Кафедра управління проектами, інформаційних технологій та телекомунікацій

Спеціальність 122 «Комп'ютерні науки», ступінь вищої освіти «бакалавр»,
II семестр

Дисципліна «Основи програмування»

ТЕМА №6. ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

ЛЕКЦІЯ №6.9. ІЄРАРХІЯ КЛАСІВ. НАСЛІДУВАННЯ

План:

Вступ. Введення в наслідування
1. Вкладені та внутрішні класи
2. Наслідування
3. Ключове слово super

Література:

1. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с. ***(наявне в віртуальному середовищі)***

2. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с. ***(наявне в віртуальному середовищі)***

3. Програмування на Java (рос.) [Електронний ресурс]. – http://kostin.ws/java **(*наявне в віртуальному середовищі)***

4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java ***(наявне в віртуальному середовищі)***

5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java **(*наявне в віртуальному середовищі)***

Вступ. Введення в наслідування

Процес, в результаті якого один об'єкт (екземпляр класу) отримує властивості іншого, називається наслідуванням. Це дуже важливий принцип об'єктно-орієнтованого програмування, оскільки успадкування забезпечує принцип ієрархічної класифікації. Наприклад ноутбук – частина класифікації персональних комп'ютерів, які, своєю чергою, є частиною класифікації електронних обчислювальних машин. Без ієрархії кожен об'єкт мав би визначати всі свої характеристики окремо. Проте завдячуючи наслідуванню, об'єкт може визначати лише ті характеристики, які роблять його унікальним. Ноутбук може наслідувати загальні атрибути від свого батьківського класу – ПК, проте унікальність його забезпечується за рахунок наявності батареї, точ-паду, можливості мобільного застосунку тощо. Решта основних компонентів,

таких як пам'ять, материна плата, клавіатура не є унікальним та можуть бути унаслідувані від батьківського класу, а відтак їх чергове перевизначення при створенні об’єкту «ноутбук», який є дочірнім класом «персонального комп’ютера» не має сенсу.

Наслідування зв’язано також з інкапсуляцією. Якщо окремий клас інкапсулює окремі властивості, то любий його підклас (дочірній клас) буде мати ті самі властивості плюс будь які інші додаткові, що визначені безпосередньо в дочірньому класі. Новий підклас наслідує атрибути всіх своїх батьківських класів і тому не містить непередбачуваних взаємодій з рештою коду системи.

1. Вкладені та внутрішні класи

В мові програмування Java дозволяється визначати один клас в межах іншого. Такі класи мають назву **вкладених**. Область дії вкладених класів обмежується областю дії зовнішнього класу. Так до прикладу, якщо клас B визначений в класі A, то клас B не може існувати незалежно від класу A. Вкладений клас має доступ до членів зовнішнього класу. Проте зовнішній клас не має доступу до кленів вкладеного класу.

Існує два типи вкладених класів: ***статичні*** та ***нестатичні***. Статичним називається вкладений клас, який оголошено з модифікатором `static`. Оскільки такі класи вважатимуть статичними, то звернення до нестатичних членів свого зовнішнього класу має забезпечуватись через екземпляр класу (об’єкт). Це означає те, що вкладений статичний клас не може посилатись безпосередньо на нестатичні члени свого зовнішнього класу. Зважаючи на можливу плутанину та складність такої реалізації, статичні вкладені класи застосовуються не часто.

Більш важливим типом вкладеного класу являється **внутрішній** клас. Внутрішній клас – це нестатичний вкладений клас. Такі класи мають доступ до всіх змінних та методів свого зовнішнього класу.

