишемо реалізацію засобами C# операції із звичайними дробами:

```

public class Rational
{
    int a; // Чисельник
    int b; // Знаменник

    // Конструктори й інші методи
    .....
}

```

Наведемо приклади перевантаження унарних операторів для нашого класу Rational. Перевантаження операнду - (зміна знака):

```

public static Rational operator - (Rational x)
{
    Rational t = new Rational();
    t.a = - x.a;
    t.b = x.b;
    return t;
}

```

Тут у методі створюється новий об'єкт t типу Rational і потім обчислюються значення полів цього об'єкта. Метод повертає результат — змінну t типу Rational, вихідний об'єкт залишається незмінним. Приклад використання операції:

```

Rational x = new Rational(2,5);
Rational y = new Rational();
y = -x;

```

Після цих дій змінна y буде мати значення -2/5.

Перевантаження операнду ++ (інкремент):

```

public static Rational operator ++ (Rational x)
{
    Rational t = new Rational();
    t.a = x.a + x.b;
}

```

```
t.b = x.b;  
return t;  
}
```

Перевантажимо бінарний операнд +:

```
public static Rational operator + (Rational x, Rational y)  
{  
    int t1 = x.a * y.b + x.b * y.a;  
    int t2 = x.b * y.b;  
    socrat(ref t1, ref t2);  
    return new Rational(t1, t2);  
}
```

Завдяки тому, що при виконанні операції додавання ми можемо отримати дріб, для якого необхідне скорочення, то викликаємо закритий статичний метод Euklid(). Цей метод по скороченню дробу необхідно робити статичним, тому що зі статичного методу (а метод, що перевантажує додавання, зобов'язаний бути статичним!) можна викликати теж лише статичний метод.

Перевантажимо оператори «>» і «<» і реалізуємо методи Equals() і GetHashCode():

```
public static bool operator > (Rational x, Rational y)  
{  
    if(x.a * y.b > x.b * y.a)  
        return true;  
    else  
        return false;  
}  
public static bool operator < (Rational x, Rational y)  
{  
    if(x.a * y.b < x.b * y.a)  
        return true;  
    else  
        return false;  
}  
public override bool Equals(object o)  
{  
    if(o.GetType() != GetType())  
        return false;  
    Rational t = (Rational) o;  
    return (a == t.a && b == t.b);  
}  
public override int GetHashCode()  
{  
    return 0;  
}
```

Остаточно програма має такий вигляд:

```
using System;  
namespace OperandOver  
{  
    public class Rational  
    {
```

```
int a; // Чисельник
int b; // Знаменник

public Rational()
{
    a = 0;
    b = 1;
}
public Rational(int a0, int b0)
{
    a = a0;
    b = b0;
    if (b == 0)
    {
        b = 1;
        Console.WriteLine("Неприпустиме_значення_знаменника:{0}._\nЗаміняємо_на_1", b0);
    }
    else if (b < 0)
    {
        a = -a;
        b = -b;
    }
    Euklid(ref a, ref b);
}
// Заміна знака
public static Rational operator -(Rational x)
{
    Rational t = new Rational();
    t.a = -x.a;
    t.b = x.b;
    return t;
}
// Інкремент
public static Rational operator ++(Rational x)
{
    Rational t = new Rational();
    t.a = x.a + x.b;
    t.b = x.b;
    return t;
}
// Додавання дробів
public static Rational operator +(Rational x, Rational y)
{
    int t1 = x.a * y.b + x.b * y.a;
    int t2 = x.b * y.b;
    Euklid(ref t1, ref t2);
    return new Rational(t1, t2);
}
// Множення дробів
public static Rational operator *(Rational x, Rational y)
{
    int t1 = x.a * y.a;
    int t2 = x.b * y.b;
    Euklid(ref t1, ref t2);
    return new Rational(t1, t2);
}
// Скорочення дробі
```

```
private static void Euklid(ref int a, ref int b)
{
    int x = Math.Abs(a), y = b;
    while (x > 0 && y > 0)
    {
        if (x > y)
            x = x % y;
        else
            y = y % x;
    }
    int k = x + y;
    if (k > 1)
    {
        a /= k;
        b /= k;
    }
}
// Друк дробі
public void Display()
{
    Console.WriteLine("{0}/{1}", a, b);
}
// Перевантаження оператора порівняння
public static bool operator >(Rational x, Rational y)
{
    if (x.a * y.b > x.b * y.a)
        return true;
    else
        return false;
}
public static bool operator <(Rational x, Rational y)
{
    if (x.a * y.b < x.b * y.a)
        return true;
    else
        return false;
}
public static bool operator ==(Rational x, Rational y)
{
    if (x.a == y.a && x.b == y.b)
        return true;
    else
        return false;
}
public static bool operator !=(Rational x, Rational y)
{
    if (x.a != y.a || x.b != y.b)
        return true;
    else
        return false;
}
public static bool operator >=(Rational x, Rational y)
{
    if (x.a * y.b >= x.b * y.a)
        return true;
    else
        return false;
}
```

```

public static bool operator <=(Rational x, Rational y)
{
    if (x.a * y.b <= x.b * y.a)
        return true;
    else
        return false;
}
public override bool Equals(object o)
{
    if (o.GetType() != GetType())
        return false;
    Rational t = (Rational)o;
    return (a == t.a && b == t.b);
}
public override int GetHashCode()
{
    return 0;
}
}
class Program
{
    public static void Main(string[] args)
    {
        Rational x = new Rational(2, 5);
        Console.Write("Вихідний_дріб_x=");
        x.Display();
        Rational y = new Rational();
        y = -x;
        Console.Write("y=-x:_y=");
        y.Display();
        if (x > y)
            Console.WriteLine("x_>_y");
        else
            Console.WriteLine("x_не_більше_y");
        if (x >= y)
            Console.WriteLine("x_>=_y");
        else
            Console.WriteLine("x_не_більше_або_дорівнює_y");
        if (x == y)
            Console.WriteLine("x_дорівнює_y");
        else
            Console.WriteLine("x_не_дорівнює_y");
        x++;
        Console.Write("x++:_x=");
        x.Display();
        Rational z = new Rational();
        z = x + y;
        Console.Write("Сума_дробів_z=");
        z.Display();
        Rational z2 = new Rational();
        z2 = x * y;
        Console.Write("Добуток_дробів_z2=");
        z2.Display();
        Console.ReadKey();
    }
}
}

```

### 6.8.4 Контрольні запитання

1. Що таке поліморфізм?
2. Які є види поліморфізму?
3. Перевантаження операторів.
4. Спеціальний поліморфізм.
5. Наведіть приклад поліморфізму.

## 6.9 Деякі особливості поліморфізму методів

Перевантаження методів.

Розглянемо приклад. Нехай необхідно написати програму роботи для виконання операції «додавання». Заздалегідь невідомо скільки буде операндів і їхні типи. Але відомо, що операндів буде не більше трьох. Ми хочемо, щоб ім'я методів було однаковим, тобто уніфікованим. Ми можемо представити сигнатури методів так:

```
Add(int , int )
Add(int , int , int )
Add(double , double )
Add(double , double , double )
```

Як ми вже казали, перевантаження методу полягає в тому, що методи мають різну сигнатуру, в якій збігається тільки назва методу. Тобто методи мають відрізнятися за:

- кількістю параметрів;
- типами параметрів;
- порядком параметрів.

Тут представлені чотири різні версії методу `Add`, тобто визначені чотири перевантаження даного методу. Перші дві версії методу відрізняються за кількістю параметрів. Третя та четверта версія збігається з першою та другою за кількістю параметрів, але відрізняється за їх типом. Наведемо приклад того, як може бути перевантажений метод:

```
public void SomeMethod()
{
    // тіло методу
}
public void SomeMethod(int a)
// від першого відрізняється наявністю параметру
```

```
{
// тіло методу
}
public void SomeMethod(string s) // від другого відрізняється типом параметру
{
// тіло методу
}
public int SomeMethod(int a, int b)
// від попередніх відрізняється кількістю параметрів
// (додатково змінений тип значення, що повертається)
{
// тіло методу
return 0;
}
```

Наведемо приклад того, як не може бути перевантажений метод:

```
public void SomeMethod(int a)
{
// тіло методу
public void SomeMethod(int b) // імені параметра недостатньо
// тіло методу
public int SomeMethod(int a) // типу значення, що повертається недостатньо
// тіло методу
return 0;
}
```

Тепер можна розв'язувати нашу задачу.

```
class Calculator
{
    public void Add(int a, int b)
    {
        int result = a + b;
        Console.WriteLine($"Result is {result}");
    }
    public void Add(int a, int b, int c)
    {
        int result = a + b + c;
        Console.WriteLine($"Result is {result}");
    }
    public int Add(int a, int b, int c, int d)
    {
        int result = a + b + c + d;
        Console.WriteLine($"Result is {result}");
        return result;
    }
    public void Add(double a, double b)
    {
        double result = a + b;
        Console.WriteLine($"Result is {result}");
    }
}
```

Перевизначення методів.

При спадкуванні нерідко виникає необхідність змінити в класі-спадкоємці функціонал методу, який був успадкований від базового класу. У цьому випадку клас-спадкоємець може перевизначати методи і властивості базового класу. Знов таки, як нам вже відомо, ті методи і властивості, які ми хочемо зробити доступними для перевизначення, в базовому класі позначається модифікатором `virtual`.

Такі методи і властивості називають віртуальними. А щоб перевизначити метод в класі-спадкоємці, цей метод визначається з модифікатором `override`. Перевизначення методу в класі-спадкоємці повинен мати той самий набір параметрів, що і віртуальний метод в базовому класі.

```
class Person
{
    public string Name { get; set; }
    public Person(string name)
    {
        Name = name;
    }
    public virtual void Display()
    {
        Console.WriteLine(Name);
    }
}
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company) : base(name)
    {
        Company = company;
    }
    public override void Display()
    {
        Console.WriteLine("Співробітник ( " + Name + "працюєв" + Company);
    }
}
```

Щоб зробити метод `Display` доступним для перевизначення, цей метод позначений з модифікатором `virtual`. Тому ми можемо перевизначати цей метод, але можемо і не перевизначати. Припустимо, нас влаштовує реалізація методу з базового класу. У цьому разі об'єкти `Employee` використовуватимуть реалізацію методу `Display` з класу `Person`:

```
class Person
{
    public string Name { get; set; }
    public Person(string name)
    {
        Name = name;
    }
    public virtual void Display()
    {
        Console.WriteLine(Name);
    }
}
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string name, string company) : base(name)
```

```
    {  
        Company = company;  
    }  
}
```

Основна програма:

```
static void Main(string[] args)  
{  
    Person p1 = new Person("Bill");  
    p1.Display(); // виклик методу Display з класу Person  
  
    Employee p2 = new Employee("Tom", "Microsoft");  
    p2.Display(); // виклик методу Display з класу Person  
  
    Console.ReadKey();  
}
```

Приховування методів. Іншим способом змінити функціональність методу, що успадкований від базового класу, є приховування (shadowing / hiding). Фактично приховування представляє визначення в класі-спадкоємці методу або властивості, які співпадають за сигнатурою методу або властивостями базового класу. Для приховування членів класу застосовується ключове слово new.

```
class Person  
{  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public Person(string firstName, string lastName)  
    {  
        FirstName = firstName;  
        LastName = lastName;  
    }  
  
    public void Display()  
    {  
        Console.WriteLine($"{FirstName}_{LastName}");  
    }  
}  
  
class Employee : Person  
{  
    public string Company { get; set; }  
    public Employee(string firstName, string lastName, string company)  
        : base(firstName, lastName)  
    {  
        Company = company;  
    }  
    public new void Display()  
    {  
        Console.WriteLine($"{FirstName}_{LastName}_работает_в_{Company}");  
    }  
}
```

У даному прикладі метод `Display` базового класу не є віртуальним, тому ми не можемо його перевизначити, але, припустимо, нас не влаштовує його реалізація для похідного класу, тому можемо скористатися приховуванням, щоб визначити потрібний нам функціонал.

Деякі особливості застосування при приведенні екземплярів спадкоємців до батьківського класу:

- при використанні віртуалізації для спадкоємця і спадкоємця, приведеного до батьківського класу, — викликається метод спадкоємця.
- При приховуванні (`new`) для спадкоємця, приведеного до батьківського класу — викликається метод батьківського.
- Без використання `new` для спадкоємця, приведеного до батьківського класу — викликається метод батьківського.

### 6.9.1 Контрольні запитання

1. Як повинні відрізнятися методи при перевантаженні?
2. Наведіть приклад методу, який не можна перевантажити.
3. Приховування методів та властивостей.

## 6.10 Параметричний поліморфізм

Розглянемо ще один вид поліморфізму. До речі, науковці описують більше 20 видів поліморфізму. Ось де починається творчість.

Параметричний поліморфізм — це можливість визначення таких узагальнених структур даних і методів, поведінка яких не залежить від типів конкретних їхніх значень. Класичним прикладом поліморфного типу є список елементів довільного типу. У випадку функцій саме алгоритмічна поведінка функції не залежить від типів аргументів, що їй передаються. Розглянемо такий приклад. Нехай необхідно описати клас для роботи із декартовою системою координат. Якщо ми будемо використовувати цілі значення, то це можна зробити так:

```
class Point2DInteger
{
    public T X { get; private set; }
    public T Y { get; private set; }
    public Point2DInteger (int x, int y)
```

```
{
    this.X = x;
    this.Y = y;
}
```

Якщо додатково необхідно використовувати ще окрему систему для дійсних значень, необхідно буде описати інший клас. Але, використовуючи параметричний поліморфізм, можна забезпечити цю узагальненість таким чином:

```
class Point2DAnyType<T>
{
    public T X { get; private set; }
    public T Y { get; private set; }
    public Point2D(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

Однак у цьому описі в якості значень координат можна задавати будь-які числові й нечислові значення. Для ліквідації цього недоліку будемо використовувати обмеження типів:

```
class Point2DValueType<T> where T : struct
{
    public T X { get; private set; }
    public T Y { get; private set; }
    public Point2D(T x, T y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

При такому описі класу компілятор буде відслідковувати використання лише значимого типу. Обмеження визначаються за допомогою ключового слова `where`. У таблиці наведено шість типів обмежень.

### 6.10.1 Контрольні запитання

1. Що таке параметричний поліморфізм?
2. Узагальненість при параметричному поліморфізмі.
3. Наведіть приклади параметричного поліморфізму.

## 6.11 Властивості статичних класів та статичних членів класів

Якщо весь клас є статичним:

1. Неможливо утворювати екземпляр класу з використанням ключового слова `new`.
2. Усі члени класу мають бути статичні.
3. Цей клас не підтримує спадкування.
4. Неможливо перевантажити методи.

Якщо клас не є статичним, але має статичні методи, то на ці методи поширюються такі обмеження:

1. Не дозволяється використовувати нестатичні члени цього ж класу зі статичних.
2. Спадкування й поліморфізм для статичних членів не підтримуються.

### 6.11.1 Контрольні запитання

1. Які властивості статичного класу?
2. Які властивості класу, що містить статичні методи?

## 6.12 Узагальнені типи та узагальнені методи

Припустимо, що ми визначаємо клас для представлення автомобіля:

```
class Car
{
    public string name {get; set; }
    public int number {get; set; }
}
```

Клас `Car` визначає дві властивості: `number` – унікальний номер та `name` – назву. Тут номер визначається як числове значення. Однак як номер може використовуватися комбінація літер та цифр. На момент написання класу ми не знаємо, у якому типі даних краще зберігати значення номера. Але ми знаємо, що тип `object` є універсальним типом, від якого успадковуються всі типи і, відповідно, ми можемо зберегти і рядки, і числа, і взагалі різні значення:

```
class Car
{
    public object name { get; set; }
    public object number { get; set; }
}
```

Перевіримо, як це працює у програмі:

```
static void Main()
{
    Car car1 = new Car { name = "BMW" };
    Car car2 = new Car { name = "Volvo" };
    car1.number = 1234;
    car2.number = "KA5678CZ";
    int number1 = (int)car1.number;
    string number2 = (string)car2.number;
    Console.WriteLine(number1);
    Console.WriteLine(number2);
}
```

У результаті отримаємо:

```
1234
KA5678CZ
```

Але таке рішення є не дуже гарним. Коли ми працюємо з класом `object`, відбувається упаковка (`boxing`) та розпакування (`unboxing`) значень.

```
car1.number = 1234; // Упаковка значення int в тип Object
car2.number = "KA5678CZ"; // упаковка значення string в тип Object
```

Щоб отримати значення, необхідно виконати розпакування:

```
int number1 = (int)car1.number; // Розпакування у тип int
string number2 = (string)car2.number; // Розпакування в тип string
```

При упаковці відбувається перетворення об'єкта значущого типу `object` у тип посилання, і нове значення зберігається у керованій купі. Розпакування перетворює значення типу посилання до значущого типу. Ми знаємо, що упаковка та розпакування ведуть до зниження ефективності програми. Для покращення роботи `C#` пропонує використовувати узагальнені типи. Вони дають змогу не вказувати конкретний тип, який буде використовуватися.

```
class Car<T>
{
public string name {get; set; }
public T number { get; set; }
}
```

Запис <T> говорить, що клас є узагальненим, а тип T, записаний у кутових дужках, використовуватиметься цим класом. Замість нього можна потім підставити будь-який тип.

```
static void Main()
{
Car<int> car1 = new Car<int> { name="BMW", number=1234};
Car<string> car2 = новий Car<string> { name="Volvo", number="KA4567KA"
};

int number1 = car1.number;
string number2 = car2.number;
Console.WriteLine(number1);
Console.WriteLine(number2);
}
```

У цьому прикладі вже упаковка-розпакування не потрібні. Ми при визначенні змінних car1 та car2 після назви класу в кутових дужках вказували той тип, який використовуватиметься замість універсального параметра T:

```
Car<int> car1 = new Car<int>
Car<string> car2 = новий Car<string>
```

Аналогічно можна використовувати кілька універсальних параметрів одночасно, які можуть бути різними типами. Тут клас AutoPark використовує два параметри, що вказують на тип.

```
class AutoPark<U, V>
{
public U car { get; set; }
public V driver {get; set; }
}
```

Подивимося, як це працюватиме.

```
class Car<T>
{
public string name {get; set; }
public T number { get; set; }
}
AutoPark<U, V>
{
public U car { get; set; }
public V driver {get; set; }
}
class Program
{
static void Main()
{
```

```
Car<int> car1 = new Car<int> { name="BMW", number=1234};
AutoPark <Car<int>, string> auto1 = новий AutoPark <Car<int>, string>
{
    car = car1,
    driver="Копылко"
};
}
```

Тут примірник auto1 класу AutoPark типізується типами Car<int> і string, оскільки як параметр U використовується клас Car<int>, а параметр V - тип string. При цьому, як можна помітити, клас, яким типізується AutoPark, є узагальненим. Даваймо підемо далі і ще більше узагальнимо наші класи за допомогою узагальнених методів. Розглянемо такий метод

```
public static void Swap<T> (ref T x, ref T y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

Він буде міняти місцями значення своїх параметрів. Але їм не зазначені типи. Точніше вказано, що тип буде узагальненим. Тому можна використовувати будь-які типи. У наступному прикладі в методі Main викликаємо метод Swap, типізуємо його типом int і передаємо значення.

```
class Program
{
    public static void Swap<T> (ref T x, ref T y)
    {
        T temp = x;
        x = y;
        y = temp;
    }
    private static void Main(string [] args)
    {
        int x = 7;
        int y = 25;
        Swap<int>(ref x, ref y);
        Console.WriteLine($"x={x}_y={y}");
        string s1 = "black";
        string s2 = "white";
        Swap<string>(ref s1, ref s2);
        Console.WriteLine($"s1={s1}_s2={s2}");
        Console.Read();
    }
}
```

Значення помінялись місцями. Статичні поля узагальнених класів. Цікава ситуація виникає при оголошенні статичних полів. Нагадаємо, що такі поля є спільними для всіх екземплярів зазначе-

ного класу. Таким чином, при типізації узагальненого класу для кожного конкретного класу буде створюватися свій набір статичних членів. Наприклад:

```
class Car<T>
{
    public static T number;
    public string name {get; set; }
}
```

Тепер типізуємо клас двома типами `int` та `string`:

```
static void Main()
{
    Car<int> car1 = new Car<int> {name = "BMW"};
    Car<int>.number = 1234;

    Car<string> car2 = новий Car<string> {name = "Volvo"};
    Car<string>.number = "KA4567KA";
    Console.WriteLine(Car<int>.number);
    Console.WriteLine(Car<string>.number);
}
```

Ми бачимо, що для класів для `Car<string>` і `Car<int>` буде створено свою статичну змінну `number`.

### 6.12.1 Контрольні запитання

1. Що таке упакування?
2. Що таке розпакування?
3. Що таке узагальнений тип?
4. Що таке узагальнений метод?
5. Наведіть приклади узагальнених класів та методів.

## 6.13 Перетворення типів

Перетворення типів

Ми розглядали деякі аспекти перетворення типів. Тепер розглянемо складніші, але необхідні. Тим більше, що такі перетворення дуже часто використовуються і часто програмісти роблять помилки. Про помилки читай у наступних розділах. На початку розглянемо перетворення екземплярів класів. Припустимо, у нас є така ієрархія класів (стандартний наш приклад):

```
class Person
{
    public string Name {get; set;}
    public Person(string name)
    {
        Name = name;
    }
    public void Display()
    {
        Console.WriteLine($"Person_{Name}");
    }
}

class Employee : Person
{
    public string Company {get; set;}
    public Employee(string name, string company) : base(name)
    {
        Company = company;
    }
}

class Client : Person
{
    public string Bank { get; set; }
    public Client(string name, string bank) : base(name)
    {
        Bank = bank;
    }
}
```

Ми пам'ятаємо, що всі класи явно успадковуються від типу `Object`. Далі виходить такий ланцюжок успадкування:

```
Object -> Person -> Employee
Object -> Person -> Client
```

Upcast або висхідне перетворення.

Екземпляри похідного типу можна використовувати як екземпляри базового типу. Наприклад, екземпляр `Employee` водночас є і екземпляром класу `Person`:

```
static void Main(string[] args)
{
    Employee employee = new Employee("Petr", "Roga");
    Person person = employee;

    Console.WriteLine(person.Name);
    Console.ReadKey();
}
```

У нашому прикладі змінній `person` має тип `Person` та їй надається посилання на екземпляр `Employee`. Оскільки `Employee` успадковується від класу `Person`, то автоматично виконується неявне висхідне перетворення до базового класу. У результаті змінні `employee` і `person` будуть вказувати один екземпляр у пам'яті, але змінній

person буде доступна тільки та частина, яка представляє функціонал типу Person. Таким чином, наприклад, поле Company не буде доступне із змінної person.

Downcast — нисхідне перетворення.

Розглянемо ще один приклад. У ньому person зберігає посилання на екземпляр типу Employee:

```
Employee employee = new Employee("Petr", "Roga");
Person person = employee;
```

І ми пам'ятаємо, що не всі атрибути екземпляра типу Employee доступні через змінну типу Person. Щоб знову можна було звертатися до недоступних атрибутів, потрібно зробити явне перетворення:

```
Employee employee2 = (Employee)person;
```

Але не завжди можна робити downcast. Наприклад, не можна цього робити, якщо перед цим не робили upcast:

```
Person person = new Employee("Petr", "Roga");
Client client = (Client)person;
```

Ми отримаємо помилку, оскільки змінна person зберігає посилання на екземпляр Employee. Цей екземпляр є також екземпляром типів object і Person, але не типу Client. Тому ми не можемо робити такого перетворення. Постає питання, як правильно виконувати перетворення? Можна, наприклад, використовувати ключове слово as. При його використанні буде зроблено спробу виконати перетворення. Якщо воно не вдасться, надається значення null:

```
Person person = new Person("Simon");
Employee emp=person as Employee;
if (emp == null)
{
    Console.WriteLine("Перетворення_пройшло_невдало");
}
else
{
    Console.WriteLine(emp.Company);
}
```

Ще один спосіб полягає у перевірці допустимості перетворення за допомогою ключового слова is:

```
Person person = new Person("Tom");
if (person is Employee)
{
    Employee emp = (Employee)person;
    Console.WriteLine(emp.Company);
}
else
{
}
```

```
Console.WriteLine("Перетворення_не_допустимо");  
}
```

Ми розглянули правила мови C# для виконання правильних перетворень. Однак навіть при правильному виконанні змін можуть бути в програмі помилки.

### 6.13.1 Контрольні запитання

1. Що таке висхідне перетворення?
2. Що таке нисхідне перетворення?
3. Ключове слово `is`.
4. Ключове слово `as`.
5. Наведіть приклади перетворень.

## 6.14 Приховування та перевизначення

Приховування означає, що метод у похідному класі приховує метод із тією самою сигнатурою (може відрізнятись), що у базовому класі. У C# для методів похідних класів, які приховують методи базових класів, використовується ключове слово `new`. Наприклад,

```
class A  
{  
    public void Hello()  
{  
    Console.WriteLine("I'm_the_base_class");  
}  
}  
  
class B: A  
{  
    public new void Hello()  
{  
    Console.WriteLine("I'm_the_derived_class");  
}  
}
```

У данному випадку для змінних базового класу `A` завжди буде викликатись метод `Hello()` базового класу, а змінних похідного класу `B` викликатиметься метод `Hello()` похідного класу.

