

Лабораторна робота №3

Інкапсуляція, успадкування, поліморфізм

Концепції ООП

Об'єктно-орієнтоване програмування (ООП) у Java — це підхід до розробки програмного забезпечення, у якому основними структурними одиницями є об'єкти, що поєднують дані та методи для їх обробки. Основними принципами ООП є **інкапсуляція, успадкування та поліморфізм**.

Інкапсуляція — це механізм об'єднання даних (змінних) і методів, які працюють із цими даними, в єдину сутність — клас. Інкапсуляція дозволяє приховати внутрішню реалізацію об'єкта від зовнішнього світу, відкриваючи лише необхідний інтерфейс. Це досягається за допомогою модифікаторів доступу (**private, public, protected**). Наприклад:

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

Успадкування — це механізм, який дозволяє створювати нові класи на основі вже існуючих, наслідуючи їх властивості та поведінку. Завдяки цьому можна повторно використовувати код і розширювати функціональність. У Java для успадкування використовується ключове слово **extends**. Наприклад:

```
public class Student extends Person {  
    private int grade;  
    // додаткові методи та поля  
}
```

Поліморфізм — це властивість, завдяки якій одна й та сама дія може виконуватися по-різному для різних об'єктів. У Java розрізняють перевантаження методів та перевизначення методів. Наприклад, якщо клас **Student** перевизначає метод **toString()**:

```
@Override  
public String toString() {  
    return "Student: " + getName();  
}
```

Якщо створити масив типу `Person[]` і додати туди як об'єкти `Person`, так і `Student`, то при виклику `toString()` для кожного з них буде виконуватися "свій" метод відповідно до реального типу об'єкта. Це і є прояв поліморфізму: кожен об'єкт поводиться згідно своєї природи.

Ще одним класичним прикладом поліморфізму є ієрархія класів для тварин. Наприклад, маємо базовий клас `Animal` із методом `makeSound()`, а класи `Dog` і `Cat` його перевизначають: собака гавкає, а кіт нявкає. Якщо створити масив `Animal[]` і додати туди різних тварин, то при виклику `makeSound()` для кожного об'єкта буде виконано саме той метод, який відповідає конкретній тварині.

```
public abstract class Animal {
    public abstract void makeSound();
}

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Гав-гав");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Няв-няв");
    }
}
```

Таким чином, поліморфізм дозволяє писати код, який працює з різними об'єктами, не знаючи їх точного типу.

Модифікатори доступу

Table 1. Модифікатори доступу для членів класу (поля та методи)

Модифікатор	Доступність
public	Доступний всюди.
protected	Доступний для будь-якого класу в тому ж пакеті, що й клас, а також лише для підкласів цього класу в інших пакетах.
default (без модифікатора)	Доступний лише для класів, включаючи підкласи, у тому ж пакеті, що й клас (доступність у межах пакету).
private	Доступний лише у власному класі і більше ніде.

Table 2. Модифікатори доступу для типів верхнього рівня (класи, інтерфейси, перерахування)

Модифікатор	Доступність
default (без модифікатора)	Доступний у межах власного пакету (доступність у межах пакету)
public	Доступний всюди

Table 3. Інші модифікатори доступу для типів верхнього рівня (класи, інтерфейси, перерахування)

Модифікатор	Класи	Інтерфейси	Перерахування (enum)
abstract	Не фінальний клас може бути оголошений як abstract. Клас з абстрактним методом має бути оголошений як abstract. Абстрактний клас не можна створити (інстанціювати).	Дозволено, але надлишково.	Не дозволено.
final	Не абстрактний клас може бути оголошений як final. Клас з фінальним методом не обов'язково має бути final. Клас final не можна наслідувати.	Не дозволено.	Не дозволено.

Table 4. Інші модифікатори доступу для членів класу (поля та методи)

Модифікатор	Поля	Методи
static	Визначає змінну класу.	Визначає метод класу.
final	Визначає константу.	Метод не може бути перевизначений.
abstract	Не застосовується.	Тіло методу не визначено. Клас також має бути оголошений як abstract.
synchronized	Не застосовується.	Метод може виконуватись лише одним потоком одночасно.
native	Не застосовується.	Метод реалізовано іншою мовою програмування.
transient	Значення поля не буде включено при серіалізації об'єкта.	Не застосовується.
volatile	Компілятор не буде оптимізувати доступ до значення поля.	Не застосовується.

