ІНКАПСУЛЯЦІЯ ТА ПРИХОВУВАННЯ ІНФОРМАЦІЇ В JAVA

Питання 2.4.

Представлення поведінки за допомогою методів

Java представляє поведінку за допомогою методів.

- Це іменовані блоками коду, оголошені всередині тіла класу.
- Поведінка, що пов’язана з класом, описується методами класу (class methods),
- Поведінка, асоційована з об’єктами, описується за допомогою методів екземпляру (object methods, instance methods)
- За домовленістю ім’я методу починається з малої літери, а перша буква кожного слова у багатослівній назві методу буде великою.

Оголошення методу класу

- Метод класу містить поведінку, що пов’язана з класом.
  - Всі об’єкти, що створюються з цього класу, мають відповідні методи.
  - Метод класу не має прямого доступу до полів екземпляру.
  - Єдиний спосіб отримання такого доступу – в контексті конкретного об'єкта.
  - Метод класу має наступний синтаксис:

```
static return_type name(parameter_list)
{
   // statements to execute
}
```

- Назва методу та кількість, типи та порядок параметрів також називають ***сигнатурою***.
  - Як і з конструктором, заголовок методу класу містить список параметрів, який дозволяє задавати дані, що будуть передаватись у метод для обробки.
  - Після заголовку у фігурних дужках слідує тіло методу, яке складається з набору інструкцій, які виконуються, коли метод класу викликається.

Приклад: клас Utilities та його метод dumpMatrix()

```
public class Utilities
{
   static void dumpMatrix(float[][] matrix)
   {
      for (int row = 0; row < matrix.length; row++)
      {
         for (int col = 0; col < matrix[row].length; col++)
            System.out.print(matrix[row][col] + " ");
         System.out.print("\n");
      }
   }

   public static void main(String[] args)
   {
      float[][] temperatures = {
                                  { 37.0f, 14.0f, -22.0f },
                                  { 0.0f, 29.0f, -5.0f }
                               };
      dumpMatrix(temperatures);
      System.out.println();
      Utilities.dumpMatrix(temperatures);
   }
}
```

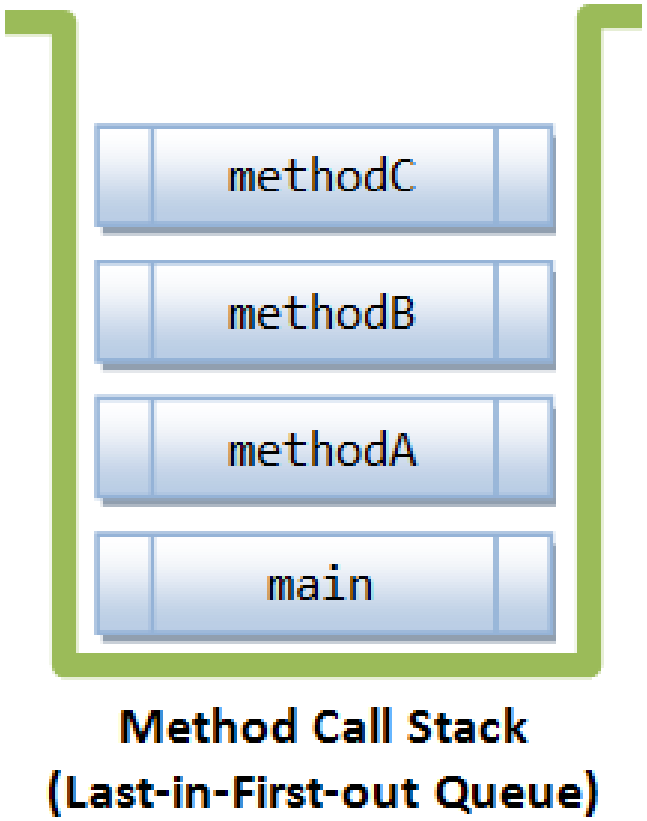
- Метод класу dumpMatrix() скидує (dump) вміст 2D-масиву в табульованому форматі в стандартний потік виводу (standard output stream).
  - Метод не повертає нічого (void), називається dumpMatrix та містить список параметрів, який складається з одного параметру з назвою matrix, який має тип float[][].

```
37.0 14.0 -22.0
0.0 29.0 -5.0

37.0 14.0 -22.0
0.0 29.0 -5.0
```

