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 №6.7. ІНКАПСУЛЯЦІЯ. УПРАВЛІННЯ ДОСТУПОМ

**План:**

Вступ. Введення в інкапсуляцію
1. Управління доступом
2. Ключові слова static та final

**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 інкапсуляцію

Механізм, що зв'язує код та дані, якими він маніпулює, захищаючи ці два компоненти від зовнішнього втручання називається **інкапсуляцією**. Інкапсуляцію можна рахувати захисною оболонкою, яка запобігає хаотичному доступу до програмного коду та даних зі сторони іншого коду, що знаходиться за межами цієї оболонки. Доступ до програмного коду та даних в середині оболонки суворо контролюється визначеним інтерфейсом. Для того, щоб провести аналогію з реальним світом розглянемо скриньку перемикання передач автомобіля. Вона інкапсулює чимало відомостей про автомобіль. Користувач (в даному випадку – водій) може здійснювати вплив на цю інкапсульовану систему лише одним способом – перемикати важіль швидкостей. На скриньку не можна подіяти, наприклад, за допомогою керма або двірників. Таким чином, важіль перемикання передач являється суворо

визначеним єдиним інтерфейсом для роботи із скринькою передач. Крім того внутрішній вміст скриньки не доступний для користувача. Водієві не має необхідності для приведення в рух автомобіля здійснювати вплив на внутрішні елементи скриньки, для цього існує важіль перемикання (інтерфейс). Також внутрішній вміст скриньки не впливає на об'єкти, що знаходяться за її межами (перемикання передач не вмикає фари). Функція перемикання передач інкапсульована, саме тому будь-який виробник автомобілів може реалізувати її як завгодно. Але з точки зору водія всі скриньки перемикання передач працюють однаково. Аналогічний принцип можна застосовувати і в програмуванні. Сильна сторона інкапсульованого коду полягає в тому, що всім відомо як можна отримати доступ до нього та його використовувати незалежно від особливостей його внутрішньої реалізації.

Оскільки класи містять в собі структуру програми існують механізми її закриття від впливу інших класів. Кожен метод або змінна в класі можуть бути помічені як закриті або відкриті. Відкритий інтерфейс класу надає доступ до своїх методів та змінних з будь-якого іншого місця програми (класу). Закритий інтерфейс – навпаки, надає доступ для компонентів лише в середині класу де визначені закриті методи або змінні (членів того самого класу). Закриті елементи класу можуть бути доступними іншим частинам програми тільки через спеціально написані відкриті методи. За рахунок такого доступу виключається можливість виконання неправомірних або помилкових дій, що можуть зашкодити кодові програми. Відкритий інтерфейс має бути ретельно спроектований та не розкривати лишніх подробиць внутрішнього механізму роботи класу.

## 1. Управління доступом

Інкапсуляція дозволяє керувати доступом до членів класу з різних частин програми, а відповідно і запобігати несанкціонованій зміні програмного коду. Якщо клас реалізований вірно, він створює прототип «чорної скриньки», якою можливо користуватись, проте неможливо переглянути її вміст (внутрішній механізм «чорної скриньки» захищений від пошкоджень).

Доступ до членів класу реалізується за допомогою ***модифікаторів доступу***. Модифікатор доступу **завжди** зазначають при оголошенні членів класу (змінних та методів). В Java визначені такі модифікатори доступу: `public` (відкритий), `private` (закритий), `protected` (захищений), а також рівень доступу за замовчуванням (`package`). Модифікатор `protected` використовується лише у випадках наслідування. Решту модифікаторів розглянемо далі.