```
static void Main()  
{  
    A a = new A();  
    a.Hello();  
}
```

```
B b = New B ();  
b.Hello();  
a = b;  
a.Hello();  
}
```

Буде надруковано

```
I'm the base class  
I'm the derived class  
I'm the base class
```

Зауважимо, що після upcast

```
a = b;
```

викликався метод базового класу.

Перевизначення застосовується до віртуальних методів. У похідному класі віртуальна функція перевизначається. Відмінність від приховування полягає в тому, що якщо змінна базового класу вказує на екземпляр похідного класу, то в цьому випадку буде викликано перевизначений метод похідного класу:

```
class A  
{  
    public virtual void Hello()  
    {  
        Console.WriteLine( "I'm_the_base_class" );  
    }  
}  
  
class B: A  
{  
    public override void Hello()  
    {  
        Console.WriteLine( "I'm_the_derived_class" );  
    }  
}  
  
class Program  
{  
  
    static void Main()  
    {  
        A a = new A();  
        a.Hello();  
  
        B b = New B ();  
        b.Hello();  
  
        a = b;  
        a.Hello();  
  
    }  
}
```

У результаті буде надруковано

```
I'm the base class
I'm the derived class
I'm the derived class
```

Таким чином, перевизначення віртуального методу базового класу — це створення методу в класі спадкоємця, який має таку саму сигнатуру. У цьому виклик методу з допомогою змінної базового класу призводить до виклику методу класу спадкоємця. Це називається пізні зв'язування. Приховування методу - це визначення у похідному класі методу з таким же ім'ям і сигнатурою, що, можливо, відрізняється. При цьому який із двох методів буде викликаний, відомо під час компіляції.

### 6.14.1 Контрольні запитання

1. Що таке приховування?
2. Що таке перевизначення?
3. Наведіть приклад приховування та перевизначення.

## 6.15 Абстрактні класи, методи, властивості

Абстрактні класи.

Нехай в компанії працюють співробітники та є клієнти. Необхідно друкувати їхні прізвище, ім'я та для співробітників назву відділу. А для клієнтів назву їхньої компанії. Знов розглянемо клас Person. Програма буде працювати з об'єктами співробітники й клієнти. Тому утворювати екземпляри класу Person немає необхідності. У такому разі краще працювати з абстрактними класами.

Такий клас має наступні особливості: - не можна створювати екземпляри (об'єкти) абстрактного класу; - абстрактний клас може містити як абстрактні методи/властивості, так і звичайні; - у класі спадкоємці, якщо він не абстрактний, повинні бути реалізовані всі абстрактні методи й властивості, що оголошені в базовому класі.

Абстрактний клас оголошується із ключовим словом `abstract`:

```
abstract class імя[ 'класу_']
{
    //тіло
}
```

Абстрактний клас не може бути запечатаним `sealed` або статичним `static`.

Абстрактний метод – це метод, який не має своєї реалізації в базовому класі, і він **ПОВИНЕН** бути реалізований у класі-спадкоємці (за умови, що клас-спадкоємець не абстрактний). Абстрактний метод може бути оголошений тільки в абстрактному класі.

Оголошення абстрактного методу відбувається за допомогою ключового слова `abstract`, і при цьому фігурні дужки упускаються, крапка з комою ставиться після заголовка методу:

```
модифікатор  
[ доступу] abstract тип[] імя[ ' методу] аргументи ([]);
```

Модифікаторами доступу до абстрактних методів можуть бути: `public`, `protected`, `internal`, `protected internal`.

Реалізація абстрактного методу в класі-спадкоємці відбувається так само, як і перевантаження віртуального методу – за допомогою ключового слова `override`:

```
модифікатор  
[ доступу] override тип[] імя[ ' методу] аргументи ([])  
{  
    // реалізація методу  
}
```

Розглянемо приклад:

```
abstract class Person  
{  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
  
    public Person(string NameLast, string NameFirst)  
    {  
        FirstName = NameLast;  
        LastName = NameFirst;  
    }  
  
    public abstract void Display();  
}  
  
class Employee : Person  
{  
    public string Department { get; set; }  
    public string Salary { get; set; }  
    public string Position { get; set; }  
    public Employee(string NameFirst, string NameLast, string  
        Dept, string  
        Sal, string Pos) : base(NameFirst, NameLast)  
    {  
        Department = Dept;  
        Salary = Sal;  
        Position = Pos;  
    }  
}
```

```
public override void Display()
{
    Console.WriteLine(FirstName + "_" + LastName + "
        Department:" + Department);
}
class Client : Person
{
    public string Company { get; set; }

    public Client(string NameFirst, string NameLast, string
        Comp)
        : base(NameFirst, NameLast)
    {
        Company = Comp;
    }

    public override void Display()
    {
        Console.WriteLine(FirstName + "_" + LastName + "_" +
            Company);
    }
}
```

Розглянемо ще один приклад використання абстрактних класів та методів.

Розглянемо задачу рисування та стирання фігур на екрані. Нам необхідно якось виділити ті методи базового класу, які мають бути обов'язково перевизначені у всіх класах спадкоємців (у кожному дочірньому класі по-своєму). Це необхідно тому, що деякий загальний елемент класу інтерфейсу (група методів) не може бути уніфіковано описаний в базовому класі і вимагає специфіки для всіх класів спадкоємців. Кожна фігура має рисуватись Draw() за власним алгоритмом. Але при цьому стерати фігури Erase() можна уніфіковано. Для цього визначимо абстрактний клас AFigure, в якому уніфіковані методи опишемо, а специфічні лише оголосимо як абстрактні:

```
abstract class AFigure
{
    public abstract void Draw();
    public void Erase()
    {
        Console.WriteLine("Erase()_method_from_abstract_class_
            AFigure");
    }
}
```

Тепер при створенні класу для роботи з колом ми можемо використовувати абстрактний клас AFigure:

```
class Circle : AFigure
{
    public override void Draw()
```

```
    {  
        Console.WriteLine("Draw(_) _method_from_class_Circles");  
    }  
}
```

Як бачимо, у цьому класі ми навели реалізацію абстрактного методу Draw(). Тепер напишемо саму програму:

```
class Program  
{  
    static void Main(String[] args)  
    {  
        Circle a = new Circle();  
        a.Draw();  
        a.Erase();  
    }  
}
```

Програма виведе на екран:

```
Draw( ) method from class Circles  
Erase() method from abstract class AFigure
```

Будь-який клас-спадкоємець абстрактного класу або зобов'язаний надати реалізацію всіх абстрактних методів свого базового суперкласу, або сам має бути оголошений абстрактним.

Використання абстрактних класів дозволяє зняти деяку зовнішню невизначеність коду, що виникає при заміщенні методів базового класу в загальному випадку успадкування та використання механізму динамічного зв'язування (поліморфізму).

Абстрактні властивості

Так само як і методами, можна користуватись і абстрактними властивостями. Створення абстрактних властивостей майже не відрізняється від абстрактних методів:

```
модифікатор  
[ доступу] abstract тип[] імя[' властивості] { get; set; }
```

Для реалізації в класі-спадкоємці необхідно записати:

```
модифікатор  
[ доступу] override тип[] імя[' властивості]  
{  
    get { тілоаксесора get }  
    set { тілоаксесора set }  
}
```

Розглянемо приклад використання абстрактних класів для створення сімейства класів. Нехай необхідно уніфіковано отримувати якусь інформацію про різних тварин: кішок, птахів, риб і т.д.

Наприклад, так:

```
foreach (Animals animal in animals)
    animal.GetInfo();
```

У даному фрагменті ми у сімействі `animals` перебираємо всіх тварин `animal` і виводимо інформацію про екран.

Розглянемо клас `Animals` (тварини). Цей клас описує абстрактну сутність Тварина, екземпляр якої створювати не можна і немає сенсу, тому що ми працюватимемо не з тваринами, а з деякими видами тварин: кішками, птахами, рибами тощо.

У такому разі, має сенс описати абстрактний клас `Animals` і від нього успадковувати вже необхідні нам класи для опису кішок, птахів, риб.

```
abstract class Animals
{
    public string Name { get; set; }
    public string Type { get; protected set; }
    public abstract void GetInfo(); // оголошення абстрактного
        методу
}
class Bird : Animals
{
    public Bird(string name)
    {
        Name = name;
        Type = "Птах";
    }
    public override void GetInfo() // реалізація абстрактного методу
    {
        Console.WriteLine("Тип:␣" + Type + "\n" + "Им'я:␣" +
            Name + "\n");
    }
}
class Cat : Animals
{
    public Cat(string name)
    {
        Name = name;
        Type = "Кішка";
    }
    public override void GetInfo() // реалізація абстрактного методу
    {
        Console.WriteLine("Тип:␣" + Type + "\n" + "Им'я:␣" + Name + "
        \n");
    }
}
class Fish : Animals
{
    public Fish(string name)
    {
        Name = name;
        Type = "Риба";
    }
    public override void GetInfo() // реалізація абстрактного методу
    {
```

```
        Console.WriteLine("Тип:_" + Type + "\n" + "Им'я:_" + Name + "\n" +
            ");
    }
}
class Program
{
    static void Main(string[] args)
    {
        List<Animals> animals = new List<Animals>();
        animals.Add(new Bird("Піпат"));
        animals.Add(new Cat("Мурка"));
        animals.Add(new Fish("Мовчун"));

        foreach (Animals animal in animals)
            animal.GetInfo();

        Console.ReadKey();
    }
}
```

У даному прикладі ми працюємо з одним списком тварин, і, викликаючи один метод `GetInfo()`, ми отримуємо інформацію про відповідну тварину.

### 6.15.1 Контрольні запитання

1. Що таке абстрактний клас?
2. Особливості абстрактного класу.
3. Чи можна створити екземпляр абстрактного класу?
4. Модифікатори доступу в абстрактному класі.
5. Реалізація абстрактних методів та властивостей.

## 6.16 Інтерфейси

Нехай необхідно написати програму, яка буде рисувати різні фігури (квадрат - `Square`, еліпс - `Ellipse`, коло - `Diamond`) на екрані, обчислювати їхню площу й периметр. Можна для рисування фігур з кожного класів `Square`, `Ellipse`, `Diamond` написати такі фрагменти:

```
for (Square s: squares)
{
    s.draw();
}
for (Ellipse e: ellipses)
{
    e.draw();
}
```

```
for (Diamond d: diamonds)
{
    d.draw();
}
```

У цих фрагментах squares, ellipses, diamonds списки екземплярів відповідних класів: Square, Ellipse, Diamond. Якщо нам потім потрібно буде додати ще фігури, то буде необхідно дописувати програму. У такій ситуації краще буде оголосити щось спільне для фігур з однаковим за функціоналом методом draw().

Створимо список figures екземплярів різних класів із загальним методом draw(). Однак кожна фігура має свій особистий метод рисування, тому цей метод треба реалізовувати в кожному класі індивідуально, а в загальній для всіх класів частині вказати на необхідність такого опису. Цю задачу й вирішують інтерфейси:

```
for (Ifigure f: figures)
{
    f.draw();
}
```

У даному фрагменті Ifigure є інтерфейсом (буква I на початку імені – це просто загальноприйнята угода в іменуванні інтерфейсів). Інтерфейси потрібні, щоб об'єднати кілька класів чимось спільним. У нашому прикладі цим спільним є наявність методів друку, обчислення периметра й площі. Синтаксис інтерфейсів такий:

```
interface імя' інтерфейсу
{
    // тіло інтерфейсу
}
```

Інтерфейси оголошують методи, властивості й події. Вони можуть бути або public, або internal. Однак вкладені інтерфейси можуть мати будь-який рівень доступу. Інтерфейси не можуть містити поля або типи через те, що не визначають реалізацію. Напишемо інтерфейс для нашого прикладу.

```
interface Ifigure // оголошення інтерфейсу
{
    void Draw();
    void Perimeter();
    void Area();
}
```

У тілі інтерфейсу ми записали прототип методів для рисування фігур, обчислення площі й периметра, але природно ми не стали його реалізувати. Цією реалізацією займеться кожний клас, який буде використовувати цей інтерфейс. Наприклад:

```
class Rectangle : Ifigure //спадкування інтерфейсу
{
    double a=4.8;
    double b=3.7;
    public void Draw() //реалізація інтерфейсу
    {
        Console.WriteLine("Rectangle , a={0}, b={1}", a, b);
    }
    public void Perimeter() //реалізація інтерфейсу
    {
        Console.WriteLine("L={0}", 2*a+2*b );
    }

    public void Area() //реалізація інтерфейсу
    {
        Console.WriteLine("S={0}", a * b / 2);
    }
}
class Circle : Ifigure //реалізація інтерфейсу
{
    double r = 2.7;
    public void Draw() //реалізація інтерфейсу
    {
        Console.WriteLine("Circle , r={0}", r);
    }
    public void Perimeter() //реалізація інтерфейсу
    {
        Console.WriteLine("L={0}", 2*3.14*r);
    }

    public void Area() //реалізація інтерфейсу
    {
        Console.WriteLine("S={0}", 3.14*r*r);
    }
}
public void Area()
{
    Console.WriteLine("a*b");
}
}
```

Тепер нам залишилося тільки створити список фігур і перебрати їх усіх для розв'язання задачі. Приведемо реалізацію всієї програми.

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

namespace Interface
{
    interface Ifigure // оголошення інтерфейсу
    {
        void Draw();
        void Perimeter();
        void Area();
    }

    class Rectangle : Ifigure //спадкування інтерфейсу
```

```

{
    double a = 4.8;
    double b = 3.7;
    public void Draw() // неявна реалізація інтерфейсу
    {
        Console.WriteLine("Rectangle , a={0}, b={1}", a, b);
    }
    public void Perimeter() // неявна реалізація інтерфейсу
    {
        Console.WriteLine("L={0}", 2 * a + 2 * b);
    }

    public void Area() // неявна реалізація інтерфейсу
    {
        Console.WriteLine("S={0}", a * b / 2);
    }
}
class Circle : Ifigure // спадкування інтерфейсу
{
    double r = 2.7;
    public void Draw() // неявна реалізація інтерфейсу
    {
        Console.WriteLine("Circle , r={0}", r);
    }
    public void Perimeter() // неявна реалізація інтерфейсу
    {
        Console.WriteLine("L={0}", 2 * 3.14 * r);
    }

    public void Area() // неявна реалізація інтерфейсу
    {
        Console.WriteLine("S={0}", 3.14 * r * r);
    }
}
class Program
{
    static void Main(string[] args)
    {
        List<Ifigure> figures = new List<Ifigure>();
        figures.Add(new Rectangle());
        figures.Add(new Circle());
        foreach (Ifigure f in figures)
        {
            f.Draw();
            f.Perimeter();
            f.Area();
        }
        Console.ReadKey();
    }
}
}

```

Якщо ми прагнемо вказати, що реалізуємо якийсь метод саме з конкретного інтерфейсу (наприклад, при збігу імен), то слід указати ім'я цього інтерфейсу: `void Ifigure.Draw()` Це так звана явна реалізація інтерфейсу. У такому випадку рівень доступу не вказується і ця явна реалізація завжди є закритою (точніше `private virtual sealed`).

Щоб отримати доступ до реалізації необхідно зробити так:

```
Rectangle a = new Rectangle();  
((Ifigure)a).Draw();
```

Тут ми створили екземпляр а класу Rectangle і привели його тип до типу Ifigure.

У випадку неявної реалізації для виклику методу рисування для екземпляра а досить просто написати:

```
a.Draw();  
як і показано в нашому прикладі.
```

Але ми можемо звертатись до методу інтерфейсу, створивши екземпляр типу цього інтерфейсу, і записати в нього таке посилання :

```
Rectangle a = new Rectangle();  
Ifigure PointerDraw = (Ifigure)a;
```

За допомогою посилання PointerDraw можна звертатися до методів інтерфейсу Ifigure, що реалізовані в класі Rectangle :

```
PointerDraw.Draw();  
PointerDraw.Perimeter();
```

Неявна реалізація інтерфейсу може бути віртуальною (virtual), що дозволяє переписувати її в класах-нащадках. Явна імплементація завжди статична. Вона не може бути переписана (override) або перекрита (new) у класах-нащадках.

Однак явна реалізація може бути переписана в класі-нащадку, якщо той же самий інтерфейс реалізувати ще раз. За такої умови використовується інший механізм, ніж override, але результат аналогічний.

```
interface Ifigure  
{  
    void Draw();  
}  
class Rectangle : Ifigure  
{  
    void Ifigure.Draw() // Явна реалізація  
    {...}  
}  
}  
class Square : Rectangle, Ifigure  
{  
    void Ifigure.Draw() // Аналог override у вигляді ще однієї явної реалізації  
    {...}  
}  
}
```

Таким чином, похідні класи можуть спадкувати методи й інтерфейси базового класу та змінювати їх.

Інтерфейси є типом посиланням і їх можна реалізовувати в класах і структурах. Однак у випадку структур їх необхідно використовувати із обережністю. Якщо ми маємо посилання на структуру, що має тип інтерфейсу, то ми маємо посилання на деяку упаковку (boxing), яка зберігає копію структури.

### 6.16.1 Контрольні запитання

1. Що таке інтерфейс?
2. Синтаксис інтерфейсів.
3. Необхідність використання інтерфейсів.
4. Що можна оголосити в інтерфейсі?
5. Реалізація членів інтерфейсу.

## 6.17 Комбінований інтерфейс

Інтерфейси описують спільну функціональність класів. C# надає можливість комбінувати інтерфейси, щоб створювати нові інтерфейси на базі існуючих. Наприклад нехай у нас є інтерфейс для рисування фігур

```
interface IDraw    // оголошення інтерфейсу
{
    void Draw();
}
```

і інтерфейс для обчислення геометричних властивостей фігур

```
interface Igeometric // оголошення інтерфейсу
{
    void Perimeter();
    void Area();
}
```

Можна побудувати комбінований інтерфейс вигляду:

```
interface Ifigure : IDraw, Igeometric // оголошення комбінованого
    інтерфейсу
{
}
```

Тепер при написанні класу `Rectangle` ми можемо вказувати інтерфейс `Ifigure` і використовуємо явну реалізацію

```
class Rectangle : Ifigure //спадкування комбінованого інтерфейсу
{
    double a = 4.8;
    double b = 3.7;
    void Idraw.Draw() //явна реалізація інтерфейсу
    {
        Console.WriteLine("Rectangle , a={0}, b={1}", a, b);
    }
    void Igeometric.Perimeter() // явна реалізація інтерфейсу
    {
        Console.WriteLine("L={0}", 2 * a + 2 * b);
    }

    void Igeometric.Area() // явна реалізація інтерфейсу
    {
        Console.WriteLine("S={0}", a * b / 2);
    }
}
```

Покажемо, як користуватись комбінованим інтерфейсом. Для початку створюємо екземпляр класу `Rectangle`.

```
Rectangle Figure = new Rectangle();
```

Потім оголошуємо змінну для посилання на інтерфейс `Ifigure`.

```
Ifigure PointerIfigure = Figure as Ifigure;
```

І за допомогою цього посилання `PointerIfigure` тепер можна звертатися до методів інтерфейсу `Ifigure`:

```
PointerIfigure.Draw();
PointerIfigure.Area();
PointerIfigure.Perimeter();
```

Або остаточно можна записати таку програму:

```
using System;

namespace CombiInterface
{
    interface Idraw // оголошення інтерфейсу
    {
        void Display();
        void Draw();
    }

    interface Igeometric // оголошення інтерфейсу
    {
        void Display();
        void Perimeter();
        void Area();
    }

    interface Ifigure : Idraw, Igeometric // оголошення комбінованого
        інтерфейсу
    {
    }
}
```

```
{
}

class Rectangle : Ifigure //спадкування комбінованого інтерфейсу
{
    string Name="Rectangle";
    double a = 4.8;
    double b = 3.7;
    void Idraw.Draw() //явна реалізація інтерфейсу
    {
        Console.WriteLine("Rectangle , a={0}, b={1}", a, b);
    }
    void Idraw.Display() //явна реалізація інтерфейсу
    {
        Console.WriteLine(Name);
    }
    void Igeometric.Perimeter() // явна реалізація інтерфейсу
    {
        Console.WriteLine("L={0}", 2 * a + 2 * b);
    }
    void Igeometric.Display() //явна реалізація інтерфейсу
    {
        Console.WriteLine(Name);
    }
    void Igeometric.Area() // явна реалізація інтерфейсу
    {
        Console.WriteLine("S={0}", a * b / 2);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Rectangle Figure = new Rectangle();
        Ifigure PointerIfigure = Figure as Ifigure;
        if (PointerIfigure != null)
        {
            PointerIfigure.Draw();
            PointerIfigure.Area();
            PointerIfigure.Perimeter();
        }
    }
}
```

В інтерфейсах `Igeometric` і `Idraw` описано метод `Display()` для друку на екрані назви геометричної фігури. При використанні комбінованого інтерфейсу може виникнути помилка через неоднозначний виклик методу `Display()` з двох різних інтерфейсів `Igeometric` і `Idraw`, якщо написати так:

```
PointerIfigure.Display();
```

Тому необхідно записувати так:

```
((Idraw)PointerIfigure).Display();
```

Тобто ми привели посилання на комбінований інтерфейс до типу того інтерфейсу, для якого необхідно викликати метод `Display()`.

### 6.17.1 Використання властивостей у інтерфейсах

Покажемо, як використовувати властивості у інтерфейсах. Для початку необхідно їх описати:

```
interface Ifigure // оголошення комбінованого інтерфейсу
{
    string IName
    {
        get;
        set;
    }
    double Ia
    {
        get;
        set;
    }
    double Ib
    {
        get;
        set;
    }
}
```

Тут ми описали властивості `IName`, `Ia`, `Ib` для роботи з полями

```
string Name = "Rectangle";
double a = 4.8;
double b = 3.7;
```

класу

```
class Rectangle : Ifigure //спадкування комбінованого інтерфейсу
{
    string Name = "Rectangle";
    double a = 4.8;
    double b = 3.7;...
}
```

Тепер необхідно в класі реалізувати ці властивості:

```
class Rectangle : Ifigure //спадкування комбінованого інтерфейсу
{
    string Name = "Rectangle";
    double a = 4.8;
    double b = 3.7;
    string Ifigure.IName
    {
        get
        {
```

```
        return Name;
    }
    set
    {
        Name = value;
    }
}
double Ifigure.Ia
{
    get
    {
        return a;
    }
    set
    {
        a = value;
    }
}
double Ifigure.Ib
{
    get
    {
        return b;
    }
    set
    {
        b = value;
    }
}
}
```

При реалізації інтерфейсу ми не вказували модифікатор доступу, а ім'я властивості мало префіксом ім'я інтерфейсу. Остаточно програма виглядає так:

```
using System;

namespace AccsesorInterface
{
    interface Ifigure // оголошення комбінованого інтерфейсу
    {
        string IName
        {
            get;
            set;
        }
        double Ia
        {
            get;
            set;
        }
        double Ib
        {
            get;
            set;
        }
    }
}
```

```
}

class Rectangle : Ifigure //спадкування комбінованого інтерфейсу
{
    string Name = "Rectangle";
    double a = 4.8;
    double b = 3.7;
    string Ifigure.IName
    {
        get
        {
            return Name;
        }
        set
        {
            Name = value;
        }
    }
    double Ifigure.Ia
    {
        get
        {
            return a;
        }
        set
        {
            a = value;
        }
    }
    double Ifigure.Ib
    {
        get
        {
            return b;
        }
        set
        {
            b = value;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle Figure = new Rectangle();
        Ifigure PointerIfigure = Figure as Ifigure;
        PointerIfigure.IName = "New_rectangle";
        PointerIfigure.Ia = 10;
        PointerIfigure.Ib = 20;

        Console.WriteLine(PointerIfigure.IName + " _a=" +
            PointerIfigure.Ia + " _b=" + PointerIfigure.Ib);
    }
}

}
```

## 6.17.2 Використання індексаторів у інтерфейсах

Індексатор також може бути включений в опис інтерфейсу. Повернемося до нашого прикладу з ліфтом. Тепер будемо використовувати індексатори. Нехай інтерфейс включає в себе кількість поверхів (властивість) `Size` і індексатор `this[uint index]`. Опишемо спочатку інтерфейс:

```
interface IfloorNumber
{
    string this[uint index]
    {
        get;
        set;
    }
    uint Size
    {
        get;
    }
}
```

Тепер індексатор необхідно реалізувати у конструкторі класу `ElevatorWithIndexer`:

```
public ElevatorWithIndexer(uint Count)
{
    // опис конструктора
    floor = new string[Count];
    AmmountOfFloor = Count;
}
uint IfloorNumber.Size
{
    // отримання розміру масиву
    get
    {
        return AmmountOfFloor;
    }
}
string IfloorNumber.this[uint index]
{
    // реалізація індексатора
    get
    {
        if (index > -1 && index < floor.Length)
            return floor[index]; // код для отримання
            елементу
        else return "Wrong_number_of_floor ";
    }
    set
    {
        floor[index] = value; // код для задання елементу
    }
}
}
```

При реалізації індексатора ми для його імені використовували префікс `IfloorNumber`. Використання індексаторів здійснюється та-

ким же чином як і раніше, але додатково необхідно було задати посилання fN на інтерфейс IfloorNumber

```
IfloorNumber fN = MyElevator as IfloorNumber;
```

і звертатись до поверхів через це посилання:

```
fN[0] = "first";  
      fN[1] = "second";  
      fN[2] = "third";
```

Програма набуде такого вигляду:

```
using System;  
  
namespace IndexerInterface  
{  
    interface IfloorNumber  
    {  
        // інтерфейс з індексатором  
        string this[uint index]  
        {  
            get;  
            set;  
        }  
        uint Size  
        {  
            get;  
        }  
    }  
    class ElevatorWithIndexer : IfloorNumber  
    {  
        private string[] floor;  
        private uint AmmountOfFloor;  
  
        public ElevatorWithIndexer(uint Count)  
        {  
            // опис конструктора  
            floor = new string[Count];  
            AmmountOfFloor = Count;  
        }  
        uint IfloorNumber.Size  
        {  
            // отримання розміру масиву  
            get  
            {  
                return AmmountOfFloor;  
            }  
        }  
        string IfloorNumber.this[uint index]  
        {  
            // реалізація індексатора  
            get  
            {  
                if (index > -1 && index < floor.Length)  
                    return floor[index]; // код для отримання  
                    елементу  
                else return "Wrong_number_of_floor";  
            }  
            set  
            {  

```

```
        floor[index] = value;    // код для задання елементу
    }
}

class Program
{
    static void Main(string[] args)
    {
        ElevatorWithIndexer MyElevator = new ElevatorWithIndexer
            (3);
        IfloorNumber fN = MyElevator as IfloorNumber;
        if (fN != null)
        {
            fN[0] = "first";
            fN[1] = "second";
            fN[2] = "third";
        }
        for (uint i = 0; i < fN.Size; i++)
        {
            Console.WriteLine("Number_of_floor_is:{0}", fN[i]);
        }
        Console.ReadKey();
    }
}
```

Зауваження. Якщо є можливість виділити не лише спільну специфікацію, але й реалізувати спільну функціональність, то необхідно використовувати абстрактний клас.

