Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

**Дворецький М. Л., Боровльова С. Ю.,
Нездолій Ю. О., Дворецька С. В.**

ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ НА МОВІ JAVA

**Методичні вказівки до виконання лабораторних робіт
з дисципліни «Об'єктно-орієнтоване програмування»**

*Для підготовки бакалаврів у галузі знань
«Інформаційні технології»*

Випуск 268

ЧНУ
ВИДАВНИЦТВО

Миколаїв – 2019

УДК 004.43Java(076)
С 75

Рекомендовано до друку вченою радою Чорноморського національного університету імені Петра Могили (протокол № 6 від 28 лютого 2019 р.).

Рецензенти:
Латанська Л. О. – завідувач кафедри ТЕЕС НУК, доктор технічних наук, професор;
Рябенький В. М. – доцент кафедри ПЗАС НУК, кандидат фізико-математичних наук, доцент.

С 75 Дворецький М. Л. Основи об'єктно-орієнтованого програмування на мові JAVA. Методичні вказівки до виконання лабораторних робіт з дисципліни «Об'єктно-орієнтоване програмування»; для підготовки бакалаврів у галузі знань «Інформаційні технології» / Дворецький М. Л., Боровльова С. Ю., Нездолій Ю. О., Дворецька С. В. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2019. – 68 с. (Методична серія ; вип. 268).

У методичних вказівках до виконання лабораторних робіт з дисципліни «Об'єктно-орієнтоване програмування» розглянуто поняття та основні положення об'єктно-орієнтованого програмування, такі, як клас, об'єкт, поле та метод. Висвітлено основні принципи ООП: абстрагування, інкапсуляція, наслідування та поліморфізм. У ході практичних робіт увага приділяється перевантаженню конструкторів та методів, модифікаторам доступу, створенню гетерів і сетерів та перевизначенню методів. Розглядаються абстрактні та фінальні методи і класи, універсальні методи та класи і поняття інтерфейсів. Також увага приділяється анонімним класам та лямбда-виразам, і особливостям роботи із ними. Під час роботи з множинами об'єктів використовуються масиви та колекції.

УДК 004.43Java(076)

ISSN 1811–492X

ЗМІСТ

ВСТУП

Об’єктно-орієнтована парадигма програмування не нова. Її витоки сягають Симула-67, хоча вперше вона була повністю реалізована в Smalltalk-80. Об’єктно-орієнтоване програмування набуло популярності у другій половині 80-х разом з такими мовами, як C ++, Objective C, Object Pascal і Turbo Pascal, Eiffel, Ada і недавно – в Java.

На відміну від традиційних поглядів, коли програму розглядали як набір підпрограм, або як перелік інструкцій комп’ютеру, ООП програми можна вважати сукупністю об’єктів. Відповідно до парадигми об’єктно-орієнтованого програмування, кожен об’єкт здатний отримувати повідомлення, обробляти дані, та надсилати повідомлення іншим об’єктам. Кожен об’єкт – своєрідний незалежний автомат з окремим призначенням та відповідальністю.

Java на сьогодні залишається найпопулярнішою мовою програмування, яка дозволяє створювати різні застосунки широкого спектру: веб-сайти і веб-сервіси, десктопні програми, мобільні застосунки для ОС Android, сучасні програми з багатим інтерфейсом (Java FX). Java – універсальна крос-платформна мова, тому застосунки на Java працюватимуть на більшості відомих платформ, таких як Windows, Linux, MacOS.

1. ЗАГАЛЬНІ ВИЗНАЧЕННЯ

Об’єктно-орієнтоване програмування – це метод програмування, заснований на поданні програми у вигляді сукупності взаємодіючих об’єктів, кожен з яких є екземпляром певного класу, а класи є членами певної ієрархії наслідування.

Об’єктно-орієнтований підхід полягає в наступному наборі основних принципів:

– Усе є об’єктами.

– Усі операції виконуються шляхом взаємодії (обміну даними) між об’єктами, при якій один об’єкт потребує, щоб інший виконав певну дію.

– Об’єкти взаємодіють, надсилаючи і отримуючи повідомлення. Повідомлення – це запит на виконання дії, доповнений набором аргументів, які можуть знадобитися під час виконання.

– Кожен об’єкт має незалежну пам’ять, яка може складатись з інших об’єктів.

– Кожен об’єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об’єктів.

– У класі задається поведінка (функціональність) об’єкта. Таким чином усі об’єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії.

– Класи організовані у єдину деревоподібну структуру з загальним корінням, яка називається ієрархією успадкування.

Таким чином, програма являє собою набір об’єктів, що мають стан та поведінку.

Базові поняття (визначення)

Клас – визначає абстрактні характеристики певної сутності, включаючи характеристики самої сутності (її атрибути або властивості) та дії, які вона здатна виконувати (її поведінки, методи або можливості).

Наприклад, клас Собака може характеризуватись рисами, притаманними усім собакам, зокрема: порода, колір хутра, здатність гавкати.

Об’єкт – окремий екземпляр класу (створюється після запуску програми та ініціалізації полів класу).

Клас Собака відповідає усім собакам шляхом опису їхніх спільних рис; об’єкт Сірко є одним окремим собакою, окремим варіантом значень

характеристик. Собака має хутро; Сірко має коричнево-біле хутро. Об'єкт Сірко є екземпляром (примірником) класу Собака.

Сукупність значень атрибутів окремого об'єкта називається станом.

На основі класу Собака можна також створити інший об'єкт Дружок, який відрізнятиметься від об'єкта Сірко своїм станом (наприклад, кольором хутра). Обидва об'єкти (Сірко і Дружок) є екземплярами класу Собака.

Методи – можливості об'єкта. Оскільки Сірко – собака, він може гавкати. Тому гавкати () є одним із методів об'єкта Сірко. Він може мати й інші методи, зокрема: місце (), або їсти ().

У межах програми, використання методу має впливати лише на один об'єкт; усі собаки можуть гавкати, але треба щоб гавкав лише один окремий собака.

Основні принципи (парадигми) ООП

Абстрагування – спрощення складної дійсності шляхом моделювання класів, що відповідають проблемі, та використання найприйнятнішого рівня деталізації окремих аспектів проблеми.

Наприклад, собака Сірко більшу частину часу може розглядатись як собака загалом, та не містити даних, що специфічні для собак окремих видів.

Інкапсуляція – приховування деталей про роботу класів від об'єктів, що їх використовують або надсилають їм повідомлення.

Так, наприклад, клас Собака має метод гавкати(). Реалізація цього методу описує як саме повинно відбуватися гавкання (приміром, спочатку вдихнути (), а потім видихнути () на обраній частоті та гучності). Хазяїну пса Сірка, не обов'язково знати як він гавкає, щоб віддати команду.

Інкапсуляція потрібна для того, аби запобігти використанню користувачами інтерфейсу тих частин реалізації, які, швидше за все, будуть змінюватись; або використовуючи які можна отримати неправильні результати роботи.

Наприклад, інтерфейс може гарантувати, що цуценята можуть додаватися лише до об'єктів класу Собака кодом самого класу.

Успадкування – клас може мати «підкласи», спеціалізовані, розширені версії надкласу. Можуть навіть утворюватись цілі дерева успадкування.

Наприклад, клас Собака може мати підкласи Коллі, Пекінес, Вівчарка і т. п. Так, Сірко може бути екземпляром класу Вівчарка.

Підкласи успадковують атрибути та поведінку своїх батьківських класів, і можуть вводити свої власні. Успадкування може бути одиничне (один безпосередній батьківський клас) та множинне (кілька батьківських класів).

Поліморфізм означає залежність поведінки від класу, в якому вона задекларована, тобто, два або більше класів можуть реагувати по-різному на однакові повідомлення.

Наприклад, якщо Собака отримує команду голос(), то у відповідь можна отримати Гав; якщо Свиня отримує команду голос(), то у відповідь можна отримати Рох-рох.

На практиці це досягається шляхом різної реалізації методів класів-нащадків спільного суперкласу або інтерфейсу.

2. СТВОРЕННЯ ПРОСТИХ КЛАСІВ ТА ОБ'ЄКТІВ

Пригадаємо, згідно з об'єктно-орієнтованим підходом, програма являє собою набір об'єктів, які мають стан та поведінку. А кожен об'єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об'єктів. У класі задається поведінка (функціональність) об'єкта.

Отже створимо клас Собака, що буде мати наступні властивості: кличку, колір хутра, вагу тіла та буде вміти засинати, просинатися, бігати та гавкати (рис. 2.1):

```
public class Dog {
    public String name;
    public String color;
    public int weight;

    public void wakeUp() {
        System.out.println("Dog " + name + " woke up");
    }
    public void sleap() {
        System.out.println("Dog " + name + " fall asleap");
    }
    public void bark() {
        System.out.println("Dog " + name + " says 'BARK !'");
    }
    public void run() {
        System.out.println("Dog " + name + " is running");
    }
}
```

Рис. 2.1. Простий клас Собака

Використаємо створений клас в основному коді програми (рис. 2.2):

```
public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.name = "Spike";
    dog1.sleap();
    dog1.wakeUp();
    dog1.bark();
    dog1.run();
}
```

Рис. 2.2 Приклад створення екземпляра класу Собака та робота із ним.

Виконаємо програму та переглянемо результат (рис. 2.3):

```
Dog Spike fall asleap
Dog Spike woke up
Dog Spike says 'BARK !'
Dog Spike is running
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 2.3. Результат виконання коду, наведеного на рис. 2.2

Виконаємо невеликі зміни нашого коду. Зробимо так, щоб собака не міг бігати та гавкати, коли він спить. Крім цього, великий собака нехай голосно гавкає, але повільно бігає, а маленький навпаки (рис. 2.4):

```
public class Dog {
    public String name;
    public String color;
    public int weight;

    private boolean isSleaping = false;

    public void wakeUp() {
        if (isSleaping) {
            System.out.println("Dog " + name + " woke up");
            isSleaping = false;
        } else {
            System.out.println("Dog " + name + " has already woken up");
        }
    }
    public void sleap() {
        if (!isSleaping) {
            System.out.println("Dog " + name + " fall asleap");
            isSleaping = true;
        } else {
            System.out.println("Dog " + name + " is already sleaping");
        }
    }
    public void bark() {
        if (!isSleaping) {
            if (weight > 20) {
                System.out.println("Dog " + name + " says 'BAAARK !'");
            } else {
                System.out.println("Dog " + name + " says 'av av!'");
            }
        } else {
            System.out.println("Dog " + name + " can't bark. It's sleaping");
        }
    }
    public void run() {
        if (!isSleaping) {
            if (weight > 20) {
                System.out.println("Dog " + name + " is running, but slowly");
            } else {
                System.out.println("Dog " + name + " is running very fast");
            }
        } else {
            System.out.println("Dog " + name + " can't run. It's sleaping");
        }
    }
}
```

Рис. 2.4. Додавання у методи умовних операторів

В основному коді створимо 2 об’єкти цього класу та виконаємо декілька методів для кожного з них (рис. 2.5):

```
public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.name = "Spike"; dog1.weight = 30;
    dog1.sleap();
    dog1.bark();
    dog1.wakeUp();
    dog1.run();

    Dog dog2 = new Dog();
    dog1.name = "Tom"; dog1.weight = 10;
    dog1.bark();
    dog1.wakeUp();
}
```

Рис. 2.5. Створення 2-х екземплярів «Dog» та робота із ними

Переглянемо результат роботи програми (рис. 2.6)

```
run:
Dog Spike fall asleap
Dog Spike can't bark. It's sleaping
Dog Spike woke up
Dog Spike is running, but slowly
Dog Tom says 'av av!'
Dog Tom has already woken up
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 2.6. Результат роботи коду, наведеного на рис. 2.5

В об’єктно-орієнтованому програмуванні конструктор класу – спеціальний метод класу, який автоматично викликається під час створення об’єкта. Призначення конструктора – встановити початковий стан об’єкта шляхом ініціалізації атрибутів об’єкта. У більшості мов програмування конструктор схожий з іншими методами, але відрізняється тим, що не має явним чином визначеного типу даних, що повертаються. У більшості мов програмування конструктор може бути перевантаженим, що дозволяє використовувати декілька конструкторів в одному класі.

