Лабораторна робота №4

Тема: Спадкування класів в Java. Ієрархія класів.

Мета: Вивчити особливості роботи з класами та їх спадкуванням у Java.

Завдання: Здобути навички створення обробки даних з використанням спадкування класів мовою ***Java***.

(max: 10 балів)

Теоретичні відомості:

Одним з ключових аспектів об'єктно-орієнтованого програмування є **успадкування**. За допомогою успадкування можна розширити функціонал вже наявних класів за рахунок додавання нового функціоналу або перевизначення старого. Наприклад, є клас *Person*, що описує людину.

Припустимо потрібно створити клас, який описує співробітника підприємства – клас *Employee*. Цей клас реалізує той же функціонал, що і клас *Person*, оскільки співробітник – це також і людина. Таким чином, буде раціонально зробити клас *Employee* похідним (спадкоємцем, підкласом) від класу *Person*, який, в свою чергу, називається базовим класом, батьком або суперкласом.

Щоб оголосити один клас спадкоємцем іншого, потрібно записати після імені класу-спадкоємця ключове слово *extends*, після чого ім'я базового класу. Для класу *Employee* базовим є *Person*, і тому клас *Employee* успадковує всі поля і методи, які є в класі *Person*.

Похідний клас має доступ до всіх методів і полів базового класу (навіть якщо базовий клас знаходиться в іншому пакеті) крім тих, які визначені з модифікатором *private*. При цьому у похідний клас можуть бути додані нові поля і методи:

```
public class Program {
    public static void main(String [] args) {
        Employee george = new Employee("George", "Microsoft");
        george.print(); // Ім'я: George
        george.work(); // George працює на Microsoft
    }
}
class Person {
    private String name;
    public String getName() {return name; }
    public Person(String name) { this.name = name; }
    public void print() { System.out.println("Ім'я: " + name); }
}
class Employee extends Person {
    private String company;
    public Employee(String name, String company) {
        super(name); this.company = company;
    }
    public void work() {
        System.out.printf("%s працює на %s \n", getName(), company);
    }
}
```

В даному випадку клас *Employee* доповнюється новим полем *company*, яке зберігає місце роботи співробітника, а також методом *work*. Якщо в базовому класі визначено конструктори, то в конструкторі похідного класу необхідно викликати один з конструкторів базового класу за допомогою **ключового слова *super***. Зокрема, клас *Person* має конструктор, який приймає один параметр, тому в класі *Employee* в конструкторі потрібно викликати конструктор класу *Person*. Після ключового слова *super* в дужках зазначається перерахування переданих аргументів. Таким чином, задання імені співробітника делегується конструктору базового класу. При цьому виклик конструктора базового класу повинен розміщуватися на початку конструктора похідного класу.

Похідний клас може визначати свої методи, а може перевизначати методи, які успадковані від базового класу. Наприклад, можна перевизначити в класі *Employee*, метод *print*:

```
@Override
public void print() {
   System.out.printf("Ім'я: %s \n", getName());
   System.out.printf("Працює на %s \n", company);
}
```

Перед методом, який перевизначається, вказується **анотація *@Override***, яка є необов'язковою. Перевизначений метод повинен мати рівень доступу не менший, ніж рівень доступу в базовому класі. Наприклад, якщо в базовому класі метод має модифікатор *public*, то і в похідному класі метод повинен мати модифікатор *public*.

Однак в даному випадку видно, що частина методу *print* в *Employee* повторює дії з методу *print* базового класу. Оскільки немає сенсу писати повторно вже написаний код, варто викликати метод *display* з батьківського класу *Person*:

```
@Override
public void print() {
   super.print();
   System.out.printf("Працює на %s \n", company);
}
```

За допомогою ключового слова *super* можна звернутися до реалізації методів базового класу. Хоча успадкування дуже цікавий і ефективний механізм, але в деяких ситуаціях його застосування може бути небажаним. І в цьому випадку можна заборонити успадкування за допомогою **ключового слова *final***, наприклад:

```
public final class Person { }
```

Якщо визначити клас *Person* таким чином, то наступний код буде помилковим і не спрацює, тому що успадкування заборонене:

```
class Employee extends Person { } // error
```

Крім заборони успадкування можна також заборонити перевизначення окремих методів. Наприклад, в прикладі вище перевизначений метод *print()*, заборонимо його перевизначення в базовому класі *Person*:

```
public class Person {
   public final void print() { System.out.println("Ім'я:" + name); }
}
```

В цьому випадку, клас *Employee* не зможе перевизначити метод *print*.