### 6.17.3 Контрольні запитання

1. Що таке інтерфейс?
2. Синтаксис інтерфейсів.
3. Необхідність використання інтерфейсів.
4. Що можна оголосити у інтерфейсі?
5. Реалізація членів інтерфейсу.

## 6.18 Інтерфейси. Більш детально та глибоко.

Головне завдання інтерфейсу - визначити поведінку, яка згодом буде реалізована в якомусь конкретному класі. Клас має можливість

підтримувати необмежену кількість інтерфейсів, а, отже, і мати безліч поведінок. В C# за допомогою інтерфейсу визначають

- Події
- Властивості (не автоматичні).
- Методи.
- Константи.
- Індексатори.
- Статичний конструктор.
- Статичні поля
- Події.

Починаючи з C# 11, інтерфейс може оголошувати `static abstract` елементи та `static virtual` для всіх типів елементів, крім полів. Інтерфейси не можуть містити стан екземпляра. Хоча статичні поля наразі дозволені, поля екземпляра не дозволені в інтерфейсах. Автовластивості екземпляра не підтримуються в інтерфейсах, оскільки вони явно оголошують приховане поле.

Зазвичай, інтерфейсу встановлюють `internal` рівень доступу. Щоб зробити його загальнодоступним, застосовується модифікатор `public`. Якщо у властивостях та методах інтерфейсу відсутні модифікатори доступу, за замовчуванням вони є `public`. Екземпляри інтерфейсів не можна створити за допомогою конструктора. Інтерфейси C# реалізуються у класах та структурах.

```
interface IMovable
{
    void Move();
}
class Person : IMovable
{
    public void Move()
    {
        Console.WriteLine("Людина_побігла");
    }
}
struct Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Автомобіль_поїхав");
    }
}
```

```
class Program
{
    static void Action(IMovable movable)
    {
        movable.Move();
    }
    static void Main(string[] args)
    {
        Person person = new Person();
        Car car = new Car();
        Action(person);
        Action(car);
        Console.Read();
    }
}
```

У програмі є метод `Action()`, він приймає екземпляр інтерфейсу `IMovable`, але зверніть увагу, що ми не утворюємо екземпляр ТИПУ `IMovable`. При написанні коду нічого про екземпляр невідомо, але ми знаємо, що метод `Move` буде обов'язково реалізовано. Тому ми можемо викликати цей метод.

Що ми знали раніше про інтерфейси (у версіях до 8-ї)? • вони містять лише методи, властивості, події, індексатори, • вони не містять полів, • за замовчуванням вони `public` і `abstract`. Але починаючи з версії `C# 8.0`, методи та властивості можуть реалізовуватися в стандартних інтерфейсах. Припустимо, що інтерфейс реалізується класами. Якщо ми додаємо новий член в інтерфейс, потрібно оновити всі класи для його підтримки або створити розширений інтерфейс з новим методом, який буде успадкований від попереднього:

```
interface ICar
{
    void GetSpeed();
    void GetMileage();
}
interface INewCar : ICar
{
    void SendCommand();
}
```

Але чим більше методів і функцій у розширеному інтерфейсі, тим швидше він стає некерованим. Цю проблему вирішують інтерфейси, які реалізують деякі методи за замовчуванням. Тобто приходимо до абстрактних класів. Інтерфейси із реалізацією за замовчуванням корисні при додаванні нових членів, не порушуючи існуючу реалізацію:

```
interface ICar
{
    void GetSpeed();
```

```
void GetMileage();  
public void SomeCommand()  
{Тут реалізовано тіло методу  
  
}  
}
```

Після введення в C# інтерфейсів із реалізацією за замовчуванням, немає потреби у застосуванні нових інтерфейсів і змін в існуючій реалізації, але є певні обмеження у використанні. Зрозуміло, що класи можуть перевизначити реалізацію за замовчуванням. Знов отримуємо абстрактний клас. Наприклад, якщо клас `Garage` реалізує метод `SendCommand`, буде викликана саме ця реалізація:

```
Public class Garage : ICar  
{  
    public void GetMileage()  
    {  
        Console.WriteLine("10_KM_Per_Liter");  
    }  
  
    public void GetSpeed()  
    {  
        Console.WriteLine("200_KMPH");  
    }  
  
    public void SendCommand()  
    {  
        Console.WriteLine("Command_Sent_via_Morris_Garage_Class");  
    }  
}
```

Інтерфейси в C# можуть:

- реалізовувати методи за умовчанням та мати `private` члени:

```
public interface ICar  
{  
    private void Initialize()  
    {  
        //Initialize the Command Center  
    }  
    void GetSpeed();  
    void GetMileage();  
    public void SendCommand()  
    {  
        Initialize();  
        Console.WriteLine("Command_Sent_via_Interface");  
    }  
}
```

- реалізовувати в інтерфейсах C# статичні поля та члени:

```
public interface ICar  
{  
    private static string commandName = "LOCK_CAR";  
    private void Initialize(string cName)
```

```
{
    commandName = cName;
    //Initialize the Command Center
}
void GetSpeed();
void GetMileage();
public void SendCommand(string cName)
{
    Initialize(cName);
    Console.WriteLine("Command_Sent_via_Interface");
}
}
```

• мати protected члени, які доступні у похідному інтерфейсі, але не у похідному класі:

```
public interface ICar
{
    public void SendCommand()
    {
        Console.WriteLine("Command_Sent_via_Interface");
    }
    protected void SendCriticalCommand()
    {
        Console.WriteLine("Critical_Command_Sent_via_Interface");
    }
}

public interface IAnotherCar : ICar
{
    public void Send(bool bCritical)
    {
        if (bCritical)
            this.SendCriticalCommand();
        else
            { }
    }
}
```

• мати virtual члени без можливості перевизначення методу класом:

```
public interface ICar
{
    public virtual void SendCommand()
    {
        Console.WriteLine("Command_Sent_via_Interface");
    }
}

public interface IAnotherCar : ICar
{
    void ICar.SendCommand()
    {
        Console.WriteLine("Command_Sent_via_another_Interface");
    }
}

class Garage : ICar, IAnotherCar
```

```
{
}

class Program
{
    static void Main()
    {
        ICar mg = new Garage();
        mg.SendCommand(); //Calls the virtual implementation.

        IAnotherCar mgOverridden = new Garage();
        mgOverridden.SendCommand(); //Calls the overridden implementation.
    }
}
```

Ми з вами бачимо, що розробники C# намагаються покращити мову програмування, але вносять певні обмеження на використання нових можливостей. До того ж, ці обмеження пов'язані не з теоретичними аспектами теорії програмування, а з неможливістю розробниками коректно їх впроваджувати. Множинна реалізація інтерфейсів У C# немає підтримки множинного спадкування - спадкувати клас можна лише від одного класу. Раніше програмісти казали, що завдяки тому, що у C# клас може реалізувати відразу кілька інтерфейсів, обмеження з неможливістю множинного спадкування можна обійти. Крім того, наводили це як основну відмінність від абстрактного класу. Але це не так. Тому що СПАДКУВАННЯ – це перш за все повторне використання. А інтерфейси необхідно було РЕАЛІЗОВУВАТИ. Зараз ситуація трохи змінилась.

Усі інтерфейси при множинній реалізації/спадкуванні в новому класі вказуються через кому. Наприклад. Тут реалізовано інтерфейси: IAccount - визначає властивість CurrentSum, яку необхідно реалізувати в класі та методи Put - надходження суми на рахунок, Withdraw - вилучення суми; IClient - визначає властивість Name, Depos – обчислення значення для зарахування. Клас Client реалізує обидва інтерфейси та перевантажує метод Depos.

```
interface IAccount
{
    int CurrentSum { get; }
    void Put(int sum);
    void Withdraw(int sum);
    public int Depos(int x, int y)
    {
        return x * y;
    }
}
interface IClient
{
    string Name { get; set; }
}
```

```
class Client : IAccount, IClient
{
    int isum;
    public string Name { get; set; }

    public int Depos(int a, int b)
    {
        return 2 * a + 2 * b;
    }
    public Client(string name, int sum)
    {
        Name = name;
        isum = sum;
    }

    public int CurrentSum { get { return isum; } }

    public void Put(int sum) { isum += sum; }

    public void Withdraw(int sum)
    {
        if (isum >= sum)
        {
            isum -= sum;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        IAccount client = new Client("Tom", 200);
        Client clientNew = new Client("Tim", 400);
        client.Put(50);
        int x=client.Depos(1, 1);
        int y = clientNew.Depos(1, 1);
        int z= client.Depos(2, 2);
        Console.WriteLine("x=" + x);
        Console.WriteLine("y=" + y);
        Console.WriteLine("z=" + z);
        Console.WriteLine(client.CurrentSum);
        client.Withdraw(100);
        Console.WriteLine(client.CurrentSum);
        Console.Read();
    }
}
```

Інтерфейси у перетвореннях типів.

Як бачимо з прикладу "Множинна реалізація інтерфейсів IAccount — інтерфейс реалізується класом Client. Посилання на об'єкт Client може зберігатися у змінній IAccount. Усі екземпляри класу Client є екземпляри типу інтерфейс IAccount. Але не всі екземпляри типу інтерфейс IAccount є екземплярами класу Client, тому необхідно явне приведення

```
IAccount account = new Client("Tom", 200);
```

```
account.Put(200);  
Console.WriteLine(account.CurrentSum);  
Client client = (Client)account;
```

ІAccount інтерфейс не має властивості Name, тому необхідне явне приведення для її застосування:

```
string clientName = ((Client)account).Name;
```

Перетворення від класу до інтерфейсу виконується автоматично та аналогічно перетворенню від похідного типу до базового класу, оскільки клас Client реалізує IAccount — інтерфейс. У класі описано метод Depos. Він стає новим і не має відношення до методу, який визначено за замовчуванням у інтерфейсі і тому у викликах

```
int x=client.Depos(1, 1);  
int y = clientNew.Depos(1, 1);  
int z= client.Depos(2, 2);
```

використовується цей новий перевизначений метод. Але зверніть увагу на таке. Якщо ми в класі не реалізуємо метод інтерфейсу, який має реалізацію за замовчуванням, то це ще не означає, що ми можемо в екземплярі класу цей метод використовувати.

```
int y = clientNew.Depos(1, 1);
```

Буде помилка 'Client' does not contain a definition for 'Depos' and no accessible extension method 'Depos' ('Client' не містить визначення для 'Depos' и не має доступного методу розширення 'Depos'). Тобто ми бачимо, що «повного» успадкування не відбувається. Але ми можемо використовувати метод, реалізований у інтерфейсі за допомогою приведення типів:

```
int y = ((IAccount)clientNew).Depos(1, 1);
```

Перетворення від інтерфейсу до класу ідентичне перетворенню базового типу на похідний, оскільки не кожен об'єкт IAccount є екземпляром Client. Тут теж потрібна операція перетворення типів.

Давайте ще раз розглянемо поняття клас, спадкування, інтерфейс, реалізація, щоб зрозуміти коли що використовувати. Успадкування класів моделює відношення «Є» («is») («Кнопка меню» є «Кнопкою»), а базові класи зазвичай містять певний функціонал, тісно пов'язаний з функціоналом похідного класу. Базові класи не просто моделюють групу типів, а й дозволяють повторно використовувати існуючий функціонал. Інтерфейси також можуть моделювати відношення «Є» («Кнопка меню» є «ІКнопкою»), але більш правильно, з точки зору проектування програмного забезпечення, казати, що до інтерфейсів стосується менш жорстке від-

ношення «Має робити» («MUST DO»). Через один інтерфейс екземпляр класу може порівнюватися для сортування, через другий — обчислювати хеш, через інші — ітеруватися, клонуватися і т.д. Один і той же клас може підтримувати купу інтерфейсів з однаковими іменами членів - але реалізованих окремими незалежними алгоритмами - доступ до потрібної реалізації однієї і тієї ж сигнатури виклику члена визначатиметься тільки після приведення до потрібного інтерфейсу - така собі квінтесенція множинного поліморфізму ООП. Метод може повертати Інтерфейс - створюючи (або отримуючи ззовні) екземпляр деякого класу - і код, що викликає цей метод, не буде знати що це за клас - для нього доступний просто тип Інтерфейсу - з ним він і буде працювати. Такий підхід має лише одну істотну проблему - метод може приймати невідомий екземпляр класу як аргумент і може його потім повернути. Але не може створити - адже саме про вихідні типи вона нічого не знає - типи Інтерфейсів не створюють екземплярів (як і абстрактні класи). Для ефективного вирішення цієї проблеми потрібна чергова революція у програмуванні, а саме через вивчення теорії програмування розроблення правильних мов і відповідно правильних компіляторів. Розглянемо більш детально явну реалізацію методів в інтерфейсі. Нехай є такий приклад:

```
interface IA
{
    void M() { Console.WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { Console.WriteLine("IB.M"); } // явна реалізація
}
interface IC : IA
{
    void M() { Console.WriteLine("IC.M"); } // створення нового методу M
}
class Derived : IA, IB, IC // allowed?
{
    static void Main()
    {
        IA a = new Derived();
        a.M(); // що роздрукується?
        ((IA)a).M(); // що роздрукується?
        ((IB)a).M(); // що роздрукується?
        ((IC)a).M(); // що роздрукується?
    }
}
```

У перших трьох випадках буде роздруковано

IB.M

Через те що метод реалізовано явно і він замінив існуючий метод в інтерфейсі IA, в інтерфейсі IC ми описуємо новий інший метод, який не замінює метод інтерфейсу IA, тому в четвертому рядку буде надруковано

IC.M

Зверніть увагу також на таке використання реалізацій методів в інтерфейсі:

```
interface IA
{
    void M() { Console.WriteLine("IA.M"); }
}

interface IC : IA
{
    void M() { Console.WriteLine("IC.M"); } // створення нового методу M
}
class Derived : IA, IC // allowed?
{
    static void Main()
    {
        IA a = new Derived();
        a.M(); // що роздрукується?
        ((IA)a).M(); // що роздрукується?
        ((IC)a).M(); // що роздрукується?
    }
}
```

Буде роздруковано

IA.M  
IA.M  
IC.M

Ми бачимо, що залишається можливість використовувати реалізацію методу за замовчуванням з інтерфейсу та надається можливість викликати нову реалізацію з класу. Тепер нам залишилось розглянути «діамантову проблему» множинного спадкування. По-перше, згадаємо, що все ж таки спадкування інтерфейсів не має. Повторюю, можна використовувати реалізовані методи в інтерфейсах, зробивши приведення типів. Яка реалізація буде використовуватись при діамантовому спадкуванні?

```
interface IA
{
    void M() { Console.WriteLine("IA"); }
}
interface IB : IA
{
    void IA.M() { Console.WriteLine("IB"); }
}
class Base : IA
```

```
{
    void IA.M() { Console.WriteLine("Base"); }
}
class Derived : Base, IB
{
    static void Main()
    {
        IA a = new Derived();
        a.M(); // яка реалізація з класу Base чи інтерфейсу IB?
    }
}
```

Відповідь проста: клас має пріоритет над інтерфейсом, тобто роздруковане буде слово `Base`. Якщо ми приберемо явну реалізацію методу `M()` в інтерфейсі `IB` та класі `Base`, то в результаті отримаємо відповідь `IA`, тому що ми утворюємо екземпляр типу `Derived` та присвоюємо посилання на його змінний тип `IA` і відповідно буде викликатись метод, який реалізовано в цьому інтерфейсі.

Давайте тепер розглянемо «чисту» діамантову проблему, тобто без визначення проміжного класу:

```
interface IA
{
    void M() { Console.WriteLine("IA"); }
}
interface IB : IA
{
    void M() { Console.WriteLine("IB"); }
}
class Derived : IB, IA
{
    static void Main()
    {
        Derived a = new Derived();
        ((IA)a).M();
        ((IB)a).M();
    }
}
```

Навіть із синтаксису ми бачимо, що ніякої проблеми не існує, буде роздруковано:

IA  
IB

І вам буде зрозуміло, що й для цього прикладу проблем немає:

```
interface IA
{
    void M() { Console.WriteLine("IA"); }
}
interface IB
{
    void M() { Console.WriteLine("IB"); }
}
```

```
class Derived : IB, IA
{
    static void Main()
    {
        Derived a = new Derived();
        ((IA)a).M();
        ((IB)a).M();
    }
}
```

Буде теж роздруковано:

IA  
IB

Діамантова проблема не виникає через те, що ми використовуємо явне приведення типів. Без цього програма просто не запрацює. Але діамантова проблема знов з'являється у такому прикладі:

```
interface IA
{
    void M() { Console.WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { Console.WriteLine("IB.M"); } // явна реалізація
}
interface IC : IA
{
    void IA.M() { Console.WriteLine("IC.M"); } // явна реалізація
}
class Derived : IA, IB, IC // припустимо IA?
{
    static void Main()
    {
        IA a = new Derived();
        a.M(); // що роздрукується?
        ((IA)a).M(); // що роздрукується?
        ((IB)a).M(); // що роздрукується?
        ((IC)a).M(); // що роздрукується?
    }
}
```

Ми хочемо мати три різні реалізації методу M() але не можемо, через присутність двох явних реалізацій цього методу в інтерфейсах IB, IC. У даному прикладі ще з'являється помилка про відсутність реалізації методу інтерфейсу IA в класі Derived, через те, що ми зробили явну реалізацію цього методу в інтерфейсах IB, IC і саме ці явні реалізації можна буде викликати через екземпляр класу Derived. Щоб мати три різні реалізації, необхідно переробити їх на утворення нового методу:

```
interface IB : IA
{
    void M() { Console.WriteLine("IB.M"); }
```

```
}  
interface IC : IA  
{  
    void M() { Console.WriteLine("IC.M"); }  
}
```

Тоді буде надруковано

```
IA.M  
IA.M  
IB.M  
IC.M
```

Ще один приклад для самостійного розбору.

```
interface IWork  
{  
    void ToDo()  
    { Console.WriteLine("I'm working ..."); }  
}  
interface IPerson: IWork  
{  
    static string m_name="Bob";  
    static int m_age;  
  
    string getName() { return m_name; }  
    int getAge() { return m_age; }  
}  
  
interface IEmployee: IWork  
{  
    static string m_name = "Tom";  
    static string m_employer= "I'm Employer";  
    static double m_wage;  
  
    string getEmployer() { return m_employer; }  
    string getName() { return m_name; }  
    double getWage() { return m_wage; }  
}  
  
// Teacher спадкує Person и Employee  
class Teacher : IPerson , IEmployee  
{  
    private int teachesGrade;  
    void Print()  
    { Console.WriteLine("Teacher_is_working ..."); }  
}  
class Program  
{  
    static void Main(string [] args)  
    {  
        Teacher teacher = new Teacher();  
        Console.WriteLine(((IPerson)teacher).getName());  
        Console.WriteLine(((IEmployee)teacher).getName());  
        Console.WriteLine(((IEmployee)teacher).getEmployer());  
        ((IWork)teacher).ToDo();  
        Console.Read();  
    }  
}
```

}

Проект з розробки гри. Нехай у нас є задача спроектувати гру Warcraft з діючими особами варан, дракон, лев, грифон, танк, ... , вогнеметник. Як це зробити, щоб писати мінімальну кількість коду? Один з простих підходів (тобто не треба додатково щось вивчати) це використовувати спадкування.

### 6.18.1 Контрольні запитання

1. Яке головне завдання інтерфейсу?
2. Що визначають за допомогою інтерфейсів?
3. Який рівень доступу встановлюється інтерфейсу?
4. Чи можуть інтерфейси реалізовувати методи, статичні поля та члени за замовчуванням?
5. В чому відмінність між абстрактним класом та інтерфейсом?
6. Інтерфейси у перетворенні типів.
7. «Діамантова» проблема.

## 6.19 Абстрактні класи, Інтерфейси - крок до уніфікації

Існують різні рівні абстрагування або ігнорування деталей. Розглянемо три сутності: клас, абстрактний клас і інтерфейс.

Клас містить опис характеристик екземпляра і його поведінку.

Абстрактний клас – опис характеристик екземпляру, частковий опис поведінки й вказівка, яка поведінка повинна бути у екземплярів класів спадкоємців.

Інтерфейси вказують, яка повинна бути загальна (уніфікована) поведінка (методи) екземплярів класів, що реалізують даний інтерфейс.

Наявність інтерфейсів і абстрактних класів дозволяє думати про задачу більш абстрактно. Абстрактність інтерфейсів дозволяє зосередитися на головному, ігноруючи другорядні деталі.

Нехай у компанії працюють співробітники та є клієнти. Необхідно друкувати їх прізвище, ім'я та для співробітників назву відділу, а для клієнтів назву їхньої компанії. Знов розглянемо клас `Person`. Програма буде працювати з об'єктами співробітники й клієнти. Тому утворювати екземпляри класу `Person` не має необхідності. У такому випадку краще працювати з абстрактними класами. Такий клас має такі особливості:

- не можна створювати екземпляри абстрактного класу;
- абстрактний клас може містити як абстрактні методи/властивості, так і звичайні;
- у класі спадкоємці, якщо він не абстрактний, повинні бути реалізовані всі абстрактні методи й властивості, що оголошені в базовому класі.