Стек виклику методів (Method-call stack)

- Виклики методів вимагають стеку виклику методів (method-call stack, method-invocation stack) для відстежування тих інструкцій, до яких повинно повертатись виконання.
  - Коли метод викликається, віртуальна машина додає (push) його аргументи та адресу першої інструкції для виконання following the invoked method у стек виклику методів.
  - Також віртуальна машина виділяє місце в стеку для локальних змінних методу.
  - Коли метод повертає значення, віртуальна машина очищає пам'ять з локальним змінними, видаляє (pop) адресу та аргументи методу зі стеку та перенаправляє процес виконання до інструкції за цією адресою.



Оголошення та виклик методів екземпляру

- *Метод екземпляру* містить поведінку, яка асоціюється з об'єктом.
  - На відміну від методу класу, instance метод напряму має доступ до полів екземпляра.
- Синтаксис методу екземпляра:

```
return_type name(parameter_list)
{
    // statements to execute
}
```

- Крім відсутності зарезервованого слова static, синтаксис такий же, як і в методу класу.

```
class Car
{
   String make;
   String model;
   int numDoors;
   Car(String make, String model)
   {
      this(make, model, 4);
   }

   Car(String make, String model, int nDoors)
   {
      this.make = make;
      this.model = model;
      numDoors = nDoors;
   }

    void printDetails()
    {
       System.out.println("Make = " + make);
       System.out.println("Model = " + model);
       System.out.println("Number of doors = " + numDoors);
       System.out.println();
    }
 }
public class Cars
{
   public static void main(String[] args)
   {
      Car myCar = new Car("Toyota", "Camry");
      myCar.printDetails();
      Car yourCar = new Car("Mazda", "RX-8", 2);
      yourCar.printDetails();
   }
}
```

Рефакторинг класу Car

- Метод printDetails() не повертає значень, а лише виводить виробника, модель, кількість дверей автомобіля та порожній рядок.
- Метод main() створює екземпляр класу (instantiate) Car, передаючи підходящі рядки тексту для make та model з конструктора з двома параметрами.
  - Кількість дверей за замовчуванням прирівнюється 4.
  - Потім викликається метод printDetails() для отриманого посилання на об'єкт (object reference) для виведення цих даних.

Далі створюється другий об'єкт Car за допомогою конструктора з трьома параметрами та виводить дані.

- Виклик у методі main()'s методу printDetails() демонструє, що метод екземпляру завжди викликається в контексті посилання на об'єкт.

Повернення з методу за допомогою оператора return

```
public class Employee
{
   String name;

   Employee(String name)
   {
      setName(name);
   }
   void setName(String name)
   {
      if (name == null)
      {
         System.out.println("name cannot be null");
         return;
      }
      else
         this.name = name;
   }

   public static void main(String[] args)
   {
      Employee john = new Employee(null);
   }
}
```

- Оператор return припиняє виконання методу та передає управління назад до того, хто викликав метод (*caller*).

Конструктор Employee(String name) викликає метод екземпляру setName() для ініціалізації поля екземпляру name.

Створення окремого методу – хороша ідея, оскільки дозволяє ініціалізувати поле екземпляру не лише під час конструювання, але й пізніше.

Метод setName() використовує оператор if для відстежування спроб не заповнювати ім’я.

Проблема unreachable code

- Компілятор повідомляє про проблему «unreachable code», коли знаходить код, що ніколи не буде виконуватись та марно займає пам'ять.
- Зустрітись з цією проблемою можна при роботі з оператором switch.
  - Наприклад, Ви задали як частину тіла оператору

```
case 2: printUsageInstructions();
return;
break;
```

  - Компілятор повідомить про помилку, оскільки оператор break, який знаходиться після return стає недосяжним, тобто ніколи не буде виконуватись.

