ЗАТВЕРДЖУЮ
Завідувач кафедри управління проектами, інформаційних технологій та телекомунікацій, д.т.н., професор
Євген МАРТИН
«_ _»_____20___ р.

МЕТОДИЧНА РОЗРОБКА

для проведення лабораторного заняття зі студентами (курсантами) 1-го курсу спеціальності 122 «Комп'ютерні науки» з дисципліни «Основи програмування»

Лабораторне заняття 6.10: «Застосування наслідування. Робота з вкладеними та внутрішніми класами»

Мета закріпити теоретичні знання та практичні навики щодо побудови ієрархії класів застосовуючи властивість наслідування та побудови нестатичних вкладених класів. А також закріплення навиків щодо застосування ключового слова super в наслідуванні.

Час: 2 годин

Місце заняття: комп'ютерна лабораторія

Матеріальне забезпечення:
- ПК з відповідним ПЗ та доступом до мережі Internet;
- Методичні рекомендації щодо виконання лабораторної роботи.

Після лабораторного заняття студенти (курсанти) повинні:
вміти: здійснювати побудову багаторівневої ієрархічної структури класів використовуючи основні принципи наслідування та будувати вкладені класи із можливістю їх використання в межах зовнішнього класу.
знати: принципи і особливості застосування вкладених класів та наслідування в обє'ктно-орієнтованому програмуванні.

Література:
1. Віртуальне навчальне середовище ЛДУ БЖД «Віртуальний університет» [Електронний ресурс]. – Доступний з : http://virt.ldubgd.edu.ua

План заняття:
1. Застосування наслідування: побудова однорівневої ієрархії
2. Застосування наслідування: побудова багаторівневої ієрархії
3. Побудова вкладених класів
4. Видача індивідуальних практичних завдань

1. Застосування наслідування: побудова одноріневої ієрархії

Для першого застосунку та побудови одноріневої ієрархії класів із принципом наслідування обираємо вже відомий приклад, а саме клас для розрахунку параметрів кругової пожежі (з попереднього лабораторного заняття):

```
class FireCircular {
	private double radius;

	FireCircular(double radius) {
		this.radius = radius;
	}

	double square() {
		double s;
		s = Math.PI * Math.pow(radius, 2);
		return s;
	}

	void perimeter() {
		double p = 2 * Math.PI * radius;
		System.out.println(p);
	}

	void front() {
		double f = 2 * Math.PI * radius;
		System.out.println(f);
	}

	public double getRadius() {
		return radius;
	}

	public void setRadius(double radius) {
		this.radius = radius;
	}
}
```

На даному етапі наведений приклад не сповідує принципу наслідування. Далі означений клас ми будемо використовувати в якості батьківського (супер) класу. Відповідно наступним кроком є побудова його дочірнього класу `FireCircularDemo` та зазначення взаємозв'язку між ними за допомогою ключового слова **`extends`**:

```
class FireCircularDemo extends FireCircular {

	double n;

	public FireCircularDemo(double radius, double n) {
		super(radius);
		this.n = n;
```

```
        }
    }
```

Очевидно, що при створенні екземплярів класу `FireCircularDemo` необхідно буде ініціалізувати поля (змінні екземплярів) `n` та `radius`. Для цього ми генеруємо конструктор. Змінна екземпляра класу `n` введена безпосередньо у дочірньому класі, а змінна `radius` успадкувалась від батьківського класу `FireCircular`. Для того, щоб не розмножувати безліч однотипних конструкторів та зберегти можливість використання конструктора класу `FireCircular` при створенні екземпляра класу `FireCircularDemo`, в генерованому конструкторі передбачено використання ключового слова **`super`**.

Парадигма наслідування передбачає не лише доступ до змінних та можливість використання конструктора суперкласу, а й доступ до усіх його методів. Нагадаємо чудовий вислів: дочірній клас знає та уміє все те, що і його батьківський клас (знає – доступ до змінних, вміє – доступ до методів).