Реалізуємо декілька конструкторів для нашого класу Dog (рис. 2.7):

```
private boolean isSleaping = false;

public Dog(String name) {
    this.name = name;
}
public Dog(String name, String color) {
    this(name);
    this.color = color;
}
public Dog(String name, int weight) {
    this(name);
    this.weight = weight;
}
public Dog(String name, String color, int weight) {
    this(name, color);
    this.weight = weight;
}
public void wakeUp() {
    if (isSleaping) {
```

Рис. 2.7. Перевантаження конструктора у класі Собака

Перевіримо роботу конструкторів, змінивши основний код програми (рис. 2.8):

```
Dog dog1 = new Dog("Spike", 30);
dog1.sleap();
dog1.bark();
dog1.wakeUp();
dog1.run();

Dog dog2 = new Dog("Tom", "black", 10);
dog1.bark();
dog1.wakeUp();
```

Рис. 2.8. Робота конструкторів класу Собака

Результат роботи програми має залишитися без змін (рис. 2.9):

```
Dog Spike fall asleap
Dog Spike can't bark. It's sleaping
Dog Spike woke up
Dog Spike is running, but slowly
Dog Tom says 'av av!'
Dog Tom has already woken up
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 2.9. Результат роботи коду, наведеного на рис. 2.8.

3. МОДИФІКАТОРИ ДОСТУПУ, ПОНЯТТЯ ІНКАПСУЛЯЦІЇ, ГЕТЕРИ ТА СЕТЕРИ

Інкапсуляція – приховування деталей про роботу класів від об’єктів, що їх використовують або надсилають їм повідомлення. Реалізується інкапсуляція через застосування до властивостей та методів класу модифікаторів доступу.

Усі члени класу в мові Java – поля і методи, властивості – мають модифікатори доступу. У минулих темах ми вже стикалися з модифікатором public. Модифікатори доступу дозволяють задати допустиму область видимості для членів класу, тобто контекст, у якому можна використовувати цю змінну або метод.

У Java використовуються наступні модифікатори доступу:

public: публічний, загальнодоступний клас або член класу. Поля і методи, оголошені з модифікатором public, видно іншим класам з поточного пакета і з зовнішніх пакетів.

private: закритий клас або член класу, протилежність модифікатора public. Закритий клас або член класу доступний тільки з коду в тому ж класі.

protected: такий клас або член класу доступний з будь-якого місця в поточному класі або пакеті, або в похідних класах, навіть якщо вони знаходяться в інших пакетах.

Модифікатор за замовчуванням. Відсутність модифікатора у поля або методу класу передбачає застосування до нього модифікатора за замовчуванням. Такі поля або методи доступні в усіх класах поточного пакету.

Почнемо з того, що закриємо прямий доступ до всіх полів нашого класу Dog (рис. 3.1):

```
public class Dog {
    private String name;
    private String color;
    private int weight;

    private boolean isSleaping = false;

    public Dog(String name) {
        this.name = name;
    }
```

Рис. 3.1. Використання модифікатора private для полів класу.

Тепер в основному коді ми не можемо, наприклад, змінити кличку собаки після її створення, і це є логічним. Ця операція буде приводити до помилки (рис. 3.2):

```
public static void main(String[] args) {
    Dog dog1 = new Dog("Spike", 30);
    dog1.sleap();
    dog1.b
    dog1.w
    dog1.r
    dog1.name = "Bob";
```

name has private access in Dog

(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)

Рис. 3.2. Спроба звернення до приватного поля за межами класу.

Колір собаки теж навряд зміниться, однак можливо, що собака набере, або навпаки, скине вагу. Для таких випадків передбачені спеціальні методи, які прийнято називати «сетерами». Їх призначення – запис значень до приватних (інкапсульованих) полів класу. Реалізуємо сетер для поля вага нашого класу (рис. 3.3):

```
public void setWeight(int weight) {
    this.weight = weight;
}
```

Рис. 3.3. Реалізація методу-сетера для приватного поля «вага».

Скористаємося ним в основному коді (рис. 3.4).

```
public static void main(String[] args) {
    Dog dog1 = new Dog("Spike", 22);
    dog1.sleap();
    dog1.bark();
    dog1.wakeUp();
    dog1.run();
    dog1.setWeight(19);
    dog1.run();
}
```

Рис. 3.4. Приклад використання сетера.

Маємо наступний результат (рис. 3.5)

```
run:
Dog Spike fall asleap
Dog Spike can't bark. It's sleaping
Dog Spike woke up
Dog Spike is running, but slowly
Dog Spike is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Рис. 3.5. Результат роботи коду, наведеного на рис. 3.4.

Розглянемо протилежну ситуацію, коли нам потрібно не змінити, а просто отримати значення якогось поля нашого об’єкта. Наприклад, ми

оголошуємо масив собак та хочемо вивести на екран клички тих собак, що мають білий колір хутра. Звернення напряму до приватних властивостей на читання також веде до помилки (рис. 3.6).

```
 */
public static void main(String[
    Dog dog1 = new Dog("Spike",
    System.out.println(dog1.name);
}
```

name has private access in Dog

(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)

Рис. 3.6. Спроба прочитати приватне поле за межами класу

Для таких випадків зазвичай використовуються спеціальні методи – гетери, що надають доступ на читання до закритих полів класу. Реалізуємо такі методи для кольору та клички, а після цього використаємо в основному коді для реалізації задачі, поставленої вище (рис. 3.7–3.9):

```
public String getName() {
    return name;
}
public String getColor() {
    return this.color;
}
```

Рис. 3.7. Приклад реалізації методів-гетерів для полів «кличка» та «колір»

```
public static void main(String[] args) {
    Dog[] dogs = {
        new Dog("Spike", "black", 22),
        new Dog("Bob", "white", 12),
        new Dog("Jina", "white", 15),
        new Dog("Wolf", "Gray", 20)
    };
    for(Dog dog: dogs) {
        if (dog.getColor() == "white") {
            System.out.println(dog.getName());
        }
    }
}
```

Рис. 3.8. Використання реалізованих гетерів

```
run:
Bob
Jina
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 3.9. Результат роботи коду, наведеного на рис. 3.8

4. ПОНЯТТЯ СИГНАТУРИ МЕТОДІВ ТА ПЕРЕВАНТАЖЕННЯ

Далі розберемо поняття сигнатури методу та перевантаження методів. Код, який описує метод, називається сигнатурою методу.

Загальний вигляд сигнатури можна описати так:

«модифікатор доступу, тип значення, ім'я методу» (список параметрів) { // тіло методу.
}

Бувають ситуації, коли в нашій програмі потрібно декілька варіантів роботи методу. Так, наприклад, метод bark() класу Dog() визначає гучність гавкання зважаючи на масу собаки. Але один і той же собака може гавкати по-різному. Навчимо цього наш клас, визначивши ще 2 методи bark(), що будуть відрізнятись від попереднього своєю сигнатурою (рис. 4.1).

```
public void bark() {
    if (!isSleaping) {
        if (weight > 20) {
            System.out.println("Dog " + name + " says 'BAAARK !'");
        } else {
            System.out.println("Dog " + name + " says 'av av!'");
        }
    } else {
        System.out.println("Dog " + name + " can't bark. It's sleaping");
    }
}
public void bark(int loudness) {
    if (loudness < 5) {
        System.out.println("Dog " + name + " says 'av av!'");
    } else if (loudness < 10) {
        System.out.println("Dog " + name + " says 'bark bark !'");
    } else {
        System.out.println("Dog " + name + " says 'BAAARK !'");
    }
}
public void bark(String bark) {
    System.out.println("Dog " + name + " says '" + bark + "");
}
```

Рис. 4.1. Перевантаження методу «гавкати» класу Собака

Використаємо нові методи в основному коді та переглянемо результат (рис. 4.2–4.3):

```
    public static void main(String[] args) {
        Dog dog = new Dog("Spike", "black", 25);
        dog.bark();
        dog.bark(5);
        dog.bark("rrrr-aav !");
    }
```

Рис. 4.2. Використання перевантажених методів класу Собака

```
run:
Dog Spike says 'BAAARK !'
Dog Spike says 'bark bark !'
Dog Spike says 'rrrr-aav !
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Рис. 4.3. Результат роботи коду, наведеного на рис. 4.2.

5. НАСЛІДУВАННЯ. БАЗОВІ КОНСТРУКТОРИ. МОДИФІКАТОР PROTECTED

Наслідування – клас може мати «підкласи», спеціалізовані, розширені версії надкласу. Підкласи успадковують атрибути та поведінку своїх батьківських класів, і можуть вводити свої власні. Успадкування може бути одиничне (один безпосередній батьківський клас) та множинне (декілька батьківських класів).

Додамо до класу Dog конструктор без параметрів (рис. 5.1):

```
public Dog() {
    this.name = "Street dog";
}
```

Рис. 5.1. Конструктор без параметрів (за замовчуванням) класу Собака

Після цього створимо клас Вівчарка, що наслідує клас Dog (рис. 5.2):

```
public class Shepherd extends Dog {

}
```

Рис. 5.2. Клас Вівчарка, що розширює клас Собака.

Використаємо клас (рис. 5.3) та наведемо результат роботи коду (рис. 5.4):

```
public static void main(String[] args) {
    Dog dog = new Shepherd();
    dog.bark();
    Shepherd dog2 = new Shepherd();
    dog2.run();
}
```

Рис. 5.3. Приклад використання похідного класу Вівчарка

```
run:
Dog Street dog says 'av av!'
Dog Street dog is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 5.4. Результат роботи коду, наведеного на рис. 5.3.

Розберемо роботу полями базового класу. Наприклад, ми хочемо створити вівчарку з кличкою Сірко (рис. 5.5):

```
public class Shepherd extends Dog {
    public Shepherd(String name) {
        this.name = name;
    }
}
```

```
name has private access in Dog
----
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)
```

Рис. 5.5. Помилка при зверненні до приватної властивості за межами класу

При зверненні до поля ім’я маємо помилку, оскільки означене поле має модифікатор доступу «private», що робить неможливим звернення до нього за межами класу Dog. Якщо ми хочемо звертатись до поля також у похідних класах, нам підійде модифікатор «protected». Виконаємо зміни для цього та інших полів (рис. 5.6):

```
    protected String name;
    protected String color;
    protected int weight;
```

Рис. 5.6. Зміна модифікатора доступу до полів класу Собака та «protected»

Тепер реалізуємо конструктори в класі Shepherd (рис. 5.7) та перевіримо результат роботи в основному коді:

```
public class Shepherd extends Dog {
    public Shepherd() {
        this.name = "Street dog";
    }
    public Shepherd(String name) {
        this.name = name;
    }
    public Shepherd(String name, String color) {
        this(name);
        this.color = color;
    }
    public Shepherd(String name, int weight) {
        this(name);
        this.weight = weight;
    }
    public Shepherd(String name, String color, int weight) {
        this(name, color);
        this.weight = weight;
    }
}
```

Рис. 5.7. Реалізація конструкторів для класу Вівчарка

Отже, клас вівчарка не має власних полів для імені, кольору та ваги, але має їх, оскільки наслідує від Dog. Створимо вівчарку з іменем та використаємо її в основному коді (рис. 5.8–5.9):

```
    public static void main(String[] args) {
        Dog dog = new Shepherd("Sirko");
        dog.bark();
        dog.run();
    }
```

Рис. 5.8. Створення екземпляра похідного класу та його використання

```
run:
Dog Sirko says 'av av!'
Dog Sirko is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 5.9. Результат роботи коду, наведеного на рис. 5.8

Але можна звернути увагу, що ми дублюємо код конструкторів із базового класу. Замість цього можна просто викликати їх із використанням ключового слова «super» (рис. 5.10):

```
public class Shepherd extends Dog {
    public Shepherd() {
    }
    public Shepherd(String name) {
        super(name);
    }
    public Shepherd(String name, String color) {
        super(name, color);
    }
    public Shepherd(String name, int weight) {
        super(name, weight);
    }
    public Shepherd(String name, String color, int weight) {
        super(name, color, weight);
    }
}
```

Рис. 5.10. Звернення до конструктора суперкласу в похідному класі

Перевіримо роботу наших конструкторів після виконаних змін (рис. 5.11–5.12):

```
    public static void main(String[] args) {
        Dog dog = new Shepherd("Sirko", "black", 20);
        dog.bark("RRR-AAVV !!!");
        dog.run();
    }
```

Рис. 5.11. Створення екземпляра похідного класу та його використання

```
run:
Dog Sirko says 'RRR-AAVV !!!
Dog Sirko is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 5.12. Результат роботи коду, наведеного на рис. 5.11

Немає сенсу створювати новий клас, якщо він повністю повторює попередній. Звичайно новий клас «уміє» те, чого не «умів» попередній – тобто якісь додаткові поля чи методи. У такому випадку говорять, що він «розширює» базовий. Додамо до класу Вівчарка метод «охороняти» (рис. 5.13):

```
public class Shepherd extends Dog {
    public boolean enemyExists;
    public void guard() {
        if (enemyExists) {
            run();
            bark();
        } else {
            System.out.println("Shepherd " + name + " keep silence and wait");
        }
    }
}
```

Рис. 5.13. Метод «охороняти», що розширює базовий клас Собака

Далі наведемо приклад роботи зі створеним методом (рис. 5.14):

```
public static void main(String[] args) {
    Shepherd dog = new Shepherd("Sirko", "black", 20);
    dog.guard();
    dog.enemyExists = true;
    dog.guard();
}
```

Рис. 5.14. Використання методу «охороняти» для класу Вівчарка

```
run:
Shepherd Sirko keep silence and wait
Dog Sirko is running very fast
Dog Sirko says 'av av!'
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 5.15. Результат роботи коду, наведеного на рис. 5.14

Розглянемо також другий варіант створення екземпляра класу Вівчарка – через клас Dog (рис. 5.16):

```
public static void main(String[] args) {
    Dog dog = new Shepherd("Sirko", "black", 20);
    dog.bark();
    dog.guard();
}
```

```
cannot find symbol
  symbol:   method guard()
  location: variable dog of type Dog
----
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)
```

Рис. 5.16. Створення екземпляра класу Вівчарка із типом Собака

Бачимо, що тепер наш собака вміє гавкати та бігати, але не вміє охороняти. Для того, щоб навчити її це робити, треба виконати явне перетворення типів (рис. 5.17):

```
public static void main(String[] args) {
    Dog dog = new Shepherd("Sirko", "black", 20);
    dog.bark();
    ((Shepherd)dog).guard();
}
```

Рис. 5.17. Явне перетворення типів

6. ПОЛІМОРФІЗМ. ПЕРЕВИЗНАЧЕННЯ МЕТОДІВ

Поліморфізм – означає залежність поведінки від класу, в якому вона задекларована, тобто, два або більше класів можуть реагувати по-різному на однакові повідомлення.