Напишемо таку програму:

```
abstract class Person
{
    public string Firstname { get; set; }
    public string Lastname { get; set; }

    public Person(string Namelast, string Namefirst)
    {
        Firstname = Namelast;
        Lastname = Namefirst;
    }

    public abstract void Display();
}

class Employee : Person
{
    public string Department { get; set; }
    public string Salary { get; set; }
    public string Position { get; set; }
    public Employee(string Namefirst, string Namelast, string
        Dept, string
        Sal, string Pos) : base(Namefirst, Namelast)
    {
        Department = Dept;
        Salary = Sal;
        Position = Pos;
    }
    public override void Display()
    {
        Console.WriteLine(Firstname + "_" + Lastname + "
            Department:" + Department);
    }
}

class Client : Person
{
}
```

```
public string Company { get; set; }

public Client(string Namefirst, string Namelast, string
    Comp)
    : base(Namefirst, Namelast)
{
    Company = Comp;
}

public override void Display()
{
    Console.WriteLine(Firstname + "_" + Lastname + "_" +
        Company);
}
```

Розглянемо ще один приклад використання абстрактних класів і методів. Розглянемо задачу рисування й видалення фігур на екрані. Нам треба якось виділити ті методи базового класу, які необхідно обов'язково перевизначити у всіх класах спадкоємця (у кожному дочірньому класі). Це необхідно через те, що деякий загальний елемент інтерфейсу класу (група методів) не може бути уніфіковано описаний у базовому класі й вимагає специфіки для всіх класів спадкоємців. Кожна фігура повинна рисуватися Draw() по власному алгоритму. Але при цьому стирати фігури Erase() можна уніфіковано. Для цього визначимо абстрактний клас Afigure, у якому уніфіковані методи опишемо, а специфічні тільки оголосимо як абстрактні:

```
abstract class Afigure
{
    public abstract void Draw();
    public void Erase()
    {
        Console.WriteLine("Erase()_method_from_abstract_class_
            Afigure");
    }
}
```

Тепер, при створенні класу для роботи з колом, ми можемо використовувати абстрактний клас Afigure:

```
class Circle : Afigure
{
    public override void Draw()
    {
        Console.WriteLine("Draw()_method_from_class_Circles");
    }
}
```

Як бачимо, у цьому класі ми привели реалізацію абстрактного методу Draw(). Тепер напишемо саму програму:

```
class Program
{
    static void Main(String[] args)
    {
        Circle a = new Circle();
        a.Draw();
        a.Erase();
    }
}
```

Програма виведе на екран:

Draw( ) method from class Circles  
Erase() method from abstract class Afigure

Будь-який клас-спадкоємець абстрактного класу або зобов'язаний надати реалізацію всіх абстрактних методів свого базового класу, або сам повинен бути оголошений абстрактним. Розглянемо приклад використання абстрактних класів для створення сімейства класів. Нехай необхідно уніфіковано одержувати якусь інформацію про різних тварин: кішках, птахам, рибах і т.д. Наприклад, так:

```
foreach (Animal animal in animals)
    animal.Getinfo();
```

У даному фрагменті ми в сімействі `animals` перебираємо всіх тварин `animal` і виводимо про них інформацію на екран. Розглянемо клас `Animal` (тварини). Цей клас описує абстрактну сутність Тварина, екземпляр якої створювати не можна й нема рації, тому що ми будемо працювати не із тваринами, а з деякими видами тварин: кішками, птахами, рибами і тощо. У такому випадку, має сенс описати абстрактний клас `Animal` і від нього успадковувати вже необхідні нам класи для опису кішок, птахів, риб.

```
abstract class Animal
{
    public string Name { get; set; }
    public string Type { get; protected set; }
    public abstract void Getinfo(); // оголошення абстрактного
        методу
}
class Bird : Animal
{
    public Bird(string name)
    {
        Name = name;
        Type = "Птах";
    }
    public override void Getinfo() // реалізація абстрактного методу
    {
        Console.WriteLine("Тип:␣" + Type + "\n" + "Імя':␣" +
            Name + "\n");
    }
}
```

```
    }
}
class Cat : Animal
{
    public Cat(string name)
    {
        Name = name;
        Type = "Кішка";
    }
    public override void Getinfo() // реалізація абстрактного методу
    {
        Console.WriteLine("Тип:_" + Type + "\n" + "Імя':_" + Name + "\n");
    }
}
class Fish : Animals
{
    public Fish(string name)
    {
        Name = name;
        Type = "Риба";
    }
    public override void Getinfo() // реалізація абстрактного методу
    {
        Console.WriteLine("Тип:_" + Type + "\n" + "Імя':_" + Name + "\n");
    }
}
class Program
{
    static void Main(string[] args)
    {
        List<Animal> animals = new List<Animal>();
        animals.Add(new Bird("Пірат"));
        animals.Add(new Cat("Мурка"));
        animals.Add(new Fish("Мовчун"));

        foreach (Animal animal in animals)
            animal.Getinfo();

        Console.ReadKey();
    }
}
```

У даному прикладі ми працюємо з одним списком тварин `animals`, і, викликаючи один метод `Getinfo()`, ми одержуємо інформацію про відповідну тварину. Абстрактні класи призначені тільки для використання як базового класу для інших класів.

Інтерфейс - це явно зазначена специфікація набору методів, які мають бути представлені в класі, що реалізує цю специфікацію. Реалізація ж цих методів в інтерфейсі відсутня, так само як в абстрактних класах відсутня реалізація абстрактних методів.

Слід зазначити, що конкретний клас може бути спадкоємцем лише одного базового класу, але в ньому може бути реалізова-

но необмежена кількість інтерфейсів. Варто помітити, що абстрактний клас успадковується (inheritance), а інтерфейс реалізується (implements).

Головна відмінність класу від інтерфейсу — у тому, що клас складається зі своєї інтерфейсної частини (методи, властивості, які забезпечують використання класу ззовні) і змістовної частини (поля).

Щоб не було термінологічної плутанини, методи, які забезпечують використання класу ззовні назовемо методами-інтерфейсами.

Таким чином, будемо розрізняти екземплярний інтерфейс, який складається з методів-інтерфейсів і типовий інтерфейс, що складається з методів, сигнатури яких описані в інтерфейсі і які потрібні для взаємодії різних за природою класів.

Звернемо увагу на те, що:

Клас говорить про те, "що таке екземпляр цього класу, і що він уміє робити і як".

Абстрактний клас говорить про те, що "чим повинен бути екземпляр успадкованого від нього класу, що він уміє робити й що він повинен уміти робити".

Інтерфейс декларує те, що "клас, реалізуючий цей інтерфейс, повинен уміти робити".

Розглянемо приклад.

Нехай ми описали абстрактний клас Ссавці й успадковували від нього клас Люди. Тоді екземпляр класу Люди має ті ж властивості, що й усі ссавці: тепла кров, народжується, а не вилуплюється з яйця, для продовження роду потрібний і самець і самка, є серце, хребет, мозок тощо.

Абстрактний метод-інтерфейс цього класу Ссавці говорить, що повинен уміти робити екземпляр успадкованого від нього класу, наприклад, годувати молоком. А метод-інтерфейс класу Люди указує нам, як саме необхідно виконувати годівлю молоком.

У той час, як, наприклад, інтерфейс Ното буде говорити, що повинен уміти робити екземпляр класу Люди, що реалізує цей інтерфейс. Наприклад, думати абстрактно, перетворювати навколишнє середовище і т.і. (чим там ще відрізняється людина від інших ссавців).

Інтерфейси потрібні, щоб об'єднати принципово різні класи чимось спільним. Тобто, припустимо, ми маємо інтерфейс Imovable. Коли клас реалізує цей інтерфейс, то він зобов'язаний реалізувати всі методи, які в ньому представлені. При цьому відразу зрозуміло, що даний клас і всі класи, які реалізують інтерфейс Imovable,

є уніфікованими класами – їх екземпляри можуть рухатися.

Можна дійти такого висновку, що інтерфейс потрібний для того, щоб уніфікувати якусь абстракцію.

Звернемося ще раз до прикладу із класом Elevator.

Нехай у нього є публічний метод Move(), який здійснює рух ліфта нагору або вниз, викликаючи приватні методи рух вгору - Turnup() і рух униз - Turndown(). Екземплярний інтерфейс цього класу складається з методу Move(). Коли ми від цього класу успадковуємо клас, наприклад, OldElevator(), то він успадковує й цей метод-інтерфейс і реалізацію методів Turnup(), Turndown(), які описують, наприклад, роботу електричного двигуна.

Тепер, коли ми захочемо ліфту додати рух, то скористаємося методом-інтерфейсом класу Move() і його «моторними» методами: вгору - Turnup() і вниз - Turndown(). Припустимо, ми прагнемо створити новий ліфт NewElevator(), який буде використовувати новий тип двигуна - антигравітаційний двигун.

Якщо ми успадковуємо Newelevator() від класу Elevator(), то для запуску ліфта будемо користуватися методом-інтерфейсом Move(). Але при цьому будуть успадковуватися й методи Turnup(), Turndown(), які, по своїй суті не відповідають новому виду двигуна.

Тому нам доведеться при перевизначенні методу-інтерфейсу Move() реалізовувати нові методи для опису роботи нового типу двигуна. При цьому, щоб не упустити деталі реалізації методів Turnup(), Turndown() (наприклад, вони зберігали номер поверху, на якому перебував ліфт або при кількох викликах – усі поверхи, на яких ліфт повинен зупинитися), нам треба буде довідатися реалізацію цих методів, тобто порушити інкапсуляцію.

Тому правильніше буде відокремити інтерфейс від реалізації. Для цього оголосимо інтерфейс IMovable, який містить метод Move().

```
interface IMovable
{
    void Move(string to);
}
```

Опишемо клас Electricturnengine, що реалізує інтерфейс IMovable, і в цьому класі опишемо реалізацію методу Move(), Turnup(), Turndown().

```
class Electricturnengine : IMovable
{
    public void Move(string to)
    {
        switch (to)
        {
            case "up":
                Turnup();
        }
    }
}
```

```
        break;
        case "down":
            Turndown();
        break;
    default:
        Console.WriteLine("Stop");
        break;
    }
}
private void Turnup()
{
    Console.WriteLine("electric_Move_to_up");
}

private void Turndown()
{
    Console.WriteLine("electric_Move_to_down");
}
}
```

Також опишемо клас Antigravengine, що реалізує інтерфейс IMovable за допомогою методів Up(), Down().

```
class Antigravengine : IMovable
{
    public void Move(string to)
    {
        switch (to)
        {
            case "up":
                Up();
                break;
            case "down":
                Down();
                break;
            default:
                Console.WriteLine("Stop");
                break;
        }
    }
    private void Up()
    {
        Console.WriteLine("antigrav_Move_to_up");
    }

    private void Down()
    {
        Console.WriteLine("antigrav_Move_to_down");
    }
}
```

Таким чином, за необхідності запустити ліфт ми будемо використовувати метод типового інтерфейсу, а не конкретний метод-інтерфейс класу, і у такий спосіб ми перейдемо на більш високий рівень абстракції. Слід помітити, що можна було б створити

абстрактний клас Engine з абстрактними методами Up(), Down() і успадковувати від нього класи Electricturnengineelevator і Anti-gravengineelevator з необхідною реалізацією методів Up(), Down(). Але тоді б порушувався принцип підстановки Лисков. Зміст ОО проєктування в тому й полягає, що треба вміти знаходити абстракції й виділяти з них найбільш загальні. У даному випадку загальним є те, що ліфт повинен рухатися. Інтерфейси також є зручним інструментом для прийняття правильних проєктних рішень. При розробці програмного забезпечення команда розробників спочатку займається обговоренням і проєктуванням, на етапах якого описується, що окремі блоки повинні робити і як між собою взаємодіяти, тобто створюються інтерфейси майбутніх класів. Після цього можна братися й до програмування. Тепер усі учасники розробки будуть використовувати типові інтерфейси, які реалізуються різними командами, не знаючи деталей реалізації, а знаючи лише те, що повинні робити методи, як одержати результат і що для цього потрібно. Можна сказати, що розробники, реалізуючи інтерфейс, реалізують спеціальний вид поліморфізму. Розглянемо приклад програми, у якій необхідно організувати доступ до різних баз даних: Mysql, ORACLE тощо. Доступ до різними БД реалізований по-різному. Але спосіб доступу повинен бути уніфікованим. Спосіб доступу і є типовим інтерфейсом. Нехай ми прагнемо для обміну даними з різними базами даних використовувати уніфікований метод Senddata(параметри методу):

```
public void Senddataпараметри( методу)
{
    Openconection();          // установлення з'єднання
    // команди обміну даними
    Closeconection();         // закриття з'єднання
}Дляцьогозадамовпрограмітиповийінтерфейс

        Iconnection , якийміститьдваметоди          : Openconection()
        длявстановленняз'єднання      ' й Closeconection()
        длязакриттяз'єднання      '.
```

```
interface Iconnection
{
public void Openconection();
public void Closeconection();
}
```

І використовуючи цей типовий інтерфейс, можете описати різні класи для з'єднання, які будуть реалізовувати інтерфейс IConnection з урахуванням специфіки різних баз даних.

```
public Class Mysqlconnection : Iconnection
// з'єднання з базою даних Mysql
```

```
{
    public void Openconnection()
    {
        //команди встановлення з'єднання з Mysql
    }

    public void Closeconnection()
    {
        //команди закриття з'єднання Mysql
    }
}

public Class Oracleconnection : Iconnection
// з'єднання з базою даних Oracle
{
    public void Openconnection()
    {
        // команди встановлення з'єднання з Oracle
    }

    public void Closeconnection()
    {
        //команди закриття з'єднання Oracle
    }
}
```

Остаточно метод Senddata(параметри методу) з інтерфейсом Iconnection буде виглядати так:

```
public void Senddatapараметри( методу)
{
    Openconnection()      // установлення з'єднання
    // команди обміну даними
    Closeconnection()     // закриття з'єднання
}
```

Таким чином, метод Senddata() може працювати з будь-якою базою даних, незалежно від її специфіки:

```
Iconnection mysql = new Mysqlconnection();
Iconnection oracle = new Oracleconnection();
mysql.Senddata(...);
oracle.Senddata(...);
```

Ми можемо додавати в програму можливість роботи з іншими базами, не міняючи код самого методу й відповідно програми. Достатньо, щоб кожний клас для встановлення з'єднання реалізовував інтерфейс Iconnection.

Підвівши попередній підсумок, можна сказати, що:

- Абстрактний клас слід використовувати, коли необхідно описати чи задати загальні характеристики й поведінку екземплярів.
- Інтерфейс – у випадку, коли потрібно задати чи описати загальний принцип взаємодії між сутностями та описати задачу на

рівні функціональності. У нашому випадку метод `Move()` і є цією спільністю, тому оформлений у вигляді типового інтерфейсу.

Якщо розглядати інтерфейси й абстрактні класи неформально, то класи будемо застосовувати для опису типів об'єктів, які використовуються у задачі – проектування програми на об'єктному рівні, а інтерфейси будемо використовувати для опису поведінки об'єктів задачі – проектування програми на функціональному рівні. Останнє проектування більш цікаве та творче.

Наприклад, давайте розглянемо проектування комп'ютерної стратегічної гри Варкрафт. При проектуванні на об'єктному рівні ми будемо описувати об'єкти, які будуть брати участь у грі: робітників, піхотинців, лучників, вершників, орків, грифонів, динозаврів, магів, тощо. Цей опис проводимо за допомогою абстрактних класів та класів. Потім будемо встановлювати, як ці об'єкти будуть воювати.

При проектуванні на функціональному рівні ми будемо описувати, як об'єкти будуть воювати: стріляти, пускати стріли, літати, чаклувати, стріляти вогнем, бігати, тощо. А потім будемо цю функціональність присвоювати класам і в класах реалізовувати.

З формального погляду можна сказати, що:

1) Клас може реалізовувати кілька типових інтерфейсів, але може успадковуватися тільки від одного базового абстрактного класу.

2) Структури можуть реалізовувати типові інтерфейси, але не можуть успадковуватися від довільного абстрактного класу (завжди неявно успадковуються від `System.ValueType`).

3) Типові інтерфейси можуть успадковуватися від типових інтерфейсів, але не можуть успадковуватися від абстрактних класів.

4) Доступність (`public`, `internal`...) реалізованих класом типових інтерфейсів може бути будь-якою, у той час як базовий абстрактний клас повинен бути не менш доступний, ніж похідний.

5) При визначенні `user defined conversions` тип їх аргументу може бути абстрактним класом, але не може бути типовим інтерфейсом.

6) Абстрактні класи можуть містити конструктори, а типові інтерфейси — ні.

7) Абстрактний клас може мати неабстрактні екземплярні методи (і ін. члени) і поля, а інтерфейси — ні.

8) Абстрактні класи `C#` можуть мати статичні методи (і ін. члени), а інтерфейси — ні.

9) Члени інтерфейсу можуть бути реалізовані як явно, так і не явно. При спадкуванні від абстрактного класу такий вибір відсутній.

10) Абстрактний клас може встановлювати різну доступність (public, internal...) для своїх членів, у той час як інтерфейс не може вказувати доступність для своїх, і вони завжди неявно public.

Слід зазначити ще один аспект використання інтерфейсів і абстрактних класів. При створенні бібліотек необхідно враховувати, що інтерфейси не можна змінити. Одного разу описаний інтерфейс неможливо змінити, не змінюючи клієнтів, що його реалізують.

Інтерфейси активно використовуються в бібліотеках для розширення певної поведінки. Абстрактний базовий клас занадто сильно обмежує клієнта, що може зробити його використання неможливим або занадто обмеженим. Не можна також не відзначити схожість інтерфейсів і абстрактних класів. Прийнято вважати, що сімейство класів створюють на основі абстрактного класу. Але також для цієї мети можна використовувати й інтерфейси. Наприклад, при визначенні сімейства «сортуємих» класів, ці класи можуть реалізовувати один інтерфейс ISort з індивідуальною для кожного класу реалізацією операції порівняння. Розглянемо приклад створення, на основі загального інтерфейсу, сімейства ліфтів з різною системою руху: за допомогою електричного двигуна й за допомогою антигравітаційного.

```
class Program
{
    static void Main(string[] args)
    {
        List<IMovable> elevators = new List<IMovable>();
        elevators.Add(new Electricturnengine());
        elevators.Add(new Antigravengine());

        foreach (IMovable el in elevators)
            el.Move("up");

        Console.ReadKey();
    }
}
```

Остаточно програма набуде такого вигляду:

```
interface IMovable
{
    void Move(string to);
}
class Electricturnengine : IMovable
{
    public void Move(string to)
    {
        switch (to)
        {
            case "up":
                Turnup();
        }
    }
}
```

```
        break;
        case "down":
            Turndown();
        break;
    default:
        Console.WriteLine("Stop");
        break;
    }
}
private void Turnup()
{
    Console.WriteLine("electric_Move_to_up");
}

private void Turndown()
{
    Console.WriteLine("electric_Move_to_down");
}
}

class Antigravengine : IMovable
{
    public void Move(string to)
    {
        switch (to)
        {
            case "up":
                Up();
                break;
            case "down":
                Down();
                break;
            default:
                Console.WriteLine("Stop");
                break;
        }
    }
    private void Up()
    {
        Console.WriteLine("antigrav_Move_to_up");
    }

    private void Down()
    {
        Console.WriteLine("antigrav_Move_to_down");
    }
}

class Program
{
    static void Main(string[] args )
    {
        List< IMovable > elevators = new List< IMovable >();
        elevators.Add(new Electricturnengine());
        elevators.Add(new Antigravengine());

        foreach (IMovable el in elevators)
```

```
        el.Move("up");  
        Console.ReadKey();  
    }  
}
```

### 6.19.1 Контрольні запитання

1. Що таке абстрактний клас?
2. Особливості абстрактного класу.
3. Чи можна створити екземпляр абстрактного класу?
4. Модифікатори доступу в абстрактному класі.
5. Реалізація абстрактних методів та властивостей.

## 6.20 Класи, абстрактні класи, інтерфейси. Спадкування чи реалізація?

Приклад об'єктно-орієнтованого конструювання програм

Барбарой Лісков у 1987 році на науковій конференції був запропонований принцип "правильного" використання спадкування. Цей принцип є найважливішим критерієм для оцінки якості прийнятих рішень при побудові ієрархій спадкування.

*Якщо для кожного об'єкта  $o1$  типу  $S$  існує об'єкт  $o2$  типу  $T$ , такий, що для будь-якої програми  $P$ , коректної в термінах  $T$ , поведінка  $P$  не змінюється при заміні  $o2$  на  $o1$ , то  $S$  є підтипом  $T$ .*

Іншими словами, властивість  $q$ , що виконується для об'єктів множини (підтипу)  $S$ , винна виконуватися для об'єктів підмножини (типу)  $T$ .

Спадкування повинне розширювати функціональність програмного забезпечення. Однак, як впливає із принципу Лісков, розширена функціональність має бути реалізована введенням додаткових методів, а не зміною поведінки старих.

При цьому необхідно враховувати такі фактори:

1. Похідні класи не повинні підсилювати передумови (не повинні вимагати більшого від своїх клієнтів).

2. Похідні класи не повинні послабляти постумови (повинні гарантувати, як мінімум теж саме, що й базовий клас).

3. Похідні класи не повинні порушувати інваріанти базового класу (інваріанти базового класу й спадкоємців доповнюють один одного).

4. Похідні класи не повинні генерувати виключення, що не описані базовим класом.

Давайте розглянемо як реалізувати принцип Барбари Лісков (LSP) мовою C#. Будемо використовувати класичний приклад: «квадрат – прямокутник» і наші знання із взаємовідносин класів, абстрактних класів, звичайних класів, інтерфейсів, тощо.

1. Класичне використання спадкування у відношенні «квадрат є прямокутником» з `virtual` дає у відповіді 25. Наведемо програму, яку ми можемо написати без глибоко розуміння реалізації спадкування мовою програмування.

```
class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle
{
    public override int Width
    {
        get
        {
            return base.Width;
        }

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public override int Height
    {
        get
        {
            return base.Height;
        }

        set
        {
            base.Height = value;
        }
    }
}
```

```
        base.Width = value;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle rect = new Square();
        rect.Height = 10;
        rect.Width = 5;
        Console.WriteLine("Area_of_square_with_up_is:{0}", rect
            .GetArea());
    }
}
Area of square with up is: 25
```

Принцип LSP порушується.

2. Приклад без virtual дає у відповіді 50. Класичний принцип LSP не порушується.

```
class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle
{
    public int Width
    {
        get
        {
            return base.Width;
        }

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public int Height
    {
        get
        {
            return base.Height;
        }
    }
}
```

```
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Rectangle rect = new Square();
        rect.Height = 10;
        rect.Width = 5;
        Console.WriteLine("Area_of_square_with_up_is:{0}", rect
            .GetArea());
    }
}

Area of square with up is: 50
```

Тобто отримали коректну відповідь для прямокутника.

