Міністерство освіти і науки України
Національний технічний університет
«Харківський політехнічний інститут»

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ JAVA

Методичні вказівки до лабораторних робіт
з курсу «Об'єктно-орієнтоване програмування»

для студентів спеціальності
122 – Комп'ютерні науки
126 – Інформаційні системи та технології

Затверджено
редакційно-видавничою
радою університету,
протокол № 2 від 29.06.2021

Харків
НТУ «ХПІ»
2022

Об’єктно-орієнтоване програмування мовою Java. Методичні вказівки до лабораторних занять з курсу «Об’єктно-орієнтоване програмування» для студентів спеціальностей 122 – Комп’ютерні науки, 126 – Інформаційні системи та технології / уклад. Нікуліна О. М., Іванов Л. В., Коцюба Н. В. – Х. : НТУ «ХПІ», 2022. – 64 с.

Укладачі: О. М. Нікуліна
Л. В. Іванов
Н. В. Коцюба
Рецензент В. В. Москоленко

Кафедра інформаційні системи та технології

ВСТУП

Вказівки призначені для студентів, які вивчають мову Java на аудиторних заняттях або самостійно. Класи, узагальнення, успадкування, винятки, стандартна бібліотека розглядаються на прикладах, супроводжуваних необхідними теоретичними відомостями.

Предметом навчальної дисципліни «Об'єктно-орієнтованого програмування» є методи алгоритмізації та програмування мовами C++, C# та Java. Завдання дисципліни – освоєння методів та засобів об'єктно-орієнтованого програмування у середовищах Visual Studio та Eclipse.

Мета цих методичних вказівок – допомогти студентам засвоїти основи об'єктно-орієнтованого програмування мовою Java.

Лабораторні роботи передбачають вивчення розділів: побудова та об'ява класів, вкладені класи та композиція, успадкування, інтерфейси та абстрактні класи, винятки, використання контейнерів, узагальнень, написання програм з використанням графічного інтерфейсу користувача. Для виконання завдань студент повинен в достатньому ступені володіти програмуванням мовою Java. До кожної лабораторної роботи в методичних вказівках наведені: мета роботи, теоретичні основи та варіанти контрольних задач за розділами курсів, що вивчаються. Варіанти робіт відповідають вимогам навчального плану та навчальної програми і можуть бути застосовані для здійснення контролю знань студентів зі спеціальностями 122 – «Комп'ютерні науки» та 126 – «Інформаційні системи та технології».

ЛАБОРАТОРНА РОБОТА 1

КЛАСИ ТА ОБ'ЄКТИ

Мета лабораторної роботи – опанувати написання програм з використанням об'єктів.

1.1. Теоретичні основи

Клас – це структурований тип даних, набір елементів даних різних типів і функцій для роботи з цими даними. Опис класу складається зі специфікаторів (наприклад, public, final), імені, імені базового класу, списку інтерфейсів і тіла у фігурних дужках.

Тіло класу містить поля (їм відповідають елементи даних у C++) і методи (функції-елементи в C++). Поля і методи разом іменуються елементами (членами) класу. Нижче наводиться приклад опису класу:

```
class Rectangle {
    double width;
    double height;
    double area() {
        return width * height;
    }
}
```

Після останньої фігурної дужки, що закривається, не слід ставити крапку з комою. Методи завжди реалізуються усередині визначення класу.

Під час створення об'єкта класу поля ініціалізуються усталеними значеннями (нулями або `null` для посилань). Java допускає ініціалізацію полів початковими значеннями:

```
class Rectangle {
    double width = 10;
    double height = 20;
    double area() {
        return width * height;
    }
}
```

Можна створити спеціальний блок ініціалізації усередині тіла класу. Такий блок виконуватиметься щораз під час створення нового об'єкта:

```
class Rectangle {
    double width;
    double height;
    {
        width = 10;
        height = 20;
    }
    double area() {
        return width * height;
    }
}
```

Для того, щоб працювати з полями і методами класу, необхідно створити об'єкт. Для цього спочатку створюють посилання на об'єкт, а потім за допомогою операції **new** створюють сам об'єкт шляхом виклику конструктора. Ці дії можна поєднати. Після цього можна викликати методи і використовувати поля:

```
Rectangle rect = new Rectangle();
double a = rect.area();
rect.width = 15
double b = rect.area();
```

Під час виклику методів аргументи передаються за значенням.

Ключове слово **this** використовується як посилання на об'єкт, для якого викликаний метод. Усі нестатичні методи неявно отримують посилання на об'єкт для якого вони використані. Ключове слово **this** використовувати явно, наприклад, коли треба повернути з функції посилання на поточний об'єкт, або запобігти конфлікту імен.

Як і C++, Java підтримує закритий (**private**), пакетний, захищений (**protected**) і відкритий (**public**) рівні доступу. Сам клас може бути оголошений як **public**. На відміну від C++, Java вимагає окремої специфікації доступу для кожного елемента, або групи полів одного типу:

```
public class Rectangle {
    private double width = 10;
    private double height = 20;
    {
        width = 30;
        height = 40;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getWidth() {
        return width;
    }
    public void setHeight(double height) {
        this.height = height;
    }
    public double getHeight() {
        return height;
    }
    public double area() {
        return width * height;
    }
}
```

У Java немає ключового слова **`friend`**, яке у C++ забезпечує доступ до закритих елементів ззовні класу.

Елементи класу без атрибутів доступу мають пакетну видимість. Такий доступ ще називають "дружнім". Всі інші класи цього пакета мають доступ до таких елементів як до відкритих. Ззовні пакета такі елементи взагалі недоступні.

Інкапсуляція (приховування даних) – одна з трьох парадигм об'єктно-орієнтованого програмування. Зміст інкапсуляції полягає у приховуванні від зовнішнього користувача деталей реалізації об'єкту. Зокрема доступ до даних (полів), які зазвичай описані з модифікатором **`private`**, здійснюється через відкриті функції доступу. Як правило, це так звані сеттери та геттери. Якщо поле має ім'я `name`, відповідні

функції доступу мають імена `setName` та `getName`.

Екземпляр класу створюється шляхом застосування операції **`new`** до конструктора. Конструктор – це функція, яка здійснює ініціалізацію даних об'єкта. Ім'я конструктора збігається з ім'ям класу. Не можна вказувати типу результату конструктора. У класі може бути визначено кілька конструкторів. Якщо жоден конструктор явно не визначений, автоматично створюється усталений конструктор (без параметрів). Такий конструктор ініціалізує всі поля усталеними початковими значеннями. Після визначення принаймні одного конструктора усталений конструктор автоматично не створюється.

Один конструктор можна викликати з іншого з використанням слова **`this`**, після якого випливають необхідні аргументи. Спочатку здійснюється ініціалізація в місці опису, після якої значення можуть бути перевизначені в блоці ініціалізації, а потім перевизначені в конструкторі

```
public class Rectangle {
    private double width = 10;
    private double height = 20;
    {
        width = 30;
        height = 40;
    }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public Rectangle() {
        this(50, 60); // виклик іншого конструктора
    }
}
Rectangle rectangle = new Rectangle();
```

У Java немає конструкторів копіювання і деструкторів. Можна створити спеціальний метод `finalize()`, який викликається збирачем сміття перед ліквідацією об'єкта. У деяких випадках об'єкт може бути

не вилучений збирачем сміття ніколи (пам'яті вистачало до кінця програми), отже метод `finalize()` може бути ніколи не викликаний.

Методи і поля можуть бути оголошені з ключовим словом **`static`**. Звернення до таких полів і методів може здійснюватися без створення екземпляра класу. Статичні поля є альтернативою відсутнім у Java глобальним змінним. На відміну від C++, статичні елементи даних не потрібно окремо визначати в глобальній області видимості. Статичні поля можуть бути проініціалізовані під час створення:

```
class SomeClass {
    static double x = 10;
    static int    i = 20;
}
```

Можна створити окремий блок статичної ініціалізації:

```
class SomeClass {
    static double x;
    static int i;
    static {
        x = 10;
        i = 20;
    }
}
```

На відміну від нестатичної ініціалізації, створення й ініціалізація статичних полів здійснюється під час першого звернення до класу (створенні екземпляра класу чи зверненні до статичних елементів). Java не створює статичних полів для класів, які не використовуються.

Статичні методи не отримують посилання на об'єкт і не можуть використовувати посилання **`this`**. Звернення до статичних елементів може здійснюватися як через ім'я класу, так і через посилання на об'єкт:

```
SomeClass.x = 30;
SomeClass s = new SomeClass();
s.x = 40;
```

Всередині класів можна визначати константи. Константи можуть бути двох видів – статичні й нестатичні. Статичну константу створює

компілятор. За угодою її ім'я має містити лише великими літерами:

```
public static final double PI = 3.14159265;
```

Значення нестатичної константи слід визначити, причому один раз – у місці визначення, в блоці ініціалізації (тоді це значення буде однаковим для всіх екземплярів), або в конструкторі:

```
public class ConstDemo {
    public final int one = 1;
    public final int two;
    {
        two = 2;
    }
    public final int other;
    public ConstDemo(int other) {
        this.other = other;
    }
}
```

Цілком безпечно визначати константи як **`public`**, оскільки компілятор не дозволить змінити їх значення.

1.2. Задачі

У всіх завданнях, крім зазначених операцій, повинні бути реалізовані наступні методи: конструктор, введення з клавіатури *Read*, виведення на екран *Display*, перетворення в рядок *toString*. Всі поля закриті.

Задача 1.1

Створити клас *Vector3d*, що задається трійкою координат. Реалізувати методи: додавання і віднімання векторів, скалярний добуток векторів, множення на скаляр, порівняння векторів, обчислення довжини вектора, порівняння довжини вектора.

Задача 1.2

Створити клас *Money* для роботи з грошовими сумами. Число представлено двома полями: гривні і копійки. Реалізувати методи: додавання і віднімання, ділення сум, розподіл суми на дробове число, множення суми на дробове число, операції порівняння.