Приклад. Птахи

Завдання

Створити абстрактний клас **Bird** і класи **Sparrow**, **Penguin**, **CamelBird**, що його розширюють.

Клас **Bird** містить змінну **name** та абстрактні методи **move**, **eat**, **getDescription**. Метод **move**, наприклад, повертає спосіб руху птаха (біжить, летить, ходить). Метод **eat** повертає масив

того, чим харчується цей птах. Метод **getDescription** повертає опис тварини.

Sparrow, Penguin, CamelBird перевизначають методи **move, eat, getDescription**.

Створіть клас **Explorer**, у якому визначте метод `void treatBird(Bird bird)`. Нехай цей метод виводить **name** та опис птаха, якого він побачив.

У методі **main** створіть масив типу **Bird**, у який запишіть птахів усіх наявних у вас типів. У циклі відправляйте їх на перегляд дослідникові. В окремому циклі викличте методи **move, eat** для кожного птаха.

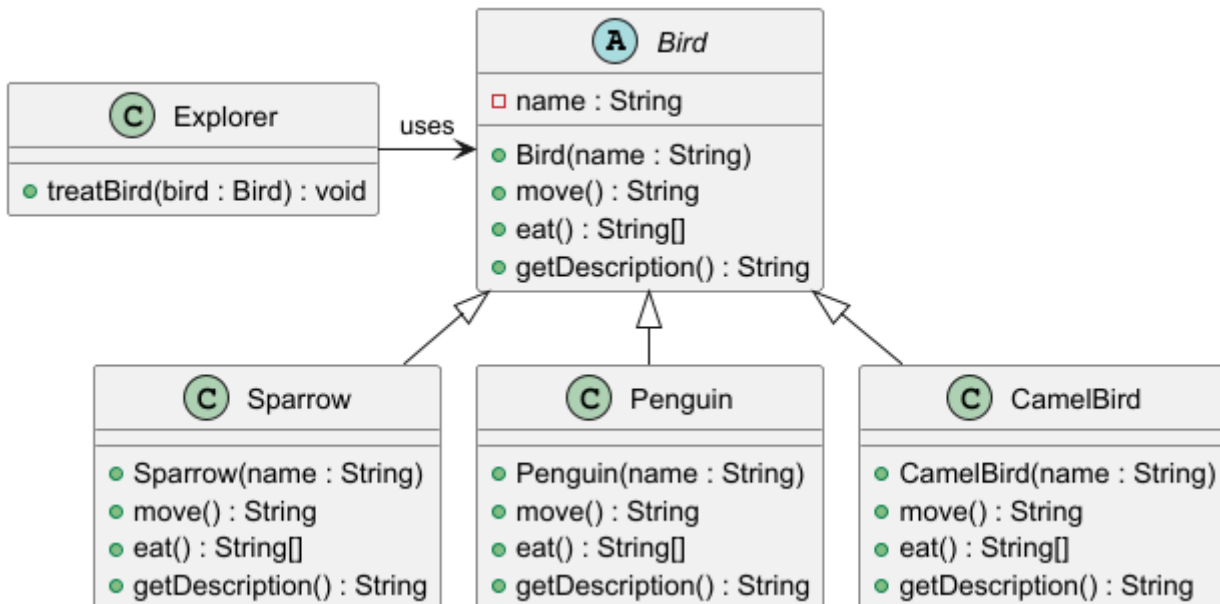
Діаграма класів

Розробимо діаграму класів відповідно до предметної області, у центрі якої стоїть абстрактний клас **Bird**. Він виконує роль концептуальної основи для всіх типів птахів, задаючи спільний контракт через абстрактні методи та забезпечуючи базовий рівень інкапсуляції. Поле **name**, оголошене як **protected**, демонструє контрольований доступ до внутрішнього стану об'єкта, тоді як абстрактні методи **move()**, **eat()** і **getDescription()** формують основу поведінки, яку конкретні підкласи мають реалізувати кожен по-своєму. Таким чином, клас **Bird** уособлює принципи абстракції, інкапсуляції та поліморфізму, створюючи фундамент для подальшого розширення.