Успадкування і можливість перевизначення методів відкривають великі можливості. Перш за все можна передати **змінній суперкласу посилання на об'єкт підкласу**:

```
Person alex = new Employee("Alex", "Oracle");
```

Оскільки *Employee* успадковується від *Person*, то об'єкт *Employee* є в той же час і об'єктом *Person*. Грубо кажучи, будь-який працівник підприємства, одночасно, є людиною.

Однак незважаючи на те, що змінна представляє об'єкт *Person*, віртуальна машина бачить, що в реальності змінна *alex* вказує на об'єкт *Employee*. Тому при виклику методів для цього об'єкта буде викликатися та версія методу, яка визначена в класі *Employee*, а не в *Person*.

При виклику перевизначеного методу віртуальна машина динамічно знаходить і викликає саме ту версію методу, яка визначена в підкласі. Даний процес ще називається **dynamic method lookup/динамічний пошук методу/динамічна диспетчеризація методів**.

Крім звичайних класів в Java є **абстрактні класи**. В абстрактному класі також можна визначити поля і методи, але в той же час не можна створити об'єкт або екземпляр абстрактного класу. Абстрактні класи існують для надання базового функціоналу класам-спадкоємцям, а похідні класи вже реалізують цей функціонал.

При визначенні абстрактних класів використовується **ключове слово *abstract***:

```
public abstract class Human {
   private String name;
   public String getName() { return name; }
}
```

Головна відмінність абстрактних класів, від звичайних, полягає в тому, що не можна створити екземпляр абстрактного класу. Наприклад, наступним чином:

```
Human h = new Human(); // помилка
```

Причина цього в тому що, крім звичайних методів, абстрактний клас може містити не визначену складову – абстрактні методи. Такі методи визначаються за допомогою ключового слова *abstract* і не мають реалізації:

```
public abstract void print();
```

Похідний клас, якщо він не є абстрактним, зобов'язаний перевизначити і реалізувати всі абстрактні методи, які є в базовому абстрактному класі. Слід врахувати, що якщо клас має хоча б один абстрактний метод, то даний клас повинен бути визначений як абстрактний.

Проаналізуємо потребу в абстрактних класах. Припустимо, що є програма для обслуговування банківських операцій, в якій є три класи: *Person*, який описує людину, *Employee*, що описує банківського службовця, і клас *Client*, який представляє клієнта банку. Очевидно, що класи *Employee* і *Client* будуть похідними від класу *Person*, тому що обидва класи мають спільні поля та методи. І оскільки всі об'єкти будуть представляти або співробітника, або клієнта банку, то безпосередньо об'єкти класу *Person* створювати не потрібно. Тому є сенс зробити його абстрактним.

Іншим хрестоматійним прикладом є система геометричних фігур. Кожна геометрична фігура: коло, прямокутник, квадрат буде реалізована у вигляді класу, потрібно написати також методи для обчислення площі, периметру кожної фігури. Тому можна виділити в якості базового класу абстрактний клас, і визначити в ньому інтерфейс методів, які притаманні кожній фігурі, але без реалізації, тому що для кожного класу-нащадка буде своя реалізація. Самі ж фігури успадкувати від абстрактного класу:

```
abstract class Figure {   // абстрактний клас “фігура”
   float x; // x-координата точки
   float y; // y-координата точки
   Figure(float x, float y) { this.x = x; this.y = y; }
   // абстрактний метод для отримання периметра
   public abstract float getPerimeter();
   // абстрактний метод для отримання площі
   public abstract float getArea();
}
class Rectangle extends Figure // похідний клас “прямокутник”
{
   private float width; private float height;
    //конструктор зі зверненням до конструктора класу Figure
    Rectangle(float x, float y, float width, float height) {
       super(x, y);
       this.width = width;
       this.height = height;
    }
    public float getPerimeter() { return width * 2 + height * 2; }
    public float getArea() { return width * height; }
}
```