З метою наочності розглянемо нижче приклад використання внутрішнього класу. В класі `Outer` оголошено одну змінну екземпляра класу `outer_x` та визначено один метод `test()`, а також оголошено внутрішній клас `Inner`.

```
class Outer {
	int outer_x = 100;

	void test() {
		Inner inner = new Inner();
		inner.display();
	}

	class Inner {
		void display() {
			System.out.println(outer_x);
		}
```

```
	}
}
class InnerClassDemo {
	public static void main(String args[]) {
		Outer outer = new Outer();
		outer.test();
	}
}
```

Результат роботи програми:

```
100
```

В програмі з наведеного прикладу внутрішній клас `Inner` оголошено (визначено) в області дії класу `Outer`. Зважаючи на це будь яка частина коду внутрішнього класу `Inner` може звертатись до змінної `outer_x`. Метод внутрішнього класу `display()` виводить у консоль значення змінної `outer_x`. Проте так як у зовнішньому класі викликати метод `display()` з внутрішнього класу не можливо, то ця конструкція реалізується через екземпляр вкладеного класу в методі `test()`. В `main()`–методі виклик методу `display()` реалізовано через виклик методу `test()` до екземпляра класу `Outer`. А безпосередньо в методі `test()` створено екземпляр класу `Inner`, та реалізовано виклик на виконання методу `display()`.

Слід зважати на те, що екземпляр класу `Inner` може бути створений тільки в середині класу `Outer`. В противному випадку компілятор Java видасть помилку.

Як було згадано раніше внутрішній клас має доступ до усіх членів зовнішнього класу, проте НЕ НАВПАКИ. Члени внутрішнього класу доступні лише в області дії цього класу та не можуть бути використані зовнішнім класом. Розглянемо приклад:

```
class Outer {
	int outer_x = 100;

	void test() {
		Inner inner = new Inner();
		inner.display();
	}

	class Inner {
		int y = 10; //локальна змінна класу Inner
		void display() {
			System.out.println(outer_x);
		}
	}

	void showy () {
		System.out.println(y); // помилка
	}
```

```
}
```

З наведеного прикладу видно, що змінна `y` оголошена як змінна екземпляра класу `Inner`, тому вона не доступна за межами цього класу та не може використатись в методі `showy()`.

Усі попередні приклади розглядали випадок, коли внутрішній клас оголошено як член в області дії зовнішнього класу. Проте внутрішні класи можливо оголошувати і в області дії будь якого блока коду. Наприклад внутрішній клас можна оголосити в блоці коку певного методу, або навіть в тілі циклу for:

```
class Outer {
      int outer_x = 100;

      void test() {
            for (int i = 0; i < 5; i++) {
                  class Inner {
                        void display() {
                              System.out.println(outer_x);
                        }
                  }
                  Inner inner = new Inner();
                  inner.display();
            }
      }
}

class InnerClassDemo {
      public static void main(String args[]) {
            Outer outer = new Outer();
            outer.test();
      }
}
```

Результат роботи програми:

```
100
100
100
100
100
```

І на завершення. Спочатку розробники Java (версія 1.0) не допускали створення вкладених класів, лише з появою чергової версії Java 1.1 з’являється така можливість.

2. Наслідування

Ключове місце у створенні ієрархічної структури класів займає наслідування – це один з основних принципів об'єктно-орієнтованого програмування.

Використовуючи наслідування можливо створити клас, що може унаслідуватись іншими, більш спеціалізованими класами, кожен з яких буде додавати свої особливі характеристики. В термінології об'єктно-орієнтованого програмування клас від якого унаслідуються називають ***суперкласо***м, або батьківським класом, а унаслідуваний клас – ***підкласом***, або дочірнім класом. Відповідно до зазначеного твердження, підклас – це спеціалізована версія суперкласу. Підклас наслідує усі члени, що оголошені в суперкласі, додаючи до них власні елементи.

Для того, щоб наслідувати клас достатньо ввести назву батьківського класу після імені дочірнього класу використавши ключове слово **extends**. Для наочності розглянемо короткий приклад де підклас B наслідує суперклас A.

```
class A {
      int i, j;

      void showij () {
            System.out.println(i+" "+j);
      }
}

class B extends A {
      int k;

      void showk() {
            System.out.println(k);
      }

      void sum() {
            System.out.println("Сума: " + (i+j+k));
      }
}

class SimpleInheritance {

      public static void main(String[] args) {
            A superOb = new A();
            B subOb = new B();

            superOb.i = 10;
            superOb.j = 20;
            superOb.showij();

            subOb.i = 7;
            subOb.j = 8;
```

```
            subOb.k = 9;
            subOb.showij();
            subOb.showk();
            subOb.sum();
        }
}
```