На цій основі утворюються конкретні реалізації — **Sparrow, Penguin** та **CamelBird**. Кожен із цих класів успадковує структуру базового класу, але водночас спеціалізує її відповідно до особливостей конкретного виду птахів. Горобець уособлює класичний приклад птаха, що літає, живиться зерном та комахами й має характерний опис як маленький та спритний. Пінгвін, навпаки, демонструє, як спадкування може враховувати виняткові випадки: він не літає, але чудово ходить і плаває, що підкреслює гнучкість поліморфізму. Страус (у моделі представлений як **CamelBird**) показує ще один варіант спеціалізації — це найбільший птах, який не літає, але швидко бігає. Усі ці підкласи реалізують однаковий набір методів, але кожен робить це у власний спосіб, що ілюструє силу динамічного зв'язування та *принцип підстановки Лісков*.

Окреме місце в системі займає клас **Explorer**, який не є частиною ієрархії птахів, але активно з нею взаємодіє. Його метод **treatBird()** приймає будь-який об'єкт типу **Bird**, що демонструє практичне застосування поліморфізму. **Explorer** працює не з конкретними реалізаціями, а з абстракцією, завдяки чому залишається відкритим до розширення: додавання нових видів птахів не потребує змін у його коді. Це втілює *принцип відкритості/закритості* та підкреслює важливість залежності від абстракцій, а не від конкретних класів.

Завершальним елементом є клас **Main**, який виступає точкою входу в застосунок. Саме тут поліморфізм проявляється найяскравіше: масив посилань типу **Bird** може містити об'єкти різних підкласів, а виклики методів виконуються відповідно до фактичного типу об'єкта під час виконання програми. Це демонструє динамічне зв'язування та підтверджує, що всі підкласи повністю відповідають контракту базового класу.



Узагальнюючи, ця модель наочно демонструє ключові принципи ООП: інкапсуляцію через захищені поля та публічні методи доступу, спадкування як механізм повторного використання коду, поліморфізм як основу гнучкої поведінки та абстракцію як спосіб визначення спільного інтерфейсу. Структура легко розширюється, а нові види птахів можуть бути додані без змін у класах, що вже існують. Таким чином, цей приклад є втіленням шаблону **Template method**, де базовий клас задає загальну структуру, а підкласи наповнюють її конкретним змістом.

Абстрактний клас Bird

```

public abstract class Bird {
    protected String name;

    public Bird(String name) {
        this.name = name;
    }

    public abstract String move();
    public abstract String[] eat();
    public abstract String getDescription();

    public String getName() {
        return name;
    }
}
  
```

Клас Sparrow

```
public class Sparrow extends Bird {  
  
    public Sparrow(String name) {  
        super(name);  
    }  
  
    @Override  
    public String move() {  
        return "летить";  
    }  
  
    @Override  
    public String[] eat() {  
        return new String[]{"зерно", "комахи"};  
    }  
  
    @Override  
    public String getDescription() {  
        return "Горобець – маленький птах, що добре літає.";  
    }  
}
```

Клас Penguin

```
public class Penguin extends Bird {  
  
    public Penguin(String name) {  
        super(name);  
    }  
  
    @Override  
    public String move() {  
        return "ходить та плаває";  
    }  
  
    @Override  
    public String[] eat() {  
        return new String[]{"риба", "креветки"};  
    }  
  
    @Override  
    public String getDescription() {  
        return "Пінгвін – нелітний птах, чудово плаває.";  
    }  
}
```

Клас CamelBird

```
public class CamelBird extends Bird {  
  
    public CamelBird(String name) {  
        super(name);  
    }  
  
    @Override  
    public String move() {  
        return "бігає";  
    }  
  
    @Override  
    public String[] eat() {  
        return new String[]{"листя", "трава", "зерно"};  
    }  
  
    @Override  
    public String getDescription() {  
        return "Страус (CamelBird) – найбільший птах, що швидко бігає.";  
    }  
}
```

Клас Explorer

```
public class Explorer {  
  
    public void treatBird(Bird bird) {  
        System.out.println("Дослідник бачить птаха: " + bird.getName());  
        System.out.println("Опис: " + bird.getDescription());  
        System.out.println();  
    }  
}
```