Задача 1.3

Створити клас *Triangle* для подання трикутника. Поля: сторони і кути. Реалізувати операції: отримання і зміни полів даних, обчислення площі, обчислення периметра, обчислення висот, визначення виду трикутника.

Задача 1.4

Створити клас *Angle* для роботи з кутами на площині. Поля: градуси і хвилини. Реалізувати методи: переклад в радіани, приведення до діапазону 0–360, збільшення і зменшення кута на задану величину, отримання синуса, порівняння кутів.

Задача 1.5

Створити клас *Data* для роботи з датами. Поля: рік, місяць, день. Реалізувати операції: віднімання, порівняння дат, визначення високосного року, переклад в кількість днів.

Задача 1.6

Створити клас *Time* для роботи з часом. Поля: години, хвилини, секунда. Реалізувати операції: віднімання, порівняння часу, переклад в хвилини і секунди.

Задача 1.7

Створити клас *Account*, який представляє собою банківський рахунок. Поля: прізвище власника, номер рахунку, відсоток нарахування, сума в гривнях. Реалізувати операції: зміна власника рахунку, зняття деякої суми з рахунку, покласти гроші на рахунок, нарахувати відсоток.

Задача 1.8

Створити клас *Fraction*, для роботи з дробовими числами. Поля: ціла частина і дрібна. Реалізувати операції: додавання, віднімання, множення, операції порівняння.

Задача 1.9

Створити клас *Goods* (Товар). Поля: найменування товару, дата оформлення, ціна товару, кількість одиниць товару, номер накладної. Реалізувати методи: зміни ціни товару, зміни кількості товару, обчислення вартості товару.

Задача 1.10

Створити клас *Payment* (Зарплата). Поля: ПІБ, оклад, рік надходження на роботу, прибутковий податок, відсоток надбавки, кількість відпрацьованих днів на місяць, нарахована і утримана суми. Реалізувати методи: обчислення нарахованої суми, обчислення утриманої суми, сума, що видається, обчислення стажу.

ЛАБОРАТОРНА РОБОТА 2
ВКЛАДЕНІ КЛАСИ. КОМПОЗИЦІЯ

Мета лабораторної роботи – опанувати написання програм з вкладеними класами.

2.1. Теоретичні основи

Визначення класу може бути розміщене всередині іншого класу. В такий спосіб можуть бути створені вкладені класи, які можуть бути статичними вкладеними або внутрішніми. Вкладені класи можуть використовуватися як усередині обхопного класу, так і поза ним.

```
class Outer {
    class Inner {
        int i;
    };
    Inner inner = new Inner();
}
class Another {
    Outer.Inner i;
}
```

Вкладені класи можуть бути оголошені зі специфікаторами **public, private** або **protected**.

Локальні класи створюють всередині блоків. Існує також спеціальний різновид локальних класів – безіменні класи.

Окрема категорія – статичні вкладені класи, використання яких аналогічне вкладеним класам C++ і C#.

Нестатичні вкладені класи називають також внутрішніми. Голов-

ною відмінністю внутрішніх класів у Java є те, що об'єкти цих класів отримують посилання на об'єкт обхопного класу. З цього факту випливає два важливих висновки: об'єкти внутрішніх класів мають прямий доступ до даних об'єкта обхопного класу та для створення об'єкта внутрішнього класу обов'язково мати в наявності об'єкт обхопного класу.

У зв'язку з цим в Java запропонований спеціальний механізм створення об'єктів внутрішніх класів. Цей механізм проілюстрований на наведеному нижче прикладі.

```
class Outer {
    int k = 100;
    class Inner {
        void show() {
            System.out.println(k);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.show();
    }
}
```

Нестатичні внутрішні класи не можуть містити статичних елементів.

Слід пам'ятати, що об'єкт внутрішнього класу автоматично не створюється. Створення об'єкта може бути передбачене в конструкторі чи у будь-якому методі обхопного класу, а також поза ним (якщо цей клас не оголошений як **`private`**). Можна також створити масив об'єктів внутрішнього класу. Кожен з таких об'єктів матиме доступ до посилання на обхопний об'єкт.

Внутрішні класи можуть мати свої базові класи. В такий спосіб за допомогою внутрішніх класів можна змоделювати відсутній у Java

механізм множинного спадкування:

```
class FirstBase {
    int a = 1;
}
class SecondBase {
    int b = 2;
}
class Outer extends FirstBase {
    int c = 3;
    class Inner extends SecondBase {
        void show() {
            System.out.println(a);
            System.out.println(b);
            System.out.println(c);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.show();
    }
}
```

Наведений приклад має суто теоретичний сенс, оскільки множинне успадкування класів незалежно від способів його реалізації є небезпечним з точки зору можливого конфлікту імен.

До внутрішніх класів також відносяться локальні. До таких класів не можна звернутися ззовні блоку, у якому вони визначені. Локальні класи найчастіше поміщають у тіло функції:

```
void f() {
    class Local {
        int j;
    }
    Local l = new Local();
```

```
    l.j = 100;
    System.out.println(l.j);
}
```

Можна також розміщати локальні класи всередині окремих блоків.

Безіменний клас може реалізовувати певний інтерфейс, перекривати абстрактні функції базового класу чи розширювати його. Для створення об'єкта безіменного класу здійснюється виклик конструктора базового класу, або вказується ім'я інтерфейсу з круглими дужками, після чого розташовують тіло безіменного класу:

```
new Object() {
    // Додавання нового методу:
    void hello() {
        System.out.println("Привіт!");
    }
}.hello();
System.out.println(new Object() {
    // Перевизначення методу:
    @Override public String toString() {
        return "Це безіменний клас.";
    }
});
```

Безіменні класи не можуть бути абстрактними. Безіменний клас завжди є внутрішнім класом; він не може бути статичним. Безіменні класи автоматично є фінальними (**final**). У наведеному нижче прикладі безіменний клас створюється для визначення способу сортування масиву рядків:

```
void sortByABC(String[] a) {
     Arrays.sort(a, new Comparator<String>() {
        public int compare(String s1, String s2) {
             return (s1).compareTo(s2);
         }
     });
}
```

У безіменних класів не може бути явних конструкторів. Разом з

тим, завжди створюється усталений безіменний конструктор. Якщо в базового класу немає конструктора без параметрів, необхідні параметри конструктора вказуються в дужках під час створення об'єкта. Для того, щоб безіменні класи мали доступ до локальних елементів зовнішніх блоків, ці елементи повинні бути описані як **`final`**.

Статичні вкладені класи мають доступ тільки до статичних елементів обхопних класів. Об'єкти таких класів можуть бути створені без створення об'єктів обхопних класів:

```
class Outer {
    int k = 100;
    static int m = 200;
    static class Inner {
        void show() {
            // k недоступно
            System.out.println(m);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.show();
    }
}
```

Статичні вкладені класи можуть містити свої статичні елементи, у тому числі свої вкладені статичні і нестатичні класи.

Класи можна створювати всередині інтерфейсів. Такі класи автоматично є статичними. Усередині класів також можна створювати інтерфейси, що є також статичними.

Під композицією класів розуміють створення нових класів з використанням об'єктів інших класів як полів. Java не дозволяє розміщення об'єктів усередині інших об'єктів, можна тільки описувати посилання. Композиція класів у цьому випадку припускає створення об'єктів безпосередньо (у тілі класу при оголошенні полів) чи в конструкторах.

```
class X {
}
class Y {
}
class Z {
    X x = new X();
    Y y;
    Z() {
        y = new Y();
    }
}
```

Можна також створити внутрішній об’єкт безпосередньо перед його першим використанням. Відношення, що моделюється композицією, часто називають відношенням "has-a".

Агрегування – це різновид композиції, який передбачає, що сутність (екземпляр) міститься в іншій сутності або не може бути створена та існувати без сутності, яка її охоплює. При цьому сутність, що охоплює, може існувати без внутрішньої, тобто час життя зовнішньої та внутрішньої сутностей може не збігатися. Більш строге трактування композиції (власне композиція) передбачає, що час життя зовнішньої та внутрішньої сутностей збігається. На рівні Java агрегування передбачає можливість створення внутрішнього об’єкта перед його використанням, тоді як строга композиція передбачає створення внутрішнього об’єкта в тілі класу, в блоці ініціалізації або в конструкторі.

2.2. Задачі

У всіх завданнях необхідно реалізувати по два класи. Реалізувати завдання двома способами:

1) композиція класів – допоміжний клас повинен бути визначений як незалежний, об’єкти цього класу використовуються в якості полів основного;

2) вкладені класи.

Задача 2.1

Реалізувати клас *Vector3d* (задача 1.1), використовуючи клас Frac-

tion (задача 1.8).

Задача 2.2

Реалізувати клас *Money* (задача 1.2), використовуючи клас Fraction (задача 1.8).

Задача 2.3

Реалізувати клас *Triangle* (задача 1.3), спираючись на клас *Angle* (задача 1.4) для подання кутів.

Задача 2.4

Реалізувати клас *Quadrangle*, спираючись на класи *Triangle* (задача 1.3) і Angle (задача 1.4) для подання кутів.

Задача 2.5

Реалізувати клас *Goods* (задача 1.9), додати поле-дату надходження товару на склад використовуючи клас Data (задача 1.5).

Задача 2.6

Реалізувати клас *Data* (задача 1.5), використовуючи клас *Time* (задача 1.6).

Задача 2.7

Реалізувати клас *Account* (задача 1.7), використовуючи суми клас Money (задача 1.2).

Задача 2.8

Реалізувати клас *Calculator* з повним набором арифметичних операцій, використовуючи клас (задача 1.8).

Задача 2.9

Реалізувати клас *Goods* (задача 1.9), використовуючи для ціни клас Money (задача 1.2).

Задача 2.10

Реалізувати клас *Payment* (задача 1.10), використовуючи для полів з датою клас Data (задача 1.5).