Продемонструємо поліморфізм на прикладі нашого класу «вівчарка», створеного у попередній роботі. Для класу Собака метод «бігати» працює залежно від ваги собаки, та чим вона важче, тим повільніше біжить. Для нашої вівчарки, ситуація буде протилежна – якщо вона велика, то й біжить швидше.

Для цього нам потрібно визначити метод run() у класі Shepherd. Також можна сказати, що ми перевизначаємо метод run() базового класу (рис. 6.1):

```
public void run() {
    if (weight > 20) {
        System.out.println("Dog " + name + " is running as wind");
    } else {
        System.out.println("Dog " + name + " is running very fast");
    }
}
```

Рис. 6.1. Перевизначення методу «бігти» у класі «вівчарка»

Виконаємо створений метод в основному коді (рис. 6.2–6.3):

```
public static void main(String[] args) {
    Dog dog = new Shepherd("Sirko", "black", 22);
    dog.run();
    Shepherd dog2 = new Shepherd("Wolf", "gray", 18);
    dog2.run();
}
```

Рис. 6.2. Приклад роботи перевизначеного методу

```
run:
Dog Sirko is running as wind
Dog Wolf is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 6.3. Результат роботи коду, наведеного на рис. 6.2

Як бачимо, навіть якщо ми визначаємо вівчарку із типом собака, то все одно викликається метод похідного класу. Це є однією з ключових особливостей поліморфізму.

Можна звернути увагу, що поки для собак та вівчарок вагою менше 20 метод run() працює однаково. Щоб уникнути дублювання коду, з методу run() похідного класу вівчарка звернемося до методу run() базового класу Собака (рис. 6.4):

```
public void run() {
    if (weight > 20) {
        System.out.println("Dog " + name + " is running as wind");
    } else {
        super.run();
    }
}
```

Рис. 6.4. Звернення до методу базового класу
із метода похідного класу

У мові java, перевизначаючи метод, також бажано вказувати анотацію override (рис. 6.5):

```
@Override
public void run() {
    if (weight > 20) {
        System.out.println("Dog " + name + " is running as wind");
    } else {
        super.run();
    }
}
```

Рис. 6.5. Приклад використання анотації override
при перевизначенні методу

Це не впливає на факт перевантаження, але приведе до помилки, якщо ми випадково змінимо сигнатуру методу, тим самим виконаємо перевантаження замість перевизначення (рис. 6.6):

```
    }
}
@Override
public void run(int speed) {
    if (weight > 20) {
        System.out.println("Dog " + name + " is running as wind");
    } else {
        super.run();
    }
}
```

method does not override or implement a method from a supertype
---
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)

Рис. 6.6. Помилка під час неспівпадіння сигнатур
при перевизначенні із використанням анотації override

Продемонструємо поліморфізм на прикладі масиву об’єктів із типом базового класу (рис. 6.7–6.8):

```
public static void main(String[] args) {
    Dog[] dogs = {
        new Shepherd("Sirko", "black", 40),
        new Dog("Sharik", 15),
        new Dog("Spark", "white", 30),
        new Shepherd("Wolf", 19),
    };
    for(Dog dog: dogs) {
        dog.run();
    }
}
```

Рис. 6.7. Демонстрація поліморфізму на прикладі масиву об’єктів

```
run:
Dog Sirko is running as wind
Dog Sharik is running very fast
Dog Spark is running, but slowly
Dog Wolf is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 6.8. Результат роботи коду, наведеного на рис. 6.7

Бачимо, що важкі собаки бігають повільніше, але важкі вівчарки, навпаки, швидше. Також, окрім масивів, групи об’єктів можуть об’єднуватись у так звані колекції, які мають більш зручні методи для маніпулювання вмістом та краще працюють з великими та мінливими наборами даних. Продемонструємо це на прикладі колекції ArrayList (рис. 6.9):

```
public static void main(String[] args) {
    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Shepherd("Sirko", "black", 40));
    dogs.add(new Dog("Sharik", 15));
    dogs.add(new Dog("Spark", "white", 30));
    dogs.add(new Shepherd("Wolf", 19));

    for(Dog dog: dogs) {
        dog.run();
    }

    for(int i=0; i< dogs.size(); i++) {
        dogs.get(i).run();
    }

    dogs.forEach(
        (dog) -> dog.run()
    );
}
```

Рис. 6.9. Поліморфізм із використанням колекції ArrayList

У прикладі із ArrayList продемонстровано 3 різних способи перебору всіх елементів колекції.

Хоча ми можемо створити звичайний клас, який не є нащадком, але фактично усі класи успадковують від класу Object. Усі класи є неявно похідними від класу Object. Тому всі типи і класи можуть реалізувати ті методи, які визначені в класі Object. Розглянемо деякі з них.

Метод toString служить для отримання уявлення нашого об’єкта у вигляді рядка. Під час спроби вивести строкове подання якогось об’єкта, як правило, буде виводитися повне ім’я класу. Наприклад, на рис. 6.10 виведемо на консоль екземпляр класу Собака:

```
public static void main(String[] args) {
    Dog dog = new Shepherd("Sirko", "black", 22);
    System.out.println(dog);
}
```

Рис. 6.10. Виведення на консоль екземпляра класу Собака

```
run:
labal.Shepherd@15db9742
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 6.11. Результат роботи коду, наведеного на рис. 6.10

Звичайно означений вивід не є дуже зручною формою представлення даних щодо об’єкта нашого класу. Однак, як і інші методи базового класу, метод toString() може бути перевизначено. Зробимо це у класі Dog (рис. 6.12):

```
@Override
public String toString() {
    return "Dog " + this.name + " with weight "
            + weight + " and " + color + " fur color";
}
```

Рис. 6.12. Перевизначення методу toString у класі Собака

Потім повторимо вивід на екран (рис. 6.13–6.14):

```
public static void main(String[] args) {
    Dog dog = new Shepherd("Sirko", "black", 22);
    System.out.println(dog);
}
```

Рис. 6.13. Виведення на консоль екземпляра класу Собака

```
run:
Dog Sirko with weight 22 and black fur color
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 6.14. Результат роботи коду, наведеного на рис. 6.13

Бачимо, що зараз дані про собаку представлені у зручній та зрозумілій для користувача формі.

Для порівняння рівності об’єктів використовується метод equals(), що також успадковується від object. Але об’єкти дорівнюють лише у випадку, коли дорівнюють посилання, а не тільки значення. Наприклад (рис. 6.15–6.16):

```
public static void main(String[] args) {
    Dog dog1 = new Shepherd("Sirko", "black", 40);
    Dog dog2 = new Shepherd("Sirko", "black", 40);
    Dog dog3 = dog2;

    System.out.println("dog1 = dog2 - " + dog1.equals(dog2));
    System.out.println("dog2 = dog3 - " + dog2.equals(dog3));
}
```

Рис. 6.16. Приклад порівняння 2-х пар об’єктів

```
run:
dog1 = dog2 - false
dog2 = dog3 - true
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 6.17. Результат роботи коду, наведеного на рис. 6.16

Бачимо, що dog1 не дорівнює dog2, незважаючи на те, що усі значення атрибутів співпадають. Якщо ми хочемо змінити це, треба перевизначити метод equals класу Dog (рис. 6.18–6.20):

```
@Override
public boolean equals(Object o) {
    Dog dog = (Dog)o;
    return ((name == dog.name)&&(color == dog.color)&&(weight == dog.weight));
}
```

Рис. 6.18. Перевизначення методу equals класу Собака

```
public static void main(String[] args) {
    Dog dog1 = new Shepherd("Sirko", "black", 40);
    Dog dog2 = new Shepherd("Sirko", "black", 40);
    Dog dog3 = dog2;
    Dog dog4 = new Shepherd("Dob", "black", 40);

    System.out.println("dog1 = dog2 - " + dog1.equals(dog2));
    System.out.println("dog1 = dog4 - " + dog1.equals(dog4));
    System.out.println("dog2 = dog3 - " + dog2.equals(dog3));
}
```

Рис. 6.19. Приклад порівняння 3-х пар об’єктів
після перевизначення методу equals класу Собака

```
run:
dog1 = dog2 - true
dog1 = dog4 - false
dog2 = dog3 - true
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 6.20. Результат роботи коду, наведеного на рис. 6.19.

Тепер собаки вважаються рівними, якщо вони мають однакові атрибути.

Увага! Перевизначаючи метод equals(), також рекомендуємо перевизначити і hashCode(). Інакше, більш глибинні механізми сортування та маніпуляції із елементами колекцій можуть давати неочікувані результати.

Перевизначити hashCode() для Dog можна, наприклад, наступним чином (рис. 6.21):

```
@Override
public int hashCode() {
    int hash = 37;
    hash = hash * 17 + name.hashCode();
    hash = hash * 17 + color.hashCode();
    hash = hash * 17 + weight;
    return hash;
}
```

Рис. 6.21. Приклад перевизначення методу hashCode

7. СТАТИЧНІ ПОЛЯ ТА МЕТОДИ

Крім звичайних методів і полів клас може мати статичні поля, методи, константи та ініціалізатор. Наприклад, головний клас програми має метод main, який є статичним (рис. 7.1):

```
public static void main(String[] args) {

}
```

Рис. 7.1. Статичний метод main

Для оголошення статичних полів та методів перед їх оголошенням вказується ключове слово static. У класі Dog створимо статичну змінну, в якій будемо зберігати значення маси, починаючи від якої, собака буде бігти повільно (або швидко).

Додамо статичне поле limitWeight (рис. 7.2) та змінимо методи run() та bark() (рис. 7.3–7.4):

```
public static int limitWeight = 20;
```

Рис. 7.2. Додавання статичного поля до класу.

```
public void bark() {
    if (!isSleaping) {
        if (weight > limitWeight) {
            System.out.println("Dog " + name + " says 'BAAARK !'");
        } else {
            System.out.println("Dog " + name + " says 'av av!'");
        }
    } else {
        System.out.println("Dog " + name + " can't bark. It's sleaping");
    }
}
```

Рис. 7.3. Зміна методу «гавкати»

```
public void run() {
    if (!isSleaping) {
        if (weight > limitWeight) {
            System.out.println("Dog " + name + " is running, but slowly");
        } else {
            System.out.println("Dog " + name + " is running very fast");
        }
    } else {
        System.out.println("Dog " + name + " can't run. It's sleaping");
    }
}
```

Рис. 7.4. Зміна методу «бігати»

Продемонструємо використання цього поля в основному коді (рис. 7.5–7.6):

```
public static void main(String[] args) {
    Dog dog = new Dog("Spike", "white", 20);
    dog.run();
    dog.bark();
    Dog.limitWeight = 15;
    dog.run();
    dog.bark();
}
```

Рис. 7.5. Використання статичного поля та змінених методів

```
run:
Dog Spike is running very fast
Dog Spike says 'av av!'
Dog Spike is running, but slowly
Dog Spike says 'BAAARK !'
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Рис. 7.6. Результат виконання коду, наведеного на рис. 7.5

Статичні методи також відносяться до всього класу загалом та на відміну від нестатичних, не мають доступу до властивостей окремого об’єкта. Змінимо модифікатор доступу поля limitWeight з public на protected та реалізуємо статичний сетер для цього поля (рис. 7.7–7.8):

```
protected static int limitWeight = 20;
```

Рис. 7.7. Зміна модифікатора для статичного поля limitWeight

```
public static void setLimitWeight(int limitWeight) {
    Dog.limitWeight = limitWeight;
}
```

Рис. 7.8. Реалізація сетера для статичного поля limitWeight

Виконаємо встановлення нового значення для граничної ваги, використовуючи створений сетер (рис. 7.9–7.10):