Можливість доступу до методів батьківського класу розглянемо в наступному прикладі. В класі `FireCircularDemo` напишемо метод, який маючи відомості про кількість пожеж (змінна **`double`** `n`) розраховує їх загальну площу використовуючи метод `squareN ()`:

```
class FireCircularDemo extends FireCircular {

        double n;

        public FireCircularDemo(double radius, double n) {
                super(radius);
                this.n = n;
        }

    double squareN () {
                double k;
                k = n * square ();
                return k;
        }
    }
```

Як видно з наведеного прикладу в тілі методу `squareN()` міститься посилання на метод `square()` з батьківського класу без жодних звернень до екземпляра відповідного класу. Ця властивість забезпечується за рахунок застосування парадигми наслідування. В дочірніх класах доступні усі відкриті методи батьківського класу.

Для перевірки дієвості наведеного прикладу проведемо виклик методу `squareN()` у класі з `main()`-методом:

```
public class MainFire {
        public static void main(String[] args) {
                FireCircular fc = new FireCircular(2);
                FireCircularDemo fcd = new FireCircularDemo(fc.getRadius(),
3);
                System.out.println(fcd.square());
                System.out.println(fcd.squareN());
```

```
        }
}
```

Результат роботи програми:

```
12.566370614359172
37.69911184307752
```

З наведеного прикладу видно, що метод `squareN()` є цілком дієвим, а звернення в його тілі до методу `square()` з батьківського класу є доступним і не перешкоджає роботі програми. Крім того в `main()`-методі можна спостерігати виклик до екземпляра класу `FireCircularDemo` не лише власного методу `squareN()`, а й методу `square()` із батьківського класу, що цілком відповідає парадигмі наслідування.

В аналізованому прикладі зустрічається ще один цікавий факт, який не розглядався раніше – це можливість передачі посилання на метод `getRadius()` з метою ініціалізації змінної `radius`. Для чого це потрібно? Змінну `radius` ми ініціалізуємо вперше під час створення екземпляра класу `FireCircular` та присвоюємо їй значення 2. Це значення використовується в подальших розрахунках під час виклику методу `square()`, як в бітьківському так і в дочірньому класах. З тією метою, щоб значення змінної `radius` залишалось завжди актуальним (однаковим) під час виконання методу `square()` в усіх класах, при другому, третьому … та усіх чергових зверненнях до цієї змінної слід використовувати посилання на неї. А так як ця змінна закрита модифікатором **`private`** то доступ до неї забезпечується через метод `getRadius()` відповідного екземпляра класу.

Звичайно, можливий варіант постійної ініціалізації змінної `radius` числовим значенням. Проте в такому випадку, якщо значення змінної завжди має бути актуальним (однаковим), необхідно постійно його відстежувати при створенні чергового екземпляра класу:

```
FireCircular fc = new FireCircular(2);
FireCircularDemo fcd = new FireCircularDemo(2, 3);
```

Йдемо далі. Нам відомо, що доступ до закритих (**`private`**) змінних екземпляра класу забезпечується через методи `get()` та `set()`. А що робити, якщо закритий цілий метод, а результат його роботи необхідно використати в дочірньому класі? Для цього необхідно створити додатковий метод, який приймає лише результат роботи, а не логіку закритого методу.

Розглянемо приклад. В класі `FireCircular` закриємо метод `square()` модифікатором **`private`**:

```
class FireCircular {
      private double radius;

      FireCircular(double radius) {
            this.radius = radius;
      }

      private double square() {
```

```
        double s;
        s = Math.PI * Math.pow(radius, 2);
        return s;
    }

    void perimeter() {
        double p = 2 * Math.PI * radius;
        System.out.println(p);
    }

    void front() {
        double f = 2 * Math.PI * radius;
        System.out.println(f);
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }
}
```

За такого виконання метод `square()` не буде доступним ні в класі `MainFire`, ні в класі `FireCircularDemo`. Своєрідним «геттером» для цього методу може бути інший відкритий метод `squareDemo()`, який приймає результат його роботи:

```
...
    private double square() {
        double s;
        s = Math.PI * Math.pow(radius, 2);
        return s;
    }

    double squareDemo() {
        double s = square();
        return s;
    }
...
```

