SWEN20003
Object Oriented Software Development
Workshop 3 (Solutions)

Eleanor McMurtry

Semester 2, 2020

Workshop

Questions

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

   Solution:

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

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

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

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

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

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

2. Consider the below class. Using the principle of **delegation**, create a `Point` class and replace the attributes of `Rectangle` with instances of `Point`.

   Solution:

```
public class Point {
    public final double x;
    public final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

public class Rectangle {
    private Point topLeft;
    private Point bottomRight;

    public Rectangle(double left, double top, double right, double bottom) {
        topLeft = new Point(left, top);
        bottomRight = new Point(right, bottom);
    }

    public double getLeft() {
        return topLeft.x;
    }

    public double getTop() {
        return topLeft.y;
    }

    public double getWidth() {
        return bottomRight.x - topLeft.x;
    }

    public double getHeight() {
        return bottomRight.y - topLeft.y;
    }
}
```

3. (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.

   Solution: Here we use a common trick: just because you need to specify your start time in hours and minutes, does not mean you have to *store* it as hours and minutes. Choosing the representation for your data carefully will often make your life a lot easier.

```
public class Show {
    public final String name;
    public final int duration;
    private final int startTimeMin;

    public Show(String name, int duration, int airTimeHrs, int airTimeMin) {
        this.name = name;
        this.duration = duration;
        this.startTimeMin = airTimeHrs * 60 + airTimeMin;
    }
```

```
    public int getAirTimeHrs() {
        return startTimeMin / 60;
    }

    public int getAirTimeMin() {
        return startTimeMin % 60;
    }

    public boolean overlaps(Show other) {
        // Either we start while they're on, or they start while we're on
        return (startTimeMin >= other.startTimeMin
                && startTimeMin < other.startTimeMin + other.duration)
            || (startTimeMin <= other.startTimeMin
                && startTimeMin + duration > other.startTimeMin);
    }
}

public class Channel {
    public final String name;

    private static final int MAX_SHOWS = 5;

    private int numShows = 0;
    private final Show[] shows = new Show[MAX_SHOWS];


    public Channel(String name) {
        this.name = name;
    }

    public void addShow(Show show) {
        if (numShows < MAX_SHOWS) {
            for (int i = 0; i < numShows; ++i) {
                if (show.overlaps(shows[i])) {
                    return;
                }
            }

            shows[numShows++] = show;
        }
    }
}
```

(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).

Solution:

```
public Show getShow(int hrs, int mins) {
    Show testShow = new Show("", 0, hrs, mins);

    for (int i = 0; i < numShows; ++i) {
        if (shows[i].overlaps(testShow)) {
            return shows[i];
        }
    }

    return null;
}
```

(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.

Solution:

```
public class Network {
    public final String name;

    private static final int MAX_CHANNELS = 3;
    private final Channel[] channels = new Channel[MAX_CHANNELS];
    private int numChannels = 0;

    public Network(String name) {
        this.name = name;
    }

    public void addChannel(Channel channel) {
        if (numChannels < MAX_CHANNELS) {
            channels[numChannels++] = channel;
        }
    }

    public void removeChannel(Channel channel) {
        for (int i = 0; i < numChannels; ++i) {
            if (channels[i].name.equals(channel.name)) {
                // Shift all the following elements back
                for (int j = i + 1; j < numChannels; ++j) {
                    channels[j - 1] = channels[j];
                }

                // Remove the last one
                --numChannels;
                channels[numChannels] = null;
                return;
            }
        }
    }

    public Show[] getShows(int hrs, int mins) {
        Show[] results = new Show[MAX_CHANNELS];
        int resultIndex = 0;

        for (int i = 0; i < numChannels; ++i) {
            Show show = channels[i].getShow(hrs, mins);
            if (show != null) {
                results[resultIndex++] = show;
            }
        }

        return results;
    }
}
```

(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.

Solution:

```
public Channel lookupShow(Show show) {
    for (int i = 0; i < numChannels; ++i) {
        if (channels[i].hasShow(show)) {
            return channels[i];
        }
    }

    return null;
}
// hasShow is defined in Channel as:
```

```
public boolean hasShow(Show show) {
    for (int i = 0; i < numShows; ++i) {
        if (shows[i].name.equals(show.name)) {
            return true;
        }
    }
    return false;
}
```

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

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

   Solution: All attributes should be private, and so should the `distanceToPerson` method.

   (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.