```
public static void main(String[] args) {
    Dog dog = new Dog("Spike", "white", 20);
    dog.run();
    dog.bark();
    Dog.setLimitWeight(15);
    dog.run();
    dog.bark();
}
```

Рис. 7.9. Використання сетера для статичного поля

```
run:
Dog Spike is running very fast
Dog Spike says 'av av!'
Dog Spike is running, but slowly
Dog Spike says 'BAAARK !'
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Рис. 7.10. Результат виконання коду, наведеного на рис. 7.9

Клас може складатись лише зі статичних методів та полів. У цьому випадку немає сенсу створювати екземпляри для означеного класу, і він, фактично є колекцією методів. Створимо клас «кінолог», що вмітиме віддавати команди run(), bark() та guard() групам собак (рис. 7.11) та наведемо приклад використання цього класу (рис. 7.12–7.13):

```
public class DogTrainer {
    static void run(Dog[] dogs) {
        for(Dog dog: dogs) {
            dog.run();
        }
    }
    static void bark(Dog[] dogs, String bark) {
        for(Dog dog: dogs) {
            if (bark == "") {
                dog.bark();
            } else {
                dog.bark(bark);
            }
        }
    }
    static void guard(Dog[] dogs) {
        for(Dog dog: dogs) {
            if (dog instanceof Shepherd) {
                ((Shepherd)dog).guard();
            } else {
                System.out.println("Dog " + dog.name + " can't guard");
            }
        }
    }
```

Рис. 7.11. Клас з двома статичними методами

```
    public static void main(String[] args) {
        Dog[] dogs = {
            new Dog("Spike", "white", 20),
            new Shepherd("Wolf", "gray", 30),
            new Shepherd("Ralf", "black", 35),
        };
        DogTrainer.run(dogs);
        DogTrainer.bark(dogs, "rrr-aff !");
        DogTrainer.guard(dogs);
    }
```

Рис. 7.12. Приклад використання статичних методів створеного класу

```
run:
Dog Spike is running very fast
Dog Wolf is running as wind
Dog Ralf is running as wind
Dog Spike says 'rrr-aff !
Dog Wolf says 'rrr-aff !
Dog Ralf says 'rrr-aff !
Dog Spike can't guard
Shepherd Wolf keep silence and wait
Shepherd Ralf keep silence and wait
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 7.13. Результат роботи коду, наведеного на рис. 7.12

8. АБСТРАКТНІ КЛАСИ ТА МЕТОДИ

Крім звичайних класів є також абстрактні. Абстрактний клас схожий на звичайний, тут також можна визначити поля і методи, але водночас не можна створити об’єкт або екземпляр абстрактного класу. Абстрактні класи покликані надавати базовий функціонал для класів-нащадків, а похідні вже реалізують цей функціонал.

Одна із головних відмінностей полягає у тому, що ми не можемо використовувати конструктор абстрактного класу для створення його об’єкта.

Продемонструємо це, створивши абстрактний клас Службовий собака (рис. 8.1) та спробуємо створити екземпляр такого класу (рис. 8.2):

```
abstract public class WarDog extends Dog {

}
```

Рис. 8.1. Створення абстрактного класу

```
 * @param args the c
 */
public static void m
    Dog dog = new WarDog();
}
```

WarDog is abstract; cannot be instantiated

(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)

Рис. 8.2. Спроба створення екземпляра абстрактного класу

Крім звичайних методів, абстрактний клас може містити абстрактні методи. Такі методи визначаються за допомогою ключового слова abstract і не мають ніякого функціоналу.

Наприклад, ми знаємо, що службовий собака має вміти «охороняти», але різні породи будуть робити це по-різному, тому ми не можемо на цьому етапі виконати реалізацію такого методу (рис. 8.3):

```
abstract public class WarDog extends Dog {
    public abstract void guard();
}
```

Рис. 8.3. Декларування абстрактного методу «охороняти»

Варто враховувати, якщо клас має хоча б один метод, то даний клас повинен бути визначений як абстрактний. Тобто, на цьому етапі клас WarDog не може бути неабстрактним (рис. 8.4):

```
 *
 * @author mich
 */
public class WarDog extends Dog {
    public abstract void guard();
}
```

WarDog is not abstract and does not override abstract method guard() in WarDog
---
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)

Рис. 8.4. Спроба зробити клас «службовий собака» неабстрактним

Також похідний клас зобов'язаний перевизначити і реалізувати всі абстрактні методи, які є в базовому абстрактному класі. Так, наприклад, спробуємо створити клас «доберман», що розширює клас WarDog (рис. 8.5):

```
 *
 * @author mich
 */
public class Doberman extends WarDog{

}
```

Doberman is not abstract and does not override abstract method guard() in WarDog
---
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)

Рис. 8.5. Розширення абстрактного класу

Виправимо це, та реалізуємо метод guard у класі Doberman (рис. 8.6):

```
public class Doberman extends WarDog{
    @Override
    public void guard() {
        System.out.println("Doberman " + name + " has started to guard");
    }
}
```

Рис. 8.6. Реалізація абстрактних методів базового класу в похідному.

Також створимо набір необхідних конструкторів для WarDog та Doberman (рис. 8.7–8.8):

```
abstract public class WarDog extends Dog {
    public WarDog() {
    }
    public WarDog(String name) {
        super(name);
    }
    public WarDog(String name, String color) {
        super(name, color);
    }
    public WarDog(String name, int weight) {
        super(name, weight);
    }
    public WarDog(String name, String color, int weight) {
        super(name, color, weight);
    }

    public abstract void guard();
}
```

Рис. 8.7. Набір конструкторів для класу «службовий собака»

```
public class Doberman extends WarDog{
    public Doberman() {
    }
    public Doberman(String name) {
        super(name);
    }
    public Doberman(String name, String color) {
        super(name, color);
    }
```

```
    public Doberman(String name, int weight) {
        super(name, weight);
    }
    public Doberman(String name, String color, int weight) {
        super(name, color, weight);
    }
    @Override
    public void guard() {
        System.out.println("Doberman " + name + " has started to guard");
    }
}
```

Рис. 8.8. Набір конструкторів та метод «охороняти» для класу «доберман»

Продемонструємо створення та роботу із екземпляром класу «доберман» в основному коді (рис. 8.9):

```
public static void main(String[] args) {
    Doberman dob1 = new Doberman("Lisa", 40);
    dob1.bark();
    dob1.guard();

    WarDog dob2 = new Doberman("Spark", 45);
    dob2.bark();
    dob2.guard();

    Dog dob3 = new Doberman("Chuck", 50);
    dob3.bark();
    dob3.guard();
}
```

Рис. 8.9. Робота із екземпляром класу «доберман»

Як бачимо, можна визначити добермана 3-ма способами – як добермана, як службового та звичайного собаку. Але в останньому випадку метод guard() буде для нас недоступним (рис. 8.10):

```
run:
Dog Lisa says 'BAAARK !'
Doberman Lisa has started to guard
Dog Spark says 'BAAARK !'
Doberman Spark has started to guard
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - E
Dog Chuck says 'BAAARK !'
        at labal.Labal.main(Labal.java:30)
```

Рис. 8.10. Результат роботи коду, наведеного на рис. 8.9

Тепер зробимо наших доберманів схожими на попередньо реалізований клас вівчарок – а саме реалізацію методів «бігати» та «охороняти», але щоб не дублювати код, розмістимо спільну частину в абстрактному класі WarDog, і розширимо Shepherd від WarDog. Також навчимо службового собаку кусатися (рис. 8.11–8.13):

```
abstract public class WarDog extends Dog {
    protected boolean enemyExists;

    public WarDog() {
    }
    public WarDog(String name) {
        super(name);
    }
    public WarDog(String name, String color) {
        super(name, color);
    }
    public WarDog(String name, int weight) {
        super(name, weight);
    }
    public WarDog(String name, String color, int weight) {
        super(name, color, weight);
    }
    public abstract void guard();

    @Override
    public void run() {
        if (weight > 20) {
            System.out.println("War-dog " + name + " is running as wind");
        } else {
            super.run();
        }
    }
    public void bite() {
        if (enemyExists) {
            System.out.println("War-dog " + name + " bites it's enemy");
        } else {
            System.out.println("There is nobody to bite");
        }

    }
    public void setEnemy(boolean enemy) {
        enemyExists = enemy;
    }
}
```

Рис. 8.11. Абстрактний клас Службовий собака
після додавання усіх необхідних методів

```
public class Shepherd extends WarDog {

    public Shepherd() {
    }
    public Shepherd(String name) {
        super(name);
    }
    public Shepherd(String name, String color) {
        super(name, color);
    }
```

```
    public Shepherd(String name, int weight) {
        super(name, weight);
    }
    public Shepherd(String name, String color, int weight) {
        super(name, color, weight);
    }

    public void guard() {
        if (enemyExists) {
            run();
            bark();
        } else {
            System.out.println("Shepherd " + name + " keep silence and wait");
        }
    }
}
```

Рис. 8.12. Клас Вівчарка, що тепер є похідним від класу Службовий собака

```
public class Doberman extends WarDog{
    public Doberman() {
    }
    public Doberman(String name) {
        super(name);
    }
    public Doberman(String name, String color) {
        super(name, color);
    }
    public Doberman(String name, int weight) {
        super(name, weight);
    }
    public Doberman(String name, String color, int weight) {
        super(name, color, weight);
    }

    @Override
    public void guard() {
        if (enemyExists) {
            run();
            bite();
        } else {
            System.out.println("Doberman " + name + " has started to guard");
        }
    }
}
```

Рис. 8.13. Клас Доберман, похідний від класу Службовий собака

Тепер створимо 2 службових собаки – вівчарку та добермана, і перевіримо роботу їхнього методу «охороняти» (рис. 8.14–8.15):

```
public static void main(String[] args) {
    WarDog dog1 = new Shepherd("Jack", "brown", 30);
    WarDog dog2 = new Doberman("Rob", "black", 40);
    dog1.guard();
    dog2.guard();
    dog1.setEnemy(true);
    dog2.setEnemy(true);
    dog1.guard();
    dog2.guard();
}
```

Рис. 8.14. Робота методу «охороняти» для «вівчарки» та «добермана»

```
run:
Shepherd Jack keep silence and wait
Doberman Rob has started to guard
War-dog Jack is running as wind
Dog Jack says 'BAAARK !'
War-dog Rob is running as wind
War-dog Rob bites it's enemy
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 8.15. Результат роботи коду, наведеного на рис. 8.14

Також наведемо діаграму класів для реалізованих Dog, WarDog, Doberman та Shepherd протягом попередніх розділів (рис. 8.16):

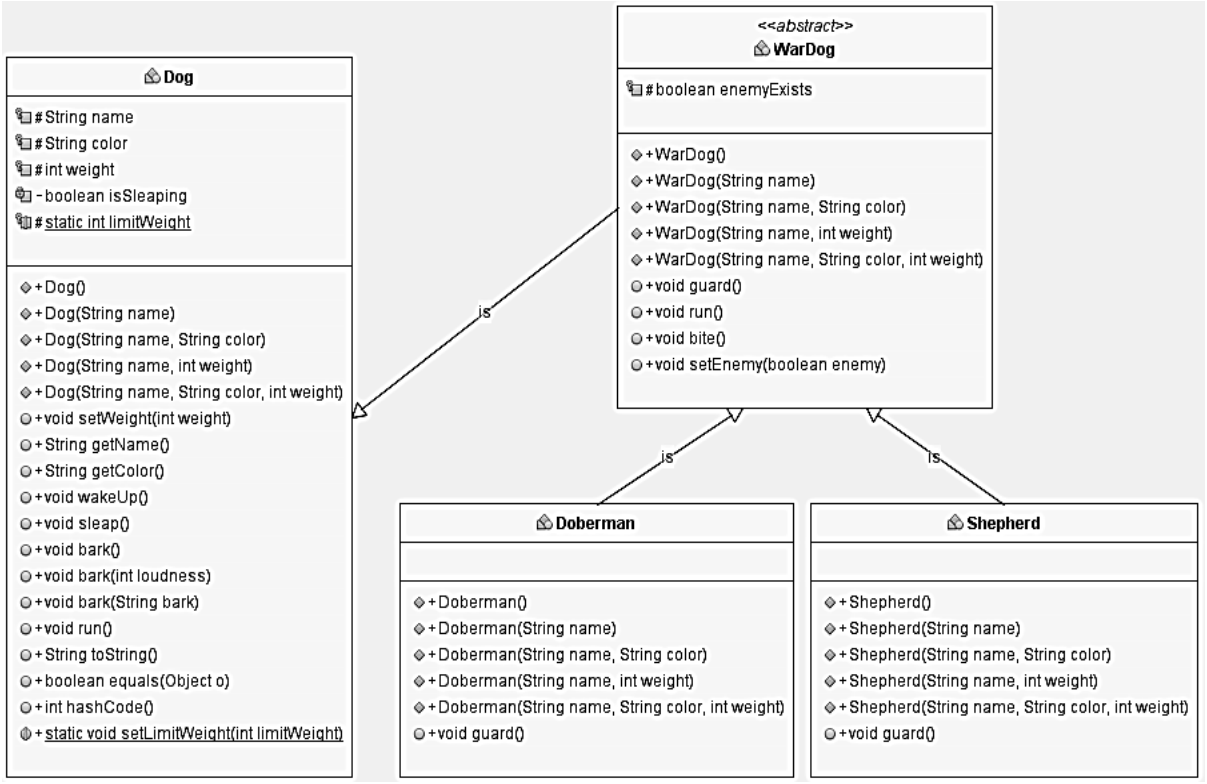

Рис. 8.16. Діаграма класів Собака,
Службовий собака, Доберман та Вівчарка

9. ВИКОРИСТАННЯ FINAL

Ключове слово final може бути застосоване до поля класу. Це дозволяє запобігти зміні вмісту поля, оскільки, воно стає константою. Фінальне поле має бути ініціалізоване під час його першого оголошення.