ЛАБОРАТОРНА РОБОТА 3
УСПАДКУВАННЯ

Мета лабораторної роботи – опанувати написання програм з

об’єктами, які успадковуються.

3.1. Теоретичні основи

Механізм успадкування полягає в породженні похідних класів від базових. Якщо один клас (похідний) є нащадком іншого (базового), то спадкоємець має можливість безпосередньо користуватися неприватними даними і функціями, визначеними в базовому класі. Відносини між класами і підкласами (нащадками) називаються *ієрархією спадкування* класів.

На відміну від C++, у Java дозволяється тільки одиничне успадкування – клас може мати тільки один базовий клас. Успадкування завжди відкрите. У Java також немає захищеного і закритого успадкування. Успадкування має такий синтаксис:

```
class DerivedClass extends BaseClass {
    // тіло класу
}
```

Функції похідного класу мають доступ тільки до елементів, описаних у як **`public`** і **`protected`**. Члени класу, оголошені як захищені, можуть використовуватися класами-нащадками, а також у межах пакета. Закриті (**`private`**) члени класу недоступні навіть для його нащадків.

Усі класи Java безпосередньо чи опосередненно походять від класу `java.lang.Object`. Цей клас надає набір корисних методів, таких як `toString()` для отримання даних будь-якого об’єкта у вигляді рядка тощо. Базовий клас `Object` не вказують явно.

Клас успадковує всі елементи базового класу, крім конструкторів. До початку виконання конструктора похідного класу викликається конструктор базового класу.

Ключове слово **`super`** використовують для доступу до елементів базового класу з похідного класу, зокрема:

– для виклику перекритого методу базового класу;

– для передачі параметрів конструктору базового класу.

Наприклад:

```
class BaseClass {
```

```
    int i, j;
    BaseClass(int i, int j) {
        this.i = i;
        this.j = j;
    }
}
class DerivedClass extends BaseClass {
    int k;
    DerivedClass(int i, int j, int k) {
        super(i, j);
        this.k = k;
    }
}
```

Доступ до базового класу з використанням **super** дозволений тільки в конструкторах і нестатичних методах.

Класи можуть бути визначені з модифікатором **final** (фінальний). Фінальні класи не можуть використовуватися як базові. Методи з модифікатором **final** не можуть бути перевизначені. Наприклад:

```
final class A {
    void f() { }
}
class B {
    final void g() { }
}
class C extends A { // Помилка! Не можна успадкувати від A
}
class D extends B {
    void g() { }    // Помилка! g() не можна перекрити
}
```

Посилання на похідний клас неявно приводяться до посилання на базовий клас. Об’єкти похідних класів завжди можна використовувати там, де потрібен об’єкт базового класу.

```
class Base {
    static void f(Base b) { }
}
```

```
class Derived extends Base {
    public static void main(String[] args) {
        Base b;
        b = new Derived(); // Неявне приведення
        Derived d = new Derived();
        f(d);              // Неявне приведення
    }
}
```

Зворотне приведення необхідно робити явно:

```
Base b = new Base();
Derived d = (Derived) b;
```

3.2. Задачі

Реалізувати класи завдань лабораторної роботи 1 (свій варіант) з конструкторами, в якості базових класів. Реалізувати для базових класів функції введення / виведення. Реалізувати класи-нащадки, в яких реалізувати різні методи.

Задача 3.1

У класі *Vector3d* реалізувати методи у класі-нащадка: додавання і віднімання векторів порівняння векторів, порівняння довжини вектора.

Задача 3.2

У класі *Money* реалізувати методи у класі-нащадка: додавання і віднімання, операції порівняння.

Задача 3.3

У класі *Triangle* реалізувати методи у класі-нащадка: обчислення площі, обчислення периметра, обчислення висот.

Задача 3.4

У класі *Angle* реалізувати методи у класі-нащадка: переклад в радіани, збільшення і зменшення кута на задану величину, порівняння кутів.

Задача 3.5

У класі *Data* реалізувати методи у класі-нащадка: додавання, віднімання, порівняння дат.

Задача 3.6

У класі *Time* реалізувати методи у класі-нащадка: додавання, віднімання, порівняння часу.

Задача 3.7

У класі *Account* реалізувати методи у класі-нащадка: зняття деякої суми з рахунку, покласти гроші на рахунок.

Задача 3.8

У класі *Fraction* реалізувати методи у класі-нащадка: додавання, віднімання, множення, операції порівняння.

Задача 3.9

У класі *Goods* реалізувати методи у класі-нащадка: зміни ціни товару, обчислення вартості товару.

Задача 3.10

У класі *Payment* реалізувати методи у класі-нащадка: сума, що видається, обчислення стажу.

ЛАБОРАТОРНА РОБОТА 4
ІНТЕРФЕЙСИ ТА АБСТРАКТНІ КЛАСИ

Мета лабораторної роботи – навчитися будувати інтерфейси та абстрактні класи.

4.1. Теоретичні основи

Поліморфізм часу виконання – це властивість класів, згідно з якою поведінка об’єктів класу може визначатися не на етапі компіляції, а на етапі виконання. Класи, що надають ідентичний інтерфейс, але реалізовані під конкретні специфічні вимоги, мають назву *поліморфних класів*.

У мовах об’єктно-орієнтованого програмування пізнє зв’язування реалізоване через механізм віртуальних функцій. *Віртуальна функція* (віртуальний метод, virtual method) – це функція, визначена в базовому класі, та перевизначена (перекрита) у похідних, так, що конкретна реалізація функції для виклику визначатиметься під час виконання програми. Вибір реалізації віртуальної функції залежить від реального (а

не оголошеного під час опису) типу об'єкта. Оскільки посилання на базовий тип може містити адресу об'єкта будь-якого похідного типу, поведінка раніше створених класів може бути змінена пізніше шляхом перевизначення віртуальних методів. Перевизначення передбачає відтворення імені, списку параметрів та специфікатора доступу. Фактично поліморфними є класи, які містять віртуальні функції.

У Java всі методи є віртуальними, за винятком конструкторів, статичних (**`static`**), фінальних (**`final`**) і закритих (**`private`**) методів. На відміну від C++, слово **`virtual`** не використовується.

Починаючи з Java 5, перед перевизначеними віртуальними методами розміщують директиву `@Override`, яка дозволяє компілятору здійснити додаткову перевірку синтаксису – відповідність сигнатури нової функції сигнатурі перекритої функції базового класу. Використання `@Override` є бажаним, але не обов'язковим.

Усі класи Java є поліморфними, оскільки таким є клас `java.lang.Object`. Зокрема, завдяки поліморфізму кожен клас може визначити свою віртуальну функцію `toString()`, яка буде викликана для автоматичного отримання даних про об'єкт у вигляді рядку.

Іноді класи створюються для представлення абстрактних концепцій, а не для створення екземплярів. Такі концепції можуть бути представлені абстрактними класами. У Java для цього використовується ключове слово **`abstract`** перед визначенням класу

```
abstract class SomeConcept {
    . . .
}
```

Абстрактний клас може містити абстрактні методи, такі, для яких не приводиться реалізація. Такі методи не мають тіла функції. Їхнє оголошення аналогічне оголошенню функцій-елементів у C++, але оголошенню повинне передувати ключове слово **`abstract`**.

Абстрактні методи аналогічні суто віртуальним функціям у C++.

Від абстрактного класу не вимагають обов'язкової наявності абстрактних методів. Але кожен клас, у якому є хоч один абстрактний метод, чи хоча б один абстрактний метод базового класу не був визна-

чений, повинен бути оголошений як абстрактний.

У Java використовується поняття інтерфейсів. Інтерфейс може розглядатися як чисто абстрактний клас, але на відміну від абстрактних класів інтерфейс ніколи не містить даних, тільки методи. Ці методи усталено вважаються публічними:

```
interface SomeFunctions {
    void f();
    int g(int x);
}
```

Кожен клас може бути створений тільки від одного базового класу, але при цьому реалізовувати один чи кілька інтерфейсів. Клас, що реалізує інтерфейс, повинен забезпечити реалізацію всіх методів, оголошених в інтерфейсі. В іншому випадку такий клас буде абстрактним і повинен бути оголошений зі специфікатором **abstract**.

Інтерфейси формально можуть містити поля, але вони є фінальними і статичними (константами часу компіляції). Вони повинні бути ініціалізовані під час створення. Інтерфейси не можуть містити конструкторів, оскільки немає ніяких даних окрім статичних констант.

Для того, щоб указати, що клас реалізує інтерфейс, ім'я інтерфейсу вказують у списку реалізованих інтерфейсів. Такий список розташовують у заголовку класу після ключового слова **implements**. Методи, визначені в інтерфейсі, є абстрактними і відкритими. У класі, що реалізує інтерфейс, такі методи повинні бути оголошені як **public**:

```
interface SomeFunctions {
    void f();
    int g(int x);
}
class SomeClass implements SomeFunctions {
    @Override
    public void f() {
    }
    @Override
    public int g(int x) {
        return x;
```

```
    }
}
```

Інтерфейс може мати кілька базових інтерфейсів. Множинне успадкування інтерфейсів є безпечним з точки зору дублювання даних і конфліктів імен.

Клас може реалізувати кілька інтерфейсів. Це – більш розповсюджений шлях, ніж створення похідного інтерфейсу:

Дуже часто в програмі створюють посилання на інтерфейс, яке ініціалізують об'єктом класу. Такий підхід є належною практикою, оскільки він дозволяє легко замінити одну реалізацію інтерфейсу іншою.

Інтерфейси не походять від класу `java.lang.Object`. Не можна створювати новий об'єкт типу інтерфейсу. Навіть для порожнього інтерфейсу треба створити клас, який його реалізує. JDK надає велику кількість стандартних інтерфейсів.

