Лабораторная работа №3

СПАДКУВАННЯ ТА ПОЛІМОРФІЗМ

Мета роботи. Отримати практичні навички створення ієрархій успадкування класів. Навчитися проектувати класи, використовуючи механізм успадкування. Мати уявлення про поліморфізм, переваги його використання при проектуванні класів.

1. Загальні відомості про реалізацію механізму наслідування в мові Java

Спадкування - один з трьох найважливіших механізмів об'єктно орієнтованого програмування (поряд з инкапсуляцией і поліморфізмом), що дозволяє описати новий клас на основі вже існуючого (батьківського), при цьому властивості і функціональність батьківського класу запозичуються новим класом. Іншими словами, клас-спадкоємець реалізує специфікацію вже існуючого класу (базовий клас). Це дозволяє звертатися з об'єктами класу-спадкоємця точно так же, як з об'єктами базового класу. Клас, від якого відбулося спадкування, називається базовим або батьківським (англ. *Base class*). Класи, які походять від базового, називаються нащадками, спадкоємцями або похідними класами (*англ. Derived class*).

Деякі мови використовують абстрактні класи. *Абстрактний клас* - це клас, що містить хоча б один абстрактний метод, він описаний в програмі, має поля, методи і не може використовуватися для безпосереднього створення об'єкта. Тобто від абстрактного класу можна тільки наслідувати. Об'єкти створюються тільки на основі похідних класів, успадкованих від абстрактного класу. Наприклад, абстрактним класом може бути базовий клас «співробітник вузу», від якого успадковуються класи «аспірант», «професор» і т.д. Так як похідні класи мають спільні поля і функції (наприклад, поле «рік народження»), то ці члени класу можуть бути описані в базовому класі. У програмі створюються об'єкти на основі класів «аспірант», «професор», але немає сенсу створювати об'єкт на основі класу «співробітник вузу».

У мові Java підтримується тільки просте спадкування: будь-який підклас є похідним тільки від одного безпосереднього суперкласу. При цьому будь-який клас може успадковуватися від декількох інтерфейсів.

Спадкування інтерфейсів реалізує деяку заміну множинного спадкоємства, коли замість того щоб один клас мав кілька безпосередніх суперкласів, цей клас успадковує кілька інтерфейсів. Інтерфейс являє собою клас, в якому все властивості - константи (тобто статичні - *static* і незмінні - *final*), а всі методи абстрактні, тобто інтерфейс дозволяє визначити певний шаблон класу: опис властивостей і методів без їх реалізації.

Мова *Java* дозволяє кілька рівнів спадкування, що визначаються безпосереднім суперкласом і непрямими суперкласу. Спадкування можна використовувати для створення ієрархії класів.

При створенні підкласу на основі одного або декількох суперкласів можливі наступні способи зміни поведінки і структури класу:

- розширення суперкласу шляхом додавання нових даних і методів;
- заміна методів суперкласу шляхом їх перевизначення;
- злиття методів з суперкласів викликом однойменних методів з відповідних суперкласів.

Приклад успадкування Лістинг 1

```
public class Person {// неявне успадкування від класу Object
    public static final String GENDER_MALE = "MALE";
```

```
    public static final String GENDER_FEMALE = "FEMALE";

    public Person() {
    // Робити більше нічого......
    }

    private String name;
    private int age;
    private int height;
    private int weight;
    private String eyeColor;
    private String gender;

}
```

Всі класи в *Java* неявно успадковують властивості і поведінки класу *Object*, тобто він є предком всіх класів, які є в *JDK* (*Java Development Kit* - комплект розробки на мові *Java*) і які будуть створюватися користувачем. Наприклад: клас *Person* в лістингу 1 неявно успадковує властивості *Object*. Так як це передбачається для кожного класу, не потрібно вводити фразу *extends Object* для кожного визначається класу. Таким чином, клас *Person* має доступ до представлених змінним і методам свого суперкласу (*Object*). В даному випадку *Person* може «бачити» і використовувати загальнодоступні методи і змінні об'єкта *Object*, а також його захищені методи і змінні.

2. Визначення ієрархії класів.

Визначимо новий клас *Employee*, який успадковує властивості *Person*. Його визначення класу (або граф успадкування) буде виглядати наступним чином:

```
public class Employee extends Person {
private String taxpayerIdentificationNumber;
private String employeeNumber;
private int salary;
// . . .
}
```

Граф успадкування для *Employee* вказує на те, що *Employee* має доступ до всіх загальнодоступних і захищених змінних і методів *Person* (так як він безпосередньо розширює його), а також *Object* (так як він фактично розширює і цей клас, хоча і опосередковано).