Результат роботи програми:

```
10 20
7 8
9
Сума: 24
```

З наведеного прикладу видно, що екземпляр суперкласу A має доступ до змінних та методів свого класу та може встановлювати їм значення і викликати відповідні методи. Що стосується екземпляра дочірнього класу B, то йому окрім власної змінної `k` також доступні змінні батьківського класу `i` та `j`. Крім того екземпляр дочірнього класу може викликати не лише методи свого класу `showk()` та `sum()`, але й метод батьківського класу `showij()`. Крім того в методі `sum()`, який розміщено в класі B, можливе безпосереднє посилання на змінні `i` та `j`, що оголошені в класі A. Таким чином можна узагальнити, що для будь якого дочірнього класу доступні усі відкриті (крім **`private`**) змінні та методи батьківського класу від якого він унаслідуваний. В цьому можна переконатись проаналізувавши результат роботи програми з наведеного прикладу.

Те, що один клас може виступати суперкласом для іншого не виключає можливості його самостійного використання без прив'язки до підкласу. Крім того підклас також може виступати суперкласом для іншого класу, забезпечуючи при цьому трирівневу ієрархічну структуру наслідування, до прикладу:

```
class A {
}

class B extends A {
}

class C extends B {
}
```

Для кожного нового підкласу потрібно вказувати лише один суперклас. В Java не підтримується наслідування декількох суперкласів в одному підкласі. Як представлено в попередньому прикладі, альтернативою багаторівневої ієрархії є послідовне наслідування. В такому разі останньому в ієрархії класі C будуть доступні усі відкриті змінні та методи як із класу B так і з класу A. Для наочності дещо видозмінимо клас `SimpleInheritance`, в якому для екземпляра класу C буде викликано усі змінні та методи класів B і A, а також розглянемо результат їх роботи:

```
class SimpleInheritance {

	public static void main(String[] args) {
		C subC = new C();

		subC.i = 1;
		subC.j = 2;
		subC.k = 3;
		subC.showij();
		subC.showk();
		subC.sum();
	}
}
```

Результат роботи програми:

```
1 2
3
Сума: 6
```

Як частково згадувалось раніше, не дивлячись на те, що підклас включає в себе усі члени суперкласу, він не може мати доступ до закритих членів (оголошених як **private**). Для прикладу розглянемо наступну ієрархію класів:

```
1 class A {
2     int i;
3     private int j;
4
5     void setij(int x, int y) {
6         i = x;
7         j = y;
8     }
9 }
```

```
1 class B extends A {
2     int total;
3
4     void sum() {
5         total = i + j; // помилка
6     }
7 }
```

```
1 class SimpleInheritance {
2
3     public static void main(String[] args) {
4         B subOb = new B();
5         subOb.setij(10, 12);
6         subOb.sum();
7         System.out.println(subOb.total);
8     }
```

```
9 }
```

Запустити цю програму не вдасться, тому що використання закритої змінної `j` з класу A в методі `sum()`, який розміщено в класі B, призводить до порушення правил доступу (інкапсуляції). Оскільки змінна `j` оголошена в класі A як **`private`** то вона доступна лише іншим члена безпосередньо класу A. Підкласи можуть мати до неї доступ лише через спеціально написані методи («геттери» та «сеттери»).

Розглянемо далі приклад, який допоможе краще продемонструвати можливості наслідування.

```
class Box {
      double width;
      double height;
      double depth;

      Box (Box ob){
            width = ob.width;
            height = ob.height;
            depth = ob.depth;
      }
      Box (double w, double h, double d){
            width = w;
            height = h;
            depth = d;
      }
      Box(){
            width = -1;
            height = -1;
            depth = -1;
      }
      Box (double len){
            width = height = depth = len;
      }

      double volume () {
            return width * height * depth;
      }
}

public class BoxWeight extends Box{
      double weight;

      BoxWeight(double w, double h, double d, double m) {
            width = w;
            height = h;
            depth = d;
            weight = m;
      }
}
```

```
public class BoxWeightDemo {

	public static void main(String[] args) {
		BoxWeight mybox = new BoxWeight(10, 20, 15, 34.3);
		System.out.println("Обєм складає: "+mybox.volume());
		System.out.println("Вага складає: "+mybox.weight);
	}
}
```