Опишемо приведення типів на прикладі наступної ієрархії успадкування класів:

```
public class Program {
   public static void main(String [] args) { ... }
}
class Person { ... } // клас “людина”
class Employee extends Person { ... } // службовець деякої компанії
class Client extends Person { } // клас “клієнт банку”
```

Ця ієрархія класів представляє собою наступний ланцюг успадкування: *Object* (всі класи неявно успадковуються від типу Object) → *Person* → *Employee* | *Client.*

На вершині успадкування знаходиться клас *Object*, а на найнижчому рівні класи *Employee* і *Client*. Об'єкт підкласу також представляє собою об'єкт суперкласу. Тому в програмі ми можемо написати наступним чином:

```
Object george = new Person("George ");
Object alex = new Employee("Alex ", "Oracle");
Object vasia = new Client("Vasia ", "DeutscheBank", 2000);
Person petia = new Client("Petia ", "DeutscheBank", 3000);
Person yuzik = new Employee("Yuzik", "Google");
```

Це так зване **висхідне перетворення** (від підкласу до суперкласу вищої ієрархії) або *upcasting*. Таке перетворення здійснюється автоматично.

Зворотнє не завжди вірно. Наприклад, об'єкт *Person* не завжди є об'єктом *Employee* або *Client*. Тому **низхідне перетворення** або *downcasting* від суперкласу до підкласу автоматично не виконується. У цьому випадку потрібно використовувати операцію перетворення типів.

```
Object alex = new Employee("Alex", "Oracle");
// низхідне перетворення від Object до типу Employee
Employee new_alex = (Employee)alex;
new_alex.print();
System.out.println(new_alex.getCompany());
```

В даному випадку змінна *alex* приводиться до типу *Employee*. І потім через об'єкт *new_alex* можна звернутися до функціоналу об'єкта *Employee*.

Можна перетворити об'єкт *Employee* по всій прямій лінії успадкування від *Object* до *Employee*. Приклади низхідного перетворення:

```
Object vasia = new Client("Vasia", "DeutscheBank", 2000);
((Person)vasia).print();
Object alex = new Employee("Alex", "Oracle");
((Employee)alex).print();
```

В даному випадку змінна типу *Object* зберігає посилання на об'єкт *Client*. Можна без помилок привести цей об'єкт до типів *Person* або *Client*. Але при спробі перетворення до типу *Employee* буде помилка під час виконання, оскільки *kate* не є об'єктом типу *Employee*. Очевидно, що об’єкт *kate* – це посилання на об'єкт *Client*, а не *Employee*.

Однак часто дані приходять ззовні, і не можливо точно знати, який саме об'єкт ці дані представляють. Виникає велика ймовірність зіштовхнутися з помилкою при приведенні типів даних. Тому перед тим, як виконувати перетворення типів, варто перевірити, чи можливе виконання приведення за допомогою оператора *instanceof*:

```
Object vasia = new Client("Vasia", "DeutscheBank", 2000);
if (vasia instanceof Employee) { ((Employee)vasia).print(); }
else { System.out.println("Перетворення не коректне"); }
```

Вираз *vasia instanceof Employee* перевіряє, чи є змінна *vasia* об'єктом типу *Employee*. В даному випадку очевидно що не є, тому така перевірка поверне значення *false*, і спроба перетворення не відбудеться.

Узагальнення або *generics* (узагальнені типи і методи) дозволяють відійти від жорсткого визначення використовуваних типів.

Припустимо, є клас *Account* для подання банківського рахунку, який має наступний вигляд:

```
class Account {
   private int id;
   private int sum;
   Account(int id, int sum) { this.id = id;  this.sum = sum; }
   public int getId() { return id; }
   public int getSum() { return sum; }
   public void setSum(int sum) {this.sum = sum; }
}
```

Клас *Account* має два поля: *id* – унікальний ідентифікатор рахунку і *sum* – сума на рахунку. В даному випадку ідентифікатор заданий як цілочисельне значення, наприклад, 1, 2, 3, 4 і так далі. Але для ідентифікатора можуть використовуватися і рядкові значення. І числові, і рядкові значення мають свої переваги і недоліки. На момент написання коду, як правило, точно не відомо, що краще вибрати для збереження ідентифікатора – рядки або числа. Або, можливо, цей клас буде

використовуватися іншими розробниками, які можуть мати свою думку з даного питання. Наприклад, в якості типу *id* захочуть використовувати якийсь свій клас.