Для поглиблення в ієрархію класів ще на один крок, можна створити третій клас, який розширює *Employee*:

```
public class Manager extends Employee {
// . . .
}
```

У мові *Java* будь-який клас може мати не більше одного суперкласу, але будь-яку кількість підкласів. Це найважливіша особливість ієрархії успадкування мови *Java*, про яку треба пам'ятати.

У *Java* всі методи конструктора використовують уточнення при перевизначенні методів по схемі зчеплення конструкторів. Зокрема, виконання конструктора починається зі звернення до конструктору суперкласу, яке може бути явним або неявним. Для явного звернення до конструктору використовується оператор *super*, який вказує на виклик суперкласу (наприклад, *super*() викликає конструктор суперкласу без аргументів). Якщо ж в тілі конструктора явне звернення відсутне, компілятор автоматично поміщає в першому рядку конструктора звернення до методу *super*(). Тобто в мові *Java* в конструкторах використовується уточнення при **перевизначенні** методів, а в звичайних методах використовується **заміщення** (лістинг 2).

Лістинг 2

```
public class Person {
 private String name;
 public Person() {}
    public Person(String name) {
             this.name = name;
 }
}
public class Employee extends Person {
 public Employee(String name) {
 super(name);
 }
}
```

3. Ключове слово this

Іноді в класі *Java* необхідно звернутися до поточного екземпляру даного класу, який в даний момент обробляється методом. Для такого звернення використовується ключове слово ***this***. Застосування цієї конструкції зручно в разі необхідності звернення до полю поточного об'єкта, ім'я якого збігається з ім'ям змінної, описаної в даному блоці (Лістинг 2).

4. Перевизначення методів.

Якщо наприклад: в базовому і дочірньому класах є методи з однаковими іменами і однаковими параметрами. В такому випадку доречно говорити про *перевизначення* методів, тобто в дочірньому класі змінюється реалізація вже існуючого в базовому класі методу. Якщо позначити метод модифікатором *final*, то метод не може бути перевизначений. Поля не можна перевизначити, їх можна тільки приховати. Розглянемо перевизначення методів Лістинг 3:

```
public class Person {
 private String FIO;
 private int age;
 public Person() {}
 public String getFIO(){
 return this.FIO;
}
 public final setAge(int age){
 this.age = age;
 }
}
public class Employee extends Person {
 public Employee(String name) {
 super();
}
public String getFIO(){
 return "My name is " + super.getFio();
 }
}
```

5. Спадкування і абстракція

Якщо підклас *перевизначає* метод з суперкласу, цей метод, по суті, прихований, тому що виклик цього методу за допомогою посилання на підклас викликає версії методу підкласу, а не версію суперкласу. Це не означає, що метод суперкласу стає недоступним. Підклас може викликати метод суперкласу, додавши перед ім'ям методу ключове слово *super* (і на відміну від правил для конструкторів, це можна зробити в будь-якому рядку методу підкласу або навіть зовсім іншого методу). За замовчуванням *Java*-програма викликає метод підкласу, якщо він викликається за допомогою посилання на підклас (Лістинг 5.3).

В контексті ООП абстрагування означає узагальнення даних і поведінки до типу, більш високого за ієрархією спадкування, ніж поточний клас. При переміщенні змінних або методів з підкласу в суперклас кажуть, що ці члени абстрагуються. Основною причиною цього є можливість багаторазового використання загального коду шляхом просування його як можна вище за ієрархією. Коли загальний код зібраний в одному місці, полегшується його обслуговування.

6. Абстрактні класи та методи

Бувають моменти, коли потрібно створювати класи, які служать тільки як абстракції, і створювати їх екземпляри ніколи не доведеться. Такі класи називаються абстрактними класами. До того ж бувають моменти, коли певні методи повинні бути реалізовані по-різному для кожного підкласу, що реалізується суперкласом. Це абстрактні методи. Ось деякі основні правила для абстрактних класів і методів:

- будь-який клас може бути оголошений абстрактним;
- абстрактні класи не допускають створення своїх примірників;
- абстрактний метод не може містити тіла методу;
- клас, що містить абстрактний метод, повинен оголошуватися як абстрактний.

7. Використання абстрагування

Наприклад: необхідно безпосередньо заборонити можливість створення екземпляра класу *Employee*. Для цього достатньо оголосити його за допомогою ключового слова *abstract*:

```
public abstract class Employee extends Person {
}
```

Після цього при виконанні цього коду, ви отримаєте помилку компіляції:

```
public void someMethodSomwhere() {
 Employee p = new Employee();// помилка компіляції !!
}
```

Компілятор вкаже на те, що *Employee* - це абстрактний клас, і його примірник не може бути створений.