Коли змінна або будь який інший клен класу оголошується із модифікатором `public` він стає доступним із будь якого місця програми (з інших класів та пакетів). А коли член класу оголошено із модифікатором `private`, доступ до нього забезпечено лише в межах цього класу. Тепер зрозуміло, чому в оголошенні головного метода `main()` завжди присутній модифікатор `public`. Цей метод викликається з коду, що знаходиться за межами програми, а саме з виконавчої системи Java (JVM). У випадках коли

модифікатор доступу не вказано (модифікатор доступу – за замовчуванням), член класу вважається доступним в межах пакету. Модифікатор за замовчуванням як правило не зазначається, проте якщо Вам зустрінеться код із модифікатором **package**, то це той самий модифікатор (його можна вказувати, а можна й не вказувати).

Коротко оглянувши основні модифікатори доступу може виникнути питання: який модифікатор застосовують найчастіше? Здебільшого, доступ до членів класу має бути закритим, надаючи до них доступ лише через спеціально підготовлені методи, а це, відповідно, потребує застосування модифікатора **private**. Крім того в низці випадків в класі з'являтиметься необхідність оголошувати закриті методи. Отже модифікатори можуть застосовуватись як для змінних екземплярів класів так і для їх методів.

Оператор оголошення члена класу має починатись з модифікатора доступу, як представлено на прикладі:

```
public int i;
private double j;

private int myMethod(int a, char b) {
      }
```

Для кращої уяви про організацію відкритого та закритого доступу в Java (відмінність модифікаторів **public** та **private**) розглянемо наступний приклад:

```
class Test {
      int a; // модифікатор за замовчуванням
      public int b; // відкритий доступ
      private int c; // закритий доступ

      void setc(int i) { // встановити значення члена c
            c = i;
      }

      int getc() { // отримати значення члена c
            return c;
      }
}


public class AccessTest {

      public static void main(String[] args) {
            Test ob = new Test();

            ob.a = 10;
            ob.b = 20;

            ob.c = 30; // помилка
```

```
            ob.setc(30);
            System.out.println(ob.getc());
        }
}
```

Як видно з наведеного прикладу в класі `Test` три змінні екземплярів класу оголошенні із різними модифікаторами доступу. Так як змінні `int a` та `int b` оголошенні з модифікаторами за замовчуванням та модифікатором `public` відповідно, то доступ до них забезпечений з іншого класу `AccessTest`. Змінна `int a` буде доступна з будь якого класу в межах одного пакету, а змінна `int b` – узагалі з будь якого місця Java-проекту.

Інша ситуація із змінною `int c`, яка закрита модифікатором `private`. При спробі задати їй значення з іншого класу компілятор видаватиме помилку, адже з-за меж класу `Test` цієї змінної видно не буде. Для доступу до закритих членів використовують спеціально підготовлені методи. В даному випадку метод `setc(int i)` використовують для ініціалізації змінної (встановлення значення), а `getc()` – для її отримання. Слід зауважити, що зазначені методи в класі `Test` оголошені із модифікатором доступу за замовчування, що надає можливість доступатись до них з інших класів в межах пакету. Подібного роду методи, які надають можливість встановити та отримати значення закритих змінних екземплярів класу із-за меж цього класу мають назву «сеттери» та «геттери». Звичайно для забезпечення доступу ці методи мають бути написані в тому ж класі, де оголошено закриті змінні. Багато середовищ розробки передбачають можливість автоматичного генерування методів `get()` та `set()`, не виключенням стало IDE Eclipse. Для цього необхідно скористатись комбінацією Shift+alt+s та обрати із контекстного меню Generate Getters and Setters …