Таким чином доступ до закритого методу `square()` забезпечено через відкритий метод `squareDemo()`. Суть такої процедури полягає в тому, що логіка методу `square()` закрита та недоступна з іншого класу, а результат роботи завжди залишається доступний. Перевіримо працездатність зміненого коду провівши відповідні зміни в класах `MainFire` та `FireCircularDemo`:

```
class FireCircularDemo extends FireCircular {

    double n;

    public FireCircularDemo(double radius, double n) {
```

```
            super(radius);
            this.n = n;
        }

    double squareN () {
            double k;
            k = n * squareDemo ();
            return k;
        }
    }

    public class MainFire {
        public static void main(String[] args) {
            FireCircular fc = new FireCircular(2);
            FireCircularDemo fcd = new FireCircularDemo(fc.getRadius(),
    3);
            System.out.println(fcd.squareDemo());
            System.out.println(fcd.squareN());

        }
    }
```

Результат роботи програми не змінився:

```
12.566370614359172
37.69911184307752
```

2. Застосування наслідування: побудова багаторівневої ієрархії

Для реалізації багаторівневої ієрархії використаємо уже відомі нам приклади з попереднього лабораторного заняття, а саме клас `FireAngular`, логіка якого націлена на визначення параметрів кутової пожежі. В попередньому прикладі ми розглядали три варіанти розрахунку параметрів пожежі: за умови кута пожежі 90^0, 180^0 та 270^0. Якщо розглянути умову коли кут пожежі сягає 360^0 то пожежа стає круговою, а розрахунок її параметрів можна проводити використовуючи уже логіку класу `FireCircular`.

За для виконання умови багаторівневої ієрархії організуємо наслідування класу `FireAngular` від класу `FireCircularDemo`. Таким чином дочірній клас отримає доступ не лише до членів класу `FireCircularDemo`, а й до `FireCircular`.

```
public class FireAngular extends FireCircularDemo {
    double angular;

    public FireAngular(double radius, double n, double angular) {
        super(radius, n);
        this.angular = angular;
    }

    double square() {
        double s = 0;
        if (angular == 90) {
            s = 0.25 * Math.PI * Math.pow(getRadius(), 2);
        } else if (angular == 180) {
```

```
                s = 0.5 * Math.PI * Math.pow(getRadius(), 2);
            } else if (angular == 270) {
                s = 0.75 * Math.PI * Math.pow(getRadius(), 2);
            }else if (angular == 360){
                s = squareDemo(); // застосунок методу з суперкласу
            }
            return s;
        }

        void perimeter() {
            if (angular == 90) {
                double p = (0.5 * Math.PI * getRadius()) + (2 * getRadius());
                System.out.println(p);
            } else if (angular == 180) {
                double p = (Math.PI * getRadius()) + (2 * getRadius());
                System.out.println(p);
            } else if (angular == 270) {
                double p = (1.5 * Math.PI * getRadius()) + (2 * getRadius());
                System.out.println(p);
            }else if (angular == 360){
                super.perimeter(); // застосунок методу з суперкласу із
ключовим словом супер
            }
        }

        void front() {
            if (angular == 90) {
                double f = 0.5 * Math.PI * getRadius();
                System.out.println(f);
            } else if (angular == 180) {
                double f = Math.PI * getRadius();
                System.out.println(f);
            } else if (angular == 270) {
                double f = 1.5 * Math.PI * getRadius();
                System.out.println(f);
            }else if (angular == 360){
                super.front(); // застосунок методу з суперкласу із ключо-
вим словом супер

            }
        }
}
```

З переставленого дочірнього класу `FireAngular` спостерігаються дві нові позиції. Перша – це можливість виклику методів порушуючи послідовність наслідування класів. Що мається на увазі? Клас `FireAngular` унаслідуваний від класу `FireCircularDemo`, проте в останньому класі не міститься методу `squareDemo()`, що викликано в дочірньому класі. Цей метод розміщено в класі `FireCircular`, який є батьківським для класу `FireCircularDemo`. Таким чином можна зробити висновок, що для класу `FireAngular` будуть доступні усі відкриті члени як класу

FireCircularDemo, так і класу FireCircular. Саме за рахунок такої властивості наслідування можливе забезпечення багаторівневої ієрархії класів.