Результат роботи програми:

```
Обєм складає: 3000.0
Вага складає: 34.3
```

З наведеного прикладу наочно видно, що клас `BoxWeight`, який унаслідувано від класу `Box` містить лише одну змінну екземпляра класу `double weight`. Проте, незважаючи на це, конструктор класу дозволяє провести ініціалізацію змінних `width`, `height`, `depth` під час створення екземпляра класу в `main()`-методі. Це говорить про те, що клас `BoxWeight` унаслідував ці змінні (змінні, а не їх занчення) від свого суперкласу. Крім того для екземпляра класу `BoxWeight` також доступний метод `volume()`, який визначено в класі `Box`, а це ні що не інакше – як **наслідування**. Узагальнюючи можна констатувати, що клас `BoxWeight` наслідує усі характеристики класу `Box`, доповнюючи до них компонент `weight`.

Головна перевага наслідування полягає в тому, що суперклас, який визначає загальні характеристики, можливо буде використовувати для розроблення низки більш спеціалізованих підкласів.

Йдемо далі. Розглянемо складніший випадок. Об’єкту (екземпляру) суперкласу може бути присвоєне посилання на об’єкт (екземпляр) будь якого його підкласу. Опираючись на попередній приклад розглянемо зазначений випадок:

```
public class BoxWeightDemo {

	public static void main(String[] args) {
		BoxWeight weightbox = new BoxWeight(10, 20, 15, 34.3);
		Box plainbox = new Box(1, 2, 3);
		double vol;
		vol = weightbox.volume();
		System.out.println("Обєм складає: "+vol);
		System.out.println("Вага складає: "+weightbox.weight);

		plainbox = weightbox;
		vol = plainbox.volume();
		System.out.println("Обєм складає: "+vol);

		// в наступні стрічці помилка, оскільки об’єкт plainbox
не визначає змінну weight (в класі BoxWeight такої змінної не має)
```

```
            System.out.println("Вага складає: "+plainbox.weight);
        }
}
```

Результат роботи програми, якщо закоментувати останню стрічку:

```
Об’єм складає: 3000.0
Вага складає: 34.3
Об’єм складає: 3000.0
```

На початку наведеного прикладу об’єкт `weightbox` містить посилання на члени класу `BoxWeigh`, а об’єкт `plainbox` - на члени класу `Box`. Але оскільки клас `Box` є суперкласом для `BoxWeigh`, то його екземпляру `plainbox` можливо присвоїти посилання на об’єкт `weightbox`. Це означає, що якщо екземпляру суперкласу присвоїти посилання на екземпляр підкласу, то його доступ забезпечується до елементів (значень) підкласу, проте лише тих, які визначені в суперкласі. Саме тому у об’єкта `plainbox` не має доступу до змінної `weight`. І це пояснюється тим, що суперкласу не відомо, що саме міститься в його дочірньому підкласі. Екземпляр класу `Box` не має доступу до змінної `weight` тому, що ця змінна не оголошена в ньому. Саме тому остання стрічка коду в наведеному прикладі буде видавати помилку.

Ще один цікавий факт з наведеного прикладу. При створенні екземпляра класу `Box` з іменем `plainbox` в конструкторі ініціалізовані три змінні. Проте значення цих змінних не приймались до уваги під час виконання програмою методу `volume()`. Це наочно видно з результатів роботи програми, а все тому, що перед викликом методу `volume()` до об’єкта `plainbox`, самому об’єкту присвоєно посилання на інший об’єкт – `weightbox`, і саме тому розрахунок об’єму «коробки» для обох екземплярів класів буде рівним.