8. Можливості абстрагування

Наприклад: необхідний метод для вивчення стану об'єкта *Employee* і перевірки його правомірності. Ця вимога може здатися загальним для всіх об'єктів *Employee*, але між усіма потенційними підкласами поведінку буде істотно відрізнятися, що дає нульовий потенціал для повторного використання. В цьому випадку ви оголошуєте метод *validate*() як абстрактний (змушуючи всі підкласи реалізовувати його):

```
public abstract class Employee extends Person {
public abstract boolean validate();
}
```

Тепер кожен прямий підклас *Employee* (такий як *Manager*) повинен реалізувати метод *validate*(). При цьому, як тільки підклас реалізував метод *validate*(), жодному з його підкласів не доведеться цього робити.

Лістинг 4

```
public abstract class Employee extends Person {
public abstract boolean validate();
}
public class Manager extends Employee {
public boolean validate() {//метод для цього класу обов'язковий.
        boolean flag = true;
```

```
// код що розраховує стан об'єкту
 return flag;
 }
}
public class Executive extends Manager {
//метод public boolean validate() можно пропустити
}
```

9. Поліморфізм

Поліморфізм в мовах програмування - це можливість об'єктів з однаковою специфікацією мати різну реалізацію. Мова програмування підтримує поліморфізм, якщо класи з однаковою специфікацією можуть мати різну реалізацію - наприклад, реалізація класу може бути змінена в процесі успадкування. Коротко зміст поліморфізму можна виразити фразою: «*Один інтерфейс, безліч реалізацій*».

У *java* існують кілька способів організації поліморфізму:

1) Поліморфізм інтерфейсів. Інтерфейси описують методи, які повинні бути реалізовані в класі, і типи параметрів, які повинен отримувати і повертати кожен член класу, але не містять певної реалізації методів, залишаючи це на реалізуючий інтерфейс класу (Лістинг 5). В цьому і полягає поліморфізм інтерфейсів. Кілька класів можуть реалізовувати один і той самий інтерфейс, в той же час один клас може реалізовувати один або більше інтерфейсів. Інтерфейси знаходяться поза ієрархії успадкування класів, тому вони виключають визначення методу або набору методів з ієрархії успадкування.

Лістингу 5

```
interface Shape {
void draw();
void erase();
}
class Circle implements Shape {
public void draw() { System.out.println("Circle.draw()");}
public void erase() { System.out.println("Circle.erase()");}
}
class Square implements Shape {
    public void draw() { System.out.println("Square.draw()");}
    public void erase() { System.out.println("Square.erase()");}
}
class Triangle implements Shape {
    public void draw() { System.out.println("Triangle.draw()");}
    public void erase() { System.out.println("Triangle.erase()");}
}
public class Shapes {
    public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
        default:
        case 0: return new Circle();
        case 1: return new Square();
        case 2: return new Triangle();
        }
    }
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        for(int i = 0; i < s.length; i++)
            s[i] = randShape();
        for(int i = 0; i < s.length; i++)
            s[i].draw();
    }
}
```

2) Поліморфізм успадкування. При спадкуванні клас отримує всі методи, властивості і події базового класу такими, якими вони реалізовані в базовому класі. При необхідності в успадкованих класах можна визначати додаткові члени або перевизначати члени, що дісталися від базового класу, щоб реалізувати їх інакше (Лістинг 6).

Наслідуваний клас також може реалізовувати інтерфейси. В даному випадку поліморфізм проявляється в тому, що функціонал базового класу присутній в успадкованих класах неявно. Функціонал може бути доповнений і перевизначений. А успадковані класи, що несуть в собі цей функціонал виступають в ролі багатьох форм.

Лістингу 6

```
class Shape {
void draw() {}
void erase() {}
}
class Circle extends Shape {
    void draw() { System.out.println("Circle.draw()");}
    void erase() { System.out.println("Circle.erase()");}
}
class Square extends Shape {
    void draw() { System.out.println("Square.draw()");}
    void erase() { System.out.println("Square.erase()");}
}
class Triangle extends Shape {
    void draw() { System.out.println("Triangle.draw()");}
    void erase() { System.out.println("Triangle.erase()");}
}
public class Shapes {
    public static Shape randShape() {
        switch((int)(Math.random() * 3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
            }
    }
public static void main(String[] args) {
    Shape[] s = new Shape[9];
    for(int i = 0; i < s.length; i++)
        s[i] = randShape();
    for(int i = 0; i < s.length; i++)
        s[i].draw();
    }
}
}
```

3) Поліморфізм за допомогою абстрактних класів. Абстрактні класи підтримують як успадкування, так і можливості інтерфейсів (лістинг 5.4). При побудові складної ієрархії, для забезпечення поліморфізму програмісти часто змушені вводити методи в класи верхнього рівня при тому, що ці методи ще не визначені. *Абстрактний клас* - це клас, екземпляр якого неможливо створити; цей клас може лише служити базовим класом при спадкуванні. Не можна оголошувати абстрактні конструктори або абстрактні статичні методи. Деякі або всі члени цього класу можуть залишатися нереалізованими, їх реалізацію повинен забезпечити успадковує клас. Похідні класи, які не переважають все абстрактні методи, повинні бути відзначені як абстрактні. Породжений клас може реалізовувати також додаткові інтерфейси.