На перший погляд можна вирішити цю проблему наступним чином: поставити *id* як поле типу *Object*, який є універсальним і базовим суперкласом для всіх інших типів. При цьому можна буде створити об'єкт класу *Account* і з рядковим і з числовим значенням *id*:

```
Account acc1 = new Account(2334, 5000); // id - число
Account acc2 = new Account("sid5523", 5000); // id - рядок
```

Однак виникає проблема безпеки типів. Наприклад, в наступному випадку буде помилка:

```
Account acc1 = new Account("2345", 5000);
int acc1Id = (int)acc1.getId(); // java.lang.ClassCastException
```

Проблема може здаватися штучною, оскільки в даному випадку видно, що в конструктор передається рядок, тому навряд чи будемо намагатися перетворити його до типу *int*. Однак в процесі розробки можлива ситуація, коли невідомо, який саме тип представляє значення в *id*, і при спробі отримати число в даному випадку отримаємо виключенням *java.lang.ClassCastException*.

Створювати для кожного окремого типу свою версію класу *Account* теж не є раціональним рішенням, оскільки в цьому випадку буде багато повторів коду. Для вирішення подібних проблем були створені **узагальнення (*generics*)**. Узагальнення дозволяють не вказувати конкретний тип, який буде використовуватися. Тому визначимо клас *Account* як узагальнений:

```
class Account <T> {
    private T id;
    private int sum;
    Account(T id, int sum) { this.id = id; this.sum = sum; }
    public T getId() {return id; }
    public int getSum() {return sum; }
    public void setSum(int sum) {this.sum = sum; }
}
```

За допомогою літери *T* у визначенні класу *class Account <T>* ми показуємо, що даний тип *T* буде використовуватися цим класом. Параметр *T* в кутових дужках називається **універсальним параметром**, тому що замість нього можна підставити будь-який тип. При цьому не відомо, який саме це буде тип: *String*, int або якийсь інший. Причому буква *T* обрана умовно, це може і будь-яка інша буква або набір символів.

Після оголошення класу можна застосувати універсальний параметр *T*: так далі в класі оголошується змінна цього типу, якій потім присвоюється значення в конструкторі. Метод *getId()* повертає значення змінної *id*, але оскільки дана змінна представляє тип *T*, то даний метод також повертає об'єкт типу *T: public T getId()*.

Використаємо даний клас:

```
public class Program {
    public static void main(String[] args) {
        Account <String> acc1 = new Account <String> ("2345", 5000);
        String acc1Id = acc1.getId();
        System.out.println(acc1Id);
        Account <Integer> acc2 = new Account <Integer> (2345, 5000);
        Integer acc2Id = acc2.getId();
        System.out.println(acc2Id);
    }
}
```

При визначенні змінної даного класу і створенні об'єкта, після імені класу в кутових дужках потрібно вказати, який саме тип буде використовуватися замість універсального параметра. При цьому потрібно враховувати, що в якості типу *T* можуть виступати тільки вказівниковий тип, не примітивний. Тобто можна написати *Account <Integer>*, але не можна використовувати тип *int* або *double*, наприклад, *Account <int>*. Замість примітивних типів треба використовувати класи-обгортки: *Integer* замість *int, Double* замість *double* і т.д.