Головний клас

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

        Bird[] birds = {
            new Sparrow("Джек"),
            new Penguin("Пінго"),
            new CamelBird("Степан")
        };

        Explorer explorer = new Explorer();

        System.out.println("=== Перегляд птахів дослідником ===");
        for (Bird bird : birds) {
            explorer.treatBird(bird);
        }

        System.out.println("=== Перевірка поведінки птахів ===");
        for (Bird bird : birds) {
            System.out.println(bird.getName() + " " + bird.move());
            System.out.print("Харчування: ");
            for (String food : bird.eat()) {
                System.out.print(food + " ");
            }
            System.out.println("\n");
        }
    }
}
```

Завдання

Створити консольну програму, що задовольняє таким вимогам:

- Використовує можливості ООП: класи, успадкування, поліморфізм, інкапсуляція.
- Кожен клас повинен мати назву, що відповідає його змісту, та містити поля та методи, указані в завданні.
- У базовому класі та його підкласах визначте додаткові поля та методи, що потрібні згідно з предметною областю.
- Під час кодування мають бути використані домовленості про оформлення коду java code convention.
- Класи мають бути грамотно розкладені за пакетами.
- Консольне меню має бути мінімальним.

Розроблена програма повинна супроводжуватися діаграмою класів.

Варіанти завдань

1. Тварини

Створити абстрактний клас **Animal** і класи **Dog**, **Cat**, **Hamster**, що його розширюють.

Клас **Animal** містить змінну **name** та абстрактні методи **makeNoise**, **eat**, **getDescription**. Метод **makeNoise**, наприклад, повертає опис звуків, що видає тварина. Метод **eat** повертає масив того, чим харчується ця тварина. Метод **getDescription** повертає опис тварини.

Dog, **Cat**, **Hamster** перевизначають методи **makeNoise**, **eat**, **getDescription**.

Створіть клас **Ветеринар**, у якому визначте метод `void treatAnimal(Animal animal)`. Нехай цей метод виводить **name** та опис тварини, яка прийшла на консультацію.

У методі **main** створіть масив типу **Animal**, у який запишіть тварин усіх наявних у вас типів. У циклі відправляйте їх на консультацію до ветеринара. В окремому циклі викличте методи **makeNoise**, **eat** для кожної тварини.

2. Лікарі

Створити абстрактний клас **Doctor** і класи **Therapist**, **Surgeon**, **Ophthalmologist**, що його розширюють.

Клас **Doctor** містить змінну **name** та абстрактні методи **cure**, **consult**, **getDescription**. Метод **cure**, наприклад, повертає способи та засоби лікування. Метод **consult** повертає масив хвороб, які діагностує цей доктор. Метод **getDescription** повертає опис задач доктора.

Therapist, **Surgeon**, **Ophthalmologist** перевизначають методи **cure**, **consult**, **getDescription**.

Створіть клас **Patient**, у якому визначте метод `void visit(Doctor doctor)`. Нехай цей метод виводить **name** та опис доктора, до якого він прийшов на консультацію.

У методі **main** створіть масив типу **Doctor**, у який запишіть лікарів усіх наявних у вас типів. У циклі відправляйте до них на консультацію пацієнта. В окремому циклі викличте методи **consult**, **cure** для кожного лікаря.

3. Музичні інструменти

Створити абстрактний клас **Instrument** і класи **Piano**, **Guitar**, **Drum**, що його розширюють.

Клас **Instrument** містить змінну **name** та абстрактні методи **play**, **type**, **getDescription**. Метод **play**, наприклад, повертає рядок нот, супроводжуючи його назвою інструмента. Метод **type** повертає тип отримання звуку з інструменту (струнний, духовий, ударний). Метод **getDescription** повертає опис інструмента.

Piano, **Guitar**, **Drum** перевизначають методи **play**, **type**, **getDescription**.

Створіть клас **Musician**, у якому визначте метод `void use(Instrument instrument)`. Нехай цей метод виводить **name** та опис інструмента, на якому він грає.

У методі **main** створіть масив типу **Instrument**, у який запишіть інструменти всіх наявних у вас типів. У циклі передавайте їх музиканту для виконання мелодії. В окремому циклі викличте методи **play**, **type** для кожного інструмента.

