SWEN20003
Object Oriented Software Development
Workshop 3

Workshop

This week, we are learning more about how to effectively structure classes in a Java program, and more practice with using two important Java types: **arrays** and **strings**.

- **Encapsulation** refers to grouping objects' data with the methods that operate on this data.
- **Information hiding** refers to hiding attributes and methods from the user of the class, e.g. using the `private` keyword.
- **Delegation** refers to the process of assigning different responsibilites to different classes.
- An **immutable** object is one whose attributes cannot be changed after it is created.

Additionally, you have been introduced to a very important version control tool called **Git**. You will get a chance to practice your git commands by **cloning** a repository that we have set up for you, **adding** your solution to it, and **pushing** it to the remote repository on Gitlab. Make sure you are familiar with the process and ask your demonstrator for help if you are confused about anything, as you are required to make regular commits in your projects.

Questions

1. Using the principle of **information hiding**, assign privacy modifiers (either `public` or `private`) to attributes and methods in the below class.

```
public class Drone {
    double homeX;
    double homeY;
    double x;
    double y;
    double altitude = 0.0;

    Drone(double homeX, double homeY) {
        this.homeX = homeX;
        this.homeY = homeY;
        x = homeX;
        y = homeY;
    }

    void flyUp(double amount) {
        altitude += amount;
    }

    void flyDown(double amount) {
        altitude = Math.max(altitude - amount, 0);
    }

    double distanceToHome() {
        return distance(x, y, homeX, homeY);
    }
```

```
    static double distance(double x1, double y1, double x2, double y2) {
        return Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -y2));
    }
}
```

2. (a) Design and implement classes to represent channels airing on TV.

   A channel has a *name*, and broadcasts up to 5 *shows* each day. (For simplicity, assume they are the same every day). A show has a *name*, a *duration* (in minutes) an *air time* (in hours and minutes).

   A channel with less than 5 shows can have a show added to its broadcast list. When doing so, it should check that two shows are not scheduled to run at the same time (otherwise adding the clashing show does nothing). A channel can also cancel a show, removing it from the broadcast list.

   (b) Add a `getShow` method to your channel class. Given a time (in hours and minutes), it should return the show that is scheduled to be running at that time (or `null` if there is no such show).

   (c) Create a class to represent a network of up to 3 channels. Networks have a *name*, and channels can be added to or removed from a network. A network has a `getShows` method that returns all shows running at a particular time on any channel in the network.

   (d) Add a `lookupShow` method to your network class that takes a show and returns which channel that show is scheduled to run on. If there are multiple channels, only return the first that you find.

3. Consider the below class (the raw code is attached to Canvas).

```
public class Person {
    public String name;
    public double x;
    public double y;
    public String householdName;

    private static Person[] people = new Person[100];
    private static int peopleCount = 0;


    public Person(String name, double x, double y, String householdName) {
        this.name = name;
        this.x = x;
        this.y = y;
        this.householdName = householdName;

        if (peopleCount < 100) {
            people[peopleCount++] = this;
        }
    }

    public double distanceToPerson(Person person) {
        return Math.sqrt((x - person.x) * (x - person.x)
                + (y - person.y) * (y - person.y));
    }

    private Person[] peopleCloserThan(double distance) {
        int numCloser = 0;
        // Count how many people are close
        for (int i = 0; i < peopleCount; ++i) {
            if (distanceToPerson(people[i]) < distance) {
                ++numCloser;
            }
        }

        // Create an appropriately-sized array, and then fill it
        Person[] result = new Person[numCloser];
        int count = 0;
        for (int i = 0; i < peopleCount; ++i) {
```

```
            if (distanceToPerson(people[i]) < distance) {
                result[count++] = people[i];
            }
        }

        return result;
    }

    public int numCloseOutsideHousehold(double distance) {
        Person[] people = peopleCloserThan(distance);
        int count = 0;
        for (int i = 0; i < people.length; ++i) {
            // If they are not from this person's household, increment counter
            if (!people[i].householdName.equals(householdName)) {
                ++count;
            }
        }
        return count;
    }
}
```

(a) If there are any public attributes or methods that should be private according to the principle of **information hiding**, make them private instead.

