OBJECT-ORIENTED PROGRAMMING
LAB 5: CLASS, OBJECT, ENCAPSULATION IN OOP (CONT.)

I. Objective

After completing this tutorial, you can:

- Understand 2D array in Java;
- Advanced practice with class, object, and encapsulation in OOP.

II. 2D array in Java

2D array is an array of array. Each element, therefore, must be accessed by a corresponding number of index values.

To ***declare*** a 2D array, you can use this statement:

```
dataType[][] arrayName;
```

There are two basic ways to **initialize** a 2D array. You can use the **new** operator or use the braces to list the values:

```
dataType[][] arrayName = new dataType[arraySize1][arraySize2];
```

```
dataType[][] arrayName = {{value00, value01, ..., value0K}, ..., {valueN0, valueN1, ..., value NK}};
```

In Java, a multidimensional array allows for rows of different lengths. Example:

```
public class MyProgram {
    public static void main(String[] args) {
        int[][] array2D = new int[2][];
        int[][] array2D_1 = {{4,5,2},
                             {1,3},
                             {7,1,5,6}};
        array2D[0] = new int[2];
        array2D[0][0] = 7;
        array2D[0][1] = 2;
        array2D[1] = new int[3];
        array2D[1][0] = 1;
        array2D[1][1] = 3;
        array2D[1][2] = 8;

        for(int i = 0; i < array2D.length; i++) {
            for(int j = 0; j < array2D[i].length; j++) {
                System.out.print(array2D[i][j] + " ");
            }
            System.out.println();
        }

        for(int[] x : array2D_1) {
            for(int y : x) {
```

```
                System.out.print(y + " ");
            }
            System.out.println();
        }
    }
}
```

Similar to a two-dimensional array, you can do the same with a multidimensional array.

III. instanceof operator and equals method

Sometimes, we need to compare two objects, but the object may have many attributes, Java doesn't know which criteria to evaluate two objects as equal or not. So you need to override the equals method from the Object class. Here are two examples:

a) You have two Rectangle objects and you said that two rectangles are equal when their area is the same. So we have class Rectangle and override the equals method:

```
public class Rectangle {
    private double width;
    private double length;

    public Rectangle(double width, double length) {
        this.width = width;
        this.length = length;
    }

    public double getArea() {
        return this.length * this.width;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Rectangle) {
            Rectangle temp = (Rectangle) obj;
            if(this.getArea() == temp.getArea()) {
                return true;
            }
        }
        return false;
    }
}
```

You can test this Rectangle class as follows.

```
public class TestRectangle {
    public static void main(String[] args) {
        Rectangle rec = new Rectangle(6,8);
        Rectangle rec1  = new Rectangle(6,8);
        Rectangle rec2 = new Rectangle(7,9);

        System.out.println(rec.equals(rec1));
        System.out.println(rec.equals(rec2));
    }
}
```

b) You have two Fraction objects and these two fractions are equal when they have their numerators and denominators are equal in reduced form. We can define the Fraction class:

```java
public class Fraction {
    private int numerator;
    private int denominator;

    public Fraction(int num, int den) {
        this.numerator = num;
        this.denominator = den;
    }
    public Fraction reducer() {
        int gcd = this.gcd(this.numerator, this.denominator);
        return new Fraction(this.numerator/gcd, this.denominator/gcd);
    }
    private int gcd(int num, int den) {
        while(num != den){
            if(num > den){
                num -= den;
            }else{
                den -= num;
            }
        }
        return num;
    }
    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Fraction){
            Fraction temp = (Fraction) obj;
            temp = temp.reducer();
            Fraction temp1 = this.reducer();
            if(temp.numerator == temp1.numerator && temp.denominator == temp1.denominator){
                return true;
            }
        }
        return false;
    }
}
```

You can test this Fraction class as follows.

```java
public class TestFraction {
    public static void main(String[] args) {
        Fraction f = new Fraction(3, 4);
        Fraction f1 = new Fraction(6, 8);
        Fraction f2 = new Fraction(3, 2);

        System.out.println(f.equals(f1));
        System.out.println(f.equals(f2));
    }
}
```

V. Exercises

1. To create an integer 2-dimensional array (matrix), you can use the following statements:

```java
int[][] arr = new int[2][3] // [[0 0 0]
                            //  [0 0 0]]
int[][] arr1 = {{1, 2, 3}, {4, 5, 6}} // [[1 2 3]
                                      //  [4 5 6]]
```

Write a Java program:

- Write a function to add two matrices of the same size.
- Write a function to multiply two matrices.
- Write a function to print a matrix to screen in matrix format.
- Write a main function and run all the above functions.