4.2. Задачі

У наступних завданнях потрібно реалізувати абстрактний базовий клас. Ці функції визначаються в похідних класах. В базових класах повинні бути оголошені віртуальні функції введення / виведення, які реалізуються в похідних класах. Замінити абстрактний клас на інтерфейс.

Задача 4.1

Створити абстрактний базовий клас Figure з віртуальними методами обчислення площі та периметра. Створити похідні класи: Rectangle (прямокутник), Trapezium (трапеція) зі своїми функціями площі і периметра. Самостійно визначити, які поля необхідні, які з них можна задати в базовому класі, а які – в похідних.

Задача 4.2

Створити абстрактний базовий клас Number з віртуальними методами – арифметичними операціями. Створити похідні класи Integer (ціле) і Real (дійсне).

Задача 4.3

Створити абстрактний базовий клас Body (тіло) з віртуальними функціями обчислення площі поверхні і об'єму. Створити похідні класи: Parallelepiped (паралелепіпед) і Ball (куля) зі своїми функціями площі поверхні і об'єму.

Задача 4.4

Створити абстрактний клас Currency (валюта) для роботи з грошовими сумами. Визначити віртуальні функції перекладу в рублі і виведення на екран. Реалізувати похідні класи Dollar (долар) і Euro (євро) зі своїми функціями перекладу і виведення на екран.

Задача 4.5

Створити абстрактний базовий клас Triangle для подання трикутника з віртуальними функціями обчислення площі та периметра. Поля даних повинні включати дві сторони і кут між ними. Визначити класи-спадкоємці: прямокутний трикутник, рівнобедрений трикутник, рівносторонній трикутник зі своїми функціями обчислення площі та периметра.

Задача 4.6

Створити абстрактний базовий клас Root (корінь) з віртуальними методами обчислення коренів і виведення результату на екран. Визначити похідні класи Linear (лінійне рівняння) і Square (квадратне рівняння) з власними методами обчислення коренів і виведення на екран.

Задача 4.7

Створити абстрактний базовий клас Function (функція) з віртуальними методами обчислення значення функції в заданій точці і виведення результату на екран. Визначити похідні класи Ellipse (еліпс), Hyperbola (гіпербола) з власними функціями обчислення у залежно від вхідного параметра.

Задача 4.8

Створити абстрактний базовий клас Integer (ціле) з віртуальними арифметичними операціями і функцією виведення на екран. Визначити похідні класи Decimal (десяткове) і Binary (двійкове), що реалізують власні арифметичні операції і функцію виведення на екран. Число представляється масивом, кожен елемент якого – цифра.

Задача 4.9

Створити абстрактний базовий клас Series (прогресія) з віртуальними функціями обчислення і-го елемента прогресії і суми прогресії. Визначити похідні класи: Linear (арифметична) і Exponential (геометрична).

Задача 4.10

Створити абстрактний базовий клас Container з віртуальними методами sort() і поелементної обробки контейнера foreach. Розробити похідні класи Bubble (бульбашка) і Choice (вибір). У першому класі сортування реалізується методом бульбашки, а поелементна обробка полягає в добуванні квадратного кореню. В другом класі сортування реалізується методом вибору, а поелементна обробка обчисленні логарифму.

ЛАБОРАТОРНА РОБОТА 5
ВИНЯТКИ

Мета лабораторної роботи – опанувати написання програм з використанням блоків з винятками.

5.1. Теоретичні основи

Використання механізму обробки винятків є дуже важливою складовою частиною практики програмування на Java. Майже кожна програма на Java містить певні частини цього механізму. Об'єкти-винятки дозволяють програмісту відокремити точки виникнення помилок часу виконання від коду, який ці помилки повинен обробити. Це дозволяє створювати більш надійно працюючі універсальні класи і бібліотеки.

Виняток – це подія, що виникає під час виконання програми і порушує нормальне виконання інструкцій коду. *Механізм генерації та обробки винятків* дозволяє передати інформацію про помилку з місця виникнення у місце, де ця помилка може бути оброблена. Винятки в Java поділяють на синхронні (помилка часу виконання, ситуація, зге-

нерована за допомогою `throw`) і асинхронні (системні, збої віртуальної машини Java). Місце виникнення другої групи винятків виявити досить складно.

Механізм винятків присутній в усіх сучасних мовах об'єктно-орієнтованого програмування. У порівнянні з C++, Java реалізує більш строгий механізм роботи з винятками.

Для генерації винятку використовується оператор `throw`. Після ключового слова `throw` міститься об'єкт классу `java.lang.Throwable`, або класів, похідних від нього. Для програмних винятків найчастіше використовується класс `java.lang.Exception` (похідний від `Throwable`). Використання `Exception` замість `Throwable` дозволяє відокремити власний виняток від системних помилок. Найкраща практика керування винятками – створювати класи, похідні від `Exception`. Такі похідні класи зазвичай відбивають специфіку конкретної програми.

```
class SpecificException extends Exception { }
```

Є також базовий клас для генерації системних помилок – клас `Error`. Класи `Exception` і `Error` мають загальний базовий клас – `Throwable`. Виняток генерується шляхом використання ключового слова `throw`, за яким розташовують об'єкт-виняток. У більшості випадків об'єкт-виняток створюється в точці генерації винятку за допомогою оператора `new`. Наприклад, типове твердження `throw` може виглядати так:

```
void f() . . .
    . . .
    if (/* помилка */) {
        throw new SpecificException();
    }
```

У заголовку функції необхідно перелічити усі типи винятків, які генерує ця функція. Це слід зробити за допомогою ключового слова `throws`:

```
void f() throws SpecificException, AnotherException {
    . . .
    if (/* помилка */) {
        throw new SpecificException();
```

```
    }
    if (/* інша помилка */) {
        throw new AnotherException();
    }
    . . .
}
```

У наведеному нижче прикладі функція `reciprocal()` генерує виняток у випадку ділення на нуль.

```
class DivisionByZero extends Exception { }
class Test {
    double reciprocal(double x) throws DivisionByZero {
        if (x == 0) {
            throw new DivisionByZero();
        }
        return 1 / x;
    }
}
```

На відміну від C++, Java не допускає створення винятків примітивних типів. Дозволені тільки об'єкти, похідні від `Throwable` або `Exception`.

Під час успадкування для перевизначених функцій список винятків повинен зберігатися.

Виняток, який був згенерований у певній частині коду, повинен бути перехоплений в іншій частині. Наприклад, якщо ми хочемо звернутися до функції, яка потенційно може згенерувати виняток, виклик цієї функції поміщають у блок **`try { }`**.

Після блоку **`try`** повинен міститись один чи декілька оброблювачів (блоків **`catch`**). Кожен такий оброблювач відповідає визначеному типу винятку:

```
catch (DivisionByZero d) {
    // обробка винятку
}
catch (Exception ex)  {
    // обробка винятку
}
```

Класи винятків утворюють ієрархію. Під час порівняння типів винятків оброблювач базового типу сприймає також винятки всіх створених від нього типів. Звідси випливає, що оброблювачі похідних типів варто розміщати до оброблювачів базових типів. Припустимо, є така ієрархія класів винятків:

```
class BaseException extends Exception { }
class FileException extends BaseException { }
class FileNotFoundException extends FileException { }
class WrongFormatException extends FileException { }
class MathException extends BaseException { }
class DivisionByZero extends MathException { }
class WrongArgument extends MathException { }
```

Залежно від логіки програми різні типи винятків можна обробляти більш детально:

```
try {
    Exceptions.badFunc();
}
catch (FileNotFoundException ex) {
    // файл не знайдено
}
catch (WrongFormatException ex) {
    // хибний формат
}
catch (FileException ex) {
    // інші помилки, пов’язані з файлами
}
catch (MathException ex) {
    // усі математичні помилки обробляємо разом
}
catch (BaseException ex) {
    // підбираємо всі інші винятки функції badFunc()
}
catch (Exception ex) {
    // про всяк випадок
}
```

Після останнього блоку `catch` можна розмістити блок `finally`. Цей код завжди виконується незалежно від того, виник чи не виник виняток, навіть якщо в якомусь з блоків був здійснений вихід з функції.

На відміну від C++, не можна використовувати `catch (...)` для перехоплення будь-якого винятку. Замість цього можна використовувати перехоплення винятків базових класів:

```
catch (Exception ex) {
    // обробка винятку
}
```

або

```
catch (Throwable ex) {
    // обробка винятку
}
```

Якщо в межах блоку `catch () { }` не можна повністю обробити виняток, його можна передати далі:

```
catch (SomeException ex) {
    // локальна обробка винятку
    throw ex;
}
```

5.2. Задачі

Реалізувати обробку винятків в трьох варіантах:

– використовувати відповідні стандартні винятки;

– використовувати винятки-нащадки від стандартних винятків;

– визначити власні винятки.

Виконати завдання лабораторної роботи 1 (свій варіант).

Задача 5.1

Клас *Vector3d* з конструкторами і обробкою винятків.

Задача 5.2

Клас *Money* з конструкторами і обробкою винятків.

Задача 5.3

Клас *Triangle* з конструкторами і обробкою винятків.

Задача 5.4

Клас *Angle* з конструкторами і обробкою винятків.

Задача 5.5

Клас *Data* з конструкторами і обробкою винятків.

Задача 5.6

Клас *Time* з конструкторами і обробкою винятків.

Задача 5.7

Клас *Account* з конструкторами і обробкою винятків.

Задача 5.8

Клас *Fraction* з конструкторами і обробкою винятків.

Задача 5.9

Клас *Goods* з конструкторами і обробкою винятків.

Задача 5.10

Клас *Payment* з конструкторами і обробкою винятків.

ЛАБОРАТОРНА РОБОТА 6
УЗАГАЛЬНЕННЯ

Мета лабораторної роботи – навчитися створювати та використовувати шаблони функцій та класів.