3. Ключове слово super

В попередніх прикладах наслідування, дочірні класи були реалізовані не зовсім ефективно та надійно. Наприклад, конструктор `BoxWeight()` явно ініціалізує поля `width`, `height`, `depth` з класу `Box`. Це веде не лише до дублювання коду суперкласу, що не є ефективно, але і передбачає наявність у підкласа доступу до цих членів суперкласу (можливість перезапису значень).

Інколи виникає необхідність створити суперклас із закритими змінними (типу `private`). В такому випадку дочірній клас не у змозі звертатись безпосередньо до змінних батьківського класу та ініціалізувати їх при створенні власного екземпляру класу. Як відомо, інкапсуляція – це один з основаних принципів ООП, саме тому в Java запропоновано спосіб вирішення такої проблеми. У всіх випадках, коли дочірній клас має мати посилання на свій батьківський клас із можливістю ініціалізації змінних, це можна реалізувати за допомогою ключового слова `super`.

Ключове слово `super` має дві форми застосування. Перша слугує для виклику конструктора суперкласу, друга – для звернення до закритої змінної суперкласу. Розглянемо ці форми більш детальніше.

З підкласу можливо викликати конструктор, який визначено в його суперкласі, використовуючи таку форму ключового слова **super**:

```
super(список аргументів);
```

де `список аргументів` визначає будь які аргументи, які потрібно дістати із суперкласу. Виклик метода **super()** має передувати в конструкторі дочірнього класу (бути першим оператором). Розглянемо приклад застосування метода **super()** шляхом удосконалення попередньо написаного класу `BoxWeight()`:

```
public class BoxWeight extends Box {
      double weight;

      BoxWeight(double w, double h, double d, double m) {
            super(w, h, d);
            weight = m;
      }
}
```

Тепер в класі `BoxWeight()` ключове слово **super** використовується для ініціалізації власних властивостей об'єкту типу `Box()`.

В наведеному прикладі метод **super()** викликається з аргументами `w, h, d` у конструкторі `BoxWeight()`. Це призводить до виклику конструктора суперкласу `Box()`, у якому вже визначено спосіб ініціалізації змінних `width`, `height`, `depth`. За такого представлення програмного коду не потрібно дублювати ініціалізацію цих змінних безпосередньо в класі `BoxWeight()`. В ньому залишається необхідність визначити спосіб ініціалізації лише власної змінної `weight`. У результаті, змінні екземпляра суперкласу `Box`, за необхідності, можуть залишатись закритими.

В наведеному прикладі метод **super()** викликається із трьома аргументами. Проте конструктори можуть бути перевизначеними, тому метод **super()** можливо викликати використовуючи будь яку форму визначену в суперкласі. Виконуватись буде той конструктор, який відповідає зазначеним аргументам. В наступному прикладі розглянемо повну реалізацію класу `BoxWeight`, у якому перевизначено конструктори для різних способів ініціалізації. У кожному випадку метод **super()** буде викликано із відповідними аргументами. Додатково слід звернути увагу, що змінні `width`, `height`, `depth` оголошені як **private**.

```
class Box {
      private double width;
      private double height;
      private double depth;

      Box (Box ob){
            width = ob.width;
            height = ob.height;
```

```
            depth = ob.depth;
        }

        Box (double w, double h, double d){
            width = w;
            height = h;
            depth = d;
        }

        Box(){
            width = -1;
            height = -1;
            depth = -1;
        }

        Box (double len){
            width = height = depth = len;
        }

        double volume () {
            return width * height * depth;
        }
}

public class BoxWeight extends Box {
        double weight;

        // створення клону об'єкта
        BoxWeight(BoxWeight ob) { // передаємо об'єкт конструктору
            super(ob);
            weight = ob.weight;
        }

        // констуктор, що застосовується при ініціалізації усіх
параметрів
        BoxWeight(double w, double h, double d, double m) {
            super(w, h, d); // викликаємо констуктор із суперкласу
            weight = m;
        }

        // конструктор, що затосовується за замовчуванням
        BoxWeight() {
            super();
            weight = -1;
        }

        // констуктор, що застосовується при створенні куба
        BoxWeight(double len, double m) {
            super(len);
            weight = m;
        }
```

```
}

public class BoxWeightDemo {

      public static void main(String[] args) {
            BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
            BoxWeight mybox2 = new BoxWeight();
            BoxWeight mycube = new BoxWeight(3, 2);
            BoxWeight myclone = new BoxWeight(mybox1);

            //демонстрація роботи методу volume()
            //викликаного до різних екземплярів класу

            System.out.println(mybox1.volume());
            System.out.println(mybox2.volume());
            System.out.println(mycube.volume());
            System.out.println(myclone.volume());
      }
}
```