2. Write a program to count how many times one word appears in the paragraph. Expect result is the 2-dimensional array in which the first column contains only one word, and the second column contains its frequency. Words are not case-sensitive.

   For example, given the string:

   "You are living on a Plane. What you style Flatland is the vast level surface of what I may call a fluid, on, or in, the top of which you and your countrymen move about, without rising above it or falling below it."

   The example output may be like this figure (the order of the words is not obligated to follow this example):

```
C:\Users\Admin\Desktop\Java Exercise\Solve\Lab3>java WordsFrequency
'You': 3,
'are': 1,
'living': 1,
'on': 2,
'a': 2,
'Plane': 1,
'What': 2,
'style': 1,
'Flatland': 1,
'is': 1,
'the': 2,
'vast': 1,
'level': 1,
'surface': 1,
'of': 2,
'I': 1,
'may': 1,
'call': 1,
'fluid': 1,
'or': 2,
'in': 1,
'top': 1,
'which': 1,
'and': 1,
'your': 1,
'countrymen': 1,
'move': 1,
'about': 1,
'without': 1,
'rising': 1,
'above': 1,
'it': 2,
'falling': 1,
'below': 1
```

3. Imagine you're developing a restaurant management system, where customers can place orders, receive invoices, and make payments. This assignment will guide you through creating a fully functional program, one step at a time. By the end, you'll have a firm grasp of designing and interacting with multiple classes, handling user input, and organizing data efficiently using arrays.

   Step 1: Designing Food and Drink Items

   Let's start with the building blocks of our restaurant – **Food** and **Drinks**. You'll create two classes: **FoodItem** and **DrinkItem**. Each item will have a **name** and a **price**, stored as private attributes. To keep things structured, implement a **no-argument constructor** and a **parameterized constructor** for initializing these objects. You'll also need **getter methods** to access their values. To make our system more interactive, override the **toString()** method so that each item is neatly displayed in the format: "*name - price*". Lastly, implement **equals()** to ensure that two items with the same name and price are recognized as identical.

   Step 2: Processing Customer Orders

   Now that we have food and drinks, it's time to take orders! Introduce an **OrderItem** class, representing a specific order placed by a customer. This class should store the **name of the item**, the **quantity ordered**, and the **price per unit**. Similar to before, provide both types of constructors and necessary getters. Enhance this class by overriding **toString()**, formatting the order details as: "*itemName - Qty: quantity - Price: price*". Also, ensure that **equals()** correctly identifies duplicate orders.

   Step 3: Managing the Invoice

   A restaurant wouldn't function without a proper bill! Here comes the **Invoice** class, responsible for storing the customer's order details. It should contain an **array** of **OrderItem** objects (let's limit it to 100 items), a **counter** to track the number of orders, and a **total amount** to keep tabs on the bill. The key functions here are:

   - **addItem(OrderItem item):** This method adds a new order to the list, but before doing so, it should check if the item already exists (using **equals()** to prevent duplicates).
   - **printInvoice():** This prints all ordered items along with the total cost.

   Step 4: Bringing It All Together in the Main Program

   It's time to make our system interactive! Create a **RestaurantManagement** class containing the main method. Here, you'll:

   - Define a **menu** with an array of **FoodItem** and **DrinkItem** objects.
   - Set up a **menu-driven** system using a while loop and switch-case statements, allowing users to:

- **Order food or drinks** by selecting items from the menu.
- **Print the invoice** to see their current bill.
- **Pay the invoice (exit the system)**.