6.1. Теоретичні основи

Парадигма *узагальненого програмування* передбачає опис правил зберігання даних і алгоритмів у загальному вигляді незалежно від конкретних типів даних. Конкретні типи даних, над якими виконуються дії, специфікуються пізніше. Механізми розділення структур даних і алгоритмів, а також формування абстрактних описів вимог до даних, визначаються по-різному в різних мовах програмування.

Для реалізації узагальненого програмування в Java використовуються узагальнення – спеціальна мовна конструкція, яка з'явилася у синтаксисі мови починаючи з версії Java 5.

Узагальнення – конструкція, що включає в себе параметр класу або функції, який містить додаткову інформацію про тип елементів та інших даних. Цей параметр беруть у кутові дужки. Узагальнення на-

дають можливість створення та використання структур даних, безпечних з точки зору типів. Класи, опис яких містить такий параметр, мають назву *узагальнених*. Під час створення об'єкта узагальненого типу у кутових дужках вказують імена реальних типів. Можна використовувати тільки типи-посилання.

```
public class Pair<T> {
  T first, second;
  public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
  public static void main(String[] args) {
   Pair<String> p = new Pair<String>("Прізвище", "Ім\'я");
   String s = p.first; // Отримуємо рядок без приведення типів
   Pair<Integer> p1 = new Pair<Integer>(1, 2);
   int i = p1.second;
    }
}
```

Примітка:Java версії 7 і вище дозволяє не повторювати фактичний параметр узагальнення після імені конструктора. Наприклад:

```
Pair<Integer> p1 = new Pair<>(1, 2);
```

Якщо ми намагаємось додати до пари дані різних типів, компілятор згенерує помилку. Помилковою є також спроба явно перетворити тип:

```
Pair<String> p = new Pair<String>("1", "2");
Integer i = (Integer) p.second; // Помилка компіляції
```

Тип даних з параметром у кутових дужках (наприклад, `Pair<String>`) має назву *параметризованого типу*.

Узагальнення по зовнішньому представленню і використанню аналогічні шаблонам C++. Але на відміну від шаблонів C++, існує не декілька різних типів `Pair`, а один. Фактично у полях класу зберігаються посилання на `Object`. Інформація про тип параметрів використовується компілятором для контролю та автоматичного приведення ти-

пів у вихідному тексті.

Окрім узагальнених класів, можна створювати *узагальнені інтерфейси*. Параметр може бути використаний в описі функцій, оголошених в інтерфейсі. Під час їх реалізації замість параметра узагальнення використовують деякий тип-посилання. Наприклад:

```
interface Function<T> {
    T func(T x);
}
class DoubleFunc implements Function<Double> {
    @Override
    public Double func(Double x) {
        return x * 1.5;
    }
}
class IntFunc implements Function<Integer> {
    @Override
    public Integer func(Integer x) {
        return x % 2;
    }
}
```

Java також дозволяє створювати *узагальнені функції* всередині як узагальнених, так і звичайних (неузагальнених) класів:

```
public class ArrayPrinter {
    public static<T> void printArray(T[] a) {
        for (T x : a) {
            System.out.print(x + "\t");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        String[] as = {"First", "Second", "Third"};
        printArray(as);
        Integer[] ai = {1, 2, 4, 8};
        printArray(ai);
    }
```

```
}
```

Як видно з прикладу, виклик узагальненої функції не вимагає явного визначення типу. Іноді таке визначення необхідне, наприклад, коли в функції немає параметрів узагальненого типу. Якщо це статична функція, необхідно явно вказувати її клас. Наприклад:

```
public class TypeConverter {
    public static <T>T convert(Object object) {
        return (T) object;
    }
    public static void main(String[] args) {
        Object o = "Some Text";
        String s = TypeConverter.<String>convert(o);
        System.out.println(s);
    }
}
```

Рекомендованими іменами формальних параметрів є імена з однієї великої літери. Узагальнення може мати два і більше параметрів.

Над даними типу параметру узагальнення можна здійснювати тільки дії, дозволені для об'єктів класу `Object`. Іноді для розширення функціональності бажаною є конкретизація типу. Наприклад, ми хочемо викликати методи, оголошені у певному класі або інтерфейсі. Тоді можна застосувати такий синтаксис опису параметру: `<T extends SomeBaseType>` або `<T extends FirstType & SecondType>` тощо. Слово **`extends`** використовують як для класів, так і для інтерфейсів.

Синтаксис узагальнень передбачає використання так званих масок (wildcard, символ '?'). Маска застосовується, наприклад, для опису посилань на поки невідомий тип. Використання масок робить узагальнені класи та функції більш сумісними. Маска надає альтернативний спосіб створення узагальнених функцій. Такі функції самі по собі не є узагальненими, але містять аргументи узагальнених типів.

6.2. Задачі

Виконати завдання лабораторної роботи 7 (свій варіант) за наступними умовами.

Задача 6.1

Реалізувати стек у вигляді узагальнення однозв'язного списку.

Задача 6.2

Реалізувати чергу у вигляді узагальнення двозв'язного списку.

Задача 6.3

Реалізувати дек у вигляді узагальнення двозв'язного списку.

Задача 6.4

Реалізувати чергу у вигляді узагальнення однозв'язного списку.

Задача 6.5

Реалізувати стек у вигляді узагальнення однозв'язного списку.

Задача 6.6

Реалізувати дек у вигляді узагальнення двозв'язного списку.

Задача 6.7

Реалізувати чергу у вигляді узагальнення однозв'язного списку.

Задача 6.8

Реалізувати чергу у вигляді узагальнення двозв'язного списку.

Задача 6.9

Реалізувати дек у вигляді узагальнення двозв'язного списку.

Задача 6.10

Реалізувати чергу у вигляді узагальнення двозв'язного списку.

ЛАБОРАТОРНА РОБОТА 7
КОНТЕЙНЕРИ

Мета лабораторної роботи – опанувати написання програм з використанням контейнерів.

7.1. Теоретичні основи

Часто виникає необхідність у створенні так званих класів-контейнерів – таких, які містять об'єкти довільних типів. При цьому над елементами контейнерів необхідно виконувати деякі однотипні дії. Код для обробки об'єктів різних типів може виглядати практично однаково. Наприклад, якщо для різних типів даних потрібно реалізувати

алгоритми на кшталт швидкого сортування або способи обробки таких структур даних, як зв'язаний список або бінарне дерево. У таких випадках код однаковий для всіх типів об'єктів.

Черга в широкому сенсі є структурою даних, яку заповнюють поелементно, та отримують з неї об'єкти за певним правилом. У вузькому сенсі цим правилом є "першим прийшов – першим вийшов" (FIFO, First In – First Out). У черзі, організованій за принципом FIFO, додавання елемента можливо лише в кінець черги, отримання – тільки з початку черги.

У бібліотеці контейнерів черга представлена інтерфейсом `Queue`.

Для реалізації черги найзручніше використовувати клас `LinkedList`, який реалізує інтерфейс `Queue` (див. приклад 7.1).

Інтерфейс `Deque` (дек, double-ended-queue) надає можливість додавати й видаляти елементи з обох кінців. Кожна з пар представляє відповідно функцію, яка генерує виняток, і функцію, яка повертає спеціальне значення. Є також методи, що дозволяють видалити перше або останнє входження заданого елемента.

Для реалізації інтерфейсу можна використовувати спеціальний класс `ArrayDeque`, або зв'язний список (`LinkedList`).

Стек – це структура даних, організована за принципом "останній прийшов – перший вийшов" (LIFO, last in – first out). Можливі три операції зі стеком: додавання елементу (push), видалення елементу (pop) і читання головного елемента (peek).

Стеки часто використовуються в різних алгоритмах. Зокрема, за допомогою стеку в деяких задачах можна позбутися рекурсії.

Множина – це колекція, що не містить однакових елементів. Три основних реалізації інтерфейсу `Set` – `HashSet`, `LinkedHashSet` і `TreeSet`. Як і списки, множини є узагальненими типами. Класи `HashSet` і `LinkedHashSet` використовують хеш-коди для ідентифікації елемента. Клас `TreeSet` використовує двійкове дерево для збереження елементів і гарантує їх певний порядок. Метод `add()` додає елемент до множини і повертає **`true`**, якщо елемент раніше був відсутній. В іншому випадку елемент не додається, а метод `add()` повертає **`false`**. Усі

елементи множини видаляються за допомогою методу `clear()`. Метод `remove()` видаляє зазначений елемент множини, якщо такий є. Метод `contains()` повертає **`true`**, якщо множина містить зазначений елемент.

Асоціативні масиви можуть зберігати пари посилань на об'єкти. Асоціативні масиви теж є узагальненими типами. Асоціативні масиви у Java представлені узагальненим інтерфейсом `Map`, який реалізовано, зокрема, класом `HashMap`. Інтерфейс `SortedMap`, похідний від `Map`, вимагає впорядкованого за ключем зберігання пар. Інтерфейс `NavigableMap`, що з'явився в Java SE 6, розширює `SortedMap` і додає нові можливості пошуку за ключем. Цей інтерфейс реалізовано класом `TreeMap`.

Кожне значення (об'єкт), яке зберігається в асоціативному масиві, зв'язується з конкретним значенням іншого об'єкта (ключа). Метод `put(key, value)` додає значення (`value`) і асоціює з ним ключ (`key`). Якщо асоціативний масив раніше містив пари з вказаним ключем, нове значення заміщає старе. Метод `put()` повертає попереднє значення, пов'язане з ключем, або **`null`**, якщо ключ був відсутній. Метод `get(Object key)` повертає по об'єкт за заданим ключем. Для перевірки знаходження ключа та значення застосовуються методи `containsKey()` і `containsValue()`.

На логічному рівні можна представити асоціативний масив через три допоміжні колекції:

`keySet` – множина значень ключа;

`values` – список значений;

`entrySet` – множина пар ключ-значення.

Через відповідні функції `keySet()`, `values()` та `entrySet()` можна здійснювати певні дії, в першу чергу послідовне проходження елементів.