Друга особливість – це застосування ключового слова **super** для виклику методу з суперкласу. Раніше це слово ми застосовували лише для виклику конструктора з суперкласу (отримання можливості використовувати конструктор батьківського класу). В наведеному прикладі в методах perimeter() та front() за умови досягнення кута пожежі = 360^0 необхідно викликати однойменні методи з батьківського класу. Проте подібний виклик без ключового слова **super** призведе до виклику методу безпосередньо з самого класу FireAngular, а не з класу FireCircular. За умови такого виклику відбудеться явище рекурсії. Для того щоб дати Java VM зрозуміти, що ми хочемо викликати методи perimeter() та front() саме з класу FireCircular застосовують ключове слово **super**. Його застосування можливе не лише за умови виклику методів батьківського класу, а і за потреби виклику змінних. В зазначених прикладах чудово поєднуються дві парадигми ОПП – це наслідування та поліморфізм.

Працездатність написаного коду перевіримо в головному класі:

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

		FireCircular fc = new FireCircular(3);
		System.out.println(fc.squareDemo());
		fc.perimeter();

		FireCircularDemo fcd = new FireCircularDemo(fc.getRadius(), 3);
		System.out.println(fcd.squareDemo());
		System.out.println(fcd.squareN());

		FireAngular fa = new FireAngular(fcd.getRadius(), fcd.n, 360);
		System.out.println(fa.square());
		fa.perimeter();
	}
}
```

Результат роботи:

```
28.274333882308138
18.84955592153876
28.274333882308138
84.82300164692441
28.274333882308138
18.84955592153876
```

3. Побудова вкладених класів

Реалізацію вкладених класів проведемо із використанням нового прикладу (без пожеж). Як відомо вкладений клас – це клас, що реалізує логіку в межах зовнішнього класу. Якщо вкладеному класу доступні члени (змінні, методи) свого зовнішнього класу, то він має назву внутрішнього або не статичного. Ключовим моментом є те, що зовнішній клас не має прямого доступу до членів вкладеного класу. Такий доступ необхідно реалізувати через спеціальні методи та екземпляри класів. Ро-

зглянемо приклад, який заснований на розрахунку деяких параметрів роботи системи масового обслуговування:

```
class Multichannel_imovKanVil {
	double Mz;
	double Mp;
	double Nv;

	public Multichannel_imovKanVil(double mz, double mp, double nv) {
		Mz = mz;
		Mp = mp;
		Nv = nv;
	}

	double zvedGust() {
		double z = (1 / Mz) / (1 / Mp);
		return z;
	}

	double fact(double i) {
		if (i == 1) {
			return 1;
		} else {
			double res = fact(i - 1) * i;
			return res;
		}
	}

	double imovKanVilDemo(double i) {
		if (i == 0) {
			return 1;
		} else {
			double tmRes = (Math.pow(zvedGust(), i)) / fact(i);
			double tmResN = imovKanVilDemo(i - 1);
			double res = tmRes + tmResN;
			return res;
		}
	}

	double imovKanVil() {
		double res = (Math.pow(imovKanVilDemo(Nv), -1));
		return res;
	}

	void test() {
		Multichannel_imovKanZad mikz = new Multichannel_imovKanZad();
		mikz.push();
	}

	class Multichannel_imovKanZad {
		double mass[] = new double[(int) Nv];