Повернення значень

```
static double divide(double dividend, double divisor)
{
   if (divisor == 0.0)
   {
      System.out.println("cannot divide by zero");
      return 0.0;
   }
   return dividend / divisor;
}
```

- Метод використовує оператор if для відстежування спроби ділення на 0.0 та виводить повідомлення про помилку, коли це стається.

Ви не можете використовувати таку форму оператора в конструкторі, оскільки конструктори не мають типу даних для повернення (return types).

Ланцюговий виклик методів екземпляру

```
public class SavingsAccount
{
   int balance;

   SavingsAccount deposit(int amount)
   {
      balance += amount;
      return this;
   }

   SavingsAccount printBalance()
   {
      System.out.println(balance);
      return this;
   }

   public static void main(String[] args)
   {
      new SavingsAccount().deposit(1000).printBalance();
   }
}
```

```
1000
```

- Двоє або більше викликів методів екземпляру можна об'єднати в ланцюг за допомогою оператора доступу «.»
- Для цього оголошується тип повернення SavingsAccount для методів deposit() та printBalance().
  - Також кожен метод задає return this; (повернути посилання на поточний об'єкт).
- У методі main() створюється об'єкт типу SavingsAccount і відбувається ланцюговий виклик

Передача аргументів у методи

- Виклик методу включає список аргументів (0 або більше), які передаються в метод.
- Java передає аргументи ***за значенням (pass-by-value)***:
  - Employee emp = new Employee("John ");
  - int recommendedAnnualSalaryIncrease = 1000;
  - printReport(emp, recommendAnnualSalaryIncrease);
  - printReport(new Employee("Cuifen"), 1500);
- Передача за значенням (Pass-by-value) передає
  - значення змінної (наприклад, посилальне значення, що зберігається в emp, або число 1000, що міститься в recommendedAnnualSalaryIncrease)
  - значення деякого іншого виразу (як для new Employee("Cuifen") або 1500) в метод.
- Ви не можете присвоїти інше посилання об'єкту типу Employee для emp зсередини printReport() за допомогою параметру printReport() для цього аргументу.
  - Ви маєте лише передану в метод копію значення emp.

Методи зі змінною кількістю параметрів

- Java дає можливість передавати змінну кількість аргументів за допомогою *varargs-методів/конструкторів*.

  Прописується «...» після назви типу останнього (rightmost) параметру конструктора або методу.

```
static double sum(double... values)
{
   int total = 0;
   for (int i = 0; i < values.length; i++)
      total += values[i];
   return total;
}
```

Рекурсивний виклик методів

- Зазвичай метод виконує інструкції, що можуть містити виклики інших методів.
- printDetails(), що викликає System.out.println().
- Сценарій, коли метод викликає сам себе, називають рекурсією.
  - Класичний приклад – знаходження факторіалу.

```
static int factorial(int n)
{
   int product = 1;
   for (int i = 2; i <= n; i++)
      product *= i;
   return product;
}
```

```
static int factorial(int n)
{
   if (n == 1)
      return 1; // base problem
   else
      return n * factorial(n - 1);
}
```

Стек викликів для факторіалу
4 * factorial(3)
3 * factorial(2)
2 * factorial(1)

Особливості рекурсії

- Рекурсія забезпечує елегантне вираження багатьох задач.
  - Може застосовуватись для пошуку конкретних значень у tree-based структурах даних та, в ієрархічних файлових системах, для знаходження та виведення назв усіх файлів, що містять конкретний текст.
- **Обережно!** Рекурсія «з'їдає» багато пам'яті стеку, тому переконайтесь, щоб рекурсія закінчувалась на базовій задачі (base problem);
  - Інакше стекова пам'ять закінчиться, а Ваш застосунок буде змушений перервати роботу.