Наприклад, перший об'єкт буде використовувати тип *String*, тобто замість *T* буде підставлятися *String*:

Account <String> acc1 = new Account <String> ("2345", 5000);

У цьому випадку в якості першого параметра в конструктор передається рядок. А другий об'єкт використовує тип int (Integer):

Account <Integer> acc2 = new Account <Integer> (2345, 5000);

Крім узагальнених типів можна також створювати узагальнені методи, які так само будуть використовувати універсальні параметри. Наприклад:

```
public class Program {
    public static void main (String[] args) {
        Printer printer = new Printer();
        String [] people = { "George", "Vasia", "Yuzik", "Alex"};
        Integer [] numbers = {23, 4, 5, 2, 13, 456, 4};
        printer. <String> print(people);
        printer. <Integer> print(numbers);
    }
}
class Printer {
    public <T> void print (T [] items) {
        for (T item: items) {
            System.out.println(item);
        }
    }
}
```

Особливістю узагальненого методу є використання універсального параметру в оголошенні методу після всіх модифікаторів і перед типом значення, що повертається:

public <T> void print(T [] items)

Потім всередині методу всі значення типу *T* представлятимуть даний універсальний параметр. При виклику подібного методу перед його ім'ям в кутових дужках вказується, який тип буде передаватися на місце універсального параметру:

printer. <String> print(people);
printer. <Integer> print(numbers);

Можна також використовувати одночасно декілька універсальних параметрів. Конструктори як і методи також можуть бути узагальненими. В цьому випадку перед конструктором також вказуються в кутових дужках універсальні параметри.

Індивідуальні завдання:

(Для виконання індивідуальних завдань № варінта є для 532 групи порядковим номером прізвища студента в списку групи, для 531 – вказаний у додатку (окремий документ). Усі проекти та, за наявності, тести до них завантажити у власні репозиторії на Git Hub. Посилання на репозиторії проектів вказати у звіті та обов'язково долучати скріни успішного виконання)

1. Розробити консольний застосунок для роботи з базою даних, що зберігається у текстовому файлі (початковий масив не менше 5 записів). Структура бази даних описується ієрархією класів згідно вашого варіанта. Для ідентифікації спроби введення з клавіатури некоректних даних описати виключення. Реалізувати методи у базовому класі для:

– додавання записів;
– редагування записів;
– знищення записів;
– виведення інформації з файла на екран;
– обчислення та виведення на екран результатів згідно свого варіанта індивідуального завдання.

Меню програми реалізувати по натисненню на певні клавіші: наприклад, Enter – вихід, n - пошук, p – редагування тощо.

Один з методів індивідуального завдання зробити віртуальним.

(7 балів)

Варіант №	Батьківський (базовий) клас		Похідний клас		Реалізувати з допомогою окремих методів обчислення та виведення на екран таких даних:
	Сутність	Обов'язкові поля	Сутність	Обов'язкові поля	
1.	Навчальний курс	Назва, наявність іспиту	Практичне заняття	Дата, тема, кількість студентів	Середня кількість студентів, заняття з максимальною кількістю студентів, список тем з певним словом у назві
2.	Трамвайна зупинка	Назва, список номерів маршрутів	Година	Кількість пасажирів, коментар	Загальна кількість пасажирів, година з найменшою кількістю пасажирів, найдовший коментар
3.	Навчальний курс	Назва, прізвище викладача	Лекція	Дата, тема, кількість студентів	Лекція з мінімальною кількістю студентів, список тем з певним словом у назві, остання літера у прізвищі викладача
4.	Конференція	Назва, місце проведення	Засідання	Дата, тема, кількість учасників	Середня кількість учасників на засіданні, засідання з найбільшою кількістю учасників, довжина назви
5.	Виставка	Назва, прізвище художника	День	Кількість відвідувачів, коментар	Сумарна кількість відвідувачів, день з найменшою кількістю відвідувачів, список коментарів з певним словом
6.	Станція метрополітену	Назва, рік відкриття	Година	Кількість пасажирів, коментар	Сумарна кількість пасажирів, години з найменшою кількістю пасажирів та найбільшою кількістю слів у коментарі
7.	Лікар	Прізвище, фах	Прийом	День, зміна, кількість відвідувачів	Загальна кількість відвідувачів, прийом з мінімальною кількістю відвідувачів, довжина прізвища
8.	Поет	Прізвище, мова, кількість збірок	Виступ	Дата, місце, кількість слухачів	Сумарна кількість слухачів, день з найбільшою кількістю слухачів, довжина прізвища
9.	Лікар	Прізвище, стаж	Прийом	День, кількість відвідувачів, коментар	Середня кількість відвідувачів, прийом з мінімальною кількістю відвідувачів, найдовшим коментарем
10.	Трамвайний маршрут	Номер, середній інтервал руху	Зупинка	Назва, кількість пасажирів	Загальна кількість пасажирів, зупинки з найменшою кількістю пасажирів, найдовшою назвою
11.	Цілодобовий кіоск	Назва, адреса	Година	Кількість покупців, коментар	Загальна кількість покупців, година з найменшою кількістю покупців, коментарями з певними словами
12.	Виконавець	Прізвище, жанр	Концерт	Дата, кількість глядачів	Загальна кількість глядачів, концерт з максимальною кількістю глядачів, кількість слів у назві жанру
13.	Музичний гурт	Назва, прізвище керівника	Гастрольна поїздка	Місто, рік, кількість концертів	Гастрольна поїздка з максимальною кількістю концертів, список гастрольних поїздок у певне місто, остання літера в прізвищі керівника
14.	Навчальний курс	Назва курсу, назва кафедри	Лекція	Дата, група, кількість студентів	Середня кількість студентів, лекції з найбільшою кількістю студентів, кількість слів у назві кафедри
15.	Виставка	Назва, прізвище скульптора	День	Кількість відвідувачів, коментар	Сумарна кількість відвідувачів, день з найбільшою кількістю відвідувачів, день з найбільшою кількістю слів у коментарі
16.	Письменник	Прізвище, мова, кількість книжок	Виступ	Дата, місце, кількість слухачів	Сумарна кількість слухачів, день з найменшою кількістю слухачів, довжина прізвища