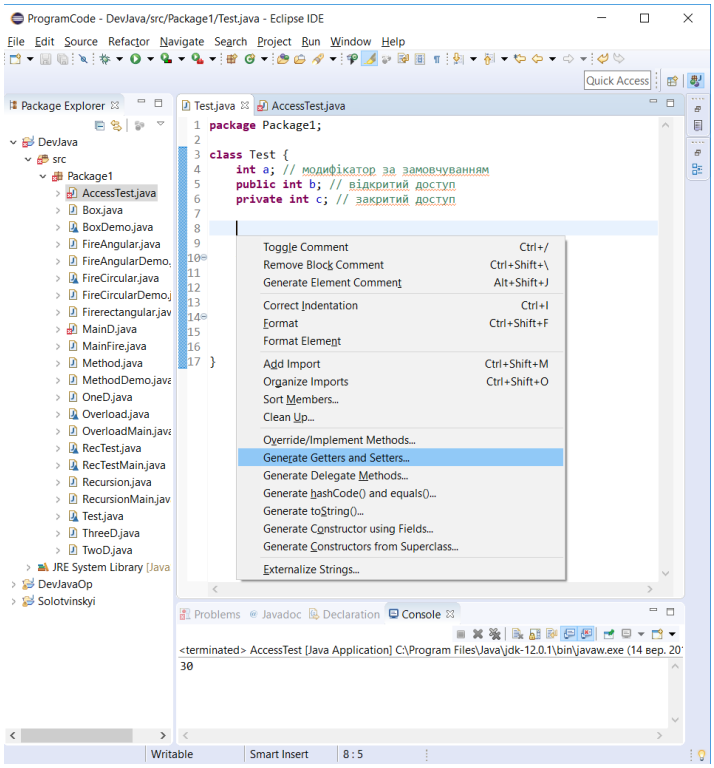


Рисунок 1 – Автоматична генерація методів `get()` та `set()`

Подібна операція також доступна через меню Source. Після виконання вказаних операцій у діалоговому вікні, яке з'явиться на екрані, користувач може обрати змінні, до яких необхідно генерувати методи, визначити місце в структурі коду де ці методи буде розміщено, визначити модифікатор доступу для методів та багато іншого.

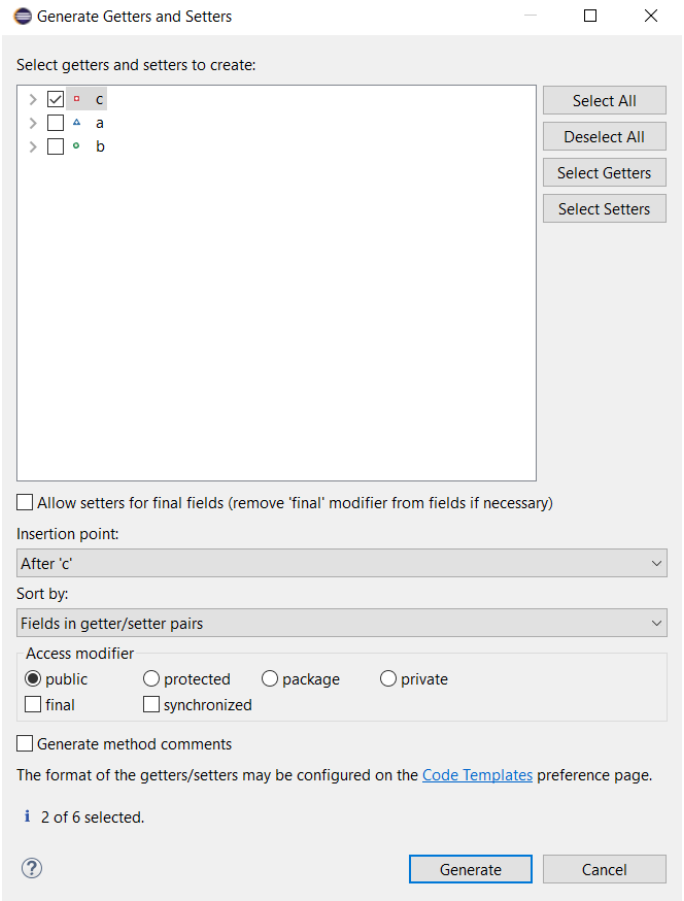


Рисунок 2 – Налаштування методів `get()` та `set()`

Для формування повноти уяви про управління доступом розглянемо реальний приклад, де доступ до закритих членів класу організовано за допомогою методів із унікальною логікою.

```
class Stack {
// оголошуємо стек на 10 елементів
private int stck[] = new int[10];
private int tos; // оголошуємо вершину стеку

// ініціалізуємо вершину стеку
Stack() {
      tos = -1;
}
// додати елемент до стеку
void push(int item) {
      if (tos == 9)
            System.out.println("Стек заповнено");
      else
```

```
            stck[++tos] = item;
    }
    // отримати елемент зі стеку
    int pop() {
        if (tos < 0) {
            System.out.println("Стек не заповнено");
            return 0;
        } else
            return stck[tos--];
    }
}
```