7.2. Задачі

Наступні завдання виконати двома способами:

– з використанням масивом, з виділенням пам'яті, розмір заздале-

гідь не відомий. Елементи можуть вставлятися в будь-яке місце масиву; може бути видалений будь-який елемент або група елементів. Реалізувати операцію індексування.

– з використанням двохзв'язного списку. Замість операції індексування реалізувати ітератор як набір методів списку.

Для роботи з контейнерами реалізувати методи: додавання та видалення елементу, введення та виведення всього контейнеру.

Задача 7.1

Створити клас «Відділ кадрів». Поля: прізвище співробітника, ім'я, по батькові, посаду, стаж роботи, оклад.

Задача 7.2

Створити клас «Червона книга». Поля: вид тварини, рід, сімейство, місце проживання, чисельність популяції.

Задача 7.3

Створити клас «Бібліотека». Поля: автор книги, назва, рік видання, код УДК, ціна, кількість в бібліотеці.

Задача 7.4

Створити клас «Супутники планет». Поля: назву, назву планети-господаря, рік відкриття, діаметр, період обертання.

Задача 7.5

Створити клас «Радіодеталі». Поля: позначення, тип, номінал, кількість на схемі, позначення можливого замінника.

Задача 7.6

Створити клас «Побут студентів». Поля: прізвище студента, ім'я, по батькові, факультет, розмір стипендії, число членів сім'ї.

Задача 7.7

Створити клас «Міський транспорт». Поля: вид транспорту, номер маршруту, початкова зупинка, кінцева зупинка, час у дорозі.

Задача 7.8

Створити клас «Спортивні змагання». Поля: прізвище спортсмена, ім'я, команда, вид спорту, заліковий результат, штрафні очки.

Задача 7.9

Створити клас «Оптова база». Поля: назву товару, кількість на

складі, вартість одиниці, назва постачальника, термін поставки.

Задача 7.10

Створити клас «Зоопарк». Поля: вид тварини, кличка, вік, категорія рідкості, вага, добовий раціон м'яса, овочів, молока.

ЛАБОРАТОРНА РОБОТА 8
ГРАФІЧНИЙ ІНТЕРФЕЙС

Мета лабораторної роботи – освоїти написання підпрограм.

8.1. Теоретичні основи

Інтерфейс користувача – це набір технічних та програмних засобів, за допомогою яких людина взаємодіє з комп'ютером. Далі йтиметься про програмні засоби інтерфейсу комп'ютеру.

Інтерфейс командного рядку – це метод взаємодії з програмою за допомогою інтерпретатору команд, які користувач уводить, як правило, у текстовому режимі, або у спеціальному консольному вікні. *Графічний інтерфейс користувача* (Graphical user interface, GUI) дає можливість користувачеві взаємодіяти з комп'ютером за допомогою графічних елементів управління (вікон, піктограм, меню, кнопок, списків тощо) та технічних пристроїв позиціонування, таких як маніпулятор "миша" Програми, які реалізують цей тип інтерфейсу, мають назву *застосунків графічного інтерфейсу користувача*.

Реалізація застосунків графічного інтерфейсу користувача базується на механізмі отримання та обробки подій. Уся програма складається з ініціалізації (реєстрації візуальних елементів управління) та основного циклу отримання та обробки подій. Події – це переміщення або натискання кнопок миші, клавіатурне введення, тощо. Кожний зареєстрований візуальний елемент управління може отримувати події, які до нього стосуються, та виконувати функції обробки цих подій.

Наразі стандартними засобами розробки додатків графічного інтерфейсу користувача в Java є бібліотеки AWT і Swing, а також платформа JavaFX, засоби якої є альтернативою Swing. Крім того, різні розробники надають альтернативні нестандартні бібліотеки, такі як Qt

Jambi, Standard Window Toolkit (SWT), XML Window Toolkit (XWT). Дві останні бібліотеки, поряд з AWT і Swing, підтримуються Eclipse.

Бібліотека `javax.swing` пропонує розробникові низку стандартних класів, які можна використовувати для проектування графічного інтерфейсу користувача. Ця бібліотека розширила попередню менш вдалу бібліотеку AWT (Abstract Window Toolkit) засоби якої використовує для обробки подій, роботи з графікою тощо. Для ідентифікації приналежності до бібліотеки `javax.swing` до імен класів візуальних компонентів додана літера `J` (наприклад, `JButton`, `JPanel` і т. д.).

На відміну від компонентів AWT, компоненти Swing є "легковагими" (lightweight). Це означає, що компоненти Swing використовують засоби Java для відображення елементів графічного інтерфейсу користувача на поверхні вікна, без використання компонентів операційної системи.

Як і більшість бібліотек графічного інтерфейсу користувача, бібліотека `javax.swing` підтримує концепцію головного вікна застосунку. Це головне вікно створюється як об'єкт класу `JFrame`, або похідного від нього. Далі до головного вікна додають візуальні компоненти – мітки (`JLabel`), кнопки (`JButton`), рядки введення (`JTextField`) тощо.

Для того, щоб створити найпростішу програму графічного інтерфейсу користувача, необхідно створити новий клас з функцією `main()`, а потім у вихідному тексті додати твердження **`import`**:

```
import javax.swing.*;
```

У функції `main()` створюємо нове вікно та вказуємо його заголовок:

```
public class HelloWorldSwing {
    public static void main(String[] args) {
       JFrame frame = new JFrame("Привіт");
       . . .
    }
}
```

Далі додаємо нову мітку до компоненту, який відповідає за вміст вікна. Створюємо новий об'єкт типу `JLabel`:

```
frame.getContentPane().add(new JLabel("Привіт, світ!"));
```

Візуальні компоненти бібліотеки Swing успадковуються від класу `javax.swing.JComponent`, спадкоємця класу `java.awt.Container`. У свою чергу, цей клас є спадкоємцем `java.awt.Component`. Клас `java.awt.Component` – базовий клас, який визначає відображення на екрані і поведінку кожного елемента інтерфейсу під час взаємодії з користувачем. Методи класу, що відповідають за управління подіями, дозволяють задати розмір, колір, шрифт та інші атрибути елементів управління.

8.2. Задачі

Задача 8.1

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма зверху вниз на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від'ємними.

Задача 8.2

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма зліва направо на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від'ємними.

Задача 8.3

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від'ємними.

Задача 8.4

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми» і 3–4 кнопки установки стилю зафарбо-

вування. Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від'ємними.

Задача 8.5

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма з прямокутників, відцентрувати щодо середини форми вибраним кольором. Числа можуть бути тільки додатними.

Задача 8.6

Дано текстовий файл з десятьма числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма з прямокутників, необхідно відцентрувати щодо середини форми за типом дитячої пірамідки, тобто масив повинен бути відсортований. Числа можуть бути тільки додатними.

Задача 8.7

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма знизу вгору на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути і від'ємними.

Задача 8.8

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма зліва направо на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути тільки додатними.

Задача 8.9

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми». Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що

числа можуть бути тільки додатними.

Задача 8.10

Дано текстовий файл з числами. На формі розмістити меню з наступних пунктів – «Прочитати з файлу», «Вибір кольору», «Побудова діаграми», «Очищення форми» і 3–4 кнопки установки стилю зафарбовування. Будується стовпчикова діаграма справа наліво на весь розмір форми вибраним кольором. Передбачити, що числа можуть бути тільки додатними.

ПРИКЛАДИ ДО ЛАБОРАТОРНИХ РОБІТ

Лабораторна робота 1

Приклад 1.1. Припустимо, необхідно реалізувати програму для роботи з точками на площині. Точка задається парою дійсних чисел. Клас для представлення точки повинен реалізовувати такі методи: обчислення відстані від точки до початку координат, обчислення відстані між двома точками. Друга функція може бути реалізована як статичний метод. Програма може бути такою:

```
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double distance() {
        return Math.sqrt(x * x + y * y);
    }
    public static double distance(Point p1, Point p2) {
        return Math.sqrt((p1.x - p2.x) * (p1.x - p2.x) +
                         (p1.y - p2.y) * (p1.y - p2.y));
    }
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        System.out.println(p1.distance());
        Point p2 = new Point(4, 5);
        System.out.println(distance(p1, p2));
```

Лабораторна робота 2

Приклад 2.1.

Статичні вкладені класи мають доступ тільки до статичних елементів обхопних класів. Об’єкти таких класів можуть бути створені без створення об’єктів обхопних класів:

```
class Outer {
int k = 100;
```

```
    static int m = 200;
    static class Inner {
    void show() {
    // k недоступно
    System.out.println(m);
    }
       }
}
    public class Test {
    public static void main(String[] args) {
    Outer.Inner inner = new Outer.Inner();
    inner.show();
    }
}
```

Статичні вкладені класи можуть містити свої статичні елементи, у тому числі свої вкладені статичні і нестатичні класи.

Лабораторна робота 3

Приклад 3.1. Припустимо, необхідно розробити ієрархію класів "Регіон" – "Населений район" – "Країна". Окремі класи цієї ієрархії можуть стати базовими для інших класів (наприклад "Незаселений острів", "Національний парк", "Адміністративний район", "Автономна республіка" і т.д.). Ієрархію класів можна доповнити класами "Місто" і "Острів". Доцільно в кожен клас додати конструктор, який ініціалізує усі поля. Можна також створити масив посилань на різні об'єкти ієрархії і для кожного об'єкта вивести на екран рядок даних про нього.