Перевантаження (Overloading) методів

- Java дозволяє створювати методи з однаковою назвою, проте різним списком аргументів в одному класі.
  - Цю можливість називають ***перевантаженням методу***.
  - Компілятор порівнює список аргументів викликаного методу з кожним списком параметрів перевантажених методів та розшукує коректне співпадіння.
  - Два методи з однаковими іменами будуть перевантаженими, коли
    - їх список параметрів відрізняється їх кількістю або порядком.
    - принаймні один параметр відрізняється своїм типом.
- Ви НЕ можете перевантажити метод, змінивши лише тип повернення методу.
  - Причина: компілятор не має достатньо інформації для вибору того, який метод викликати при появі sum(1.0, 2.0) у вихідному коді.

```
public class MO
{
   int add(int a, int b)
   {
      System.out.println("add(int, int) called");
      return a + b;
   }

   int add(int a, int b, int c)
   {
      System.out.println("add(int, int, int) called");
      return a + b + c;
   }

   double add(double a, double b)
   {
      System.out.println("add(double, double) called");
      return a + b;
   }

   public static void main(String[] args)
   {
      MO mo = new MO();
      int result = mo.add(10, 20);
      System.out.println("Result = " + result);
      result = mo.add(10, 20, 30);
      System.out.println("Result = " + result);
      double result2 = mo.add(5.0, 8.0);
      System.out.println("Result2 = " + result2);
   }
}
```

Додаток з перевантаженням

```
add(int, int) called
Result = 30
add(int, int, int) called
Result = 60
add(double, double) called
Result2 = 13.0
```

Виклик методів з різних контекстів рекомендується проводити так:

- Задавайте назву методу класу «як є» всередині відповідного класу.
  - Наприклад, dumpMatrix(temperatures);
- Дописуйте назву класу цього методу та оператор «.» ззовні класу.
  - Наприклад, Utilities.dumpMatrix(temperatures);
- Задавайте назву методу екземпляру «як є» в будь-якому методі екземпляру, конструкторі або ініціалізаторі в межах відповідного класу.
  - Наприклад, setName(name);
- Дописуйте посилання на об’єкт, якому належить метод, та оператор «.»,
  - в будь-якому методі класу
  - ініціалізаторі класу в межах того ж класу, що і метод екземпляру
  - ззовні класу.
  - Наприклад,

```
Car car = new Car("Toyota", "Camry");
car.printDetails();
```

Інкапсуляція та приховування даних

```
class Car
{
   String make;
   String model;
   int numDoors;
   Car(String make, String model)
   {
      this(make, model, 4);
   }

   Car(String make, String model, int nDoors)
   {
      this.make = make;
      this.model = model;
      numDoors = nDoors;
   }
   void printDetails()
   {
      System.out.println("Make = " + make);
      System.out.println("Model = " + model);
      System.out.println("Number of doors = " + numDoors);
      System.out.println();
   }
}
```

- Кожен клас X показує інтерфейс: конструктори, методи та, можливо, поля, до яких можна отримати доступ ззовні класу.
  - Доступ до поля model класу Car можна отримати ззовні.
  - Метод printDetails() теж можна викликати, тому він є частиною інтерфейсу.
- Інтерфейс слугує для контакту між класами та їх ***клієнтами.***
  - Зовнішніми класами, які звертаються до полів (зазвичай public static final) та викликають конструктори й методи.

Інкапсуляція та приховування даних

```
class Car
{
   String make;
   String model;
   int numDoors;

   Car(String make, String model)
   {
      this(make, model, 4);
   }

   Car(String make, String model, int nDoors)
   {
      this.make = make;
      this.model = model;
      numDoors = nDoors;
   }
}
```

- Клас X також забезпечує ***реалізацію*** (implementation – код всередині методів з опційно допоміжними методами та полями, які відкривати не можна).
  - Допоміжні (helper) методи виступають асистентами, яких не слід показувати
  - При проектуванні класу необхідно створити корисний інтерфейс, приховуючи деталі реалізації

Клас відкриває поля екзепляру make, model та numDoors з парою конструкторів.

- Багато розробників скаже, що поля екземпляру краще робити частиною реалізації - приховувати.
- Поля можуть змінювати назву, видалятись, що зруйнує залежність між класом та його клієнтами.