To handle user input, use the **Scanner** class to let customers select menu items and enter quantities. Here's the example of the application:

```
1. Order Food
2. Print Invoice
3. Pay and Exit
Select an option: 1
Select food item:
1. Chicken Rice - 50000.0
2. Beef Noodles - 40000.0
3. Coca Cola - 15000.0
4. Iced Tea - 5000.0
Enter item number: 1
Enter quantity: 1

1. Order Food
2. Print Invoice
3. Pay and Exit
Select an option: 1
Select food item:
1. Chicken Rice - 50000.0
2. Beef Noodles - 40000.0
3. Coca Cola - 15000.0
4. Iced Tea - 5000.0
Enter item number: 2
Enter quantity: 1

1. Order Food
2. Print Invoice
3. Pay and Exit
Select an option: 1
Select food item:
1. Chicken Rice - 50000.0
2. Beef Noodles - 40000.0
3. Coca Cola - 15000.0
4. Iced Tea - 5000.0
Enter item number: 3
Enter quantity: 2

1. Order Food
2. Print Invoice
3. Pay and Exit
Select an option: 2
--- Invoice ---
Chicken Rice - Qty: 1 - Price: 50000.0
Beef Noodles - Qty: 1 - Price: 40000.0
Coca Cola - Qty: 2 - Price: 15000.0
Total Amount: 120000.0

1. Order Food
2. Print Invoice
3. Pay and Exit
Select an option: 3
Payment successful!
➜  Demo
```

4. Dictionary Management System

Imagine you are building a **basic dictionary management** system using Java. This system will allow users to **store words, manage their meanings, and retrieve definitions** interactively through a console menu.

This exercise will help you understand **static methods and variables** by managing a shared dictionary without using objects for dictionary storage. Instead, all dictionary-related operations will be handled using **static** methods and a **static** array.

In this exercise, you need to define a class **Word** to contain word and its meaning, and a class **Dictionary** to contain a list of words.

System Features

a. Add a word and its meaning:

- If the word is **new**, add it to the dictionary along with its first meaning.
- If the word **already exists**, add a new meaning (if it's not duplicated).

b. Remove the meaning of a word:

- The user provides a word and selects an index of a meaning to remove.
- If it's the only meaning, removing it should keep the word entry but with no meanings left.

c. ***Look up a word:*** The user enters a word, and the program returns all stored meanings.

d. ***Check the total number of words in the dictionary*** using a **static** variable.

e. Exit the program.

Key Constraints

- Do not use Java Collections (ArrayList, HashMap, etc.).
- Use **static variables** to maintain the dictionary and count words.
- Use a **static array** to store words.
- The dictionary can store up to 100 words, and each word can have up to 10 meanings.
- **Encapsulation:** Keep attributes private and use **getter methods**.
- Override **toString()** for formatted output.
- Implement a client class containing a **main** method to handle user input and output.

Hint: Use the attribute that tracks the number of meanings to manage the meaning array. This helps avoid unnecessary iterations over empty slots and ensures that new meanings are always added in the correct position. When removing a meaning, shift the remaining elements to maintain order in the array.

Here's the example of the application:

```
1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 1
Enter word: Java
Enter meaning: An island in Indonesia.
Word/meaning added successfully.

1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 1
Enter word: Java
Enter meaning: A programming language.
Word/meaning added successfully.

1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 1
Enter word: Python
Enter meaning: A type of large snake.
Word/meaning added successfully.

1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 3
Enter word to look up: Java
Word: Java
Meanings:
0) An island in Indonesia.
1) A programming language.
```

Figure 1

```
1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 4
Total words in dictionary: 2

1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 2
Enter word: Java
Word: Java
Meanings:
0) An island in Indonesia.
1) A programming language.

Enter index of meaning to remove: 0
Meaning removed successfully.

1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 3
Enter word to look up: Java
Word: Java
Meanings:
0) A programming language.


1. Add Word (or Add Meaning to Existing Word)
2. Remove Meaning
3. Look Up Word
4. Check Word Count
5. Exit
Select an option: 5
Exiting...
→  Demo
```

Figure 2