Для того, щоб одержати рядкове представлення об'єкта, необхідно перекрити функцію `toString()`. Можна запропонувати таку ієрархію класів.

```
import java.util.*;
// Ієрархія класів
class Region {
    private String name;
    private double area;
    public Region(String name, double area) {
```

```
        this.name = name;
        this.area = area;
    }
    public String getName() {
        return name;
    }
    public double getArea() {
        return area;
    }
    public String toString() {
        return "Регіон " + name + ".\tТериторія    " + area +
" кв.км.";
    }
}
class PopulatedRegion extends Region {
    private int population;
    public PopulatedRegion(String name, double area, int pop-
ulation) {
        super(name, area);
        this.population = population;
    }
    public int getPopulation() {
        return population;
    }
    public int density() {
        return (int) (population / getArea());
    }
    public String toString() {
        return   "Населений   регіон   "   +   getName()   +
".\tТериторія   " + getArea() +
                " кв.км.   \tНаселення " + population +
                " чол.\tЩільність населення " + density() + "
чол/кв.км.";
    }
}
class Country extends PopulatedRegion {
```

```
    private String capital;
    public Country(String name, double area, int population,
String capital) {
        super(name, area, population);
        this.capital = capital;
    }
    public String getCapital() {
        return capital;
    }
    public String toString() {
        return "Країна " + getName() + ".\tТериторія    " +
getArea() +
                " кв.км.   \tНаселення " + getPopulation() +
                " чол.\tЩільність населення " + density() +
                " чол/кв.км.\tСтолиця " + capital;
    }
}
class City extends PopulatedRegion {
    private int boroughs; // Кількість районів
    public City(String name, double area, int population, int
boroughs) {
        super(name, area, population);
        this.boroughs = boroughs;
    }
    public int getBoroughs() {
        return boroughs;
    }
    public String toString() {
        return "Місто " + getName() + ".\tТериторія    " +
getArea() +
                " кв.км.   \tНаселення " + getPopulation() +
                " чол.\tЩільність населення " + density() +
                " чол/кв.км.\tРайонів – " + boroughs;
    }

}
```

```
class Island extends PopulatedRegion {
    private String sea;
    public Island(String name, double area, int population,
String sea) {
        super(name, area, population);
        this.sea = sea;
    }
    public String getSea() {
        return sea;
    }
   public String toString() {
     return "Острів" + getName() + ".\tТериторія"+ getArea()
+
              " кв.км.   \tНаселення " + getPopulation() +
             " чол.\tЩільність населення " + density() +
             " чол/кв.км.\tМоре – " + sea;
    }
}
public class Regions {
    public static void main(String[] args) {
Region[] a = { new City("Київ", 839, 2679000, 10),
             new    Country("Україна",    603700,    46294000,
"Київ"),
             new City("Харків", 310, 1461000, 9),
             new Island("Зміїний", 0.2, 30, "Чорне") };
        for (Region region : a) {
            System.out.println(region);
        }
    }
}
```

Лабораторна робота 4

Приклад 4.1. Наприклад, абстрактний клас `Shape` (геометрична фігура) реалізує поля і методи, що можуть бути використані різними похідними класами. До таких полів можна, наприклад, віднести поточну

позицію і метод переміщення по екрану `moveTo()`. У класі `Shape` також оголошені абстрактні методи, такі як `draw()`, що повинні бути реалізовані у всіх похідних класах, але по-різному. Усталена реалізація не має сенсу.

```
abstract class Shape {
    int x, y;
    . . .
    void moveTo(int newX, int newY) {
        . . .
    }
    abstract void draw();
}
```

Конкретні класи, створені від `Shape`, такі як `Circle` або `Rectangle`, визначають реалізацію методу `draw()`.

```
class Circle extends Shape
{
    void draw() {
        . . .
    }
}

class Rectangle extends Shape {
    void draw() {
        . . .
    }
}
```

Приклад 4.2. У Java 5 `Comparable` – це узагальнений інтерфейс. Завдяки узагальненням у функціях, які оголошені в інтерфейсі, замість параметрів типу `Object`, можна використовувати параметри інших типів. В нашому випадку функція `compareTo()` повинна приймати аргумент типу елементу масиву. Можна самостійно створити клас, що реалізує інтерфейс `Comparable`. Наприклад, масив прямокутників сортується за площею:

```
class Rectangle implements Comparable<Rectangle> {
    private double width, height;
```

```
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double area() {
        return width * height;
    }
    public double perimeter() {
        return 2 * (width + height);
    }

    @Override
    public int compareTo(Rectangle rect) {
        return Double.compare(area(), rect.area());
    }
    @Override
    public String toString() {
        return "[" + width + ", " + height + ", area = " +
area() + ", perimeter = " + perimeter() + "]";
    }
}
public class SortRectangles {
    public static void main(String[] args) {
        Rectangle[] a = {new Rectangle(2, 7), new Rectan-
gle(5, 3), new Rectangle(3, 4)};
        java.util.Arrays.sort(a);
        System.out.println(java.util.Arrays.toString(a));
    }
}
```

У наведеному прикладі використовується статична функція `compare()` класу `Double`. Ця функція повертає значення, необхідні методу `sort()`.

Лабораторна робота 5

Приклад 5.1. Припустимо, необхідно створити програму, яка здій-

снює копіювання текстових файлів рядок за рядком. Імена файлів задаються аргументами командного рядка.

```
import java.io.*;
public class TextFileCopy {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Необхідні аргументи!");
            return;
        }
        try (BufferedReader in = new BufferedReader
             (new FileReader(args[0]));
             PrintWriter out = new PrintWriter
             (new FileWriter(args[1]))) {
            String line;
            while ((line = in.readLine()) != null) {
                out.println(line);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Лабораторна робота 6

*Приклад 6.1. М*ожна створити узагальнену функцію обчислення середнього арифметичного в масиві деяких числових значень. Стандартні класи `Double`, `Float`, `Integer`, `Long` та інші класи-обгортки числових даних мають загальний абстрактний базовий клас `java.lang.Number`, що декларує, зокрема, метод `doubleValue()`, який дозволяє, отримати число, що зберігається в об'єкті, у вигляді значення типу **`double`**. Цей факт можна використовувати для обчислення середнього арифметичного. Створена функція може працювати з масивами чисел різних типів:

```
public class AverageTest {
    public static<E extends Number> double average(E[] arr) {
```

```
        double result = 0;
        for (E elem : arr) {
            result += elem.doubleValue();
        }
        return result / arr.length;
    }
    public static void main(String[] args) {
        Double[] doubles = { 1.0, 1.1, 1.5 };
        System.out.println(average(doubles)); // 1.2
        Integer[] ints = { 10, 20, 3, 4 };
        System.out.println(average(ints));    // 9.25
    }
}
```

Приклад 6.2. Наведемо приклад створення узагальненого класу, який зберігає масив елементів певного типу:

```
public class MyArray<T> {
    private T[] arr;
    public MyArray(T... arr) {
        this.arr = arr;
    }
    public int size() {
        return arr.length;
    }
    public T get(int i) {
        return arr[i];
    }
    public void set(int i, T t) {
        arr[i] = t;
    }
    public void printAll() {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

В іншому класі здійснюємо тестування:

```
public class TestClass {
    public static void main(String[] args) {
        MyArray<String> a = new MyArray<>(new String[]{ "1",
"2" });
        String s = a.get(a.size() - 1);
        System.out.println(s);     // 2
        a.set(1, "New");
        a.printAll();              // 1 New
    }
}
```

Не можна створювати об’єктів узагальнених типів:

```
T arr = new T[10]; // Помилка!
```

Приклад 6.3. У нашому прикладі цю проблему можна розв’язати за допомогою посилань на клас `Object`. Крім того, для зручності використання конструктор можна реалізувати як функцію зі змінним числом параметрів. Корисною також буде функція додавання елемента в кінець масиву. Альтернативна реалізація може бути такою:

```
public class MyArray<T> {
    private Object[] arr = {};
    public MyArray(T... arr) {
        this.arr = arr;
    }
    public MyArray(int size) {
        arr = new Object[size];
    }
    public int size() {
        return arr.length;
    }
    public T get(int i) {
        return (T)arr[i];
    }
    public void set(int i, T t) {
        arr[i] = t;
    }
```

```
    public void add(T t) {
        Object[] temp = new Object[arr.length + 1];
        System.arraycopy(arr, 0, temp, 0, arr.length);
        arr = temp;
        arr[arr.length - 1] = t;
    }
    public void printAll() {
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

В іншому класі здійснюємо тестування:

```
public class TestClass {
    public static void main(String[] args) {
        MyArray<String> a = new MyArray<>("1", "2");
        String s = a.get(a.size() - 1);
        System.out.println(s);     // 2
        a.set(1, "New");
        a.printAll();              // 1 New
        MyArray<Double> b = new MyArray<>(3);
        b.set(0, 1.0);
        b.set(1, 2.0);
        b.set(2, 4.0);
        b.add(8.0);
        b.printAll();
    }
}
```

Функціональність класу можна розширити методами додавання нового елемента всередині масиву, видалення існуючого тощо.

У наведеному нижче прикладі використовується раніше створений клас `MyArray`, зокрема, його конструктор зі змінним числом параметрів:

```
public class GenericArrayPrinter {
```

```
    public static void printGenericArray(MyArray<?> a) {
        for (int i = 0; i < a.size(); i++) {
            System.out.print(a.get(i) + "\t");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        MyArray<String> arr1 = new MyArray("First", "Second",
"Third");
        printGenericArray(arr1);
        MyArray<?> arr2 = new MyArray(1, 2, 3); // MyArray<?>
                                  //                    замість
MyArray<Integer>
        printGenericArray(arr2);
    }
}
```

Можна обмежити використання типу параметра функції певними похідними класами, наприклад, `MyArray<? super String>`. Тоді використання списку `MyArray<Integer>` неможливе.

Лабораторна робота 7

Приклад 7.1. Для реалізації черги найзручніше використовувати клас `LinkedList`, який реалізує інтерфейс `Queue`.

```
import java.util.LinkedList;
import java.util.Queue;
public class SimpleQueueTest {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("First");
        queue.add("Second");
        queue.add("Third");
        queue.add("Fourth");
        String s;
        while ((s = queue.poll()) != null) {
            System.out.print(s + " ");
```

```
        }
    }
}
```

Приклад 7.2. Клас `PriorityQueue` впорядковує елементи відповідно до компаратору (об'єкта класу, що реалізує інтерфейс `Comparator`), заданого в конструкторі як параметр. Якщо об'єкт створити за допомогою конструктора без параметрів, елементи будуть упорядковані в природному порядку (для чисел – за зростанням, для рядків – за абеткою).