4. Фігури на площині

Створити абстрактний клас **Figure** і класи **Circle**, **Square**, **Triangle**, що його розширюють.

Клас **Figure** містить змінну **name** та абстрактні методи **perimeter**, **area**, **getDescription**. Метод **perimeter** повинен повертати периметр фігури. Метод **area** повертає площу фігури. Метод **getDescription** повертає опис фігури.

Circle, **Square**, **Triangle** перевизначають методи **perimeter**, **area**, **getDescription**.

Створіть клас **Picture**, у якому визначте метод **void draw(Figure figure)**. Нехай цей метод виводить **name** та опис фігури.

У методі **main** створіть масив типу **Figure**, у який запишіть фігури усіх наявних у вас типів. У циклі передавайте їх у **Picture** для малювання (**draw**). В окремому циклі викличте методи **perimeter**, **area** для кожної фігури.

5. Залізничні вагони

Створити абстрактний клас **RailCar** і класи **PostCar** (поштовий вагон), **PassengerCar** (пасажирський вагон), **AnimalCar** (вагон для тварин).

Клас **RailCar** містить змінну **number** та абстрактні методи **cargo**, **volume**, **getDescription**. Метод **cargo** повертає тип вантажу (поштовий вантаж, люди, тварини). Метод **volume** повертає кількість вантажу. Метод **getDescription** повертає опис вагона та всі його характеристики.

Класи **PostCar**, **PassengerCar**, **AnimalCar** перевизначають методи **cargo**, **volume**, **getDescription**.

Створіть клас **Checker**, у якому визначте метод **check(RailCar railCar)**. Нехай цей метод виводить **number** та опис вагона.

У методі **main** створіть масив типу **RailCar**, у який запишіть вагони всіх наявних у вас типів. У циклі передавайте їх у **Checker** для перевірки (**check**). В окремому циклі викличте методи **cargo**, **volume** для кожного вагона.

6. Прикраси

Створити абстрактний клас **Jewelry** і класи **Necklace** (намисто), **Earring** (сережка), **Ring** (каблучка).

Клас **Jewelry** містить змінну **nameProduct** та абстрактні методи **price**, **size**, **getDescription**.

Метод **price** повертає вартість прикраси, наприклад, залежно від сплаву та наявності коштовного каміння тощо. Метод **size** повертає розмір, (для намиста — це довжина, для каблучки розмір визначається за діаметром, для сережки — це довжина). Метод **getDescription** повертає опис прикраси та всі її характеристики.

Класи **Necklace**, **Earring**, **Ring** перевизначають методи **price**, **size**, **getDescription**.

Створіть клас **Customer**, у якому визначте метод **jewelryOrder(Jewelry jewelry)**. Нехай цей метод виводить **nameProduct** та опис прикраси.

У методі **main** створіть масив типу **Jewelry**, у який запишіть прикраси всіх наявних у вас типів. У циклі передавайте їх у **Customer** для замовлення (**jewelryOrder**). В окремому циклі викличте методи **price**, **size** для кожної прикраси.

7. Таксопарк

Створити абстрактний клас **Vehicle** і класи **Car**, **Truck**, **Minivan**.

Клас **Vehicle** містить змінну **number** та абстрактні методи **passengers**, **speed**, **getDescription**.

Метод **passengers** повертає максимальну кількість пасажирів (для **Car** — 4, для **Truck** — 0, для **Minivan** — число вказане в конструкторі). Метод **speed** повертає максимальну швидкість автівки. Метод **getDescription** повертає опис автівки та всі її характеристики.

Класи **Car**, **Truck**, **Minivan** перевизначають методи **cargo**, **volume**, **getDescription**.

Створіть клас **Customer**, у якому визначте метод **placeOrder(Vehicle vehicle)**. Нехай цей метод виводить **number** та опис автівки.

У методі **main** створіть масив типу **Vehicle**, у який запишіть автівки всіх наявних у вас типів. У циклі передавайте їх у **Customer** для замовлення (**placeOrder**). В окремому циклі викличте методи **passengers**, **speed** для кожної автівки.

8. Меблі

Створити абстрактний клас **Furniture** і класи **Wardrobe**, **Sofa**, **Bookcase**, що його розширюють.