		void push() {
```

```
                for (int i = (int) Nv; i > 0; i--) {
                    mass[i - 1] = (Math.pow(zvedGust(), i) / fact(i)) *
imovKanVil();
                    System.out.println("Ймовірність, що " + i + " канали
задіяний: " + mass[i - 1]);
                }
            }
        }
}
```

Спочатку коротко про зміст коду, а далі про його логіку. Отже клас містить три змінні, значення яких потрібні для розрахунків. Для ініціалізації цих змінних передбачено конструктор. Методи `zvedGust()`, `fact(double i)` та `imovKanVilDemo(double i)` потрібні для розрахунку проміжних значень. У логіку цих методів не занурюємось, окрім того що два останні – це рекурсивні методи!

Метод `imovKanVil()` є основним та відповідає за визначення ймовірності того, що усі канали системи масового обслуговування будуть вільними. Теорія масового обслуговування також передбачає можливість визначення ймовірності того, що n каналів системи будуть задіяні. Логіку цього розрахунку винесено до вкладеного класу `Multichannel_imovKanZad`.

В останньому класі оголошено масив для запису результатів проміжних розрахунків, а в методі `push()` проводиться циклічне виконання відповідних розрахунків (канал 1, канал 2, канал n).

Слід відмітити, що логікою методу `push()` передбачено звернення до членів зовнішнього класу, а саме змінної `Nv`, методів `zvedGust()`, `fact(i)` та `imovKanVil()`. Таке звернення можливе і це відповідає правилам застосування не статичних вкладених класів. Проте ні у якому разі не навпаки. З зовнішнього класу `Multichannel_imovKanVil` відсутній будь який доступ до членів вкладеного класу. Такий доступ потрібно реалізовувати через екземпляр відповідного класу, що передбачено в методі `test()`.

Розглянемо наочність такого обмеження в головному класі:

```
public class MainDemo {

    public static void main(String[] args) {
        Multichannel_imovKanVil dm = new Multichannel_imovKanVil(5, 4, 2);
        System.out.println(dm.imovKanVil());
//      dm.push(); // помилка – відсутній доступ
        dm.test();
    }
}
```

Результат роботи:

```
0.4716981132075471
Ймовірність, що 2 канали задіяний: 0.15094339622641512
Ймовірність, що 1 канали задіяний: 0.3773584905660377
```

Як видно з представленого прикладу, доступ до членів вкладеного класу можливий лише через метод `test()`, який містить екземпляр класу `Multichannel_imovKanZad`. Такий спосіб реалізації класів – це додаткова можливість

закрити логіку виконання окремих частин коду від зовнішнього впливу та організовувати доступ лише до результатів виконання цієї логіки.

Отже, як підсумок практичного заняття, слід зазначити, що парадигма наслідування дозволяє значно спростити програмний код, організовувати зручну та зрозумілу ієрархію класів, надавати можливість використовувати методи батьківських класів без дублювання програмного коду тощо. Окремі приклади застосунку наслідування дозволяють уявити повну суть взаємодії різних класів між собою при створенні розгалуженої структури програмної системи. А використання вкладених класів дозволяє інкапсулювати окремі частини програмного коду без використання модифікаторів доступу.

4. Видача індивідуальних практичних завдань

У відповідності до тематики заняття для виконання лабораторної роботи необхідно розширити раніше написану об’єктно-орієнтовану програму із дотриманням таких вказівок:

1. Вихідними даними до лабораторної роботи є раніше написана об’єктно-орієнтована програма з довільною логікою за результатами виконання Індивідуального практичного заняття за темою №6.8.

2. Створити додатковий клас із довільною логікою (4-й клас) у якому передбачити використання внутрішнього (вкладеного класу). Вкладений клас має містити метод, що використовує члени свого зовнішнього класу (методи та/або змінні). Доступ до методу вкладеного класу (виклик методу) слід реалізувати через екземпляр вкладеного класу у спеціально реалізованому методі test() за прикладом, що наведений нижче:

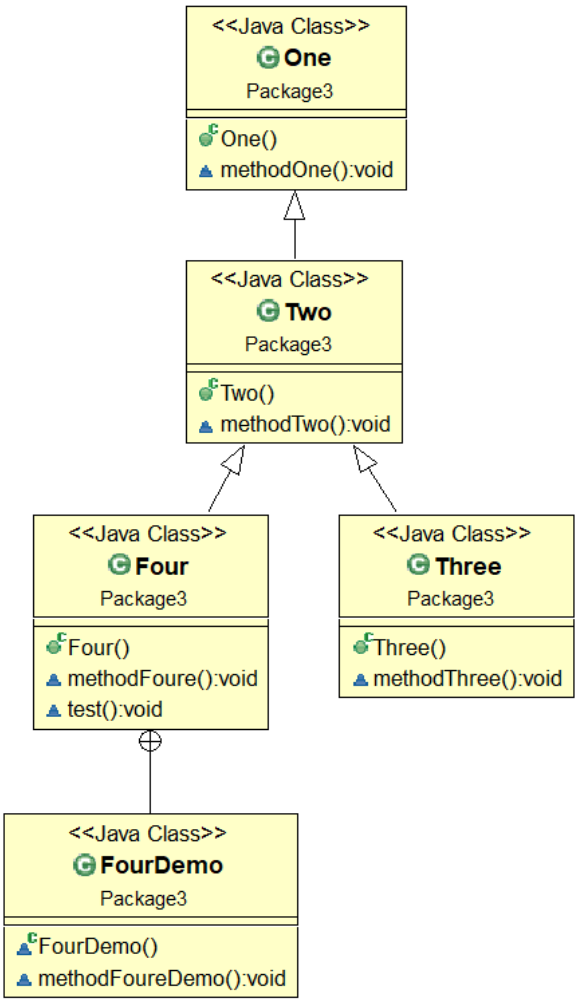
```
public class Four extends Two {