3. Розглянемо цей же приклад, але детальніше його проаналізуємо. Показані площі окремо для фігур (прямокутника і квадрата) і з приведенням квадрата до прямокутника

```
class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle
{
    public int Width
    {
        get
        {
            return base.Width;
        }

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public int Height
    {
```

```

        get
        {
            return base.Height;
        }

        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Square sq = new Square();
        sq.Height = 10;
        Console.WriteLine("Area_of_square_is:{0}", sq.GetArea());
        ;
        Rectangle rect = new Rectangle();
        rect.Height = 10;
        rect.Width = 5;
        Console.WriteLine("Area_of_rectangle_is:{0}", rect.
            GetArea());

        rect = sq;
        rect.Height = 15;
        rect.Width = 5;
        Console.WriteLine("Area_of_square_up_to_rect_is:{0}",
            rect.GetArea());

    }
}

```

```

Area of square is: 100
Area of rectangle is: 50
Area of square up to rect is: 75

```

4. Спробуємо модифікувати цей приклад. Наступний приклад обчислює площі окремо для фігур і з приведенням квадрата до прямокутника, але в класі спадкоємця ще описаний метод `GetArea()`. Новий метод не перекриє метод базового класу. Навпаки, кажуть, що він сховає метод базового класу. При цьому компілятор, вирішуючи, який метод викликати, завжди буде викликати метод з того типу даних, до якого належить змінна, який тип даних заданий при її оголошенні!

```

class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea()

```

```
        {
            return Width * Height;
        }
    }

class Square : Rectangle
{
    public int Width
    {
        get
        {
            return base.Width;
        }

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public int Height
    {
        get
        {
            return base.Height;
        }

        set
        {
            base.Height = value;
            base.Width = value;
        }
    }

    public int GetArea()
    {
        return Width * Height*100000;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Square sq = new Square();
        sq.Height = 10;
        Console.WriteLine("Area_of_square_is:{0}", sq.GetArea());
        ;
        Rectangle rect = new Rectangle();
        rect.Height = 10;
        rect.Width = 5;
        Console.WriteLine("Area_of_rectangle_is:{0}", rect.
            GetArea());

        rect = sq;
        rect.Height = 15;
        rect.Width = 5;
        Console.WriteLine("Area_of_square_up_to_rect_is:{0}",
```

```
        rect.GetArea());  
  
    }  
  
}  
Area of square is: 10000000  
Area of rectangle is: 50  
Area of square up to rect is: 75
```

Але при цьому маємо, що після «приведення» до базового типу та зміни довжин сторін з посиланням з прямокутника на квадрат, змінюються і сторони самого квадрату.

```
{  
    Square sq = new Square();  
    sq.Height = 10;  
    Console.WriteLine("Area_of_square_is:_{0}", sq.GetArea());  
    ;  
    Rectangle rect = new Rectangle();  
    rect.Height = 10;  
    rect.Width = 5;  
    Console.WriteLine("Area_of_rectangle_is:_{0}", rect.  
        GetArea());  
  
    rect = sq;  
    rect.Height = 15;  
    rect.Width = 5;  
    Console.WriteLine("Area_of_square_up_to_rect_is:_{0}",  
        rect.GetArea());  
  
    Console.WriteLine("Area_of_square_after_up_to_rect_is:_  
        {0}", sq.GetArea());  
  
}
```

Відповідно змінюється і площа квадрату.

```
Area of square is: 10000000  
Area of rectangle is: 50  
Area of square up to rect is: 75  
Area of square after up to rect is: 7500000
```

5. Давайте використаємо інтерфейси. Спробуємо за допомогою інтерфейсів виправити ситуацію. Введемо інтерфейс і в нього запишемо властивості, тому що саме властивості змінюють довжини сторін.

```
interface ISquare  
{  
    int Width  
    {  
        get;  
        set;  
    }  
    int Height  
    {  
        get;  
    }  
}
```

```
        set;
    }
}

class Rectangle: ISquare
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle, ISquare
{
    public int Width
    {
        get
        {
            return base.Width;
        }

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public int Height
    {
        get
        {
            return base.Height;
        }

        set
        {
            base.Height = value;
            base.Width = value;
        }
    }

    public int GetArea()
    {
        return Width * Height*100000;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Square sq = new Square();
        sq.Height = 10;
        Console.WriteLine("Area_of_square_is:{0}", sq.GetArea())
        ;
    }
}
```

```
Rectangle rect = new Rectangle();
rect.Height = 10;
rect.Width = 5;
Console.WriteLine("Area_of_rectangle_is:{0}", rect.
    GetArea());

rect = sq;
rect.Height = 15;
rect.Width = 5;
Console.WriteLine("Area_of_square_up_to_rect_is:{0}",
    rect.GetArea());

}

}
Area of square is: 10000000
Area of rectangle is: 50
Area of square up to rect is: 75
```

Однак незважаючи на те, що інтерфейси вимагають реалізацію методів в класі, у даному прикладі реалізацію замінює успадкування. І введення інтерфейсу нічого не дає. Та й не повинно було давати, тому що компілятор, вирішуючи, який метод викликати, завжди буде викликати метод з того типу даних, до якого належить змінна, як тип даних заданий при її оголошенні!

```
interface ISquare
{
    int Width
    {
        get;
        set;
    }
    int Height
    {
        get;
        set;
    }
}

class Rectangle : ISquare
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle, ISquare
{
    public int Width
    {
        get
        {
```

```
        return base.Width;
    }

    set
    {
        base.Width = value;
        base.Height = value;
    }
}

public int Height
{
    get
    {
        return base.Height;
    }

    set
    {
        base.Height = value;
        base.Width = value;
    }
}

public int GetArea()
{
    return Width * Height * 100000;
}
}

class Program
{
    static void Main(string[] args)
    {
        Square sq = new Square();
        sq.Height = 10;
        Console.WriteLine("Area_of_square_is:{0}", sq.GetArea());
        ;
        Rectangle rect = new Rectangle();
        rect.Height = 10;
        rect.Width = 5;
        Console.WriteLine("Area_of_rectangle_is:{0}", rect.
            GetArea());

        rect = sq;
        rect.Height = 15;
        rect.Width = 5;
        Console.WriteLine("Area_of_square_up_to_rect_is:{0}",
            rect.GetArea());

        Console.WriteLine("Area_of_square_after_up_to_rect_is:
            {0}", sq.GetArea());
    }
}

Area of square is: 10000000
Area of rectangle is: 50
Area of square up to rect is: 75
```

Area of square after up to rect is: 7500000

6. Якщо розглядати принцип Барбери Лісков у розширеному тлумаченні, тобто «При підстановці замість екземпляра базового класу екземпляра дочірнього класу властивості ПРОГРАМИ не мають змінюватися», тоді одержуємо, що представлена реалізація не задовольняє даному принципу. Тому що при підстановці замість екземпляра прямокутника, екземпляра квадрата й при роботі із квадратом, як із прямокутником:

```
rect.Height = 10;  
rect.Width = 5;
```

отримаємо, що міняються й сторони квадрата. Тим самим ЗМІНИЛИСЯ властивості ПРОГРАМИ. При цьому програма працює коректно (правильно обчислює площу прямокутника) з погляду принципу LSP у оригінальному формулюванні. Наведемо програму, для якої виконується принцип Барбери Лісков у розширеному формулюванні:

```
abstract class Figure  
{  
    protected int width;  
    protected int height;  
    public abstract int GetArea();  
}  
class Rectangle: Figure  
{  
  
    public override int GetArea()  
    {  
        return width * height;  
    }  
  
    public int Width  
    {  
        get  
        {  
            return Width;  
        }  
  
        set  
        {  
            width = value;  
        }  
    }  
  
    public int Height  
    {  
        get  
        {  
            return Height;  
        }  
    }  
}
```

```
        set
        {
            height = value;
        }
    }
}

class Square : Figure
{
public int Width
{
    get
    {
        return Width;
    }

    set
    {
        width = value;
        height = value;
    }
}

public int Height
{
    get
    {
        return Height;
    }

    set
    {
        height = value;
        width = value;
    }
}

public override int GetArea()
{
    return width * height;
}
}

class Program
{
static void Main(string[] args)
{
    List<Figure> figures = new List<Figure>();
    Square sq = new Square();
    sq.Height = 10;
    Rectangle rec = new Rectangle();
    rec.Width = 5;
    rec.Height = 10;
    figures.Add(sq);
    figures.Add(rec);
    foreach (Figure fig in figures)
    {
        Console.WriteLine("Square is : {0}", fig.GetArea());
    }
}
```

```
    }  
  
}  
  
}
```

Відповідь:

```
Square is: 100  
Square is: 50
```

У цьому проектному рішенні ми вже не розглядаємо квадрат як прямокутник. А обидва чотирикутника розглядаємо як абстрактні фігури. Цей приклад дає чітке уявлення, що ООП не слід сприймати як апарат для моделювання дійсності. При використанні ООП для вирішення конкретних задач ми маємо АБСТРАГУВАТИСЬ від конкретної предметної області.

### 6.20.1 Контрольне запитання

1. У чому полягає принцип Барбари Лісков?

## 6.21 Множинне спадкування чи реалізація інтерфейсів?

Почнемо з розгляду реалізації за замовчуванням.

Інтерфейси все ж таки відрізняються від абстрактних класів. Класи, що реалізують інтерфейси, не успадковують члени, що реалізовані за замовчуванням в інтерфейсах. Тому, якщо клас залишає член, що реалізований в інтерфейсі, без своєї реалізації, то в класі немає цього члена і, відповідно, не можна до нього звертатися.

```
class Program  
{  
    interface IA1  
    {  
        void M1()  
    }  
    {  
        Console.WriteLine("IA.M1");  
    }  
    }  
    class C : IA1  
    {  
    }  
    static void Main(string[] args)  
    {
```

```
        C i = new C();
        i.M1(); // error!
    }
}
```

Для використання необхідно екземпляр привести до типу цього інтерфейсу, щоб дістатися до таких членів.

```
static void Main(string[] args)
{
    C i = new C();
    ((IA1)i).M1(); // prints "IA.M1"
}
```

Члени інтерфейсу за замовчуванням дозволяють додавати нових членів до інтерфейсу в наступних версіях, не порушуючи сумісність вихідного коду з існуючими реалізаціями цього інтерфейсу. Тобто не виникає необхідності реалізовувати їх в усіх класах, що мають відношення до цього інтерфейсу. Інтерфейси можуть мати статичні члени.

```
interface IStaticMembers
{
    private static string commandName = "Default_command_name";
    private void build(string name)
    { commandName = name; }
    public void setCommandName(string name) { build(name); }
}
```

Інтерфейси можуть містити віртуальні члени.

Але клас, який успадковує інтерфейс, перевизначати віртуальний член не може. Перевизначати віртуальні члени інтерфейсів можуть лише інтерфейси.

```
interface IVirtualMembers
{
    public virtual void sendNotification()
    {
        Console.WriteLine("IVirtualMembers's_sendNotification");
    }
}
interface IOverrideVirtualMembers : IVirtualMembers
{
    void IVirtualMembers.sendNotification()
    {
        Console.WriteLine("IOverrideVirtualMembers's_sendNotification");
    }
}
```

Множинне спадкування

Розглянемо приклад імітації множинного спадкування засобами інтерфейсів (работает).

```
class Program
{
    interface IA1
    {
        void M1() { Console.WriteLine("IA.M1"); }
    }
    interface IA2
    {
        void M2() { Console.WriteLine("IA.M2"); }
    }
    class C : IA1, IA2 { } // OK
    static void Main(string[] args)
    {
        C i = new C();
        ((IA1)i).M1(); // prints "IA.M1"
        ((IA2)i).M2(); // prints "IA.M2"
    }
}
```

А можна просто використати композицію замість того, щоб намагатися імітувати множинне спадкування (працює).

```
class Program
{
    public interface IFoo
    {
        void Foo();
    }

    public interface IBar
    {
        void Bar();
    }

    public class FooService : IFoo
    {
        public void Foo()
        {
            Console.WriteLine("Foo");
        }
    }

    public class BarService : IBar
    {
        public void Bar()
        {
            Console.WriteLine("Bar");
        }
    }

    public class FooBar : IFoo, IBar
    {
        FooService fooService = new FooService();
        BarService barService = new BarService();

        public void Foo()
        {

```

```
        this.fooService.Foo();
    }
    public void Bar()
    {
        this.barService.Bar();
    }
}
static void Main(string[] args)
{
    FooBar fb = new FooBar();
    fb.Bar();
    fb.Foo();
}
}
```

### 6.21.1 Контрольні запитання

1. Що дозволяють робити члени інтерфейсу за замовчуванням?
2. Що таке множинне наслідування?
3. Яким чином можна замінити множинне наслідування?

## 6.22 Делегати, анонімні методи й лямбда-вирази

### 6.22.1 Делегати

Розглянемо просте завдання – сортування чисел. Нехай ми маємо кілька готових методів, які сортують дані різними алгоритмами. Хочемо, щоб у різних випадках ми використовували різні алгоритми. Щоб повідомити головному методу, яким саме алгоритмом ми хочемо відсортувати дані, у головний метод необхідно передати метод, який і виконуватиме необхідний алгоритм сортування. Тобто виникає необхідність передавати метод як звичайну змінну. Делегат — це об'єктно-орієнтований метод роботи із методом як із змінною. Його можна додати, змінити або видалити будь-коли. Працювати з методом, саме як із звичайною змінною.

Розглянемо такий приклад: одержати суму або різницю 2 цілих чисел:

```
class Program
{
    static void Main(string[] args)
    {
        //Стандартний спосіб
```

```
Console.WriteLine("10+15=_{0}", Sum(10, 15));
Console.WriteLine("5-_3=_{0}", Sub(5, 3));

Console.ReadLine();
}

//Сума двох чисел
public static int Sum(int x, int y)
{
    return x + y;
}

//Різниця двох чисел
public static int Sub(int x, int y)
{
    return x - y;
}
}
```

Результат друку на екран:

```
10 + 15 = 25
5 - 3 = 2
```

Розглянемо інший приклад. Нехай потрібно обчислити визначений інтеграл від деякої функції. Причому при реалізації методу обчислення інтегралу ще невідомо від якої він функції обчислюється. Тобто необхідно передавати функцію як параметр методу. Для розв'язання цієї проблеми використовують делегати.

Делегати представляють такі об'єкти, які вказують на методи. Тобто делегати - це покажчики на методи і за допомогою делегатів ми можемо викликати дані методи. Для оголошення делегата використовується ключове слово `delegate`, після якого йде тип, що повертається, назва і параметри. Наприклад:

```
delegate void Message ( );
```

Делегат `Message` повертає значення типу `void` (тобто нічого не повертає) і не приймає жодних параметрів. Це означає, що цей делегат може вказувати на будь-який метод, який не приймає ніяких параметрів і нічого не повертає. Об'єкт типу делегат має такі особливості:

- адресу методу, який він викликає;
- аргументи (якщо є) для цього методу;
- яке значення (якщо є) цей метод має повертати.

Як тільки делегат створений і проініціалізований, він може динамічно, під час виконання програми, викликати методи, на які вказує, наприклад:

Покажемо цей підхід на попередньому прикладі обчислення суми й різниці двох чисел.

```
class Program
{
    public delegate int Mathdelegate(int x, int y); //делегат
        з двома аргументами
    //типу int, який повертає int

    static void Main(string[] args)
    {
        //Стандартний спосіб
        Console.WriteLine("5+3={0}", Sum(5, 3));
        Console.WriteLine("5-3={0}", Sub(5, 3));

        Console.WriteLine();

        //Використання делегата
        Mathdelegate md = new Mathdelegate(Sum); //Передаємо
            делегатові посилання на метод Sum
        Console.WriteLine("5+3={0}", md(5, 3));
        md = Sub; //Тепер передаємо делегатові посилання на метод Sub
        Console.WriteLine("5-3={0}", md(5, 3));

        Console.ReadLine();
    }

    //Sums to numbers
    public static int Sum(int x, int y)
    {
        return x + y;
    }

    //Subtracts two numbers
    public static int Sub(int x, int y)
    {
        return x - y;
    }
}
```

Результат друку на екран:

```
5 + 3 = 8
5 - 3 = 2

5 + 3 = 8
5 - 3 = 2
```

Розгорнемо цей приклад у більш красивий і цікавий:

```
// Створимо делегат
delegate int IntOperation ( int i , int j);

class Program
{
    // Опишемо ряд методів
    static int Sum ( int x, int y)
```

```
{
    return x + y;
}

static int Mult ( int x, int y)
{
    return x * y;
}

static int Del ( int x, int y)
{
    return x / y;
}

static void Main( )
{
    // сконструюємо делегат
    IntOperation op1 = new IntOperation(Sum);

    int result = op1(5, 10);
    Console.WriteLine ( "Сума:\u0020" + result);

    // Змінимо посилання на метод
    op1 = new IntOperation(Mult);
    result = op1(5, 10);
    Console.WriteLine ( "Добуток:\u0020" + result);
    Console.ReadLine ();
}
}
```

Головний висновок з цього прикладу полягає в наступному: у той момент, коли відбувається звернення до примірника делегата `IntOperation`, викликається метод, на який він посилається. Отже, виклик методу дозволяється під час виконання, а не в процесі компіляції. Ще приклад:

```
class Program
{
    delegate void Message( ); // Оголошуємо делегат

    static void Main( string [ ] args )
    {
        Message mes; // Створюємо змінну типу делегат
        if (DateTime.Now.Hour <12)
        {
            mes = GoodMorning; // Надаємо цієї змінної адресу методу
        }
        else
        {
            mes = GoodEvening;
        }
        mes(); // Викликаємо метод
        Console.ReadKey();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good_Morning");
    }
}
```

```
private static void GoodEvening()
{
    Console.WriteLine("Good_Evening");
}
}
```

Виклик делегата проводиться подібно викликом методу. Делегат може служити для виклику будь-якого методу з відповідною сигнатурою і типом, що повертається. Метод, що викликається, може бути методом екземпляра або ж статичним методом. Значення має лише одне: сигнатура методу має бути узгоджена з сигнатурою, яка вказана в оголошенні делегата. Є ще один спосіб: створення екземпляра делегата за допомогою конструктора, в який передається потрібний метод:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string [ ] args )
    {
        Operation del = Add;
        Operation del2 = new Operation(Add);

        Console.Read();
    }
    private static int Add(int x, int y)
    {return x + y; }
}
```

Обидва способи рівноцінні. Як було написано вище, методи відповідають делегату, якщо вони мають один і той же тип, що повертається і один і той же набір параметрів. Наприклад, нехай у нас є делегат: `delegate void SomeDel ( int a, double b);` Цьому делегату відповідає, наприклад, такий метод:

```
void SomeMethod1(int g, double n)
{}
```

А такі методи НЕ відповідають:

```
int SomeMethod2(int g, double n) {}
void SomeMethod3(double n, int g) {}
void SomeMethod4(ref int g, double n) {}
void SomeMethod5(out int g, double n) {g = 6; }
```

У прикладах вище змінна делегата вказувала на один метод. Насправді ж делегат може вказувати на декілька методів відповідних. Можна утворити для делегата спеціальний список - список виклику або *invocation list* . І при виклику делегата всі методи з цього списку послідовно будуть викликатися:

```
class Program
{
    delegate void Message();

    static void Main(string [ ] args )
    {
        Message mes1 = Hello;
        mes1 += HowAreYou;
        // Тепер mes1 вказує на два методи
        mes1();
        // викликаються обидва методи - Hello і HowAreYou
        Console.ReadKey();
    }
    private static void Hello()
    {
        Console.WriteLine("Hello");
    }
    private static void HowAreYou()
    {
        Console.WriteLine("How_are_you?");
    }
}
```

У даному разі в список виклику делегата `mes1` додаються два методи - `Hello` і `HowAreYou`. І при зверненні в програмі `mes1` викликаються відразу обидва ці методи. Для додавання делегатів застосовується операція `+=`. Однак варто зазначити, що в реальності буде відбуватися створення нового екземпляра типу делегат, який отримає методи старої копії делегату; і новий метод, і новий екземпляр делегату буде присвоєно змінної `mes1`. При додаванні делегатів слід враховувати, що ми можемо додати посилання на один і той же метод кілька разів, і в списку виклику делегата тоді буде кілька посилань на один і той же метод.

```
Message mes1 = Hello;
mes1 += HowAreYou;
mes1 += Hello;
mes1 += Hello;
mes1();
```

Подібним чином ми можемо видаляти методи з делегата за допомогою операції `-=`:

```
static void Main(string [ ] args )
{
    Message mes1 = Hello;
    mes1 += HowAreYou;
    mes1 (); // викликаються всі методи з mes1
    mes1 -= HowAreYou; // Видаляємо метод HowAreYou
    mes1 (); // викликається метод Hello
    Console.ReadKey();
}
```

При видаленні методів фактично буде новий екземпляр, який в

списку виклику методів буде містити на один метод менше. Делегати можна об'єднувати в інші делегати, наприклад:

```
class Program
{
    delegate void Message();

    static void Main(string [ ] args )
    {
        Message mes1 = Hello;
        Message mes2 = HowAreYou ;
        Message mes3 = mes1 + mes2;
// об'єднуємо делегати
        mes3(); // викликаються всі методи з mes1 і mes2

        Console.ReadKey();
    }
}
```

Інший спосіб виклику делегата представляє метод Invoke() :

```
class Program
{
    delegate int Operation(int x, int y);
    delegate void Message();

    static void Main(string [ ] args )
    {
        Message mes = Hello;
        mes.Invoke();
        Operation op = Add;
        op.Invoke(3, 4);
        Console.ReadKey();
    }
    private static void Hello( )
    { Console.WriteLine ( "Hello"); }
    private static int Add(int x, int y)
    {return x + y; }
}
```

Якщо делегат приймає параметри, то в метод Invoke передаються значення/аргументи для цих параметрів. Слід враховувати, що якщо делегат порожній, тобто в його списку виклику немає посилань на методи (делегат дорівнює Null ), то при виклику такого делегата ми отримаємо виключення:

```
Operation op = Add;
op - = Add; // делегат op порожній
op(3, 4); //! Помилка: делегат дорівнює null
```

Тому при виклику делегата завжди краще перевіряти, чи не дорівнює він null. Або можна використовувати метод Invoke і оператор умовного null:

```
Operation op = Add;
op - = Add; // делегат op порожній
```

```
op?.Invoke(3, 4);  
// помилки немає, делегат просто не викликається
```

Також делегати можуть бути параметрами інших методів:

```
class Program  
{  
    delegate void GetMessage();  
  
    static void Main(string [ ] args )  
    {  
        if (DateTime.Now.Hour <12)  
        {  
            Show_Message(GoodMorning);  
        }  
        else  
        {  
            Show_Message(GoodEvening);  
        }  
        Console.ReadLine();  
    }  
    private static void Show_Message(GetMessage _del)  
    {  
        _Del?.Invoke();  
    }  
    private static void GoodMorning()  
    {  
        Console.WriteLine("Good_Morning");  
    }  
    private static void GoodEvening()  
    {  
        Console.WriteLine("Good_Evening_");  
    }  
}
```

Делегати можуть бути узагальненими, наприклад:

```
delegate T Operation <T, K > (K val);  
  
class Program  
{  
    static void Main(string [ ] args )  
    {  
        Operation <double, int > op = Square;  
  
        Console.WriteLine(op( 5));  
        Console.Read();  
    }  
  
    static double Square(int n)  
    {  
        return n * n;  
    }  
}
```

### 6.22.2 Анонімні методи

Розглянемо більш складний приклад. Нехай потрібно знайти суму двох чисел у другому степені. Для розв'язання цієї задачі будемо використовувати анонімні методи, що дозволяють передавати делегатові блок коду й уникнути при цьому створення окремого методу. Отже, доповнимо наш приклад, використовуючи анонімний метод для обчислення суми двох чисел у другому степені:

```
class Program
{
    public delegate int Mathdelegate(int x, int y); //делегат з двома
        аргументами                                     //типу int, який повертає int

    static void Main(string[] args)
    {
        //Стандартний спосіб
        Console.WriteLine("5+3={0}", Sum(5, 3));
        Console.WriteLine("5-3={0}", Sub(5, 3));
        Console.WriteLine();

        //Використання делегата
        Mathdelegate md = new Mathdelegate(Sum); // Передаємо делегатові посилання
            на метод Sum
        Console.WriteLine("5+3={0}", md(5, 3));
        md = Sub; //Тепер передаємо делегатові посилання на метод Sub
        Console.WriteLine("5-3={0}", md(5, 3));

        Console.WriteLine();
        //Використання анонімного методу
        //Передаємо делегатові блок коду, який і є анонімним методом

        md = delegate (int x, int y) { return ((x * x) + (y * y)); };
        Console.WriteLine("5*5+3*3={0}", md(5, 3));

        Console.ReadLine();
    }

    //Sums to numbers
    public static int Sum(int x, int y)
    {
        return x + y;
    }

    //Subtracts two numbers
    public static int Sub(int x, int y)
    {
        return x - y;
    }
}
```

Результат друку на екран:

```
5 + 3 = 8
5 - 3 = 2
```

$$5 + 3 = 8$$
$$5 - 3 = 2$$
$$5*5 + 3*3 = 34$$

Починаючи з C# 3.0, з'явилися лямбда-вирази, які замінюють анонімні методи. Додамо в нашому прикладі лямбда-вирази, функціонально ідентичні анонівному методу:

```
class Program
{
    public delegate int Mathdelegate(int x, int y); //
        Оголосимо делегат, що має два аргументи типу int, та й повертає
        int

    static void Main(string[] args)
    {
        //Стандартний спосіб
        Console.WriteLine("5+3={0}", Sum(5, 3));
        Console.WriteLine("5-3={0}", Sub(5, 3));

        Console.WriteLine();

        //Використання делегата
        Mathdelegate md = new Mathdelegate(Sum); //Передаємо
            делегатові посилання на метод Sum
        Console.WriteLine("5+3={0}", md(5, 3));
        md = Sub; //Тепер передаємо делегатові посилання на метод Sub
        Console.WriteLine("5-3={0}", md(5, 3));

        Console.WriteLine();

        //Використання анонімного методу
        md = delegate (int x, int y) { return ((x * x) + (y
            * y)); }; //Передаємо делегатові блок коду, який і є
            анонімним методом
        Console.WriteLine("5*5+3*3={0}", md(5, 3));

        Console.WriteLine();

        //Використання лямбда-вираження, замість анонімного методу
        md = (x, y) => ((x * x) + (y * y)); //Лямбда-вираз
        Console.WriteLine("5*5+3*3={0}", md(5, 3));

        Console.ReadLine();
    }

    //Sums to numbers
    public static int Sum(int x, int y)
    {
        return x + y;
    }

    // Sub of two numbers
    public static int Sub(int x, int y)
    {
        return x - y;
    }
}
```

```
}
```

Результат друку на екран:

```
5 + 3 = 8
5 - 3 = 2

5 + 3 = 8
5 - 3 = 2

5*5 + 3*3 = 34

5*5 + 3*3 = 34
```

### 6.22.3 Лямбда-вирази

Лямбда-вирази — це певний запис коротких функцій: спочатку вказується список аргументів, потім лямбда-оператор ( $\Rightarrow$ ), а після записуємо блок коду, який має виконатися. Схематично це виглядає так:

**список аргументів  $\Rightarrow$  блок коду**

Наприклад:

```
//(x, y) - список аргументів
//((x*x) + (y*y)) - що має виконувати функція
md = (x, y)  $\Rightarrow$  ((x*x) + (y*y));
```

Якщо аргументи відсутні, то в якості параметрів вказується пара порожніх дужок:

```
public delegate void Newdelegate();
Newdelegate del = ()  $\Rightarrow$  Console.WriteLine(DateTime.Now.
    ToString());
del(); //Виведе в консоль поточну дату й час
```

Для обробки декількох рядків, код в лямбда-виразі необхідно виділити за допомогою фігурних дужок:

```
del = ()  $\Rightarrow$ 
{
    Console.WriteLine("-----_Some_additional_code_-----");
    Console.WriteLine(DateTime.Now.ToString());
};
```

Типи аргументів можна вказувати, але це не обов'язково. Вони визначаються компілятором автоматично на основі сигнатури делегата, теж саме стосується й типу значення, що обчислюється й повертається в якості результату.

#### 6.22.4 Контрольні запитання

1. Що таке делегат?
2. Які особливості делегата?
3. Методи створення делегата.
4. Чи можна додавати та видаляти методи з делегата?
5. Що таке анонімний метод?
6. Наведіть приклад анонімного методу.
7. Що таке лямбда-вираз?
8. Наведіть приклад лямбда-виразу.

#### 6.23 Задачі до розділу КЛАСИ

1. Розробити програму, що реалізовує роботу зоопарку. Реалізувати класи зоопарк, тварина, різні тварини, персонал зоопарку, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
2. Розробити програму, що реалізовує роботу бібліотеки. Реалізувати класи бібліотека, каталог, книжка, персонал бібліотеки, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
3. Розробити програму, що реалізовує роботу університету. Реалізувати класи університет, факультет, кафедра, викладачі, студенти. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
4. Розробити програму, що реалізовує роботу лікарні. Реалізувати класи лікарня, відділення, лікарі, пацієнти, послуги. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.

5. Розробити програму, що реалізовує роботу аптечної мережі. Реалізувати класи аптечна мережа, аптека, каталог ліків, ліки, персонал аптеки, замовлення. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
6. Розробити програму, що реалізовує роботу галактика. Реалізувати класи галактика, зорі, планети, жителі планет, супутники планет. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
7. Розробити програму, що реалізовує роботу океан. Реалізувати класи океан, море, риби, водорості, ракоподібні. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
8. Розробити програму, що реалізовує роботу країна. Реалізувати класи країна, область, район, місто, населені пункти. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
9. Розробити програму, що реалізовує роботу супермаркет. Реалізувати класи супермаркет, відділи, товари, працівники, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
10. Розробити програму, що реалізовує роботу салон. Реалізувати класи салон краси, послуги, майстри, матеріали, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.

## 6.24 Взаємозв'язки та взаємодія класів

Нехай необхідно побудувати модель взаємозв'язків:

Університет – Факультет – Студент

за умов: студент може вчитися в декількох університетах, на декількох факультетах. Університет може мати кілька факультетів. При ліквідації університету, автоматично ліквідуються всі факультети. Студент продовжує існувати, через те, що він може вчитися в інших університетах. Після аналізу, одержуємо такі відношення між класами:

- університет складається з факультетів, тобто факультети є невід'ємною частиною (**part of**) університету;
- факультет має (**has a**) студентів, тобто студенти є самостійною сутністю й можуть належати іншим факультетам.

Прийнято вважати, що університети й факультети знаходяться у відношенні **композиція**, а факультети й студенти – у відношенні **агрегація**. Покажемо, як зв'язки між класами можна реалізувати мовою C#. Будемо вважати, що інформація про студентів надається екземплярами класу Student:

```
public class Student
{
    public string firstname { get; set; }
    public string lastname { get; set; }

    public Student(string firstname, string lastname)
    {
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public override string ToString()
    {
        return firstname + "_" + lastname;
    }
}
```

Інформація про факультети представляється екземплярами класу Faculty:

```
public class Faculty
{
    HashSet<Student> students = new HashSet<Student>();

    public void Enroll(Student p)
    {
        if (students.Contains(p))
            throw new ArgumentException(p + "_is_already_enrolled");
        students.Add(p);
    }

    public void Withdraw(Student p)
    {
        if (!students.Contains(p))
            throw new ArgumentException(p + "_is_not_enrolled");
    }
}
```

```
        students.Remove(p);
    }

    // вивести список студентів факультету на консоль (indent – відступ)
    public void ListEnrolled(string indent = "")
    {
        foreach (Student p in students)
        {
            Console.WriteLine(indent + p);
        }
    }
}
```

Інформація про університет представляється екземпляром класу University:

```
public class University
{
    public string Name { get; set; }
    Dictionary<string, Faculty> faculties = new Dictionary<string, Faculty>();

    public University(string name, string[] facultynames)
    {
        Name = name;
        // факультети створюються й утримуються всередині університету
        foreach (string facultyname in facultynames)
        {
            faculties.Add(facultyname, new Faculty());
        }
    }

    // індикатор для отримання об'єкту Факультет за назвою факультету
    public Faculty this[string name]
    {
        get
        {
            if (!faculties.ContainsKey(name))
                throw new ArgumentException("\"" + name + "\" : no such faculty");
            return faculties[name];
        }
    }

    // отримання переліку назв факультетів
    public IEnumerable<string> Getfacultynames()
    {
        return faculties.Keys.GetEnumerator();
    }

    // вивести перелік студентів за факультетами на консоль (indent – відступ)
    public void ListEnrolled(string indent = "")
    {
        Console.WriteLine(indent + Name);
        foreach (KeyValuePair<string, Faculty> p in faculties)
        {
            Console.WriteLine(indent + "\t" + p.Key);
            p.Value.ListEnrolled(indent + "\t\t");
        }
    }
}
```

```
    }  
  }  
}
```

Класи, що описані вище, можна використовувати так:

```
class Program  
{  
    static void Main()  
    {  
        University u = new University("My_University", new string  
            [] { "Faculty1", "Faculty2" });  
        Student p1 = new Student("Name1", "Surname1");  
        Student p2 = new Student("Name2", "Surname2");  
        Student p3 = new Student("Name3", "Surname3");  
        Student p4 = new Student("Name4", "Surname4");  
        u["Faculty1"].Enroll(p1);  
        u["Faculty2"].Enroll(p2);  
        u["Faculty2"].Enroll(p3);  
        u["Faculty2"].Enroll(p4);  
        u["Faculty2"].Withdraw(p4);  
        u["Faculty2"].Enroll(p4);  
        // вивести список студентів  
        u.ListEnrolled();  
    }  
}
```

У результаті на консолі буде надруковано:

```
My University  
  Faculty1  
    Name1 Surname1  
  Faculty2  
    Name2 Surname2  
    Name3 Surname3  
    Name4 Surname4
```

Усі види зв'язаності можна впорядкувати за ступенем спадання в такий спосіб:

1. Спадкування (найбільше жорстка зв'язаність).
2. Композиція.
3. Агрегування.
4. Асоціація.

Композиція – найпростіший механізм зв'язку для створення нового класу шляхом включення декількох екземплярів існуючих класів у новий клас (контейнер). При композиції включені екземпляри не можуть існувати окремо від контейнера, як було показано на прикладі університет – факультет.

```
class A  
{  
    ....  
}  
class B  
{
```

```
A a = new A();  
.....  
}
```

Агрегація — механізм зв'язку для створення нового класу із уже існуючих класів шляхом вказівки посилання на екземпляри інших класів. При агрегації обидва об'єкти можуть існувати незалежно. Наприклад, екземпляр створений зовні і в конструктор передається посилання на нього:

```
class A  
{  
.....  
}  
  
class B  
{  
    private A a;  
    public B(A a)  
{  
        this.a=a;  
    }  
}
```

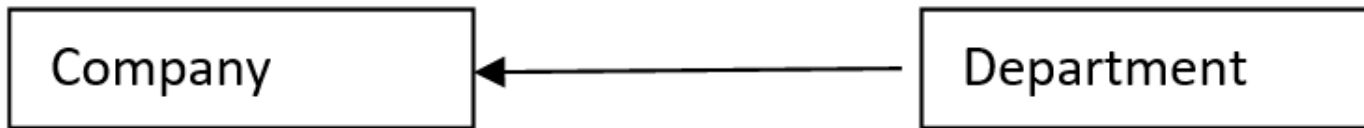
Причому агрегований екземпляр не стає частиною іншого екземпляра, як це відбувається у випадку з композицією. Крім зазначених відношень ще ми будемо застосовувати відношення узагальнення (спадкування, див. попередній матеріал) і асоціації (рівноправності). Узагальнення показує, що один із двох зв'язаних класів (підтип) є частиною іншого (надтипу), який називається узагальненням першого.

```
class A  
{  
protected static int Field=0;  
}  
class D : A  
{  
    private int Dfield = Field;  
}
```

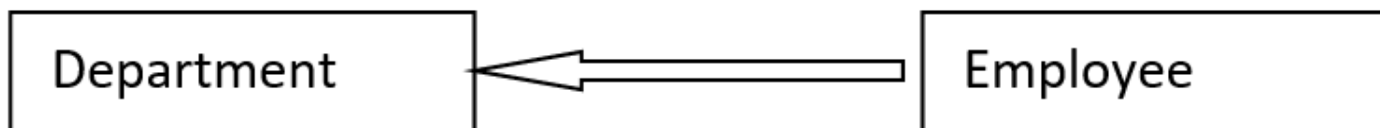
Асоціація показує, що екземпляри одного класу просто зв'язані з екземплярами іншого, так би мовити інформаційними зв'язками. Даний тип зв'язку найчастіше використовується при проєктуванні класів.

Розглянемо ще один приклад. Побудуємо модель ІТ компанії. Компанія складається з декількох відділів, у яких працюють співробітники. Співробітники можуть займати різні посади й бути приписаними до різних кабінетів. Для кожного співробітника необхідно зберігати список займаних посад (історію кар'єри) і їм видається перепустка із унікальним номером. Розглянемо зв'язки між

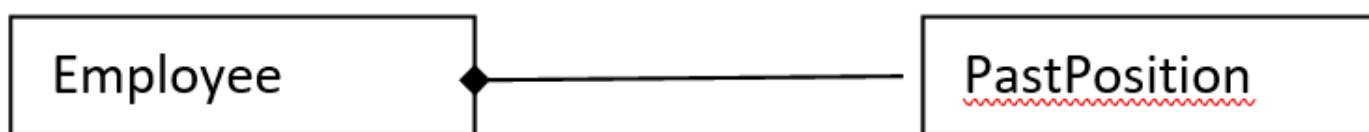
класами, які ми будемо проектувати. Отже, класи Компанія (Company) і Відділи (Department) перебувають у відношенні композиція, тому що компанія складається з відділів і відділи не можуть існувати без компанії:



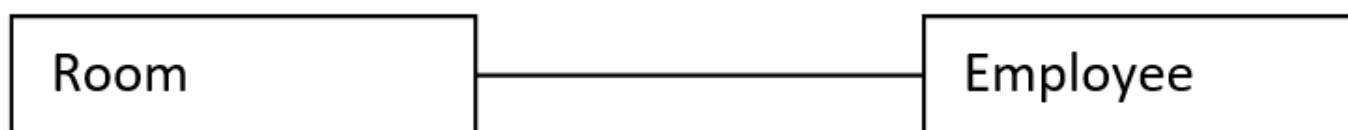
Відділи й Співробітники (Employee) перебувають у відношенні агрегація, тому що співробітники можуть існувати й при ліквідації відділу, їх можна перевести в інший відділ:



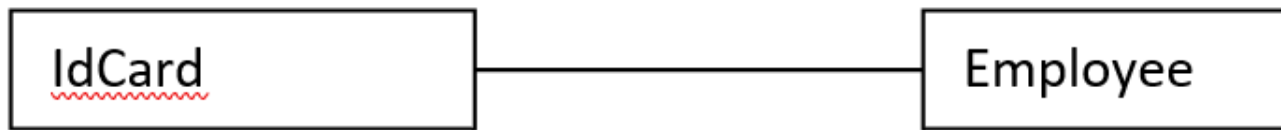
Класи Співробітники й Попередні Посади (Pastposition) перебувають у відношенні композиція, тому що при видаленні співробітника знищується весь його послужний список:



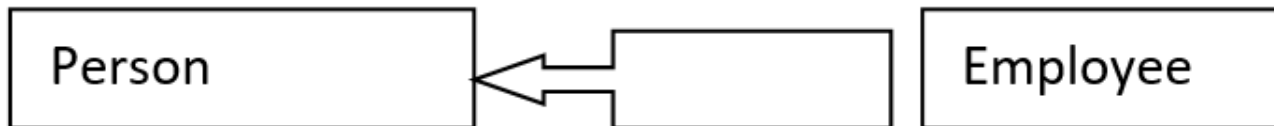
Співробітники й Кімнати (Room) перебувають у відношенні асоціація, їхній зв'язок рівноправний:



Аналогічно можна сказати, що у відношенні асоціація перебувають Співробітники й Перепустки (Idcard):



Уведемо, штучно, додатковий клас Людей (Person) для опису узагальнення співробітника, у якого є тільки Ім'я й Прізвище. Цей клас перебуває у відношенні узагальнення із класом Співробітник, тобто клас Співробітник розширює (спадкує) клас Людей.



Розглянемо клас Person. Він призначений для завдання Імені (name) і Прізвища (surname)

```
public class Person
{
    protected string name;
    protected string surname;
    public void Person(string newName, string newSurname)
    {
        name = newName;
        surname = newSurname;
    }
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name=value;
        }
    }
    public string Surname
    {
        get
        {
            return surname;
        }
    }
}
```

```
    }  
    set  
    {  
        surname = value;  
    }  
}  
}
```

Його розширенням є клас `Employee`, який дає вже більш повну інформацію про людину й тим самим описує співробітника. Цей клас успадковує дані із класу

```
Person:  
public class Employee : Person    // наследование  
{  
    private string position;  
    public Employee(string name, string surname, string position)  
        : base(name, surname)  
    {  
        this.position = position;  
    }  
}
```

Кожний співробітник працює в певному відділі, тобто відділ агрегує у собі співробітників.

```
public class Department  
{  
    private string name;  
    private HashSet<Employee> Employees = new HashSet<Employee>()  
        ; // для агрегации  
        сотрудников  
    public Department(string name)  
    {  
        this.name = name;  
    }  
}
```

Розберемося тепер із перепустками. Кожному співробітникові видана перепустка із унікальним ідентифікаційним номером:

```
public class Idcard  
{  
    private Datetime dateexpires;    // дата закінчення дії перепустку  
    private int number;              // номер перепустку  
    public Idcard(int number, Datetime dateexpires)  
    {  
        this.number = number;  
        this.dateexpires = dateexpires;  
    }  
    public bool isvalid() { return Datetime.Now.CompareTo(  
        dateexpires)>=0; }  
}
```

Аналогічно створимо клас для робочих приміщень співробітників – кабінетів:

```
public class Room
{
    private int number;    // номер кабінету
    public Room(int n)
    {
        number = n;
    }
}
```

І підготуємо набір значень для встановлення, надалі, асоціації з Employee

```
private HashSet <Room> idroom = new HashSet<Room>();
```

Як ми вже говорили, для відділу кадрів необхідно зберігати всі посади, які займав співробітник. Опишемо клас Pastposition:

```
public class PastPosition
{
    private string name;
    private Department department;
    public PastPosition(string position , Department
        department)
    {
        name = position;
        this.department = department;
    }
}
\Iвстановимокомпозиціюізкласом
\cskeyword{Employee}

\medskip\begin{lstlisting}
private HashSet <PastPosition> pastPosition = new HashSet<
    PastPosition>();...

public void setPastPosition(PastPosition p)
{
    pastPosition.Add(p);
}
```

Також опишемо методи для роботи із екземплярами класів. Остаточно програма прийме такий вигляд:

```
using System;
namespace Relations
{
    public class Person
    {
        protected string name;
        protected string surname;
        public Person(string newName, string newSurname)
        {
            name = newName;
            surname = newSurname;
        }
        public string Name
        {
            get
```

```
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    public string Surname
    {
        get
        {
            return surname;
        }
        set
        {
            surname = value;
        }
    }
}

// опис класу Employee
// наслідуюм клас Person
public class Employee : Person // спадкування
{
    private string position;
    private IdCard idCard;
    private HashSet<Room> idRoom = new HashSet<Room>();
    private Department department;
    private HashSet<PastPosition> pastPosition = new HashSet<
        PastPosition>();
    // создаем и конструктор
    public Employee(string name, string surname, string position)
        : base(name, surname)
    {
        this.position = position;
    }
    public string getPosition()
    {
        return position;
    }
    public void setIdCard(IdCard newcard)
    {
        idCard = newcard;
    }
    public IdCard getIdCard()
    {
        return idCard;
    }
    public void setRoom(Room newRoom)
    {
        idRoom.Add(newRoom);
    }
    public HashSet<Room> getRoom()
    {
        return idRoom;
    }
}
```

```
    }
    public void setDepartment(Department department)
    {
        this.department = department;
    }
    public Department getDepartment()
    {
        return department;
    }
    public void setPosition(String newPosition)
    {
        PastPosition pastPosition = new PastPosition(this.
            getPosition(), this.getDepartment());
        this.setPastPosition(pastPosition);
        position = newPosition;
    }
    public void setPastPosition(PastPosition p)
    {
        pastPosition.Add(p);
    }
    public HashSet<PastPosition> getPastPosition()
    {
        return pastPosition;
    }
    public void deletePastPosition(PastPosition pos)
    {
        pastPosition.Remove(pos);
    }
}
```

// опис класу Card

```
public class IdCard
{
    private DateTime dateExpires;
    private int number;
    public IdCard(int number, DateTime dateExpires)
    {
        this.number = number;
        this.dateExpires = dateExpires;
    }
    public int Number
    {
        set
        {
            number = value;
        }
        get
        {
            return number;
        }
    }
    public DateTime Date
    {
        get
        {
            return dateExpires;
        }
    }
}
```

```
        set
        {
            dateExpires = value;
        }
    }

    public bool isValid() { return DateTime.Now.CompareTo(
        dateExpires) >= 0; }
}

// опис класу Room

public class Room
{
    private int number;
    public Room(int n)
    {
        number = n;
    }
    public int Number
    {
        get
        {
            return number;
        }
        set
        {
            number = value;
        }
    }
}

// опис класу Department

public class Department
{
    private string name;           // назва відділу
    private HashSet<Employee> Employees = new HashSet<Employee>()
    ;
    public Department(string name)
    {
        this.name = name;
    }
    public string Name
    {
        set
        {
            name = value;
        }
        get
        {
            return name;
        }
    }

    public void addEmployee(Employee newEmployee)
    {
```

```
        Employees.Add(newEmployee);
        // связываем сотрудника с этим отделом - агрегация
        newEmployee.setDepartment(this);
    }
    public HashSet<Employee> getEmployees()
    {
        return Employees;
    }
    public void removeEmployee(Employee Empl)
    {
        Employees.Remove(Empl);
    }
}
// опис класу PastPosition

public class PastPosition
{
    private string name;
    private Department department;
    public PastPosition(string position , Department department)
    {
        name = position;
        this.department = department;
    }
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
    public void setDepartment(Department department)
    {
        this.department = department;
    }
    public Department getDepartment()
    {
        return department;
    }
}
}
```

```
public class Company
{
    public string Name { get; set; }
    Dictionary<string , Department> departments = new Dictionary<
        string , Department>();

    public Company(string name, string [] departmentNames)
    {
        Name = name;
        foreach (string departmentName in departmentNames)
```

```
        {
            departments.Add(departmentName, new Department(
                departmentName));
        }
    }

    public Department this[string name]
    {
        get
        {
            if (!departments.ContainsKey(name))
                throw new ArgumentException("\"\" + name + "\":_no_
                    _such_department");
            return departments[name];
        }
    }

    public IEnumerable<string> GetDepartmentNames()
    {
        return departments.Keys.GetEnumerator();
    }
}
```

### 6.24.1 Контрольні запитання

1. Які є види зв'язності класів?
2. Що таке спадкування?
3. Що таке композиція?
4. Що таке агрегування?
5. Що таке асоціація?

## 6.25 Задачі до розділу КЛАСИ

1. Розробити програму, що реалізовує роботу зоопарку. Реалізувати класи зоопарк, тварина, різні тварини, персонал зоопарку, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
2. Розробити програму, що реалізовує роботу бібліотеки. Реалізувати класи бібліотека, каталог, книжка, персонал бібліотеки, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.

3. Розробити програму, що реалізовує роботу університет. Реалізувати класи університет, факультет, кафедра, викладачі, студенти. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
4. Розробити програму, що реалізовує роботу лікарні. Реалізувати класи лікарня, відділення, лікарі, пацієнти, послуги. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
5. Розробити програму, що реалізовує роботу аптечної мережі. Реалізувати класи аптечна мережа, аптека, каталог ліків, ліки, персонал аптеки, замовлення. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
6. Розробити програму, що реалізовує роботу галактика. Реалізувати класи галактика, зорі, планети, жителі планет, супутники планет. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
7. Розробити програму, що реалізовує роботу океан. Реалізувати класи океан, море, риби, водорості, ракоподібні. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
8. Розробити програму, що реалізовує роботу країна. Реалізувати класи країна, область, район, місто, населені пункти. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
9. Розробити програму, що реалізовує роботу супермаркет. Реалізувати класи супермаркет, відділи, товари, працівники, відвідувачі. Передбачити взаємодію між об'єктами. Використати принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.
10. Розробити програму, що реалізовує роботу салон. Реалізувати класи салон краси, послуги, майстри, матеріали, відвідувачі. Передбачити взаємодію між об'єктами. Використати

принципи ООП, абстрактні класи, інтерфейси, делегати та лямбда-вирази.

# 7 Вправи

## 7.1 Вступ до програмування

**Введення і виведення даних через консоль.**

1. Написати програму, що читає з консолі введене користувачем ціле число та виводить на консоль число, вдвічі більше від введеного числа. Виконати програму покроково у середовищі Microsoft Visual Studio. При цьому переглянути значення змінних програми.

**Математичні операції та умовний оператор**

2. Написати програму для розв'язання рівняння вигляду  $ax = b$ , де  $a, b$  – цілі числа, що вводяться користувачем через консоль. Програма має вивести дійсне значення  $x$ , що задовольняє рівняння, якщо таке значення існує і єдине, або вивести повідомлення про те, що рівняння не має розв'язків, або має не один розв'язок.
3. Написати програму для розв'язання рівняння вигляду  $ax^2 + bx + c = 0$ , де  $a, b, c$  – цілі числа, що вводяться користувачем через консоль. Програма має вивести два дійсних значення  $x_1, x_2$ , що задовольняють рівня і  $x_1 < x_2$ , якщо такі значення існують і єдині, або вивести повідомлення про те, що рівняння не має дійсних розв'язків, або має один дійсний розв'язок, або має більше ніж два дійсних розв'язки.

Для обчислення квадратного кореня дійсного невід'ємного числа використати метод `Math.Sqrt` (метод стандартного класу `Math` для обчислення квадратного кореня), наприклад, рядок `Console.WriteLine(Math.Sqrt(4.0))` виводить значення 2.

4. Написати програму для обчислення значення виразу

$$\sin \left( \frac{x + y^2 + z^3}{1 + (x + y + z)^2} \right) \sqrt{|x + y + z|}$$

для заданих значень  $x, y, z$  типу `double`. Для обчислення функції `cos` скористатися методом `Math.Cos`.

5. Написати програму для обчислення площі рівнобедреної трапеції за її основами рівнобедреної трапеції і кутом при більшій основі.
6. Написати програму для обчислення суми цифр заданого натурального трицифрового числа.

## Оператори циклу

7. Написати консольну програму для моделювання руху матеріальної точки у замкненій прямокутній області на площині. Лівий нижній кут прямокутної області має координати  $(0,0)$ . Матеріальна точка переміщується з часом в середині області. Вона рухається по прямій з постійною швидкістю до зіткнення зі стінкою коробки, після чого пружньо відбивається від стінки і продовжує рух з тією ж швидкістю у відповідному напрямку до нового зіткнення зі стінками і т.д. Розміри області (ширина  $w$  та висота  $h$ ), початкове положення матеріальної точки (координати  $x_0 \in [0, w]$ ,  $y_0 \in [0, h]$ ), проєкції  $v_x$ ,  $v_y$  її початкового вектора швидкості на осі  $x$  та  $y$ , та час  $t_{max}$  (натуральне число), до якого програма моделює рух матеріальної точки вводяться через консоль користувачем програми. Програма має вивести послідовність рядків вигляду `t x y` де  $t=0,1,2,\dots,t_{max}$  – послідовні моменти часу (від початку руху), а  $x, y$  – координати положення матеріальної точки в час  $t$ . Одиниці виміру довжин та часу обрати на власний розсуд. Наприклад, програма може вивести:

```
0 1.0 1.0
1 1.1 1.5
2 1.2 2.0
```

що означає, що в початковий момент часу ( $t = 0$ ) координати матеріальної точки –  $(1, 1)$ , після чого в момент часу  $t = 1$  її координати –  $(1.1, 1.5)$ , а момент  $t = t_{max} = 2$  вона досягає точки  $(1.2, 3.0)$ .

8. Написати програму для обчислення суми цифр у десятковому записі заданого числа типу `ulong`. Число вводиться користувачем. Програма має вивести суму цифр на консоль.

- 9\*. Написати програму для наближеного обчислення значення квадратного кореня дійсного невід'ємного числа, введеного користувачем з консолі, не використовуючи методи класу Math (наприклад, Math.Sqrt), а користуючись операторами циклу, умовним оператором та арифметичними операціями над дійсними числами (+, -, \*, /).

## Інтерактивні програми

10. Створити програму для такого сценарію. Уявімо, що є стіл, на якому лежать предмети. Користувач розповідає програмі про предмети, що є на столі (наприклад, чашка, ложка і т.і.) і їх взаємне розташування (ложка знаходиться праворуч / ліворуч / перед /... від чашки) вводячи інформацію через консоль. При введенні інформації програма запам'ятовує її. Після введення необхідних відомостей, користувач може попросити програму схематично зобразити стіл на консолі. Наприклад, якщо користувач повідомив програмі, що на столі є чашка (Ч) і ложка (Л), і що ложка знаходиться праворуч від чашки, і після цього попросив програму вивести схему столу на екран, то програма має зобразити таку схему: Ч Л.

Взаємодію користувача з програмою організувати через вибір варіантів, наприклад:

**Програма:** Виберіть, що Ви хочете зробити  
(1 – розповісти про предмет на столі,  
2 – побачити схему столу,  
3 – вийти з програми):

**Користувач:** 1

**Програма:** Виберіть тип речення, яке Ви хочете мені сказати:

1 – я бачу на столі \_\_\_\_  
2 – предмет \_\_\_\_ знаходиться \_\_\_\_\_ від предмету \_\_\_\_\_

**Користувач:** 1

**Програма:** Повідомте мені, який предмет, Ви бачите на столі (1 - ложку (Л), 2 – чашку (Ч)):

**Користувач:** 1

**Програма:** ОК. Ви мені повідомили: “я бачу на столі ложку”. Виберіть, що Ви хочете зробити далі  
(1 – розповісти про предмет на столі,  
2 – побачити схему столу,  
3 – вийти з програми):

**Користувач:** 2

**Програма:** Схема столу:

Л

Виберіть, що Ви хочете зробити далі  
(1 – розповісти про предмет на столі,  
2 – побачити схему столу,  
3 – вийти з програми):

**Користувач:** 3

Деталі щодо сенсу відношень між предметами (наприклад, чи слово "ліворуч" означає "безпосередньо ліворуч" і т.п.) продумати самостійно.

## Масиви

11. Написати програму, що забезпечує введення з консолі послідовності з не більше 100 цілих чисел та знаходження найбільшого серед них. Зберігати введені числа в масиві.
12. Написати програму для знаходження трикутника з найбільшим можливим периметром, вершини якого належать заданій користувачем з консолі послідовності з попарно різних 10 точок на площині, заданих цілочисельними координатами  $x, y$ . Якщо такого трикутника не існує (усі точки розташовані на одній прямій), програма має вивести відповідне повідомлення.
13. Написати програму для транспонування прямокутної матриці цілих чисел розміру не більше  $50 \times 50$ . Розмір та елементи матриці вводяться через консоль, після чого транспонована матриця виводиться програмою на консоль. Деталі способу введення та форму виведення даних продумати самостійно.

Наприклад, при введенні матриці

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},$$

програма має вивести матрицю

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

- 14\*. Написати програму знаходження кількості зв'язних областей у чорно-білому зображенні розміру не більше  $20 \times 20$ , заданому двовимірним масивом чисел 0, 1, що вводиться користувачем через консоль.

Наприклад, при вводі зображення

```
0 0 1 0 0 0 0 0 0
0 1 1 1 0 0 0 0 0
1 1 1 1 1 0 1 1 1
0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0
```

програма має виводити 2, тобто зображення містить дві зв'язні області (трикутник ліворуч та прямокутник праворуч).

## Прості структури в C#

15. Написати програму для додавання і віднімання грошових сум. Програма повинна надати можливість користувачу програми ввести дві грошові суми, кожна з яких вводиться як два невід'ємних цілих числа – кількість гривень і кількість копійок в межах від 0 до 99, та повинна вивести суму цих двох грошових сум та різницю між більшою з них і меншою (або 0 грн 0 коп, якщо вони однакові) у вигляді двох пар невід'ємних цілих чисел – кількості гривень і кількості копійок в межах від 0 до 99.

Наприклад, якщо ввести суми 1 грн 20 коп та 2 грн 10 коп, програма має вивести у якості суми цих двох грошових сум 3 грн 30 коп, а у якості різниці між більшою і меншою сумою – 0 грн 90 коп.

У програмі подати грошові суми екземплярами структури Money, наведеної нижче, з двома цілочисельними полями – grn (кількість гривень) і kop (кількість копійок в межах від 0 до 99).

```
public struct Money
{
    public uint grn;
    public byte kop;
}
```

16. Написати програму для визначення типу трикутника (прямокутний, рівнобедрений, і т.і.) заданого координатами  $x, y$  його вершин на площині. Координати трьох вершин вводяться користувачем з консолі у вигляді пар цілих чисел. Програма має

вивести на консоль тип трикутника (напр. “Прямокутний”), або повідомлення про те, що введені точки не утворюють трикутник (лежать на одній прямій).

У програмі подати введені користувачем точки екземплярами структури Point, наведеної нижче:

```
public struct Point
{
    public int x;
    public int y;
}
```

- 17\*. Написати програму для визначення, чи задана послідовність з не більше 100 точок (подана парами цілих чисел – координат  $x, y$  на площині) є переліком вершин простого багатокутника (без самоперетинів). Програма має надавати користувачу можливість ввести точки з консолі і має виводити на консоль повідомлення Так чи Ні. У програмі подати введені точки масивом, елементами якого є екземпляри структури з двома полями (координатами точки).

### Символи і рядки

18. Написати програму для обчислення арифметичного виразу вигляду число операція число, де число є натуральним числом, операція – одна з  $+$ ,  $-$ ,  $*$ ,  $/$ . Вираз вводиться через консоль користувачем. Числа від операції відділені однією або більше прогалів. Програма має вивести значення виразу.

## 7.2 Алгоритми

19. Написати програму, яка за заданими користувачем цілим числом  $n$  в діапазоні  $1 \dots 20$  та числом  $a \in \{2, 3\}$  обчислює  $a^n$  (не користуючись методами класу Math), при цьому виконуючи не більше  $2 \log(n)$  операцій множення.

Наприклад,  $a^7$  можна обчислити, виконавши  $4 < 2 \log_2(7)$  операції множення:

```
b = a * a
c = b * b
result = a * b * c
```

20. Написати програму, яка за заданим масивом  $A$  з  $N$  дійсних чисел та дійсним числом  $S$  визначає, чи існують натуральні числа  $i, j$  в діапазоні  $1 \dots N$ , такі, що  $A[i] + A[j] = S$ .

21. Інверсією в масиві  $A$  з  $N$  цілих чисел назвемо таку пару натуральних чисел  $(i, j)$ , що  $1 \leq i < j \leq N$  та  $A[i] > A[j]$ . Написати програму, яка визначає кількість інверсій в заданому масиві  $A$  з  $N$  цілих чисел.
22. Задано масив  $A$  зі 101 натурального числа в діапазоні  $1 \dots 200$  та натуральне число  $X$  в діапазоні  $1 \dots 200$ . Написати програму, яка обчислює остачу від ділення числа  $A[100]X^{100} + A[99]X^{99} + \dots + A[1]X + A[0]$  на 997, використовуючи схему Горнера. Визначити кількість операцій додавання та множення, які виконує програма.
23. Написати програму для обчислення цілої частини максимального значення, яке приймає многочлен  $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  заданий масивом коефіцієнтів  $(a_n, \dots, a_0)$ , на дійсному проміжку  $[0, 1]$ , при умові, що коефіцієнти  $a_n, \dots, a_0$  є цілими числами діапазоні  $-100 \dots 100$ .
24. Написати програму для переведення натурального числа, заданого масивом цифр у двійковій системі числення в десяткову систему. Кількість цифр в двійковому записі вхідного числа не перевищує 10000.
25. Написати програму для переведення натурального числа, заданого масивом цифр у десятковій системі числення в двійкову систему. Кількість цифр в десятковому записі вхідного числа не перевищує 10000.
26. Написати програму, яка додає два натуральні числа, задані в двійковій системі числення масивами цифр. Кількість цифр в двійковому записі кожного з вхідних чисел не перевищує 10000.
27. Написати програму, яка знаходить добуток двох натуральних чисел, заданих в двійковій системі числення масивами цифр. Кількість цифр в двійковому записі кожного з вхідних чисел не перевищує 10000.
28. Написати програму для обчислення цілої частини та остачі від ділення невід'ємного числа  $M$  в діапазоні  $1 \dots 10^{10000} - 1$ , заданого десятковим записом у вигляді рядка з не більше ніж 10000 цифр на ціле число в діапазоні від  $1 \dots 2^{30}$ .

29. Написати програму для виведення подання заданого цілого числа  $N$  в діапазоні  $1...2^{30}$  у вигляді добутку простих чисел.
30. Написати програму для знаходження найбільшого спільного дільника та найменшого спільного кратного двох цілих чисел в діапазоні  $1...2^{30}$ . Скористатись алгоритмом Евкліда.
31. Написати програму для виведення усіх простих чисел в межах від 1 до заданого числа  $N$  в діапазоні  $1...2^{30}$ .
32. Написати програму для обчислення  $n$ -го члена ( $F_n$ ) послідовності Фібоначчі, визначеної таким чином:  $F_0 = 1$ ,  $F_1 = 1$ ,  $F_n = F_{n-1} + F_{n-2}$ ,  $n \geq 2$ . Число  $n$  в діапазоні  $1...100$  вводиться у якості вхідних даних програми. Програма має вивести число  $F_n$ .
33. Написати програму для сортування за неспаданням заданої користувачем послідовності з  $N$  ( $2 \leq N \leq 100$ ) цілих чисел. Програма отримує на вхід число  $N$  та  $N$  цілих чисел та виводить на консоль вказані числа, впорядковані за неспаданням.
34. Написати програму для виводу усіх способів подання заданого натурального числа  $N$  в діапазоні  $1...2^{30}$  у вигляді суми неспадаючої послідовності натуральних чисел.
35. Написати програму, яка за двома заданими рядками довжини не більше 25, що складаються з символів A-Z, визначає кількість способів, якими можна отримати перший рядок з другого шляхом викреслення деяких символів. Наприклад, якщо задано рядки ab та abcaб, то перший рядок можна отримати з другого двома способами: або викресленням перших трьох символів, або останніх трьох символів. Якщо задано рядки abcaб та ab, то другий рядок не можна отримати з першого шляхом викреслення символів, тому в такому випадку відповідь програми має бути нуль.
36. Написати програму для визначення кількості способів, якими можна подати задане ціле число  $N$  у вигляді суми строго зростаючої послідовності натуральних чисел.
37. Написати програму, яка для заданого користувачем рядка з не більше ніж 6 символів виводить усі рядки, що можуть бути отримані із заданого рядка шляхом перестановки деяких символів. Наприклад, якщо користувач задав програмі рядок

abc, то програма має вивести такі рядки (у довільній послідовності): abc, acb, bac, bca, cab, cba.

38. Написати програму, яка за заданими натуральними числами  $n$ ,  $x$  та  $a_1, a_2, \dots, a_n$  знаходить один з способів розстановки знаків “+” або “-” між числами послідовності  $a_1, a_2, \dots, a_n$ , так, щоб значення отриманого виразу було рівне  $x$  (якщо такий спосіб існує). Якщо вказаного способу розстановки знаків не існує, програма має виводити відповідне повідомлення.
39. Написати програму обчислення площі простого  $n$ -кутника ( $n \leq 100$ ) на площині, заданого послідовністю  $n$  пар цілих чисел – координат  $x, y$  своїх вершин у порядку одного з варіантів їх обходу.
40. Написати програму, яка за заданим користувачем числом  $N$  в межах  $1 \dots 2^{30}$  знаходить кількість різних прямокутників з цілочисельними довжинами сторін, які мають площу  $N$ .
41. Написати програму, яка за заданою користувачем квадратною таблицею розміру  $N \times N$  ( $N$  в діапазоні  $2 \dots 20$ ), у клітинках якої знаходяться цифри 0-9, знаходить і виводить шлях з лівої нижньої у праву верхню клітинку таблиці, який є послідовністю клітинок, в якій кожна наступна клітинка знаходиться або безпосередньо над, або безпосередньо праворуч від попередньої, для якого сума цифр у клітинках вздовж шляху є мінімально можливою (якщо таких шляхів більше одного, програмі достатньо знайти лише один з них). Наприклад, якщо користувач вводить таблицю

```
0 0 1
1 2 1
0 0 1
```

то програма може вивести знайдений шлях у такому вигляді (зірочками відмічено клітини, що належать шляху):

```
* * *
*
*
```

При цьому сума цифр у клітинках такого шляху є 2 і є мінімально можливою серед усіх шляхів з лівої нижньої у праву верхню клітинку таблиці, в яких кожна наступна клітинка знаходиться або безпосередньо над, або безпосередньо праворуч від попередньої.

42. Написати програму, яка визначає, чи можна отримати заданий користувачем рядок, що складається з символів [ , ], ( , ) шляхом виконання послідовності таких дій:

- а) заключення порожнього або одного з вже отриманих рядків у круглі, квадратні, або фігурні дужки;
- б) зчеплення двох вже отриманих рядків (наприклад, з ( ) і [] отримується ( )[]).

Програма має виводити повідомлення Так або Ні. Наприклад, рядок [ ]( ) [ ] [ ] можна отримати виконанням дій а) та б): можна отримати ( ) та [] за допомогою операції а), з них за допомогою б) можна отримати [ ]( ), звідки за допомогою операції а) можна отримати [ ]( ) [ ] [ ], а з [ ]( ) [ ] [ ] і [] можна отримати [ ]( ) [ ] [ ] за допомогою операції б). Тому для вхідного рядка [ ]( ) [ ] [ ] програма має вивести Так. Але, наприклад, для рядка ( ) програма має вивести Ні.

## 7.3 Комплексна вправа до розділів 1-6

Дана консольна програма на мові C# (див. нижче), яка при запуску запитує ім'я користувача (англ. user name) та кодове слово (англ. code word). У залежності від введеного імені користувача та кодового слова, програма може вивести деякий рядок символів на консоль і успішно завершитися, або аварійно завершитися, або “зависнути”.

### Завдання:

- 1) Ввести, скомпілювати та запустити наведену нижче програму на комп'ютері.
- 2) Підібрати таке кодове слово, щоб при вводі імені користувача Peter та цього кодового слова наведена нижче програма виводила на консоль повідомлення ОК та успішно завершувалась.
- 3) Написати програму на мові C# , що дозволить для будь-якого заданого імені користувача, що складається з не більше ніж 5 алфавітно-цифрових символів (0-9, a-z, A-Z) знайти одне з можливих кодових слів, таке, щоб наведена нижче програма при вводі цього імені користувача та кодового слова виводила повідомлення ОК та успішно завершувалась.

```
using System;

class Program
{
    public struct Proposition
    {
        public bool IsAssertion;
        public string Operation;
        public int Arg1;
        public int Arg2;
    }

    static void Main()
    {
        Console.WriteLine("User_name:_");
        string name = Console.ReadLine();
        Console.WriteLine("Enter_code_word:_");
        string codeword = Console.ReadLine();
        Console.WriteLine();
        int i = 0; byte c1 = 0, c2 = 0;
        for (i = 0; i < name.Length; i++)
            c1 += (byte)name[i];
        for (i = 0; i < codeword.Length; i++)
            c2 += (byte)codeword[i];
        if (c1 != c2)
            Console.WriteLine("Wrong_code_word");
        else
        {
            codeword = "bakbbkbckbbbambcbbbmbcbambdtbetbbjbfbu" +
                codeword;
            byte[] s = new byte[256];
            byte top = (byte)(s.Length - 1);
            s[--top] = 0;
            Proposition[] proof = new Proposition[256];
            bool cheat_mode = true;
            int pc = 0, goal = -1;
            for (i = 0; i < codeword.Length; i++)
            {
                int code = codeword[i] - 'a';
                if (!cheat_mode)
                {
                    if ((codeword[i] >= '0' && codeword[i] <= '9') ||
                        (codeword[i] >= 'A' && codeword[i] <= 'Z'))
                        code = codeword[i] ^ c1;
                    else if (((code ^ c1) >= '0' && (code ^ c1) <=
                        '9') || ((code ^ c1) >= 'A' && (code ^ c1) <=
                        'Z'))
                        break;
                }
                int A, B; if (i == 0) cheat_mode = true;
                switch (code)
                {
                    case 0: i = codeword.Length; break;
                    case 1: byte val = (byte)(codeword[++i] - 'a');
                        if (!cheat_mode)
                        {
                            if ((codeword[i] >= '0' && codeword[i] <=
                                '9') || (codeword[i] >= 'A' &&
```

```

        codeword[i] <= 'Z'))
            val = (byte)(codeword[i] ^ c1);
    }
    s[--top] = val;
    break;
case 2: top++; break;
case 3: s[++top] += s[top - 1]; break;
case 4: s[++top] -= s[top - 1]; break;
case 5: s[++top] *= s[top - 1]; break;
case 6: s[++top] /= s[top - 1]; break;
case 7: s[--top] = s[top + 1]; break;
case 8:
    if (goal < 0 || proof[goal].IsAssertion)
        Console.WriteLine((char)(s[top++] + 'A'));
    break;
case 9:
    int target = s[top++], value = s[top++];
    if (target > 127) target = target - 256;
    if (value != 0)
        i += target;
    break;
case 10:
    proof[pc].IsAssertion = false;
    proof[pc].Operation = s[top++].ToString();
    pc++;
    break;
case 11:
    proof[pc].IsAssertion = false;
    proof[pc].Operation = "NOT";
    proof[pc].Arg1 = s[top++];
    pc++;
    break;
case 12:
    proof[pc].IsAssertion = false;
    proof[pc].Operation = "IMPLICATION";
    proof[pc].Arg1 = s[top++];
    proof[pc].Arg2 = s[top++];
    pc++;
    break;
case 13:
case 14:
    A = s[top++]; B = s[top++];
    int[] x = new int[256], y = new int[256];
    x[0] = A; y[0] = B;
    int count = 1;
    while (count > 0)
    {
        count--;
        if (proof[x[count]].Operation != proof[y[
            count]].Operation)
        {
            count = -1;
            break;
        }
        else if (proof[x[count]].Operation == "
            NOT")
        {
            x[count] = proof[x[count]].Arg1; y[

```

```

        count] = proof[y[count]].Arg1;
        count++;
    }
    else if (proof[x[count]].Operation == "
        IMPLICATION")
    {
        int xc = x[count], yc = y[count];
        x[count] = proof[xc].Arg1; y[count] =
            proof[yc].Arg1; count++;
        x[count] = proof[xc].Arg2; y[count] =
            proof[yc].Arg2; count++;
    }
}
if (count == 0)
{
    if (code == 13)
    {
        proof[pc].IsAssertion = true;
        proof[pc].Operation = "IMPLICATION";
        proof[pc].Arg1 = A;
        proof[pc].Arg2 = B;
        pc++;
    }
    else if (proof[A].IsAssertion || proof[B
        ].IsAssertion)
    {
        proof[A].IsAssertion = true;
        proof[B].IsAssertion = true;
    }
}
break;
case 15: ;
A = s[top++]; B = s[top++];
if (proof[B].Operation == "IMPLICATION" &&
    proof[B].Arg2 == A)
{
    proof[pc].IsAssertion = true;
    proof[pc].Operation = "IMPLICATION";
    proof[pc].Arg1 = A;
    proof[pc].Arg2 = B;
    pc++;
}
break;
case 16:
A = s[top++]; B = s[top++];
if (proof[A].Operation == "IMPLICATION" &&
    proof[B].Operation == "IMPLICATION" &&
    proof[proof[A].Arg2].Operation == "
        IMPLICATION" &&
    proof[proof[B].Arg1].Operation == "
        IMPLICATION" &&
    proof[proof[B].Arg1].Arg1 == proof[A].
        Arg1 &&
    proof[proof[B].Arg1].Arg2 == proof[proof[
        A].Arg2].Arg1 &&
    proof[proof[B].Arg2].Operation == "
        IMPLICATION" &&
    proof[proof[B].Arg2].Arg1 == proof[A].

```



## 7.4 Вправи з об'єктно-орієнтованого програмування

1. Написати програму, яка для  $n=2,3,\dots,100$  забезпечує наближене виконання таких дій над векторами  $n$ -вимірному Евклідового простору, що подаються  $n$ -ками дійсних чисел – координат у деякому фіксованому ортонормованому базисі: додавання двох векторів, множення вектора на скаляр, обчислення норми вектора, розклад вектора у лінійну комбінацію двох векторів (якщо такий розклад існує). Спосіб введення та виведення даних обрати на власний розсуд.
2. Написати програму, яка для  $n=2,3,\dots,100$  забезпечує наближене виконання таких дій над векторами  $n$ -вимірному векторного простору над полем дійсних чисел, що подаються  $n$ -ками елементів поля – координат у деякому фіксованому базисі: додавання двох векторів, множення вектора на скаляр, віднімання двох векторів, розклад вектора у лінійну комбінацію двох векторів (якщо такий розклад існує). Спосіб введення та виведення даних обрати на власний розсуд.
3. Написати програму, яка для  $n=2,3,\dots,100$  забезпечує наближене виконання таких дій над векторами  $n$ -вимірному векторного простору над полем комплексних чисел, що подаються  $n$ -ками елементів поля – координат у деякому фіксованому базисі: додавання двох векторів, множення вектора на скаляр, віднімання двох векторів, обчислення лінійної комбінації векторів. Спосіб введення та виведення даних обрати на власний розсуд.
4. Написати програму, яка забезпечує наближене виконання таких дій над прямокутними матрицями розміру не більше  $100 \times 100$  з цілими елементами та з дійсними елементами: додавання двох матриць, добуток двох матриць, віднімання двох матриць, транспонування матриці. Спосіб введення та виведення даних обрати на власний розсуд.
5. Написати програму, яка забезпечує наближене виконання таких дій над прямокутними матрицями розміру не більше  $100 \times 100$  з комплексними елементами: додавання двох матриць, добуток двох матриць, віднімання двох матриць, Ермітове спряження матриці (тобто транспонування матриці із насту-

пною заміною кожного елемента результату на комплексно-спряжений). Спосіб введення та виведення даних обрати на власний розсуд.

6. Написати програму, яка забезпечує наближене виконання таких дій над многочленами степеня не більше 100 над полем комплексних чисел, що подаються послідовностями своїх коефіцієнтів: додавання двох многочленів, множення двох многочленів, віднімання двох многочленів, обчислення многочлена в точці. Спосіб введення та виведення даних обрати на власний розсуд.
7. Написати програму, яка забезпечує наближене виконання таких дій над раціональними функціями – відношеннями двох многочленів, кожен з яких має степінь не більше 100, над полем дійсних чисел: додавання двох раціональних функцій, множення двох раціональних функцій, обчислення раціональної функції в точці. Спосіб введення та виведення даних обрати на власний розсуд.
8. Написати програму, яка забезпечує наближене виконання таких дій над раціональними функціями – відношеннями двох многочленів, кожен з яких має степінь не більше 100, над полем комплексних чисел: додавання двох раціональних функцій, віднімання двох раціональних функцій, множення двох раціональних функцій. Спосіб введення та виведення даних обрати на власний розсуд.
9. Написати програму, яка забезпечує перевірку рівності двох булевих функцій  $n$  ( $n=2,3,\dots,8$ ) аргументів, заданих булевыми виразами, що складаються з булевих констант і змінних та операцій диз'юнкції, кон'юнкції та заперечення, на всіх можливих значеннях своїх аргументів. Спосіб введення та виведення даних обрати на власний розсуд.
10. Написати програму, яка забезпечує перевірку рівності двох булевих функцій  $n$  ( $n=2,3,\dots,8$ ) аргументів, заданих виразами, що складаються з булевих констант і змінних та операцій імплікації та диз'юнкції, на всіх можливих значеннях своїх аргументів. Спосіб введення та виведення даних обрати на власний розсуд.
11. Написати програму, яка забезпечує перевірку рівності двох

- булевих функцій  $n$  ( $n=2,3,\dots,8$ ) аргументів, заданих виразами, що складаються з булевих констант і змінних та операцій імплікації та додавання за модулем 2, на всіх можливих значеннях своїх аргументів. Спосіб введення та виведення даних обрати на власний розсуд.
12. Написати програму, яка забезпечує перевірку рівності двох булевих функцій  $n$  ( $n=2,3,\dots,8$ ) аргументів, заданих поліномами Жегалкіна, на всіх можливих значеннях своїх аргументів. Спосіб введення та виведення даних обрати на власний розсуд.
  13. Написати програму, яка забезпечує перевірку, чи приймає булева функція  $n$  ( $n=2,3,\dots,8$ ) аргументів, задана булевим виразом, що складається з булевих констант і змінних та операцій диз'юнкції, кон'юнкції та заперечення, значення 0 на всіх можливих значеннях своїх аргументів. Спосіб введення та виведення даних обрати на власний розсуд.
  14. Написати програму, яка забезпечує перевірку, чи приймає булева функція  $n$  ( $n=2,3,\dots,8$ ) аргументів, задана поліномом Жегалкіна значення 0 на всіх можливих значеннях своїх аргументів. Спосіб введення та виведення даних обрати на власний розсуд.
  15. Написати програму, яка виводить таблицю істинності для даної булевої функції  $n$  ( $n=2,3,\dots,8$ ) аргументів, заданої булевим виразом, що може включати операції диз'юнкції, кон'юнкції та заперечення. Спосіб введення та виведення даних обрати на власний розсуд.
  16. Написати програму, яка виводить таблицю істинності для даної булевої функції  $n$  ( $n=2,3,\dots,8$ ) аргументів, заданої поліномом Жегалкіна. Спосіб введення та виведення даних обрати на власний розсуд.
  17. Написати програму, яка для  $n=2,3,\dots,8$  та будь-якої заданої таблично булевої функції  $n$  аргументів буде один з можливих булевих виразів. Допустимими операціями у булевому виразі є диз'юнкція, кон'юнкція та заперечення. Спосіб введення та виведення даних обрати на власний розсуд.
  18. Розробити просту мову програмування, яка надає програмісту можливість виконання присвоєння цілочисельних кон-

стант змінним, порівняння змінних між собою ( $<$ ,  $>$ ), інкремент змінної, декремент змінної, послідовне виконання кількох присвоєнь,

умовний оператор та циклічний оператор, вивід значення змінної на екран. Написати інтерпретатор розробленої мови. Синтаксис мови визначити на власний розсуд. Наприклад, можна використати синтаксис такого вигляду (вважаємо, що синтаксис мови вимагає, щоб кожна інструкція мови записувалась на окремому рядку і кількість прогалів на початку рядка відповідала рівню вкладеності інструкції):