Клас **Furniture** містить змінну **name** та абстрактні методи **quantity**, **sizes**, **getDescription**.

Метод **quantity** повертає деяку числову характеристику: (для **Wardrobe** — кількість дверей, для **Sofa** — кількість спальних місць, для **Bookcase** — кількість полиць). Метод **sizes** повертає габарити. Метод **getDescription** повертає опис предмета та його характеристики.

Класи **Wardrobe**, **Sofa**, **Bookcase** перевизначають методи **quantity**, **sizes**, **getDescription**.

Створіть клас **Designer**, у якому визначте метод **design(Furniture furniture)**. Нехай цей метод виводить **name** та опис предмета.

У методі **main** створіть масив типу **Furniture**, у який запишіть меблі всіх наявних у вас типів. У циклі передавайте їх у **Designer** для проектування (**design**). В окремому циклі викличте методи **quantity**, **sizes** для кожного предмета.

9. Дитячі іграшки

Створити абстрактний клас **Toy** і класи **Doll**, **Ball**, **Constructor**, що його розширюють.

Клас **Toy** містить змінну **name** та абстрактні методи **size**, **age**, **price**, **getDescription**.

Метод **size** повертає деяку числову характеристику: (для **Doll** — зріст, для **Ball** — діаметр, для **Constructor** — кількість деталей). Метод **age** повертає діапазон віку дитини. Метод **price** повертає вартість іграшки, що залежить від її інших характеристик. Метод **getDescription** повертає опис іграшки та її характеристики.

Класи **Doll**, **Ball**, **Constructor** перевизначають методи **size**, **age**, **price**, **getDescription**.

Створіть клас **Child**, у якому визначте метод **play(Toy toy)**. Нехай цей метод виводить **name** та опис іграшки.

У методі **main** створіть масив типу **Toy**, у який запишіть іграшки усіх наявних у вас типів. У циклі передавайте їх у **Child** для гри (**play**). В окремому циклі викличте методи **size**, **age**, **price** для кожної іграшки.

10. Парк розваг

Створити абстрактний клас **Amusement** і класи **FerrisWheel**, **RollerCoaster**, **PanicRoom**, що його розширюють.

Клас **Amusement** містить змінну **name** та абстрактні методи **age**, **price**, **getDescription**.

Метод **age** повертає мінімальний вік людини, що допускається до атракціону. Метод **price** повертає вартість розваги, що залежить від її інших характеристик. Метод **getDescription** повертає опис атракціону та його характеристики.

Класи **FerrisWheel**, **RollerCoaster**, **PanicRoom** перевизначають методи **age**, **price**, **getDescription**.

Створіть клас **Visitor**, у якому визначте метод **haveFun**(Amusement amusement). Нехай цей метод виводить **name** та опис атракціону.

У методі **main** створіть масив типу **Amusement**, у який запишіть атракціони всіх наявних у вас типів. У циклі передавайте їх у **Visitor** для розваги (**haveFun**). В окремому циклі викличте методи **age**, **price** для кожного атракціону.

11. Побутова техніка

Створити абстрактний клас **Appliance** і класи **Fridge**, **Microwave**, **WashingMachine**, що його розширюють.

Клас **Appliance** містить змінну **name** та абстрактні методи **powerUsage**, **function**, **getDescription**.

Метод **powerUsage** повертає споживання електроенергії на рік, наприклад, у кВт·год. Метод **function** повертає основні функції приладу — масив рядків. Метод **getDescription** повертає опис приладу та його характеристики.

Класи **Fridge**, **Microwave**, **WashingMachine** перевизначають методи **powerUsage**, **function**, **getDescription**.

Створіть клас **Technician**, у якому визначте метод **check**(Appliance appliance). Нехай цей метод виводить **name** та опис приладу, який технік перевіряє.

У методі **main** створіть масив типу **Appliance**, у який запишіть прилади всіх наявних у вас типів. У циклі передавайте їх **Technician** для перевірки (**check**). В окремому циклі викличте методи **powerUsage**, **function** для кожного приладу.

12. Населені пункти

Створити абстрактний клас **Settlement** і класи **City**, **Town**, **Village**, що його розширюють.