В наведеному прикладі оголошено стек на 10 елементів та вершину цього стеку із модифікатором доступу **private**. Для додавання та отримання елементів із стеку в класі написано методи `push()` та `pop()` відповідно. За такої реалізації класу доступ до стеку із-за його меж може бути виконаний лише за допомогою методів `push()` та `pop()`. Наприклад, оголошення змінної **int tos** закритою попереджує випадковому встановленню її значення за межами масиву `stck[]` з інших частин програми. Розглянемо дієвість такого підходу в класі `TestStack` із `main()`–методом.

```
public class TestStack {

    public static void main(String[] args) {
        Stack mystack1 = new Stack ();

        //додаємо числа до стеку
        for(int i = 0; i<10; i++) mystack1.push(i);

        //отримуємо числа зі стеку
        for(int i = 0; i<10; i++)
            System.out.println(mystack1.pop());

        //встановлюємо вершину стеку
        mystack1.tos = - 2; // помилка

        // додаємо число в стек
        mystack1.stck[3] = 100; //помилка
    }
}
```

Як видно з наведеного прикладу прямий доступ до вершини стеку, або запис числа в комірку з індексом 3 є неможливим, адже із класу `TestStack` доступ як до самого стеку, так і до його вершини закритий. Єдиним можливим доступом до закритих членів класу `Stack` є застосування завчасно написаних методів `push()` та `pop()` (ці методи заміняють так звані «геттери» і «сеттери»).

## 2. Ключові слова static та final

Як правило, звернення до члена класу має реалізовуватись лише в поєднанні з об'єктом цього класу (через оператор виклику «.»). Проте в об'єктно-орієнтованому програмуванні є виключення, які дозволяють створювати члени класу та користуватись ними автономно, без прив'язки до екземпляра класу (об'єкту). Щоб створити такий член на початку його оголошення необхідно розмістити ключове слово **static**. Відповідно, коли член класу оголошено статичним він стає доступним без посилання на його екземпляр. Статичними можуть бути оголошені як змінні екземплярів класу так і методи. Ще одна особливість статичних членів закладена в їх назві. Якщо змінну або метод оголошено як **static** то це означає, що вони є незмінними (статичними) у всій структурі коду, де використовуються. Найбільш розповсюдженим прикладом статичного методу є `main()`–метод, оскільки він є доступним і незмінним з будь якої частини програмного коду.

Змінні екземплярів класу, які оголошені як **static**, по суті, являються також глобальними. При оголошені екземпляра класу (класу в якому розміщені статичні змінні), копії змінних НЕ створюються. Всі об'єкти такого класу спільно використовують одну й ту саму статичну змінну.

На методи, які оголошені як **static**, накладаються такі обмеження:

- можуть викликати лише інші статичні методи;
- їм доступні лише статичні змінні;
- не можуть містити посилання типу **this** або **super**.