	void methodFoure () {
		methodTwo();
		System.out.println("method foure does something");
	}

	void test () {
		FourDemo fd = new FourDemo();
		fd.methodFoureDemo();
	}

	class FourDemo {
		void methodFoureDemo () {
			System.out.println("method foureDemo does something");
			methodFoure ();
		}
	}
}
```

3. Загальну ієрархію класів (4-х класів з довільною логікою) необхідно реалізувати використовуючи принципи наслідування за структурою, що наведена на UML-діаграмі:



4. У кожному з існуючих класів необхідно ввести (написати) по одному методу, який реалізовуватиме власну довільну логіку та логіку методів батьківських класів. Для кращого розуміння завдання означеного пункту наведемо приклад подібної реалізації:

```
public class One {
	void methodOne () {
		System.out.println("method one does something");
	}
}

public class Two extends One {
	void methodTwo () {
		methodOne();
		System.out.println("method two does something");
	}
}

public class Three extends Two {
	void methodThree () {
		methodOne();
```

```
            System.out.println("method three does something");
        }
    }
```

5. Конструктори у класах необхідно згенерувати таким чином, щоб забезпечити можливість у них конструкторів із батьківських класів (з використанням ключового слова `super`).

6. В головному класі з `main()`-методом (5-й клас) викликати усі новостворені методи за результатами виконання лабораторної роботи.

Результатом виконання лабораторної роботи є розширена конструкція об'єктно-орієнтованої програми з довільною логікою та дотриманням принципів наслідування, яка міститиме методи, що використовують логіку батьківських та вкладених класів.

Звіт до лабораторної роботи подається п'ятьма файлами із розширенням *.java. Назви файлів мають відповідати назвам класів. Зміст файлів має відповідати поставленій умові в п.4. Програмний код рекомендовано супроводжувати коментарями щодо його змісту та логіки.

Звіт за результатами виконання лабораторної роботи (5 файлів) завантажується у відповідну категорію Віртуального університету.

Методичну розробку розробив:
Заступник начальника кафедри УПІТтаТ
підполковник служби цивільного захисту Олександр ПРИДАТКО

Методична розробка обговорена на засіданні кафедри УПІТтаТ
Протокол № ____ від "_ __" _____ 20__ р.