Наприклад, у класі Собака оголосимо фінальне поле «кількість лап» (рис. 9.1):

```
public class Dog {
    protected String name;
    protected String color;
    protected int weight;

    public final int paws = 4;
```

Рис. 9.1. Приклад реалізації фінального поля

Тепер ми можемо звертатись до цього поля, але при спробі змінити значення отримаємо помилку (рис. 9.2):

```
    public static void main(String[] args) {
        Dog dogl = new Shepherd("Bob");
        System.out.println("Dog has " + dogl.paws + " pows");
        dogl.paws = 3;
    }
```

Рис. 9.2. Приклад роботи з фінальним полем

Так само слово final можна застосувати до методів, щоб запобігти його перевизначенню. Так, наприклад, ми хочемо, щоб усі службові собаки бігали однаково, незалежно від породи. На цей момент ми, наприклад, можемо перевизначити метод run() у класі Doberman (рис. 9.3):

```
public class Doberman extends WarDog{
    public Doberman() {
    }

    @Override
    public void run() {
        System.out.println("Dobermans run as fast as nobody can");
    }
```

Рис. 9.3. Перевизначення методу «бігати» у класі Доберман

Тепер, під час роботи з вівчаркою та доберманом, коли бігтиме вівчарка, вона використовуватиме метод базового класу WardDog, а доберман – власний метод (рис. 9.4–9.5):

```
public static void main(String[] args) {
    DogTrainer.run(new WarDog[]
        {
            new Doberman("Spike", 40),
            new Shepherd("Rob", 35)
        }
    );
}
```

Рис. 9.4. Виконання методу «бігати» для добермана та вівчарки

```
run:
Dobermans run as fast as nobody can
War-dog Rob is running as wind
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 9.5. Результат виконання коду, наведеного на рис. 9.4

Заборонимо перевизначати run() у класі WarDog (рис. 9.6):

```
@Override
final public void run() {
    if (weight > 20) {
        System.out.println("War-dog " + name + " is running as wind");
    } else {
        super.run();
    }
}
```

Рис. 9.6. Фіналізація методу run

Тепер під час спроби зробити це у класі Doberman маємо помилку (рис. 9.7):

```
    }
}

@Override
public void run() {
    System.out.println("Dobermans run as fast as nobody can");
}
```

```
run() in Doberman cannot override run() in WarDog
 overridden method is final
---
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие подсказки)
```

Рис. 9.7. Спроба перевизначення фінального методу

Виконаємо ще одну цікаву реалізацію із використанням final. Закладемо наступну загальну поведінку для службового собаки. При виконанні методу «охороняти» – якщо є ворог, вона поводить себе одним чином, якщо немає – іншим. І це незмінне. А як саме вона веде себе у тому чи іншому випадку – визначається вже залежно від конкретної породи.

Спочатку змінимо базовий клас WarDog (рис. 9.8):

```
abstract public class WarDog extends Dog {
    protected boolean enemyExists;

    public final void guard() {
        if (enemyExists) {
            makeGuardAction();
        } else {
            waitForEnemy();
        }
    }

    protected abstract void makeGuardAction();

    protected abstract void waitForEnemy();
```

Рис. 9.8. Визначення фінального методу в базовому класі,
що використовує абстрактні методи

Тепер у класах Sheperd та Doberman маємо помилки – по-перше, реалізований фінальний метод guard, а по-друге не реалізовані абстрактні makeGuardAction та waitForEnemy. Виправимо це (рис. 9.9–9.10):

```
public class Doberman extends WarDog{
    public Doberman() {
    }

    @Override
    public void makeGuardAction() {
        run();
        bite();
    }

    @Override
    protected void waitForEnemy() {
        System.out.println("Doberman " + name + " has started to guard");
    }
```

Рис. 9.9. Реалізація абстрактних методів,
що використовує фінальний guard у класі «доберман»

```
public class Shepherd extends WarDog {

    public Shepherd() {
    }
    @Override
    public void makeGuardAction() {
        run();
        bark();
    }

    @Override
    protected void waitForEnemy() {
        System.out.println("Shepherd " + name + " keep silence and wait");
    }
```

Рис. 9.10. Реалізація абстрактних методів,
що використовує фінальний guard у класі «вівчарка»

Перевіримо роботу змінених класів (рис. 9.11–9.12):

```
public static void main(String[] args) {
    WarDog dog1 = new Doberman("Spike", 40);
    WarDog dog2 = new Shepherd("Rob", 35);
    WarDog[] dogs = {dog1, dog2};
    DogTrainer.guard(dogs);
    dog1.setEnemy(true);
    dog2.setEnemy(true);
    DogTrainer.guard(dogs);
}
```

Рис. 9.11. Перевірка роботи нового фінального методу

```
run:
Doberman Spike has started to guard
Shepherd Rob keep silence and wait
War-dog Spike is running as wind
War-dog Spike bites it's enemy
War-dog Rob is running as wind
Dog Rob says 'BAAARK !'
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 9.12. Результат виконання коду, наведеного на рис. 9.11

Ще один варіант використання ключового слова final – запобігання наслідування класу. При цьому неявно всі методи класу також стануть фінальними. Тому не можна одночасно оголосити клас абстрактним і фінальним, оскільки абстрактний клас є лише шаблоном і тільки його підкласи реалізують методи.

Оголосимо клас Doberman, як фінальний (рис. 9.13):

```
final public class Doberman extends WarDog{
    public Doberman() {
    }
    public Doberman(String name) {
        super(name);
    }
```

Рис. 9.13. Фіналізація класу

Тепер, під час спроби створити клас Доберман-пінчер, що наслідуватиме від добермана, отримаємо наступну помилку (рис. 9.14):

```
/**
 *
 * @author michael
 */
public class DobermanPinscher extends Doberman {

}
```

cannot inherit from final Doberman

(при нажатии сочетания клавиш AL

Рис. 9.14. Спроба розширення фінального класу

При цьому можемо спокійно створити клас Кавказька вівчарка, що наслідуватиме від класу Вівчарка (рис. 9.15).

```
 * @author michael
 */
public class CaucasianShepherdDog extends Shepherd {

}
```

Рис. 9.15. Розширення нефіналізованого класу «вівчарка»

Наведемо діаграму класів для наших собак після усіх виконаних змін у нашій роботі (рис. 9.16):

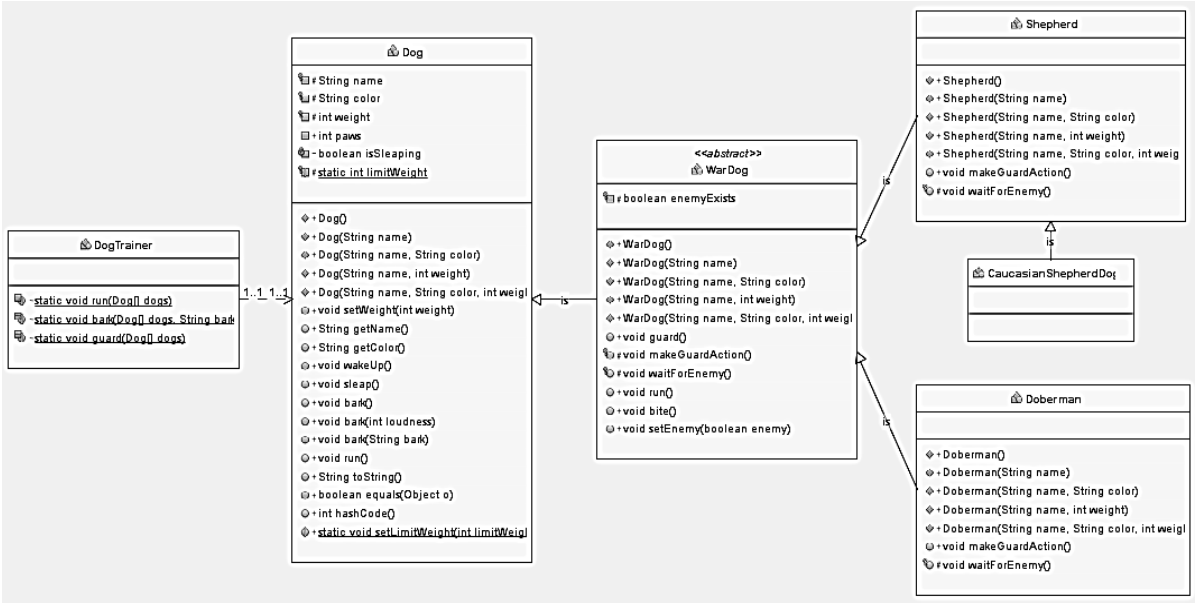

Рис. 9.16. Діаграма класів для Собака, Службовий собака, Доберман, Вівчарка, Кавказька вівчарка та Кінолог.

10. ІНТЕРФЕЙСИ

Механізм наслідування дуже зручний, але він має свої обмеження. Зокрема у java ми можемо наслідувати тільки від одного класу, на відміну, наприклад, від мови C ++, де є множинне наслідування.

У мові Java подібну проблему частково дозволяють вирішити інтерфейси. Інтерфейси визначають деякий функціонал, який не має конкретної реалізації, який потім реалізують класи, що застосовують ці інтерфейси. І один клас може застосувати багато інтерфейсів.

Щоб визначити інтерфейс, використовується ключове слово interface. Наприклад (рис. 10.1):

```
public interface Runable {
    void run();
}
```

Рис. 10.1. Приклад простого інтерфейсу

Цей інтерфейс називається Runable. Він може визначати константи і методи, які можуть мати або не мати реалізації. Методи без реалізації схожі на абстрактні методи абстрактних класів. Так, у нашому випадку оголошений один метод, який не має реалізації.

Усі методи інтерфейсу не мають модифікаторів доступу, але фактично за замовчуванням доступ public, оскільки мета інтерфейсу – визначення функціоналу для реалізації його класом. Тому весь функціонал повинен бути відкритий для реалізації.

Інтерфейси – це не класи. За допомогою ключового слова new можна створити екземпляр інтерфейсу (рис. 10.2):

```
    /**
     * @param args the comma
     */
    public static void main(
        Runable runner = new Runnable();
    }
}
```

Runnable is abstract; cannot be instantiated

(при нажатии сочетания клавиш ALT+ВВО

Рис. 10.2. Спроба створити екземпляр інтерфейсу.

Але можна оголошувати інтерфейсні змінні (рис. 10.3):

```
public static void main(String[] args) {
    Runable runner;
    ArrayList<Runable> runners;
}
```

Рис. 10.3. Використання інтерфейсу в якості типу даних

Щоб клас застосував інтерфейс, треба використовувати ключове слово implements. Імплементуємо інтерфейс Runnable до класу Dog (рис. 10.4).

```
public class Dog implements Runable {
    protected String name;
    protected String color;
    protected int weight;

    @Override
    public void run() {
        if (!isSleaping) {
            if (weight > limitWeight) {
                System.out.println("Dog " + name + " is running, but slowly");
            } else {
                System.out.println("Dog " + name + " is running very fast");
            }
        } else {
            System.out.println("Dog " + name + " can't run. It's sleaping");
        }
    }
```

Рис. 10.4. Імплементація інтерфейсу в класі

У цьому випадку клас Dog реалізує інтерфейс Runable. При цьому треба враховувати, якщо клас імплементує інтерфейс, то він має реалізувати усі методи інтерфейсу, як у випадку вище реалізованого методу run(). Потім, у методі main, ми можемо створити об’єкт класу Book і викликати його метод run().

Якщо клас не реалізує якісь методи інтерфейсу, то такий клас має визначатися як абстрактний, а його неабстрактні класи-нащадки потім реалізовуватимуть ці методи.