(b) Using the principle of **encapsulation**, define a `Point` class with an x- and y-coordinate, and a method `double distanceTo(Point other)` to calculate the distance to another point.

(c) Using the principle of **delegation**, replace the `x` and `y` attributes of `Person` with an instance of `Point`. Update the methods of `Person` accordingly.

(d) Using the principle of **delegation**, define a `Household` class with an appropriate `equals` method. Each household has a *name* and up to 5 *people* (set in the constructor). Replace the `householdName` attribute of `Person` with an instance of `Household`. Update the methods of `Person` accordingly. (You may assume household names are unique, and you may need to add a setter for the household.)

(e) Using the principle of **encapsulation**, define a `boolean contains(Person person)` method for `Household` that returns `true` if the person is in that household.

(f) Using the principle of **delegation**, modify the `numCloseOutsideHousehold` method of `Person` to use the `contains` method defined in (e).

(g) Using the principle of **encapsulation** define a `int numCloseOutsideHousehold(double distance)` method for `Household` that calculates and returns the total number of people in the household who are close to people outside the household.

(h) Define a `main` method that creates two households and fills them each with 5 people, with random coordinates between 0 and 20. (Use `Math.random` or `java.util.Random`.) Print the result of `numCloseOutsideHousehold(10)` for one of the households.

4. Modify the following class so that it is **immutable**.

```
public class Student {
    private String name;
    public int studentNumber;

    public Student(String name, int studentNumber) {
        this.name = name;
        this.studentNumber = studentNumber;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }
}
```

5. Write a program that reads in a **single line** from standard input and calculates its acronym, writing the result in a **single line** to standard output. You **do not** need to print out "Input" and "Output" prompts, or read in more than one input. Assume that the input will not be empty and every word is separated by a space.

   Examples:

   ```
   Portable Network Graphics
   PNG

   LeAgUe oF LeGeNdS
   LOL

   situational task and yodelling hat open mustache extension
   STAYHOME
   ```

   Please follow the steps below to clone the Gitlab repository that has been set up for you and push your solution for **Question 5** to it. We will harvest all the contents in your repository at the end of this week to check that you have completed this exercise. You can find more details about Git in Lecture 6.

   (a) Log into https://swen20003.eng.unimelb.edu.au/ with your **university username** and **password**.

   (b) You will see 3 folders already created for you:

      i. `[username]-workshops`
      ii. `[username]-project-1`
      iii. `[username]-project-2`

   (c) Go to the `[username]-workshops` folder and click the `Clone` button.

   (d) Copy the URL link under `Clone with HTTPS`. You can also choose with Clone with SSH option if you are familiar with Git and prefer to do so.

   (e) Open a command window (e.g. Terminal on MAC and Command Prompt/Git Bash on Windows) on your local machine and make sure Git is installed by typing in the command `git version`. You should see the version of Git that is installed on your machine. If you have yet to install Git, follow the tutorial here.

   (f) Once you have Git installed, change the directory to where you would like the local copy of your repository to be created. You can do so with the `cd` command. If you are not familiar with this, find more information here.

   (g) Make a local copy of the repository with this command: `git clone -c http.sslVerify=false [URL that you copied from Gitlab]`.You will see a newly-created folder called `[username]-workshops`. This is the local copy of your workshop repository.

   (h) You are now ready to move your solution to your local repository. Create a new folder in the local copy of your repository called `Workshop 3`.

   (i) Open where your solution to Question 5 is stored and copy all the files to the `Workshop 3` folder.

   (j) Stage the files in your local repository by typing the command `git add .` (to stage **all** the files) or `git add [file-name]` (to stage specific files). Staging a file means adding it to the list of files that has been modified and is ready to commit and push.

   (k) Commit your changes with this command: `git commit -m "message"`. Make sure to replace `message` with a meaningful message that describes the changes you made in this commit.

   (l) Push your changes to the master branch of your remote repository with this command: `git push -u origin master`.