Клас **Settlement** містить змінну **name** та абстрактні методи **population**, **infrastructure**, **getDescription**. Метод **population** повертає кількість населення. Метод **infrastructure** повертає масив основних об'єктів інфраструктури, наприклад: лікарня, школа, ринок, ТРЦ, автобусна станція тощо. Метод **getDescription** повертає опис населеного пункту та його

характеристики.

Класи **City**, **Town**, **Village** перевизначають методи **population**, **infrastructure**, **getDescription**.

Створіть клас **Geographer**, у якому визначте метод **explore**(Settlement settlement). Нехай цей метод виводить **name** та опис населеного пункту, який географ досліджує.

У методі **main** створіть масив типу **Settlement**, у який запишіть населені пункти всіх наявних у вас типів. У циклі передавайте їх **Geographer** для дослідження (**explore**). В окремому циклі викличте методи **population**, **infrastructure** для кожного населеного пункту.

13. Працівники

Створити абстрактний клас **Worker** і класи **Laborer**, **Engineer**, **Manager**, що його розширюють.

Клас **Worker** містить змінну **name** та абстрактні методи **task**, **salary**, **getDescription**. Метод **task** повертає опис основних обов'язків працівника. Метод **salary** повертає розмір заробітної плати, наприклад, у вигляді числа або формули. Метод **getDescription** повертає опис працівника та його професійних характеристик.

Класи **Laborer**, **Engineer**, **Manager** перевизначають методи **task**, **salary**, **getDescription**.

Створіть клас **HRDepartment**, у якому визначте метод **review**(Worker worker). Нехай цей метод виводить **name** та опис працівника, якого перевіряє відділ кадрів.

У методі **main** створіть масив типу **Worker**, у який запишіть працівників усіх наявних у вас типів. У циклі передавайте їх у **HRDepartment** для перевірки (**review**). В окремому циклі викличте методи **task**, **salary** для кожного працівника.

14. Слюсарні інструменти

Створити абстрактний клас **Tool** і класи **Screwdriver**, **Hammer**, **Wrench**, що його розширюють.

Клас **Tool** містить змінну **name** та абстрактні методи **use**, **size**, **getDescription**. Метод **use** повертає опис основного призначення інструмента, наприклад, закручування гвинтів, забивання цвяхів, затягування гайок. Метод **size** повертає розмір інструмента: довжина викрутки, вага голови молотка, розмір зіву гайкового ключа тощо. Метод **getDescription** повертає опис інструмента та його характеристики.

Класи **Screwdriver**, **Hammer**, **Wrench** перевизначають методи **use**, **size**, **getDescription**.

Створіть клас **Mechanic**, у якому визначте метод **inspect**(Tool tool). Нехай цей метод виводить **name** та опис інструмента, який механік оглядає.

У методі **main** створіть масив типу **Tool**, у який запишіть слюсарні інструменти всіх наявних у вас типів. У циклі передавайте їх слюсарю для огляду (**inspect**). В окремому циклі викличте методи **use**, **size** для кожного інструмента.

15. Периферійні пристрої

Створити абстрактний клас **Peripheral** і класи **Mouse**, **Keyboard**, **BarcodeScanner**, що його розширюють.

Клас **Peripheral** містить змінну **name** та абстрактні методи **connect**, **dataRate**, **getDescription**. Метод **connect** повертає спосіб підключення пристрою (USB, Bluetooth, радіоканал тощо). Метод **dataRate** повертає швидкість передачі даних, наприклад: 125–1000 Hz для миші, 1–5 мс відгуку для клавіатури, кількість сканів за секунду для сканера. Метод **getDescription** повертає опис пристрою та його характеристики.

Класи **Mouse**, **Keyboard**, **BarcodeScanner** перевизначають методи **connect**, **dataRate**, **getDescription**.

Створіть клас **Operator**, у якому визначте метод **test(Peripheral peripheral)**. Нехай цей метод виводить **name** та опис пристрою, який оператор тестує.

У методі **main** створіть масив типу **Peripheral**, у який запишіть пристрої всіх наявних у вас типів. У циклі передавайте їх оператору для тестування (**test**). В окремому циклі викличте методи **connect**, **dataRate** для кожного пристрою.