Класи можуть реалізовувати декілька інтерфейсів. Створимо інтерфейси Barkable і Sleapable (рис. 10.5–10.6) та імплементуємо його у класі Dog (рис. 10.7):

```
public interface Barkable {
    void bark();
    void bark(int loudness);
    void bark(String bark);
}
```

Рис. 10.5. Інтерфейс «гавкаючий»

```
public interface Sleapable {
    void wakeUp();
    void sleap();
}
```

Рис. 10.6. Інтерфейс «сплячий»

```
public class Dog implements Runable, Barkable, Sleapable {
    protected String name;
    protected String color;
    protected int weight;

    public void bark() {
        if (!isSleaping) {
            if (weight > limitWeight) {
                System.out.println("Dog " + name + " says 'BAAARK !'");
            } else {
                System.out.println("Dog " + name + " says 'av av!'");
            }
        } else {
            System.out.println("Dog " + name + " can't bark. It's sleaping");
        }
    }
    public void bark(int loudness) {
        if (loudness < 5) {
            System.out.println("Dog " + name + " says 'av av!'");
        } else if (loudness < 10) {
            System.out.println("Dog " + name + " says 'bark bark !'");
        } else {
            System.out.println("Dog " + name + " says 'BAAARK !'");
        }
    }
    public void bark(String bark) {
        System.out.println("Dog " + name + " says '" + bark + "");
    }
    @Override
    public void wakeUp() {
        if (isSleaping) {
            System.out.println("Dog " + name + " woke up");
            isSleaping = false;
        } else {
            System.out.println("Dog " + name + " has already woken up");
        }
    }
    @Override
    public void sleap() {
        if (!isSleaping) {
            System.out.println("Dog " + name + " fall asleap");
            isSleaping = true;
        } else {
            System.out.println("Dog " + name + " is already sleaping");
        }
    }
```

Рис. 10.7. Імплементація Barkable та Sleapable у класі Dog

Однією з переваг використання інтерфейсів є те, що вони дозволяють додати в програму гнучкості. Так, наприклад, може бути клас «кішка», що

імплементуватиме інтерфейс Sleapable (рис. 10.8), або Кінь, що імплементуватиме Runable (10.9):

```
    public void wakeUp() {
        System.out.println("Cat " + name + " woke up");
    }
    @Override
    public void sleap() {
        System.out.println("Cat " + name + " fall asleap");
    }
}
```

Рис. 10.8. Клас «кішка», що імплементує інтерфейс «сплячий»

```
public class Horse implements Runable{
    private String name;
    public Horse(String name) {
        this.name = name;
    }
    @Override
    public void run() {
        System.out.println("The horse " + name + " is running");
    }

}
```

Рис. 10.9. Клас «кінь», що імплементує інтерфейс «бігаючий»

Тепер в основному коді створимо собаку та коня, що будуть разом бігати, і собаку та кішку, які будуть разом засинати та просинатися (рис. 10.10–10.11):

```
    public static void main(String[] args) {
        WarDog dog = new Doberman("Jina", "brown", 40);
        Horse horse = new Horse("Rodrik");
        Cat cat = new Cat("Tom");
        Runable[] runners = { horse, dog };
        for(Runable runner: runners) {
            runner.run();
        }

        Sleapable[] sleapyes = { dog, cat };
        for(Sleapable sleapy: sleapyes) {
            sleapy.sleap();
        }
        for(Sleapable sleapy: sleapyes) {
            sleapy.wakeUp();
        }
    }
}
```

Рис. 10.10. Створення масивів із типом даних інтерфейсу та робота з ними

```
run:
The horse Rodrik is running
War-dog Jina is running as wind
Dog Jina fall asleap
Cat Tom fall asleap
Dog Jina woke up
Cat Tom woke up
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 10.11. Результат роботи коду, наведеного на рис. 10.10

За необхідності можна скористатись явним перетворенням типів та виконати метод, притаманний представнику конкретного класу. Наприклад, візьмемо масив «бігунів», та змусимо гавкати тих, хто вміє це робити (рис. 10.12):

```
    public static void main(String[] args) {
        WarDog dog1 = new Doberman("Jina", "brown", 40);
        Horse horse = new Horse("Rodrik");
        Dog dog2 = new Dog("Gerbert");
        Runable[] runners = { horse, dog1, dog2 };
        for(Runable runner: runners) {
            if (runner instanceof Barkable) {
                ((Barkable)runner).bark();
            }
        }
    }
```

Рис. 10.12. Перевірка на приналежність до інтерфейсу

```
run:
Dog Jina says 'BAAARK !'
Dog Gerbert says 'av av!'
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 10.13. Результат роботи коду, наведеного на рис. 10.12

11. УЗАГАЛЬНЕНІ КЛАСИ ТА МЕТОДИ

Узагальнення або generics (узагальнені типи і методи) дозволяють нам піти від жорсткого визначення використовуваних типів. Це параметризовані типи. З їх допомогою можна оголошувати класи, інтерфейси і методи, де тип даних вказано у вигляді параметра.

Розглянемо метод run() класу DogTrainer. Зараз він працює тільки із масивом собак, але у нас є також клас кінь, що вміє бігати. Крім того, параметр може бути передано, як інстанцію інтерфейсів Sleapable або Barkable, конкретні реалізації яких також можуть вміти бігати. Перетворимо цей метод таким чином – передаємо туди масив даних одного типу, і виконуємо метод run() для всіх представників інтерфейсу runnable (рис. 11.1):

```
static <T> void run(T[] runners) {
    for(T runner: runners) {
        if (runner instanceof Runable) {
            ((Runable)runner).run();
        }
    }
}
```

Рис. 11.1. Реалізація узагальненого методу

Використаємо зазначений метод для команди «бігати» масиву собак та котів, що імплементують інтерфейс sleapable (рис. 11.2–11.3):

```
public static void main(String[] args) {
    WarDog dog1 = new Doberman("Jina", "brown", 40);
    Cat cat = new Cat("Top");
    Dog dog2 = new Dog("Gerbert");
    Sleapable[] sleapies = { dog1, cat, dog2 };
    DogTrainer.<Sleapable>run(sleapies);
}
```

Рис. 11.2. Використання узагальненого методу

```
run:
War-dog Jina is running as wind
Dog Gerbert is running very fast
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 11.3. Результат роботи коду, наведеного на рис. 11.2

Було б правильно аналогічним чином переробити також методи bark та guard для класу DogTrainer. У цьому випадку буде доцільним створювати не узагальнені методи, а зробити узагальненим весь клас DogTrainer, та прибрати модифікатор static для методів цього класу (рис. 11.3).

```
public class DogTrainer<T> {
    public void run(T[] runners) {
        for(T runner: runners) {
            if (runner instanceof Runable) {
                ((Runable)runner).run();
            }
        }
    }
    public void bark(T[] barkers, String bark) {
        for(T barker: barkers) {
            if (barker instanceof Barkable) {
                if ("".equals(bark)) {
                    ((Barkable)barker).bark();
                } else {
                    ((Barkable)barker).bark(bark);
                }
            }
        }
    }
    public void guard(T[] dogs) {
        for(T dog: dogs) {
            if (dog instanceof WarDog) {
                ((WarDog)dog).guard();
            }
        }
    }
}
```

Рис. 11.3. Реалізація узагальненого класу

Тепер під час роботи із DogTrainer доведеться створювати нову інстанцію класу вказавши тип даних, якому будуть віддаватися команди (рис. 11.4–11.5):

```
    public static void main(String[] args) {
        WarDog dog1 = new Doberman("Jina", "brown", 40);
        Horse horse = new Horse("Gerbert");
        Dog dog2 = new Dog("Bob");

        Runable[] runners = { dog1, horse, dog2 };
        DogTrainer<Runable> trainer = new DogTrainer<Runable>();
        trainer.run(runners);
        System.out.println("=================");
        trainer.bark(runners, "");
        System.out.println("=================");
        trainer.guard(runners);
    }
}
```

Рис. 11.4. Приклад використання узагальненого класу

```
run:
War-dog Jina is running as wind
The horse Gerbert is running
Dog Bob is running very fast
==========
Dog Jina says 'BAAARK !'
Dog Bob says 'av av!'
==========
Doberman Jina has started to guard
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 11.5. Результат роботи коду, наведеного на рис. 11.4

Узагальненими також можуть бути і інтерфейси. Розглянемо приклад, коли у кішки є власник, і це може бути об’єкт класу Person або звичайний рядок String. Для початку виконаємо клас Person (рис. 11.5):

```
public class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Рис. 11.5. Реалізація нового класу «Person»

Інтерфейс має забезпечити реалізацію методів для встановлення та отримання власнику (рис. 11.6)

```
public interface Ownerable<T> {
    T getOwner();
    void setOwner(T owner);
}
```

Рис. 11.6. Реалізація інтерфейсу «маючий власника»

Реалізувати цей інтерфейс можна двома шляхами. Або вказати тип для власника при створенні об’єкта (рис. 11.7):

```
public class Cat<T> implements Sleapable, Ownerable<T> {
    private String name;
    private T owner;

@Override
public T getOwner() {
    return owner;
}

@Override
public void setOwner(T owner) {
    this.owner = owner;
}
```

```
public static void main(String[] args) {
    Cat<Person> cat= new Cat<Person>("Tom");
    cat.setOwner(new Person("Jack Smit"));
}
```

Рис. 11.7. Перший шлях імплементації узагальненого інтерфейсу

Або відразу при визначенні класу (рис. 11.8):

```
public class Cat implements Sleapable, Ownerable<Person> {
    private String name;
    private Person owner;

    @Override
    public Person getOwner() {
        return owner;
    }

    @Override
    public void setOwner(Person owner) {
        this.owner = owner;
    }
    public static void main(String[] args) {
        Cat cat= new Cat("Tom");
        cat.setOwner(new Person("Jack Smit"));
    }
```

Рис. 11.8. Другий шлях імплементації узагальненого інтерфейсу

12. РЕАЛІЗАЦІЯ ІНТЕРФЕЙСІВ COMPARATOR ТА COMPARABLE

У своїй роботі програмісти часто стикаються з тим, що треба щось порівнювати або сортувати – наприклад, дізнатися, яке число більше або відсортувати множину чисел за зростанням. Але для того, щоб щось порівняти або впорядкувати, нам потрібно порівнювати об’єкти за якимись правилами. Тут, здавалося б, все просто – ми можемо порівнювати та сортувати числа, та й у сортуванні за алфавітом немає нічого складного (рис. 12.1–12.4):

```
public static void main(String[] args) {
    int x, y;
    x = 10; y = 5;
    if (x > y) {
        System.out.println("x greater then y");
    } else {
        System.out.println("x equels or less then y");
    }

}
```

Рис. 12.1. Порівняння 2-х змінних типу int

```
run:
x greater then y
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Рис. 12.2. Результат роботи коду, наведеного на рис. 12.1

```
public static void main(String[] args) {
    int[] arr = {4, -5, 8, 15, 1};
    for (int el: arr) {
        System.out.print(el + " ");
    }
    System.out.println();
    Arrays.sort(arr);
    for (int el: arr) {
        System.out.print(el + " ");
    }
    System.out.println();
}
```

Рис. 12.3. Сортування масиву типу int

```
run:
4 -5 8 15 1
-5 1 4 8 15
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 12.4. Результат роботи коду, наведеного на рис. 12.3

Так, з такими даними все легко. Але як нам порівняти два об’єкти класу Dog або Cat?