```
x = 7
y = 0
z = 0
a = 5
while x < a
  x++
  z--
  if x > z
    y = 1
  endif
endwhile
print y
```

## 7.5 Комплексна вправа до розділу 7

Дана консольна програма на C# (див. нижче), яка дозволяє обчислювати значення арифметичних виразів, що можуть включати цілі числа, унарну операцію мінус ( $-$ ), бінарні операції додавання ( $+$ ), віднімання ( $-$ ), множення ( $*$ ), ділення ( $/$ ), та круглі дужки. Наприклад, якщо у якості вхідних даних програми ввести  $1 + (2 * 3 - 10 / 2)$ , програма виводить 2 в якості результату. Операція ділення розуміється як ціла частина від ділення двох цілих чисел. Пропуски у виразі між числами, дужками та знаками операцій ігноруються.

### Завдання:

- 1) Ввести, скопіювати та запустити програму на комп'ютері. Проаналізувати вихідний код програми. Визначити призначення кожного класу, поля та методу. Задokumentувати кожний клас та метод. Визначити, чи є випадки, коли програма неправильно обчислює значення вхідного виразу. Якщо так, запропонувати спосіб модифікації коду програми для усунення таких випадків.
- 2) Додати можливість обчислення арифметичних виразів над дійсними числами, наприклад,  $(1.0 + 2.5) * 3$ .

- 3) Додати можливість обчислення арифметичних виразів, що включають дійсні числа та константи  $e$  та  $\pi$ , наприклад,  $(1.0 + 2.5)*\pi + e$ .
- 4) Додати можливість обчислення виразів над дійсними числами, що можуть включати арифметичні операції та тригонометричні функції  $\sin$ ,  $\cos$ ,  $\tan$ , наприклад,  $(1.5 + \sin(2.0 + \cos(3.5)))*3$ .
- 5) У класі `Expression` створити метод, який дозволяє визначити змінну (наприклад,  $x$ ) та надати їй ціле числове значення. Додати можливість обчислення арифметичних виразів над цілими числами, що включають як числа, так і змінні, при умові, що змінним завчасно надані цілі числові значення за допомогою зазначеного методу. Наприклад,  $(2 + x / y)*3$  при умові, що значення змінних  $x$  та  $y$  завчасно визначені за допомогою зазначеного методу.
- 6) Додати можливість обчислення булевих виразів, що включають булеві константи  $(0, 1)$ , операції диз'юнкції ( $\vee$ ), кон'юнкції ( $\wedge$ ), заперечення ( $!$ ) та дужки, наприклад,  $(0 \vee ((1))) \wedge 1$ .
- 7) Написати обґрунтовані критичні зауваження (при наявності) щодо структури та/або алгоритму роботи програми та шляхи вдосконалення програми, які Ви бачите.

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

namespace Expressions
{
    public abstract class Expression
    {
        public interface IInput : IEnumerator<char>
        {
            bool IsEmpty();
            int Position { get; }
        }

        public class StringInput : IInput
        {
            private string s;
            private int position;

            public int Position { get { return position; } }

            public static explicit operator StringInput(string s)
```

```
{
    return new StringInput(s);
}

public StringInput(string s)
{
    if (s == null)
        throw new ArgumentNullException();
    this.s = s;
    this.position = -1;
}

public bool IsEmpty()
{
    return (position >= s.Length);
}

object IEnumerator.Current {
    get { return s[Math.Max(0, position)]; }
}

public bool MoveNext()
{
    if (position + 1 >= s.Length)
    {
        position = s.Length;
        return false;
    }
    else
    {
        position++;
        return true;
    }
}

void IEnumerator.Reset() { }

public char Current {
    get { return s[Math.Max(0, position)]; }
}

public void Dispose() { }
}

public class ExpressionSyntaxError : Exception
{
    public enum ErrorReason {
        MissingSymbol,
        UnexpectedToken,
        EmptyParentheses
    }

    private static Dictionary<ErrorReason, string> messages =
        new Dictionary<ErrorReason, string>();

    private ErrorReason reason;
    private readonly int where;
```

```
static ExpressionSyntaxError()
{
    messages.Add(
        ErrorReason.MissingSymbol,
        "Missing_{0}"
    );
    messages.Add(
        ErrorReason.UnexpectedToken,
        "Unexpected_token"
    );
    messages.Add(
        ErrorReason.EmptyParentheses,
        "Empty_parentheses"
    );
}

public ErrorReason Reason {
    get { return reason; }
}

public int Where {
    get { return Math.Max(where, 0); }
}

public ExpressionSyntaxError(ErrorReason reason, int
    where, params object[] args)
    : base(string.Format(messages[reason], args))
{
    this.reason = reason;
    this.where = where;
}

}

public static class Symbols
{
    public const char Space = ' ';
    public const char Tab = '\t';
    public const char OpenBrace = '(';
    public const char CloseBrace = ')';

    public static bool IsWhite(char c)
    {
        return (c == Tab || c == Space);
    }

    public static bool IsReserved(char c)
    {
        return (IsWhite(c) || c == OpenBrace || c ==
            CloseBrace || char.IsDigit(c));
    }
}

internal interface IToken { }

public abstract class Operation : IToken
{
    public abstract char Symbol { get; }
    public abstract long Eval(params long[] args);
}
```

```
internal bool TryReadSymbol(IInput input)
{
    if (!input.IsEmpty() && input.Current == Symbol)
    {
        input.MoveNext();
        return true;
    }
    else
        return false;
}

}

public sealed class UnaryOp : Operation
{
    private readonly char symbol;
    private readonly Func<long, long> f;

    public UnaryOp(char symbol, Func<long, long> f)
    {
        this.symbol = symbol;
        this.f = f;
    }

    public override char Symbol { get { return symbol; } }

    public override long Eval(params long[] args)
    {
        if (args.Length != 1)
            throw new ArgumentException("Invalid_number_of_arguments_of_an_unary_operation");
        return f(args[0]);
    }
}

public sealed class BinaryOp : Operation
{
    private readonly int priority;
    private readonly char symbol;
    private readonly Func<long, long, long> f;

    public int Priority { get { return priority; } }
    public override char Symbol { get { return symbol; } }

    public BinaryOp(int priority, char symbol, Func<long, long, long> f)
    {
        if (priority <= 0)
            throw new ArgumentException("Priority_must_be_>_0");
        this.priority = priority;
        this.symbol = symbol;
        this.f = f;
    }

    public override long Eval(params long[] args)
    {
        if (args.Length != 2)
```

```
        throw new ArgumentException("Invalid_number_of_
            arguments_of_a_binary_operation");
        return f(args[0], args[1]);
    }
}

public sealed class BinaryExpression : Expression
{
    private readonly BinaryOp op;
    private readonly Expression left, right;

    public BinaryOp Op { get { return op; } }
    public Expression Left { get { return left; } }
    public Expression Right { get { return right; } }

    public BinaryExpression(BinaryOp op, Expression left,
        Expression right)
    {
        if (op == null || left == null || right == null)
            throw new ArgumentNullException();
        if (left is EmptyExpression || right is
            EmptyExpression)
            throw new ArgumentException("Binary_operation_
                cannot_be_applied_to_empty_expression");
        this.op = op;
        this.left = left;
        this.right = right;
    }

    public override long Eval()
    {
        return op.Eval(left.Eval(), right.Eval());
    }
}

public sealed class UnaryExpression : Expression
{
    private readonly UnaryOp op;
    private readonly Expression arg;

    public UnaryOp Op { get { return op; } }
    public Expression Arg { get { return arg; } }

    public UnaryExpression(UnaryOp op, Expression arg)
    {
        if (op == null || arg == null)
            throw new ArgumentNullException();
        if (arg is EmptyExpression)
            throw new ArgumentException("Unary_operation_
                cannot_be_applied_to_empty_expression");
        this.op = op;
        this.arg = arg;
    }

    public override long Eval()
    {
        return op.Eval(arg.Eval());
    }
}
```

```
}

public sealed class NumberExpression : Expression, IToken
{
    private readonly long value;

    public long Value { get { return value; } }

    public NumberExpression(long value)
    {
        this.value = value;
    }

    public override long Eval()
    {
        return value;
    }

    internal static bool TryRead(IInput input, out IToken
        result)
    {
        if (input.IsEmpty() || !char.IsDigit(input.Current))
        {
            result = null;
            return false;
        }
        string s = "";
        while (!input.IsEmpty())
        {
            if (!char.IsDigit(input.Current))
            {
                result = new NumberExpression(long.Parse(s));
                return true;
            }
            else
            {
                s += input.Current;
                input.MoveNext();
            }
        }
        result = new NumberExpression(long.Parse(s));
        return true;
    }
}

public sealed class EmptyExpression : Expression
{
    public override long Eval()
    {
        throw new InvalidOperationException("Empty_expression
            _cannot_be_evaluated");
    }
}

public static class OperationsRegistry
{
    private static Dictionary<char, UnaryOp> Ops1 =
        new Dictionary<char, UnaryOp>();
}
```

```
private static Dictionary<char, BinaryOp> Ops2 =
    new Dictionary<char, BinaryOp>();

public static void Register(IEnumerable<Operation> ops)
{
    foreach (Operation op in ops)
        Register(op);
}

public static void Register(Operation op)
{
    if (op == null)
        throw new ArgumentNullException();
    if (Symbols.IsReserved(op.Symbol))
        throw new ArgumentException("Reserved_symbol_
            cannot_be_used_as_an_operation_symbol");
    if (op is UnaryOp)
    {
        if (Ops1.ContainsKey(op.Symbol))
            throw new ArgumentException("Operation_is_
                already_registered");
        else
            Ops1.Add(op.Symbol, op as UnaryOp);
    }
    else if (op is BinaryOp)
    {
        if (Ops2.ContainsKey(op.Symbol))
            throw new ArgumentException("Operation_is_
                already_registered");
        else
            Ops2.Add(op.Symbol, op as BinaryOp);
    }
    else
        throw new ArgumentException("Unsupported_
            operation");
}

internal static bool TryRead1(IInput input, out IToken
    result)
{
    if (!input.IsEmpty() && Ops1.ContainsKey(input.
        Current))
    {
        result = Ops1[input.Current];
        return ((Operation)result).TryReadSymbol(input);
    }
    else
    {
        result = null;
        return false;
    }
}

internal static bool TryRead2(IInput input, out IToken
    result)
{
    if (!input.IsEmpty() && Ops2.ContainsKey(input.
```

```
        Current))
    {
        result = Ops2[input.Current];
        return ((Operation)result).TryReadSymbol(input);
    }
    else
    {
        result = null;
        return false;
    }
}

}

public sealed class GroupExpression : Expression, IToken
{
    private readonly Expression content;

    public Expression Content { get { return content; } }

    public GroupExpression(Expression content)
    {
        if (content == null)
            throw new ArgumentNullException();
        if (content is EmptyExpression)
            throw new ArgumentException("Group_expression_
                cannot_be_empty");
        this.content = content;
    }

    public override long Eval()
    {
        return content.Eval();
    }

    internal static bool TryRead(IInput input, out IToken
        result)
    {
        if (input.IsEmpty())
        {
            result = null;
            return false;
        }
        else
        {
            if (input.Current == Symbols.OpenBrace)
            {
                input.MoveNext();
                int OpenBracePos = input.Position;
                Expression body = Expression.Parse(input,
                    Symbols.CloseBrace);
                if (body is EmptyExpression)
                    throw new ExpressionSyntaxError(
                        ExpressionSyntaxError.ErrorReason.
                            EmptyParentheses, OpenBracePos);
                result = new GroupExpression(body);
                return true;
            }
            else

```

```
        {
            result = null;
            return false;
        }
    }
}

private sealed class RawExpression : Expression
{
    private List<IToken> tokens = new List<IToken>();

    private bool AcceptNum = true;
    private bool AcceptOp = true;
    private bool AcceptGrp = true;

    private RawExpression() { }

    private RawExpression(IEnumerable<IToken> tokens)
    {
        this.tokens.AddRange(tokens);
    }

    private void AddToken(IToken tok)
    {
        if (tok == null)
            throw new ArgumentNullException();
        if (tok is NumberExpression)
        {
            if (!AcceptNum)
                throw new InvalidOperationException();
            AcceptNum = false;
            AcceptOp = true;
            AcceptGrp = false;
        }
        else if (tok is UnaryOp)
        {
            if (!AcceptOp)
                throw new InvalidOperationException();
            AcceptNum = true;
            AcceptOp = false;
            AcceptGrp = true;
        }
        else if (tok is BinaryOp)
        {
            if (!AcceptOp)
                throw new InvalidOperationException();
            AcceptNum = true;
            AcceptOp = false;
            AcceptGrp = true;
        }
        else if (tok is GroupExpression)
        {
            if (!AcceptGrp)
                throw new InvalidOperationException();
            AcceptNum = false;
            AcceptOp = true;
            AcceptGrp = false;
        }
    }
}
```

```
    }
    tokens.Add(tok);
}

private IToken GetToken(IInput input)
{
    IToken result;
    if (GroupExpression.TryRead(input, out result))
        return result;
    else if (NumberExpression.TryRead(input, out result))
        return result;
    else if (tokens.Count == 0
        && OperationsRegistry.TryRead1(input, out result)
        )
        return result;
    else if (tokens.Count > 0
        && OperationsRegistry.TryRead2(input, out result)
        )
        return result;
    else
        throw new InvalidOperationException();
}

public void ReadToken(IInput input)
{
    try
    {
        AddToken(GetToken(input));
    }
    catch (InvalidOperationException)
    {
        throw new ExpressionSyntaxError(
            ExpressionSyntaxError.ErrorReason.
            UnexpectedToken, input.Position);
    }
}

private bool FindMinPriorityBinOp(out int position,
    out BinaryOp op)
{
    position = -1;
    op = null;
    int i = 0;
    foreach (IToken tok in tokens)
    {
        if (tok is BinaryOp &&
            (op == null
            || ((BinaryOp)tok).Priority <= op.Priority))
        {
            op = (BinaryOp)tok;
            position = i;
        }
        i++;
    }
    return (op != null);
}

public Expression BuildExpressionTree()
```

```
{
    if (tokens.Count == 0)
        return new EmptyExpression();
    int pos; BinaryOp op;
    if (FindMinPriorityBinOp(out pos, out op))
    {
        if (pos == 0)
            throw new Exception("Unknown_error");
        Expression left =
            new RawExpression(tokens.Take(pos)).
                BuildExpressionTree();
        tokens.RemoveRange(0, pos + 1);
        Expression right = BuildExpressionTree();
        return new BinaryExpression(op, left, right);
    }
    else
    {
        IToken tok = tokens.First();
        if (tok is UnaryOp)
        {
            tokens.RemoveAt(0);
            Expression arg = BuildExpressionTree();
            return new UnaryExpression((UnaryOp)tok, arg);
        }
        else if (tok is Expression && tokens.Count == 1)
            return (Expression) tok;
        else
            throw new Exception("Unknown_error");
    }
}

public override long Eval()
{
    throw new NotImplementedException();
}

public static new Expression Parse(IInput input, char?
    stop)
{
    RawExpression e = new RawExpression();
    while (!input.IsEmpty())
    {
        while (!input.IsEmpty()
            && Symbols.IsWhite(input.Current))
        {
            input.MoveNext();
        }
        if (!input.IsEmpty())
        {
            if (input.Current == stop)
            {
                input.MoveNext();
                return e.BuildExpressionTree();
            }
            else
            {
                e.ReadToken(input);
            }
        }
    }
}
```

```
        }
    }
    if (stop == null)
        return e.BuildExpressionTree();
    else
        throw new ExpressionSyntaxError(
            ExpressionSyntaxError.ErrorReason.
                MissingSymbol,
            input.Position,
            new string((char)stop, 1)
        );
    }
}

public abstract long Eval();

public static long Eval(string input)
{
    return Eval((StringInput)input);
}

public static long Eval(IInput input)
{
    return Parse(input).Eval();
}

private static Expression Parse(IInput input, char? stop)
{
    return RawExpression.Parse(input, stop);
}

public static Expression Parse(IInput input)
{
    input.MoveNext();
    return Parse(input, null);
}

public static Expression Parse(string input)
{
    return Parse((StringInput)input);
}
}

public class ArithmeticOps : IEnumerable<Expression.Operation>
{
    private static IEnumerable<Expression.Operation> ops =
        new Expression.Operation[] {
            new Expression.UnaryOp('-', x => -x),
            new Expression.BinaryOp(1, '+', (x, y) => x + y),
            new Expression.BinaryOp(1, '-', (x, y) => x - y),
            new Expression.BinaryOp(2, '*', (x, y) => x * y),
            new Expression.BinaryOp(2, '/', (x, y) => x / y)
        };

    private static bool Registered = false;

    public static void Register()
```

```
        {
            if (!Registered)
            {
                Expressions.Expression.OperationsRegistry.Register(
                    new ArithmeticOps()
                );
            }
        }

        private ArithmeticOps() { }

        IEnumerator<Expression.Operation>
        IEnumerable<Expression.Operation>.GetEnumerator()
        {
            return ops.GetEnumerator();
        }

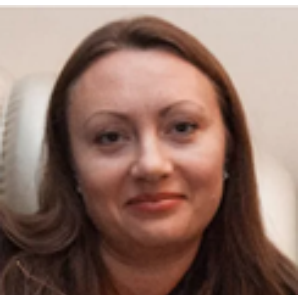
        IEnumerator IEnumerable.GetEnumerator()
        {
            return ops.GetEnumerator();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Expressions.ArithmeticOps.Register();
        Console.WriteLine("Enter arithmetic expression:");
        string expr = Console.ReadLine();
        try
        {
            Console.WriteLine("Result: {0}",
                Expressions.Expression.Eval(expr));
        }
        catch (Expressions.Expression.ExpressionSyntaxError e)
        {
            ConsoleColor c = Console.ForegroundColor;
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(new string(' ', e.Where) + '^');
            Console.WriteLine("ERROR: {0}", e.Message);
            Console.ForegroundColor = c;
        }
    }
}

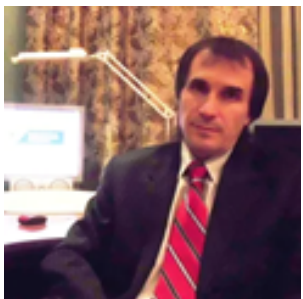
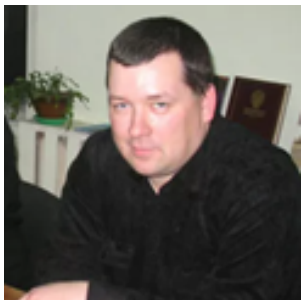
?
```



## 8 Відомості про авторів

|                                                                                     |                                                                                                                                                                                                                                                                            |
|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|    | <p><b>Бичков Олексій Сергійович</b><br/>д.т.н., професор, Лауреат державної премії України в галузі науки і техніки, завідувач кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p> |
|   | <p><b>Іванов Євген В'ячеславович</b><br/>к.ф.-м.н., доцент кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p>                                                                     |
|  | <p><b>Жабська Єлизавета Олегівна</b><br/>асистент кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p>                                                                              |
|  | <p><b>Петрівський Володимир Ярославович</b><br/>PhD, асистент кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p>                                                                  |
|  | <p><b>Меркулова Катерина Володимирівна</b><br/>к.т.н., доцент, доцент кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p>                                                          |

---

|                                                                                   |                                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <p><b>Шатирко Андрій Володимирович</b><br/>д.ф.-м.н., доцент, професор кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p> |
|  | <p><b>Порєв Геннадій Володимирович</b><br/>д.т.н., с.н.с., доцент кафедри програмних систем і технологій факультету інформаційних технологій Київського національного університету імені Тараса Шевченка.</p>      |

# Список літератури

1. Об'єктно-орієнтоване програмування мовою C# : підручник / Олексій Сергійович Бичков, Євген В'ячеславович Іванов. – Київ : ВПЦ "Київський університет 2018. – 207 с. : іл., табл. 100 пр. – Бібліогр.: с. 204 . – На укр. яз. ISBN 978-966-439-965-1.
2. Learn C#.   
OFFICIAL COLLECTION: <https://learn.microsoft.com/en-us/dotnet/csharp/>
3. Overview of .NET Framework - .NET Framework. Microsoft Learn: Build skills that open doors in your career.   
URL: <https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview> (date of access: 06. 10.2022).
4. .NET implementations - .NET. Microsoft Learn: Build skills that open doors in your career.   
URL: <https://learn.microsoft.com/en-us/dotnet/fundamentals/implementations> (date of access: 06. 10.2022).
5. What Is Windows Communication Foundation - WCF. Microsoft Learn: Build skills that open doors in your career.   
URL: <https://learn.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf> (date of access: 06. 10.2022).
6. .NET class library overview. Microsoft Learn: Build skills that open doors in your career.   
URL: <https://learn.microsoft.com/en-us/dotnet/standard/class-library-overview> (date of access: 06.10.2022).
7. Warren M. Posts By Year. Performance is a Feature!.   
URL: <https://mattwarren.org/postsByYear/> (date of access: 06.10.2022).
8. Assemblies in .NET. Microsoft Learn: Build skills that open doors in your career.   
URL: <https://learn.microsoft.com/en-us/dotnet/standard/assembly/?source=recommendations> (date of access: 06.10.2022).

- 
9. What is Xamarin? - Xamarin. Microsoft Learn: Build skills that open doors in your career.  
URL: <https://learn.microsoft.com/en-us/xamarin/get-started/what-is-xamarin> (date of access: 06.10.2022).
  10. Managed Execution Process. Microsoft Learn: Build skills that open doors in your career.  
URL: <https://learn.microsoft.com/en-us/dotnet/standard/managed-execution-process> (date of access: 06.10.2022).
  11. .NET Core Overview. TutorialTeacher - Learn Technologies.  
URL: <https://www.tutorialsteacher.com/core/dotnet-core> (date of access: 06.10.2022).
  12. Kendal S. Object Oriented Programming using C#. – 2-nd edition. – BookBoon, 2018. – 403 p. 2. Mark J. Price C# 7 and .NET Core: Modern Cross-Platform Development – Second Edition, 2017. – 594 p. 3.

Навчальне видання

# C# - творчість програмування

Том 1.

Об'єктно-орієнтоване програмування.

Підручник

Бичков Олексій Сергійович  
Іванов Євген В'ячеславович  
Жабська Єлизавета Олегівна  
Петрівський Володимир Ярославович  
Меркулова Катерина Володимирівна  
Шатирко Андрій Володимирович  
Порєв Геннадій Володимирович

e-mail: [oleksiibychkov@knu.ua](mailto:oleksiibychkov@knu.ua)  
[http: pst.knu.ua](http://pst.knu.ua)

Підписано до друку 25.04.2024 р. Формат 60x84 1/8.  
Папір офсет. Гарнітура «Computer Modern». Друк цифр.  
Ум. друк. арк. 33,48. Наклад 50 пр. Зам. 65.  
Видавництво «Волинські обереги».  
33028 м. Рівне, вул. 16 Липня, 38; тел./факс: (0362) 62-03-97;  
e-mail: [oberegi97@ukr.net](mailto:oberegi97@ukr.net)  
Свідоцтво про внесення до Державного реєстру суб'єкта  
видавничої справи ДК № 270 від 07.12.2000 р.  
Надруковано в друкарні видавництва «Волинські обереги».