Інкапсуляція та приховування даних

- Приховування спрямоване на те, щоб змінювати реалізацію міг лише розробник класу, а інтерфейс залишався незмінним.
  - Багато розробників вважають приховування інформації частиною інкапсуляції
- Java підтримує приховування інформації за допомогою 4 рівнів (модифікаторів) доступу:
  - ***public***: доступ можна отримати з будь-якого місця. Публічними можуть оголошуватись класи, зокрема, які містять точку входу public static void main(String[] args)
  - ***protected***: доступ можливий з усіх класів у межах одного пакету та підкласів незалежно від пакету
  - ***private***: доступ неможливий ззовні класу
  - ***package-private***: доступ лише в межах даного пакету. Рівень доступу за замовчуванням.

Відокремлення інтерфейсу від реалізації

```
public class Employee
{
   private String name;

   public Employee(String name)
   {
      setName(name);
   }

   public void setName(String empName)
   {
      name = empName; // Assign the empName
   }

   public String getName()
   {
      return name;
   }
}
```

- Частіше за все поля екземпляру роблять приватними і створюють набір публічних методів для задавання та отримування їх значень
  - *Геттери (getters)* дозволяють зчитувати значення
  - *Сеттери (setters)* дозволяють записувати значення
- ***Інтерфейс:*** публічні клас Employee, конструктор та setter/getter методи.
- ***Реалізація:*** приватне поле name та код конструкторів/методів

Перегляд реалізації без зміни існуючого інтерфейсу

```
public class Employee
{
   private String firstName;
   private String lastName;

   public Employee(String name)
   {
      setName(name);
   }

   public Employee(String firstName, String lastName)
   {
      setName(firstName + " " + lastName);
   }

   public void setName(String name)
   {
      // Assume that the first and last names are separated by a
      // single space character. indexOf() locates a character in a
      // string; substring() returns a portion of a string.
      setFirstName(name.substring(0, name.indexOf(' ')));
      setLastName(name.substring(name.indexOf(' ') + 1));
   }
   public String getName()
   {
      return getFirstName() + " " + getLastName();
   }

   public void setFirstName(String empFirstName)
   {
      firstName = empFirstName;
   }

   public String getFirstName()
   {
      return firstName;
   }
```

- **Навіщо створювати геттери/сеттери?**
  - Якщо можна просто пропустити private та отримувати доступ напряму до поля name.
  - Нехай необхідно створити конструктор, який відокремлює ім'я від прізвища, та нові set/get методи для роботи з іменами та прізвищами.
  - Виявляється, що доступ до окремих імені та прізвища відбувається частіше, ніж до повного ПІБ працівника!
- Уявіть інші зміни в реалізації, які додають більше коду в setFirstName(), setLastName(), getFirstName(), getLastName();
  - Не викликаючи ці методи новий код не буде виконуватись.
  - Тепер клієнтський код не постраждає

```
   public void setLastName(String empLastName)
   {
      lastName = empLastName;
   }

   public String getLastName()
   {
      return lastName;
   }
}
```

Маловідома особливість Java

```
public class PrivateAccess
{
   private int x;

   PrivateAccess(int x)
   {
      this.x = x;
   }

   boolean equalTo(PrivateAccess pa)
   {
      return pa.x == x;
   }

   public static void main(String[] args)
   {
      PrivateAccess pa1 = new PrivateAccess(10);
      PrivateAccess pa2 = new PrivateAccess(20);
      PrivateAccess pa3 = new PrivateAccess(10);
      System.out.println("pa1 equal to pa2: " + pa1.equalTo(pa2));
      System.out.println("pa2 equal to pa3: " + pa2.equalTo(pa3));
      System.out.println("pa1 equal to pa3: " + pa1.equalTo(pa3));
      System.out.println(pa2.x);
   }
}
```

- Дозволяє одному об’єкту (або методу/ініціалізатору класу) отримувати доступ до приватних полів іншого об’єкта або викликати його приватні методи.
  - Єдиний код, що здатен отримати доступ до приватного поля x знаходиться в межах класу PrivateAccess.
  - Якщо Ви намагаєтесь отримати доступ до x через об’єкт типу PrivateAccess, що був створений у контексті іншого класу, компілятор повідомить про помилку.

```
pa1 equal to pa2: false
pa2 equal to pa3: false
pa1 equal to pa3: true
20
```

ДЯКУЮ ЗА УВАГУ!

Наступне запитання: інкапсуляція та приховування інформації в Java