Результат роботи програми:

```
3000.0
-1.0
27.0
3000.0
```

Узагальнюючи вище викладене, розглянемо основні принципи, що закладені в основу метода `super()`. Коли метод `super()` викликається із дочірнього класу, викликається конструктор його безпосереднього батьківського класу. Таким чином, метод `super()` завжди звертається до свого безпосереднього суперкласу. Така властивість дотримується навіть за умови багаторівневої ієрархії класів. Крім того важливо пам'ятати, що виклик метода `super()` має завжди бути першим оператором, що виконується в тілі конструктора.

Далі перейдемо до розгляду другої форми застосування ключового слова `super`. Друга форма діє подібно до ключового слова `this`, за виключенням того, що посилання завжди відбувається на батьківський клас того дочірнього класу, в якому виконано цей оператор. Загальна форма застосування ключового слова `super` виглядає так:

```
super.змінна_метод;
```

де `змінна_метод` може бути як методом так і змінною екземпляра класу.

Така форма широко застосовується у тих випадках, коли імена змінних дочірнього класу перекривають змінні батьківського класу з такими ж іменами. Розглянемо приклад:

```
class A {
```

```
        int i;
    }

    class B extends A {
        int i; //ця змінна перекрипає змінну i з класу A

        B (int a, int b){
            super.i = a; //змінна i з класу A
            i = b; //змінна i з класу B
        }

        void show () {
            System.out.println(super.i);
            System.out.println(i);
        }
    }

    public class UseSuper {

        public static void main(String[] args) {
            B subOb = new B(1, 2);
            subOb.show();
        }
    }
```

Хоча змінна `i` з класу `B` перекриває змінну `i` з класу `A`, ключове слово `super` надає можливість звертатись безпосередньо до змінної батьківського класу. Щодо виклику методів за допомогою ключового слова `super`, то це тема наступних лекцій.

Висновок: Наслідування можна вважати однією із основних парадигм об'єктно-орієнтованого програмування, адже саме наслідування передбачає можливість створювати ієрархію класів надаючи дочірнім класам властивості своїх батьківських класів не нагромаджуючи та не дублюючи програмного коду. Парадигма наслідування є невід'ємним інструментом для написання програм за об'єктно-орієнованим принципом. На сьогодні практично не існує програмних систем, які б не містили у собі принципи наслідування класів. Крім того парадигма наслідування є обов'язковим питанням на усіх співбесідах із працевлаштування на посаду developer. Відповідно знання базових принцип ієрархії класів, наслідування та особливостей застосування ключового слова super є невід'ємними елементами у процесі формування фахових компетенцій із розробки програмного забезпечення.

Завдання на самопідготовку:

1. Опрацювати конспект лекції та підготуватись до складання тестових завдань.
2. Java 8. Полное руководство. 9-е изд.: пер. с англ. / Герберт Шилдт. – Москва : ООО "И.Д. Вильяме", 2015. – 1376 с. **(стор. 209-219)** ***(наявне в віртуальному середовищі)***

3. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с. **(стор. 198-222)** ***(наявне в віртуальному середовищі)***

Додатково:

4. Освоюємо Java. Вікіпідручник [Електронний ресурс]. – Доступний з https://uk.wikibooks.org/wiki/Освоюємо_Java **(розділ 9)** ***(наявне в віртуальному середовищі)***

5. Java. ProgLang. Самоучитель (рос.) [Електронний ресурс]. – Доступний з http://proglang.su/java **(розділ 22) (*наявне в віртуальному середовищі)***