```
public static void main(String[] args) {
    Dog dog1 = new Dog("Spike");
    Dog dog2 = new Doberman("Ralf", "black", 40);
    if (dog1 > dog2) {
        System.out.println("dog1 is greater then dog2");
    }
}
```

```
bad operand types for binary operator '>'
 first type:  Dog
 second type: Dog
```

Рис. 12.5. Спроба порівняння 2-х об’єктів з використанням «більше»

Ми не можемо взяти і порівняти 2 об’єкти, оскільки вони мають багато полів, і не завжди примітивних типів. Відповідно, не зрозуміло як саме виконувати порівняння.

Зрозуміло, що з цієї ж причини ми не можемо відсортувати масив або колекцію собак (рис. 12.6):

```
    public static void main(String[] args) {
        Dog[] dogs = {
            new Dog("Spike"),
            new Doberman("Ralf", "black", 40)
        };
        for(Dog dog: dogs) {
            System.out.println(dog);
        }
        System.out.println("============");

        Arrays.sort(dogs);

        for(Dog dog: dogs) {
            System.out.println(dog);
        }
        System.out.println("============");
    }
}
```

Рис. 12.6. Спроба сортування масиву об’єктів

```
run:
Dog Spike with weight 0 and null fur color
Dog Ralf with weight 40 and black fur color
============
Exception in thread "main" java.lang.ClassCastException: l
        at java.util.ComparableTimSort.countRunAndMakeAsce
        at java.util.ComparableTimSort.sort(ComparableTimS
```

Рис. 12.7. Результат роботи коду, наведеного на рис. 12.6

Для того, щоб можна було скористатися стандартним методом sort статичного класу Arrays необхідно, щоб клас елементів масиву, що передається у якості аргумента, реалізовував інтерфейс Comparable. Цей інтерфейс складається з одного метода compareTo, що забезпечує можливість порівняти 2 елементи між собою. Звернемо увагу, що інтерфейс є узагальненим, тобто потребує вказання типу даних. Реалізуємо цей інтерфейс у класі Dogs (рис. 12.8):

```
public class Dog implements Runable, Barkable, Sleapable, Comparable<Dog> {
    protected String name;
    protected String color;
    protected int weight;

    @Override
    public int compareTo(Dog t) {
        return weight - t.weight;
    }
```

Рис. 12.8. Реалізація узагальненого інтерфейсу Comparable у класі Dog

У методі compareTo ми вказуємо, яким саме чином порівнювати 2 об’єкти. Якщо метод повертає від’ємне число, вважається що поточний елемент менше, якщо додатне – більше, якщо 0 – то дорівнює. У нашому випадку ми впорядковуємо собак за вагою. Тепер сортування масиву собак відбувається без помилок (рис. 12.9):

```
public static void main(String[] args) {
    Dog[] dogs = {
        new Dog("Spike", 15),
        new Doberman("Ralf", "black", 40),
        new Shepherd("Wolf", "gray", 30)
    };
    for(Dog dog: dogs) {
        System.out.println(dog);
    }
    System.out.println("============");

    Arrays.sort(dogs);

    for(Dog dog: dogs) {
        System.out.println(dog);
    }
    System.out.println("============");
}
```

Рис. 12.9. Сортування масиву собак після імплементації інтерфейсу Comparable

```
run:
Dog Spike with weight 15 and null fur color
Dog Ralf with weight 40 and black fur color
Dog Wolf with weight 30 and gray fur color
==========
Dog Spike with weight 15 and null fur color
Dog Wolf with weight 30 and gray fur color
Dog Ralf with weight 40 and black fur color
==========
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 12.10. Результат роботи коду, наведеного на рис. 12.9

Також у разі необхідності за допомогою нового методу можна виконати просте порівняння об’єктів (рис. 12.11–12.12):

```
    public static void main(String[] args) {
        Dog dog1 = new Dog("Ralf", 20);
        Dog dog2 = new Shepherd("Juli", 35);
        if (dog2.compareTo(dog1) > 0) {
            System.out.println("dog2 is greater then dog1");
        }
    }
```

Рис. 12.11. Порівняння собак за допомогою методу compareTo

```
run:
dog2 is greater then dog1
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 12.12. Результат роботи коду, наведеного на рис. 12.11

Як бути, якщо тепер потрібно впорядкувати собак не за вагою, а за кличкою? Якщо переписати compareTo, перестане працювати впорядкування за вагою, а воно також потрібне.

Для таких випадків метод Sort перевантажений додатковою версією з 2-ма параметрами, де 2-й – це саме метод порівняння під час сортування. У цей параметр передається екземпляр класу, що реалізує інтерфейс Comparator, який у свою чергу містить метод compare. У цьому методі і задається правило для порівняння.

Реалізуємо клас для порівняння собак за кличкою та подальшим сортуванням в алфавітному порядку (рис. 12.13–12–15):

```
class CompareByName implements Comparator<Dog> {
    @Override
    public int compare(Dog t, Dog t1) {
        return t.name.compareTo(t1.name);
    }
}
```

Рис. 12.13. Реалізація компаратора для порівняння 2-х об’єктів

```
public static void main(String[] args) {
    Dog[] dogs = {
        new Dog("Spike", 15),
        new Doberman("Ralf", "black", 40),
        new Shepherd("Wolf", "gray", 30)
    };
    for(Dog dog: dogs) {
        System.out.println(dog);
    }
    System.out.println("============");

    Arrays.sort(dogs, new CompareByName());

    for(Dog dog: dogs) {
        System.out.println(dog);
    }
    System.out.println("============");
}
```

Рис. 12.14. Сортування собак
із використанням реалізованого компаратора

```
run:
Dog Spike with weight 15 and null fur color
Dog Ralf with weight 40 and black fur color
Dog Wolf with weight 30 and gray fur color
============
Dog Ralf with weight 40 and black fur color
Dog Spike with weight 15 and null fur color
Dog Wolf with weight 30 and gray fur color
============
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 12.15. Результат роботи коду, наведеного на рис. 12.14

13. АНОНІМНІ КЛАСИ ТА ЛЯМБДА-ВИРАЗИ

Анонімний клас – це локальний клас без імені. Можна оголосити анонімний (безіменний) клас, який може розширити (extends) інший клас або реалізувати (implements) інтерфейс. Оголошення такого класу виконується одночасно зі створенням його об’єкта за допомогою оператора new.

Анонімні класи дозволяють зробити ваш код більш стислим та одночасно оголосити і створити екземпляр класу. Вони подібні до локальних класів, за винятком того, що не мають назви. Анонімні класи зручно використовувати, якщо нам потрібно використовувати локальний клас лише один раз.

Припустимо, нам потрібен масив об’єктів, що вміють засинати та просинатися, але трьома різними способами. На поточний момент у нас імплементують інтерфейс Sleapable класи Dog та Cat. Можна, звичайно, створити ще клас Mouse, у якому імплементувати Sleapable та використати його, але якщо він надалі нам не знадобиться, робити це не дуже доцільно. Отже, скористаємось анонімним класом (рис. 13.1–13.2):

```
public static void main(String[] args) {
    Dog dog = new Shepherd("Spark", "brown", 30);
    Cat cat = new Cat("Tom");
    Sleapable sleapy = new Sleapable() {
        @Override
        public void wakeUp() {
            System.out.println("Sleapy says: 'Who woke me up? I'll kill you!'");
        }
        @Override
        public void sleap() {
            System.out.println("Sleapy is snoring like a monster");
        }
    };
    Sleapable[] sleapies = {dog, cat, sleapy};
    for(Sleapable s: sleapies) {
        s.sleap();
        s.wakeUp();
    }
}
```

Рис. 13.1. Реалізація анонімного класу,
що імплементує інтерфейс та його використання

```
run:
Dog Spark fall asleap
Dog Spark woke up
Cat Tom fall asleap
Cat Tom woke up
Sleapy is snoring like a monster
Sleapy says: 'Who woke me up? I'll kill you!'
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 13.2. Результат виконання коду, наведеного на рис. 13.1

Інший варіант – ми хочемо створити «поганого собаку», який буде під час виконання методу «бігати» не тільки бігати, а ще й гавкати та пісяти на килим (рис. 13.3–13.4):

```
public static void main(String[] args) {
    Dog BadDog = new Dog("Spike", 20) {
        @Override
        public void run() {
            super.run();
            bark();
            System.out.println("Bad dog " + name + " piss to the carpet");
        }
    };
    BadDog.run();
}
```

Рис. 13.3. Перевизначення методу в анонімному класі

```
run:
Dog Spike is running very fast
Dog Spike says 'av av!'
Bad dog Spike piss to the carpet
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 13.4. Результат роботи коду, наведеного на рис. 13.3

Анонімні класи також можуть бути похідними від абстрактних класів та реалізовувати їх абстрактні методи. Створимо скаженого службового собаку, що кусає всіх підряд (рис. 13.5–13.6):

```
public static void main(String[] args) {
    WarDog CrazyDog = new WarDog("Spike", 20) {
        @Override
        protected void makeGuardAction() {
            super.bite();
        }
        @Override
        protected void waitForEnemy() {
            System.out.println("Crazy war-dog " + name + " bites the first man it sees");
        }
    };
    CrazyDog.setEnemy(true);
    CrazyDog.guard();
    CrazyDog.setEnemy(false);
    CrazyDog.guard();
}
```

Рис. 13.5. Анонімний клас, що є похідним від абстрактного

```
ывод - laba1 (run)  ×
  run:
  War-dog Spike bites it's enemy
  Crazy war-dog Spike bites the first man it sees
  СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 13.6. Результат роботи коду, наведеного на рис. 13.5

Розглянемо ще один приклад використання анонімного класу – для виконання сортування наших собак за кольором хутра. Припустимо, що нам потрібно зробити це одноразово, тобто реалізувати окремий клас та імплементувати у ньому Comparator недоцільно. Скористаємось анонімним класом (рис. 13.7–13.8):

```
public static void main(String[] args) {
    Dog[] dogs  = {
        new Dog("Ralf", "white", 20),
        new Shepherd("Wolf", "gray", 30),
        new Doberman("Jina", "black", 40),
        new Dog("Bob", "brown", 10),
    };
    Arrays.sort(dogs, new Comparator<Dog>() {
        @Override
        public int compare(Dog t, Dog t1) {
            return t.color.compareTo(t1.color);
        }
    });
    for (Dog d: dogs) {
        System.out.println(d);
    }
}
```

Рис. 13.7. Використання анонімного класу для впорядкування масиву

```
run:
Dog Jina with weight 40 and black fur color
Dog Bob with weight 10 and brown fur color
Dog Wolf with weight 30 and gray fur color
Dog Ralf with weight 20 and white fur color
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 13.8. Результат роботи коду, наведеного на рис. 13.7.

Серед нововведень, які були привнесені в мову Java з появою JDK 8, варто особливо відмітити лямбда-вирази. Лямбда представляє набір інструкцій, які можна виділити в окрему змінну, і потім багато разів викликати їх в різних місцях програми.

Основу лямбда-виразів становить лямбда-оператор, який представляє стрілку ▸. Цей оператор розділяє лямбда-вираз на дві частини: ліва частина містить список параметрів виразу, а права власне являє тіло лямбда-виразу, де виконуються всі дії.

Лямбда-вираз не виконується самостійно, а утворює реалізацію методу, визначеного у функціональному інтерфейсі. При цьому важливо, що функціональний інтерфейс має містити тільки один єдиний метод без реалізації.

У нашій поточній реалізації є функціональний інтерфейс Runable, що містить 1 метод run(). Виконаємо реалізацію масиву бігунів, який буде складатись із 2-х собак, коня, та лямбда-виразу (рис. 13.9–13.10):

```
Runable[] runners = {
    new Dog("Wolf", 20),
    new Shepherd("Jack", "brawn", 35),
    new Horse("Night"),
    () -> System.out.println("Lambda runner is defenetly the fastest runner")
};
for(Runable r: runners) {
    r.run();
}
```

Рис. 13.9. Імплементація інтерфейсу у вигляді лямбда-виразу

```
run:
Dog Wolf is running very fast
War-dog Jack is running as wind
The horse Night is running
Lambda runner is defenetly the fastest runner
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 13.10. Результат роботи коду, наведеного на рис. 13.9

Розглянемо ще один приклад реалізації лямбда-виразу – під час сортування масиву або колекції об’єктів. Реалізуємо попередній приклад сортування собак за кольором хутра, але тепер із використанням лямбда-виразу (рис. 13.11–13.12):

```
public static void main(String[] args) {
    Dog[] dogs = {
        new Dog("Snow", "white", 10),
        new Doberman("Jack", "black", 40),
        new Shepherd("Spark", "brown", 30),
        new Dog("Rob", "Spotted", 15)
    };
    Arrays.sort(
        dogs,
        (dl, d2) -> dl.color.compareTo(d2.color)
    );
    for(Dog d: dogs) {
        System.out.println(d);
    }
}
```

Рис. 13.11. Сортування масиву об’єктів
з використанням лямбда-виразу

```
run:
Dog Rob with weight 15 and Spotted fur color
Dog Jack with weight 40 and black fur color
Dog Spark with weight 30 and brown fur color
Dog Snow with weight 10 and white fur color
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 1 секунда)
```

Рис. 13.12. Результат роботи коду, наведеного на рис. 13.11

14. ВИКЛЮЧЕННЯ

Виключення – це позаштатна ситуація, помилка під час виконання програми. Найпростіший приклад – ділення на нуль. Можна вручну відстежувати виникнення подібних помилок, а можна скористатися спеціальним механізмом виключень, який спрощує створення великих надійних програм, зменшує обсяг необхідного коду і підвищує впевненість в тому, що в застосунку не буде необробленої помилки.

У методі, в якому відбувається помилка, створюється і передається спеціальний об'єкт. Метод може або обробити виключення самостійно, або пропустити його. У будь-якому випадку виключення ловиться і обробляється. Виняток може з'явитися завдяки самій системі, або ви самі можете його створити. Системні виключення виникають в разі неправильного використання мови Java або заборонених прийомів доступу до системи. Ваші власні виключення обробляють специфічні помилки вашої програми.

Подивимося, що відбувається під час виникнення ситуації із діленням на 0 (рис. 14.1–14.2):

```
public static void main(String[] args) {
    int x = 10, y = 0;
    int z = x / y;
}
```

Рис. 14.1. Помилка ділення на 0

```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at labal.Labal.main(Labal.java:32)
C:\Users\michael\AppData\Local\NetBeans\Cache\8.2\executor-snippets\r
СБОРКА ЗАВЕРШЕНА СО СБОЕМ (общее время: 0 секунд)
```

Рис. 14.2. Результат роботи коду, наведеного на рис. 14.1

Бачимо, що основний код повертає ArithmeticException із текстом «/ by zero». При бажанні це виключення може бути оброблене для уникнення виникнення помилки (рис. 14.3–14.4):

```
public static void main(String[] args) {
    int x = 10, y = 0;
    try {
        int z = x / y;
        System.out.println("z = " + z);
    } catch(ArithmeticException e) {
        System.out.println("ArithmeticException catched. " + e.getMessage());
    }
}
```

Рис. 14.3. Обробка помилки ділення на 0

```
run:
ArithmeticException catched. / by zero
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 14.4. Результат роботи коду, наведеного на рис. 14.3

Існує п’ять ключових слів, які використовуються у виключеннях: try, catch, throw, throws, finally.