Якщо для ініціалізації (присвоєння значення) статичних змінних необхідно застосувати арифметичний вираз, то для цієї цілі достатньо оголосити статичний блок ({…}), який буде виконуватись лише один раз при першому завантаженні класу. Для наочності в наведеному прикладі розглянемо клас, який містить статичний метод, статичні змінні та статичний блок ініціалізації.

```
class UseStatic {
      static int a = 3;
      static int b;

      static void meth(int x) {
            System.out.println(x);
            System.out.println(a);
            System.out.println(b);
      }

      //Статичтий блок ініціалізації
      static {
            b = a * 4;
      }

      public static void main(String[] args) {
            meth(42);
```

```
        }
}
```

При завантажені класу `UseStatic` виконуються всі статичні оператори. Спочатку в змінну `static int a` встановлюється значення 3, потім статичний блок коду, в якому змінна `static int b` ініціалізується значенням `a * 4`, тобто 12. Після цього викликається `main()`–метод, який соєю чергою викликає метод *`meth`*`()`, передаючи параметру значення 42. В трьох викликах методів `println()` створюються посилання на дві статичні змінні *`a`* і *`b`*, а також локальну змінну `x`.

Результат роботи програми матиме вигляд:

```
42
3
12
```

За межами класу, в якому оголошені статичні методи та змінні, ними можна користуватись без звернення до екземпляра цього класу. Достатньо вказати ім’я класу, де вони розміщені, та через оператор виклику «.» звернутись до статичних членів за їх іменами:

```
Ім’я_класу.статичний_метод();
```

де `Ім’я_класу` – це клас, де оголошено статичний *`метод()`*, що викликається. На відміну від локальних, для виклику статичних (глобальних) методів НЕ потрібно створювати об’єкт та здійснювати виклик методу за його назвою. В статичних методах достатньо звертатись безпосередньо до назви самого класу. За тією ж схемою проводиться виклик статичних змінних. Саме так в Java реалізовано керування глобальними методами та змінними.

Для наочності розглянемо приклад, де в тілі методу `main()` розміщено звернення до статичного методу `callme()` та статичної змінної `int b` за іменем класу `StaticDemo`, БЕЗ оголошення та звернення до екземпляра цього класу:

```
public class StaticDemo {
      static int a = 42;
      static int b = 99;

      static void callme() {
            System.out.println(a);
      }
}

public class StaticByName {
      public static void main(String[] args) {
            StaticDemo.callme();
            System.out.println(StaticDemo.b);
      }
}
```

Результат роботи програми:

```
42
99
```

Поряд із статичними на практиці часто оголошують змінні як **final** (завершені). Такий застосунок дозволяє попередити зміну значення змінної, зробивши її константою. Відповідно змінна **final** має ініціалізовуватись під час її оголошення. Значення такій змінній можна задати і в межах конструктора, проте перший підхід найбільш розповсюджений:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Тепер в решті частин програми можливо використовувати зазначені змінні як константи, без будь якого ризику випадково змінити їх значення. В практиці об’єктно-орієнтованого програмування на Java ідентифікатори усіх завершених змінних екземплярів класів (полів) прийнято зазначати ПРОПИСНИМИ літерами.

Окрім змінних екземплярів калів, як **final** можуть оголошуватись параметри методу і локальні змінні. Оголошення параметра як **final** попереджує його зміну в межах методу, а локальної змінної – переприсвоєння їй значення більше одного разу.

**Висновок:** Як відомо найбільшої популярності за останні роки набирають саме об’єктно-орієнтовані мови програмування, які підтримують основні принципи цієї парадигми. Що стосується структури програмного коду та забезпечення доступу до членів класу з інших частин програми, то принцип інкапсуляції є дієвим інструментарієм який широко застосовується в сучасних мовах програмування. Інкапсуляція, як одна із парадигм ОПП дозволяє зберігати безпеку програмного коду від зовнішніх втручань, надаючи доступ лише до використання логіки програми без можливості її зміни. Знання основних принципів інкапсуляції надає можливість створювати безпечний та високоефективний за структурою програмний код.

**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. 189-195)** ***(наявне в віртуальному середовищі)***
3. Head First Java (изучаем Java): пер. с англ. / Kathy Sierra, Bert Bates. – Москва : «Эксмо», 2012. – 718 с. **(стор. 110-113)** ***(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 8) (*наявне в віртуальному середовищі)***