17.	Співробітник	ПІБ, посада	Робочій день	Дата, кількість годин, назва проекту	Середня кількість робочих годин за період; Кількість годин на проекті; Дні з максимальним навантаженням
18.	Лікар	ПІБ, спеціальність	Робочій день	Дата, кількість пацієнтів, час початку роботи	Середня кількість пацієнтів в день за період; Кількість днів з максимальним навантаженням; Дні, коли починав приймати після зазначеного часу
19.	Піцерія	Назва, адреса	Робочій день	Дата, кількість замовлень, піца дня	Середня кількість замовлень в день за період; Дні з максимальним відвідуванням; Сумарна кількість замовлень для днів з визначеною піцою дня
20.	Басейн	Назва, адреса	Робочій день	Дата, кількість відвідувачів, кількість доступних доріжок	Середня кількість відвідувань в день за період; Дні з мінімальною кількістю доступних доріжок; Кількість днів, коли було доступно не менше зазначеної кількості доріжок
21.	Бібліотека	Назва, адреса	Робочій день	Дата, кількість книг, що видано, кількість книг, що повернуто	Середній рух книжок в день за період; Кількість днів, коли було видано книг більше, ніж повернуто; Дні, коли видана парна кількість книг, а повернута – непарна
22.	Сайт	Назва, URL	Відвідування	Дата, кількість унікальних хостів, кількість завантажених сторінок	Середня кількість хостів в день за період; Дні з максимальною кількістю завантажених сторінок; Кількість днів, коли співвідношення хостів до сторінок перевищує задане значення
23.	Акаунт електронної пошти	E-mail, ПІБ володаря	Спам	Дата, кількість спам повідомлень, загальна кількість повідомлень	Середня кількість спаму в день за період; Кількість днів, коли відсоток спам повідомлень був менший за задане значення; Дні, коли кількість спаму збільшувалась
24.	Акція компанії на біржі	Назва компанії, код на біржі	Курс	Дата, курс відкриття, курс закриття	Середня вартість акцій по закриттю за період; Кількість днів, коли курс зростав протягом дня; Дні, коли зміна курсу за день перевищувала задане значення
25.	Телефонний номер	Номер, оператор	Дзвінки	Дата, кількість хвилин розмов, кошти, що використано на розмови	Середня платня в день за період; Кількість днів, коли вартість хвилини розмови перевищувала задане значення; Дні, коли кількість хвилин розмов була парна

2. Модифікувати консольний застосунок з попереднього завдання таким чином, щоб:
А) Базовий клас був абстрактним.
Б) Реалізація методів індивідуального завдання була перенесена у похідний клас.
(3 бала)