4) Поліморфізм методів. Здатність класів підтримувати різні реалізації методів з однаковими іменами - один із способів реалізації поліморфізму. Різні реалізації методів з однаковими іменами в *Java* називається *перевантаженням* методів (лістинг 7). На практиці часто доводиться реалізовувати один і той же метод для різних типів даних. Право вибору специфічної версії методу надано компілятору.

Окремим варіантом поліморфізму методів є поліморфізм методів зі змінним числом аргументів, введений у версії Java 2 5.0. Перевантаження методів тут передбачена неявно, тобто перевантажений метод може викликатися з різним числом аргументів, а в деяких випадках навіть без параметрів. Перевантаження методів як правило робиться для

тісно пов'язаних за змістом операцій. Відповідальність за побудову перевантажених методів і виконання ними однорідних за змістом операцій лежить на розробнику.

Лістинг 7

```
public class PrintStream extends FilterOutputStream
 implements Appendable , Closeable {
/*…*/
public void print(boolean b) {
    write(b ? "true" : "false");
}
public void print(char c) {
    write(String.valueOf(c));
}
public void print(int i) {
    write(String.valueOf(i));
}
public void print(long l) {
    write(String.valueOf(l));
}
/*…*/
public void write(int b) {
}
}
```

5) Поліморфізм через перевизначення методів. Якщо перевантажені методи з однаковими іменами знаходяться в одному класі, списки параметрів повинні відрізнятися. Але якщо метод підкласу збігається з методом суперкласу, то метод підкласу *перевизначає* метод суперкласу (лістинг 3). Збігатися при цьому повинні і імена методів і типи вхідних параметрів. В даному випадку перевизначення методів є основою концепції динамічного зв'язування (або пізне зв'язування), що реалізує поліморфізм. Суть динамічної диспетчеризації методів полягає в тому, що рішення на виклик перевизначеного методу приймається під час виконання, а не під час компіляції. Однак *final*-методи не є перевизначеним, їх виклик може бути організований під час компіляції і називається раннім зв'язуванням. Приклад поліморфізму представлений в лістингу 5.

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

За варіантами (створити класи, в них передбачити різні члени класів і методи для роботи):

1. Базовий клас - учень. Похідні - школяр і студент. Створити клас Конференція, який може містити обидва види учнів. Передбачити метод підрахунку учасників конференції окремо по школярах і по студентам (використовувати оператор *instanceof*).

2. Базовий клас - працівник. Похідні - працівник на погодинну оплату і на окладі. Створити клас Підприємство, який може містити обидва види працівників. Передбачити метод підрахунку працівників окремо на погодинну оплату і на окладі (використовувати оператор *instanceof*).

3. Базовий клас - комп'ютер. Похідні - ноутбук і смартфон. Створити клас Ремонт Сервіс, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо ремонтованих ноутбуків і смартфонів (використовувати оператор *instanceof*).

4. Базовий клас - друковані видання. Похідні - книги і журнали. Створити клас КнижковийМагазин, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо книг і журналів (використовувати оператор *instanceof*).

5. Базовий клас - приміщення. Похідні - квартира і офіс. Створити клас Будинок, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо квартир і офісів (використовувати оператор *instanceof*).

6. Базовий клас - файл. Похідні - звуковий файл і відео-файл. Створити клас Каталог, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо звукових і відео-файлів (використовувати оператор *instanceof*).

7. Базовий клас - літальний апарат. Похідні - літак і вертоліт. Створити клас Авіакомпанія, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо літаків і вертольотів (використовувати оператор *instanceof*).

8. Базовий клас - змагання. Похідні - командні змагання та особисті. Створити клас Чемпіонат, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо командних змагань і особистих (використовувати оператор *instanceof*).

9. Базовий клас - меблі. Похідні - диван і шафа. Створити клас Кімната, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо диванів і шаф (використовувати оператор *instanceof*).

10. Базовий клас - зброя. Похідні - вогнепальну та холодну. Створити клас ОружейнаПалата, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо вогнепальної і холодної зброї (використовувати оператор *instanceof*).

11. Базовий клас - оргтехніка. Похідні - принтер і сканер. Створити клас Офіс, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо принтерів і сканерів (використовувати оператор *instanceof*).

12. Базовий клас - ЗМІ. Похідні - телеканал і газета. Створити клас Холдинг, який може містити обидва види об'єктів. Передбачити метод підрахунку окремо телеканалів і газет (використовувати оператор *instanceof*).