Оператори програми, які ви хочете відслідковувати, поміщаються в блок try. Якщо виключення відбулося, то воно створюється і передається далі.

– Ваш код може перехопити виключення за допомогою блоку catch і обробити його.

– Системні виключення автоматично передаються самою системою. Для ручного виключення, використовується throw.

– Будь-яке виключення, створене і передане всередині методу, має бути вказано у його інтерфейсі ключовим словом throws.

– Код, який варто виконати обов’язково після завершення блоку try, розміщується у блоці finally.

Доповнимо наш приклад блоком finaly, в якому будемо виводити z на консоль (рис. 14.5–14.6):

```
public static void main(String[] args) {
    int x = 10, y = 0, z = 0;
    try {
        z = x / y;
    } catch(ArithmeticException e) {
        System.out.println("ArithmeticException catched. " + e.getMessage());
    } finally {
        System.out.println("z = " + z);
    }
}
```

Рис. 14.5. Приклад використання блоку finally

```
run:
ArithmeticException catched. / by zero
z = 0
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 14.6. Результат роботи коду, наведеного на рис. 14.5

Тепер виконаємо у нашому прикладі повернення нового виключення з своїм текстом помилки (рис. 14.7–14.8):

```
public static void main(String[] args) throws Exception {
    int x = 10, y = 0, z = 0;
    try {
        z = x / y;
    } catch(ArithmeticException e) {
        throw new Exception("Арифметична помилка. Перевірте змінну 'y'");
    } finally {
        System.out.println("z = " + z);
    }
}
```

Рис. 14.7. Зміна повідомлення у виключенні на власне повідомлення

```
run:
z = 0
Exception in thread "main" java.lang.Exception: Арифметична помилка. Перевірте змінну 'y'
        at labal.Labal.main(Labal.java:35)
C:\Users\michael\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java retu
СБОРКА ЗАВЕРШЕНА СО СБОЕМ (общее время: 0 секунд)
```

Рис. 14.8. Результат роботи коду, наведеного на рис. 14.7

Клас Exception може бути розширено власним класом. Реалізуємо клас MyException, метод що його повертає, та обробимо його в основному коді (рис. 14.4–14.12):

```
class MyException extends Exception {
    int code;
    public MyException() {
        super("Арифметична помилка. Перевірте змінну 'y'");
        code = 1;
    }
    @Override
    public String toString() {
        return "MyException with code " + code + " " + getMessage();
    }
}
```

Рис. 14.9. Реалізація власного класу, що наслідує від Exception

```
private static int devide(int x, int y) throws MyException {
    int z = 0;
    try {
        z = x / y;
        return z;
    } catch(ArithmeticException e) {
        throw new MyException();
    }
}
```

Рис. 14.10. Метод, що повертає виключення створеного типу

```
public static void main(String[] args) {
    int x = 10, y = 0, z = 0;
    try {
        z = devide(x, y);
    } catch(MyException e) {
        System.out.println(e);
    } finally {
        System.out.println("z = " + z);
    }
}
```

Рис. 14.11. Код, що викликає створений метод
та обробляє виключення, що повертається

```
run:
MyException with code 1 Арифметична помилка. Перевірте змінну 'y'
z = 0
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 14.12. Результат роботи коду, наведеного на рис. 14.11

Реалізуємо у класі Dog перевірку, щоб вага собаки була додатною, інакше будемо повертати виключення DogException (рис. 14.13–14.14):

```
class DogException extends Exception {
    int code;
    public DogException(String mess, int code) {
        super(mess);
        this.code = code;
    }
    @Override
    public String toString() {
        return "DogException with code " + code + " : " + getMessage();
    }
}
```

Рис. 14.13. Створення нового класу для виключення

```
    public Dog(String name, int weight) throws DogException {
        this(name);
        if (weight < 0) {
            throw new DogException("Dog weight must be positive", 1);
        }
        this.weight = weight;
    }
    public Dog(String name, String color, int weight) throws DogException {
        this(name, color);
        if (weight < 0) {
            throw new DogException("Dog weight must be positive", 1);
        }
        this.weight = weight;
    }
```

Рис. 14.14. Повернення виключення при створенні собаки із недопустимим значенням ваги

Звернемо увагу, що тепер неможливо викликати конструктор із вказанням ваги за межами try…catch (14.15):

```
    public static void main(String[
        Dog d1 = new Dog("Jack");
        Dog d2 = new Dog("Spike", 20);
    }
```

unreported exception DogException; must be caught or declared to be thrown
---
(при нажатии сочетания клавиш ALT+ВВОД отображаются всплывающие п

Рис. 14.15. Спроба створення собаки за межами try…catch

Виконаємо створення двох екземплярів класу собака у try…catc (рис. 14.16–14.17):

```
    public static void main(String[] args) {
        try {
            Dog d1 = new Dog("Jack", "white", 15);
            System.out.println(d1);
            Dog d2 = new Dog("Spike", -20);
            System.out.println(d2);
        } catch (DogException ex) {
            System.out.println(ex);
        }
    }
```

Рис. 14.16. Створення собаки у блоці try…catch

```
run:
Dog Jack with weight 15 and white fur color
DogException with code 1 : Dog weight must be positive
СБОРКА УСПЕШНО ЗАВЕРШЕНА (общее время: 0 секунд)
```

Рис. 14.17. Результат виконання коду, наведеного на рис. 14.16

ЗАВДАННЯ
ДЛЯ ІНДИВІДУАЛЬНОГО ВИКОНАННЯ

Завдання № 1.

Реалізувати клас, що має як мінімум 2 числових та 2 текстових поля, 4 методи та 4 конструктори. В основному коді створити 2 об’єкти класу та продемонструвати роботу з ними.

Завдання № 2.

У класі, реалізованому в першій роботі, зробити всі поля приватними та додати мінімум 2 гетери та 2 сетери. Перевантажити 2 методи свого класу. Продемонструвати роботу із полями та перевантаженими методами класу.

Завдання № 3.

Розширити власний клас із попередньої роботи новим класом із мінімум 3-ма конструкторами, 2-ма власними полями та 2-ма власними методами. У тілі нових методів виконати звернення до методів базового класу. Мінімум в одному конструкторі звернутись до конструктора базового класу.

В основному коді створити екземпляр нового класу та продемонструвати роботу з методами базового та поточного класу. (Створити екземпляр, із типом поточного та базового класу, показати різницю в роботі).

Завдання № 4.

Для створених у попередній роботі базового та похідного класів:

У класі-нащадку перевизначити 2 методи базового класу, в одному з яких звернутись до методу базового класу.

Об’явити масив та колекцію із типом базового класу та заповнити елементами базового та похідного класів. Обійти всі елементи масиву та колекції (різними способами) та виконати для масиву перший перевизначений метод, для колекції – другий.

Для базового або похідного класів, на власний розсуд, перевизначити методи toString та equals і продемонструвати результат в основному коді.

Завдання № 5.

У базовому та похідному класі створити по одному статичному полю та методу (можна більше). Продемонструвати звернення до них із самого класу, класу нащадку та основного коду програми.

Реалізувати клас, що складається лише зі статичних методів, серед яких такі, що мають параметри із типом даних базового та похідного класів.

Завдання № 6.

Реалізувати ієрархію – базовий абстрактний клас та декілька (мінімум 2) похідних класів. У базовому класі мінімум 2 поля та 4 методи, 2 з яких абстрактні. В одному з похідних класів перевизначити метод базового.

Продемонструвати роботу із екземплярами похідних класів, створивши колекцію з типом базового класу, та викликаючи для кожного елемента один (або декілька) методів базового класу.

Завдання № 7.

У власній ієрархії класів реалізувати фінальне поле, фінальний метод та фінальний клас. Продемонструвати особливості роботи із такими полями, методами та класами.

Завдання № 8.

У власній ієрархії класів реалізувати мінімум 3 інтерфейси та 2 класи (які не наслідують один одного), що їх імплементують. 1 клас має імплементувати 2 або більше інтерфейсів.

Продемонструвати роботу із масивом 2-х або більше класів, що не наслідують один одного, але імплементують один інтерфейс. Використати явне перетворення типів для отримання доступу до методів одного з класів.

Завдання № 9.

На власній предметній області реалізувати мінімум 1 узагальнене поле, 2 узагальнених методи, 1 узагальнений клас із 1 узагальненим полем та методом та 1 узагальнений інтерфейс. Імплементувати узагальнений інтерфейс у клас двома способами.

Продемонструвати роботу із створеними полями, методами, класами та інтерфейсами.

Завдання № 10.

Реалізувати сортування масиву (або колекції) елементів власного класу за 3-ма полями із використання методу Sort() та інтерфейсів Comparable і Comparator.

Завдання № 11.

Реалізувати сортування масиву (або колекції) елементів власного класу за 3-ма полями із використанням методу Sort(), інтерфейсів Comparable і Comparator, анонімних класів та лямбда-виразів.

Завдання № 12.

У методах власних класів реалізувати обробку виключень та використати конструкції try, catch, throw, throws, finally. Розширити класс Exception та використати отриманий клас із throw. Обробити виклик методу з throws у блоці try…catch.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Эккель Брюс. Философия Java ; 4-е изд., Питер / Брюс Эккель. 2015. – 1159 ст. ISBN- 978-5-496-01127-3.
2. Шилдт Герберт. Java 8. Полное руководство ; 9-е изд. / Герберт Шилдт. Вильямс, 2015, 1 376 с. ISBN 978-5-8459-1918-2.
3. Р. Седжвик. Алгоритмы на Java ; 4-е изд. / Р. Седжвик, К. Уэйн. Вильямс, 2013. – 848 с., ISBN-978-5-8459-1781-2.
4. Хорстман Кей. Java Библиотека профессионала т. 1. Основы ; 9-е изд. / Кей Хорстман, Гари Корнелл. Вильямс, 2014, 864 с., ISBN-978-5-8459-1869-7.
5. Хорстман Кей. Java Библиотека профессионала т. 2. Расширенные средства ; 9-е изд., Кей Хорстман, Гари Корнелл. Вильямс, 2014, 1008 с., ISBN-978-5-8459-1870-3.
6. Сьерра Кэтти. Изучаем Java ; 2-е изд. / Кэтти Сьерра, Берт Бейтс. Эксмо, 2016, 720 с., ISBN- 978-5-699-54574-2.
7. Гослинг Джеймс. Язык программирования Java SE 8. Подробное описание ; 5-е изд., Диалектика-Вильямс, 2017, 672 с., ISBN- 978-5-8459-1875-8.
8. Мартин Роберт С. Гибкая разработка программ на Java и C++: принципы, паттерны и методики, 2-е изд. / Роберт С. Мартин. Диалектика-Вильямс, 2017, 704 с., ISBN- 978-5-9908462-8-9.
9. Бэзинс Барт. Java для начинающих. Объектно-ориентированный подход ; 3-е изд. / Барт Бэзинс, Эйми Бэкил, Зеппе Ванден Бруке Питер, 2018, 688 с., ISBN- 978-5-496-02402-0.
10. Bloch Joshua. Effective Java ; 3rd Edition / Joshua Bloch. Addison-Wesley Professional, 2018, 412 p., ISBN-978-0134685991.
11. Savitch Walter. Absolute Java (6th Edition) ; 6th Edition / Walter Savitch, Kenrick Mock. Pearson, 2015, 1296 p., ISBN-13 : 978-0134041674.

ДЛЯ НОТАТОК

Навчальне видання

Дворецький *Михайло Леонідович*
Боровльова *Світлана Юріївна*
Нездолій *Юрій Олексійович*
Дворецька *Світлана Володимирівна*

ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ НА МОВІ JAVA

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

Для підготовки бакалаврів у галузі знань «Інформаційні технології»

Випуск 268

Редактор *А. Грубкіна.*
Технічний редактор, комп'ютерна верстка *Д. Кардаш.*
Друк, фальцювально-палітурні роботи *С. Волинець*.

Підп. до друку 08.04.2019
Формат 60x84$^{1}/_{16}$. Папір офсет.
Гарнітура "Times New Roman". Друк ризограф.
Ум. друк. арк. 7,90. Обл.-вид. арк. 1,30.
Тираж 5 пр. Зам. № 5707.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.