```
import java.util.PriorityQueue;
import java.util.Queue;
public class PriorityQueueTest {
    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<>();
        queue.add("First");
        queue.add("Second");
        queue.add("Third");
        queue.add("Fourth");
        String s;
        while ((s = queue.poll()) != null) {
            System.out.print(s + " ");
        }
    }
}
```

У JRE 1.1 стек представлений класом `Stack`. Наприклад:

```
import java.util.Stack;
public class StackTest {

    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("First");
        stack.push("Second");
        stack.push("Third");
        stack.push("Fourth");
        String s;
```

```
        while (!stack.isEmpty()) {
            s = stack.pop();
            System.out.print(s + " "); // Fourth Third Second
First
        }
    }
}
```

Приклад 7.3. Цей клас в даний час не рекомендований до використання. Замість нього можна використовувати інтерфейс `Deque`, який оголошує аналогічні методи.

```
import java.util.ArrayDeque;
import java.util.Deque;
public class AnotherStackTest {
    public static void main(String[] args) {
        Deque<String> stack = new ArrayDeque<>();
        stack.push("First");
        stack.push("Second");
        stack.push("Third");
        stack.push("Fourth");
        String s;
        while (!stack.isEmpty()) {
            s = stack.pop();
            System.out.print(s + " ");
        }
    }
}
```

Приклад 7.4. У наведеному нижче прикладі до множини цілих чисел додається десять випадкових значень у діапазоні від –9 до 9:

```
import java.util.*;
public class SetOfIntegers {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        Random random = new Random();
        for (int i = 0; i < 10; i++) {
            Integer k = random.nextInt() % 10;
```

```
            set.add(k);
        }
        System.out.println(set);
    }
}
```

Результуюча множина як правило містить менш, ніж 10 чисел, оскільки окремі значення можуть повторюватися. Оскільки ми використовуємо `TreeSet`, числа зберігаються та виводяться в упорядкованому (за зростанням) вигляді. Для того, щоб додати саме десять різних чисел, програму можна модифікувати, наприклад із застосуванням циклу **`while`** замість **`for`**:

```
while (set.size() < 10) {
    . . .
}
```

Можна створити масив, який містить копії елементів множини. В такий спосіб можна звертатися до елементів за індексом. Наприклад, так можна вивести елементи множини в зворотному порядку:

```
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, 4));
Object[] arr = set.toArray();
for (int i = set.size() - 1; i >= 0; i--) {
    System.out.println(arr[i]);
}
```

Приклад 7.5. Оскільки множина може містити тільки різні елементи, її можна використати для підрахунку різних слів, літер, цифр тощо – створюється множина та викликається метод `size()`. Застосовуючи `TreeSet`, можна виводити слова та літери в алфавітному порядку. У наведеному нижче прикладі вводиться речення та виводяться всі різні літери речення (не враховуючи роздільників) в алфавітному порядку:

```
import java.util.*;
public class Sentence {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Функція nextLine() читає рядок до кінця:
```

```
        String sentence = scanner.nextLine();
        // Створюємо множину роздільників:
        Set<Character> delimiters = new HashSet<Character>(
            Arrays.asList(‘ ‘, ‘.’, ‘,’, ‘:’, ‘;’, ‘?’, ‘!’,
‘-’, ‘(‘, ‘)’, ‘\"‘));
        // Створюємо множину літер:
        Set<Character> letters = new TreeSet<Character>();
        // Додаємо всі літери крім роздільників:
        for (int i = 0; i < sentence.length(); i++) {
            if (!delimiters.contains(sentence.charAt(i))) {
                letters.add(sentence.charAt(i));
            }
        }
        System.out.println(letters);
    }
}
```

Приклад 7.6. Порядок сортування елементів `TreeSet` можна задати, реалізувавши інтерфейс `Comparable`, або передавши в конструктор `TreeSet` посилання на об’єкт класу, який реалізує інтерфейс `Comparator`. Наприклад, так можна відсортувати дерево у зворотному порядку:

```
import java.util.*;
public class CompTest {
    public static void main(String args[]) {
        TreeSet<String> ts = new TreeSet<String>(new Compara-
tor<String>()
        {
            @Override
            public int compare(String s1, String s2) {
                return s2.compareTo(s1);
            }
        });
        ts.add("C");
        ts.add("E");
        ts.add("D");
        ts.add("B");
        ts.add("A");
```

```
        ts.add("F");
        for (String element : ts)
            System.out.print(element + " ");
        }
    }
}
```

Приклад 7.7. У наведеному нижче прикладі обчислюється кількість входжень різних слів у речення. Слова і відповідні кількості зберігаються в асоціативному масиві. Використання класу `TreeMap` гарантує алфавітний порядок слів (ключів).

```
import java.util.*;
public class WordsCounter {
    public static void main(String[] args) {
        Map<String, Integer> m = new TreeMap<String, Inte-
ger>();
        String s = "the first men on the moon";
        StringTokenizer st = new StringTokenizer(s);
        while (st.hasMoreTokens()) {
            String word = st.nextToken();
            Integer count = m.get(word);
            m.put(word, (count == null) ? 1 : count + 1);
        }
        for (String word : m.keySet()) {
            System.out.println(word + " " + m.get(word));
        }
    }
}
```

Використання `keySet()` передбачає окремий пошук кожного значення за ключем. Більш рекомендованим є обхід асоціативного масиву через множину пар:

```
for (Map.Entry<?, ?> entry : m.entrySet())
    System.out.println(entry.getKey()   +   "   "   +   en-
try.getValue());
```

Тут метод `entrySet()` дозволяє одержати представлення асоціативного масиву у вигляді колекції `Set`.

Порядок сортування елементів `TreeMap` також можна змінити, вказавши як параметр конструктора `TreeMap` об'єкт класу, який реалізує інтерфейс `Comparator`, або задавши ключ як об'єкт класу, який реалізує інтерфейс `Comparable.`

Лабораторна робота 8

Приклад 8.1. За допомогою методу `pack()` здійснюється припасування розмірів вікна. Після виклику функції `setVisible(true)` вікно з'являється на екрані. Можна навести весь текст програми:

```
import javax.swing.*;
public class HelloWorldSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Привіт");
        frame.getContentPane().add(new JLabel("Привіт, світ!"));
        frame.pack();
        frame.setVisible(true);
    }
}
```

Далі до програми можна додавати інші візуальні компоненти та оброблювачі подій.

Список літератури

1. Шилдт Г. Java 8. Полное руководство; 9-е изд. / Г. Шилдт. – Москва: ООО "И.Д.Вильямс", 2015. – 1376 с.

2. Дейтел Х. М. Как программировать на Java. Книга 2. Файлы, сети, базы даннях / Х. М. Дейтел. – Москва: Бином, 2006. – 672 с.

3. Хорстманн К. С. Java 2. Библиотека профессионала, том 1. Основы, 7-е изд. / К. С. Хорстманн, Г. Корнелл. – Москва: Издательский дом "Вильямс", 2007. – 896 с.

4. Хорстманн К. С. Java 2. Библиотека профессионала, том 2. Тонкости программирования, 7-е изд. / К. С. Хорстманн, Г. Корнелл. – М.: Издательский дом "Вильямс", 2007. – 1168 стр. с.

5. Шилдт Г. Искусство программирования на Java / Г. Шилдт, Д. Холмс. – Москва: Издательский дом "Вильямс", 2005. – 336 с.

6. Эккель Б. Философия Java. Библиотека программиста. 4-е изд. / Б. Эккель. – Санкт-Петербург: Питер, 2009. – 640 с.

7. Блинов И. Н. Java. Промышленное программирование : практ. пособие / И. Н. Блинов, В. С. Романчик – Минск: УниверсалПресс, 2007. – 704 с.

8. Копитко М. Ф. Основи програмування мовою Java: Тексти лекцій / М. Ф. Копитко, К. С. Іванків. – Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002. – 83 с.

9. Брнакевич І. Є. Програмування мовою Java: використання фундаментальних класів: Тексти лекцій / І. Є. Брнакевич, П. П. Вагін. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002. – 75 с.

10. Дудзяний І. М. Об'єктно-орієнтоване моделювання програмних систем: Навч. посібник. / І. М. Дудзяний. – Львів: Видавничий Центр ЛНУ імені Івана Франка, 2007. – 108 с.

11. Блинов И. Н. Java. Промышленное программирование : практ. пособие / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2007. – 704 с.

12. Об'єктно-орієнтоване програмування (частина 1). Розробник курсу Л.В. Іванов. http://www.iwanoff.inf.ua/oop_ua/index.html

ЗМІСТ

Навчальне видання

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ C#

Методичні вказівки до лабораторних робіт
з курсу «Об'єктно-орієнтоване програмування»

для студентів спеціальності
122 – Комп'ютерні науки
126 – Інформаційні системи та технології

Укладачі **НІКУЛІНА** Олена Миколаївна
ІВАНОВ Лев Вадимович
КОЦЮБА Ніна Вікторівна

Відповідальний за випуск В. В. Московленко

Роботу до видання рекомендував Годлевський

В авторській редакції

План 2021 р., поз. 206

Підп. до друку Формат 60x84 1/16. Папір друк.№2.
Гарнітура Times New Roman. Ум. друк. арк.
Тираж 50 прим. Зам. № . Ціна договірна.

Віддруковано в ТОВ «ДРУКАРНЯ МАДРИД»
61024, м. Харків, вул. Максиміліанівська, 11
Тел.: (057) 756-53-25
Свідоцтво суб'єкта видавничої справи
Серія ДК, № 4399 от 27.08.2012 р.
www.madrid.in.ua e-mail: info@madrid.in.ua 61002