   Solution:

```
public class Point {
    public final double x;
    public final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double distanceTo(Point other) {
        return Math.sqrt((x - other.x) * (x - other.x) + (y - other.y) * (y - other.y));
    }
}
```

   (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.

   Solution:

```
public class Person {
    private String name;
    private Point location;
    private String householdName;

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


    public Person(String name, Point location, String householdName) {
        this.name = name;
        this.location = location;
        this.householdName = householdName;

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

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

(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.

Solution: Note this does not enforce the maximum number of people.

```
public class Household {
    public final String name;
    private final Person[] people;

    public Household(String name, Person[] people) {
        this.name = name;
        this.people = people;
    }

    public boolean equals(Household other) {
        // Assume households are unique by name
        return name.equals(other.name);
    }
}
public class Person {
    private String name;
    private Point location;
    private Household household;

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


    public Person(String name, Point location) {
        this.name = name;
        this.location = location;

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

```
    // Allow people to move now :-)
    public void setHousehold(Household household) {
        this.household = household;
    }

    private Person[] peopleCloserThan(double distance) {
        int numCloser = 0;
        // Count how many people are close
        for (int i = 0; i < peopleCount; ++i) {
            if (location.distanceTo(people[i].location) < 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 (location.distanceTo(people[i].location) < 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].household.equals(household)) {
                ++count;
            }
        }
        return count;
    }
}
```

(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.

Solution:

```
public boolean contains(Person person) {
    for (Person other : people) {
        if (person.equals(other)) {
            return true;
        }
    }
    return false;
}
// Person.equals:

public boolean equals(Person other) {
    return name.equals(other.name) && household.equals(other.household);
}
```

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

Solution:

```
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 (!household.contains(people[i])) {
            ++count;
        }
    }
    return count;
}
```

(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.

Solution:

```
public int numCloseOutsideHousehold(double distance) {
    int count = 0;
    for (Person person : people) {
        if (person.numCloseOutsideHousehold(distance) > 0) {
            ++count;
        }
    }
    return count;
}
```

(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.

Solution:

```
public static void main(String[] args) {
    Person aliceYan = new Person("Alice", randomPoint());
    Person bobYan = new Person("Bob", randomPoint());
    Person carterYan = new Person("Carter", randomPoint());
    Person dariusYan = new Person("Darius", randomPoint());
    Person eyreYan = new Person("Eyre", randomPoint());

    Person[] yans = new Person[] {
            aliceYan, bobYan, carterYan, dariusYan, eyreYan
    };

    Household yanHousehold = new Household("Yan", yans);
    for (Person person: yans) {
        person.setHousehold(yanHousehold);
    }


    Person aliceMcMurtry = new Person("Alice", randomPoint());
    Person bobMcMurtry = new Person("Bob", randomPoint());
    Person carterMcMurtry = new Person("Carter", randomPoint());
    Person dariusMcMurtry = new Person("Darius", randomPoint());
    Person eyreMcMurtry = new Person("Eyre", randomPoint());

    Person[] mcMurtrys = new Person[] {
            aliceMcMurtry, bobMcMurtry, carterMcMurtry, dariusMcMurtry, eyreMcMurtry
    };

    Household mcMurtryHousehold = new Household("McMurtry", mcMurtrys);
    for (Person person: mcMurtrys) {
        person.setHousehold(mcMurtryHousehold);
    }

    System.out.println("Number violating: " + yanHousehold.numCloseOutsideHousehold(10));
}
```

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

   Solution:

```java
public class Student {
    public final String name;
    public final int studentNumber;

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

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

6. We want to assign a numerical score to a given string. The score is the sum of the value of its constituent letters. The value of each letter is defined as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	3	3	2	1	4	2	4	1	8	5	1	3	1	1	3	10	1	1	1	1	4	4	8	4	10

   Write a program that reads in a single line from standard input and calculates its score, writing the result to standard output. The score of an empty input is 0.

7. 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.

   Examples:

```
Portable Network Graphics
PNG

LeAgUe oF LeGeNdS
LOL

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

   Assume that the input will not be empty and every word is separated by a space.

   Solution:

```java
import java.util.Scanner;

public class StringScoring {

    private static final char[] letterScores =
        {1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3, 1, 1, 3, 10, 1, 1, 1, 1, 4, 4, 8, 4, 10};


    public static int score(String word) {
        // Assumes well-formed input
        int sum = 0;
        for (char c : word.toCharArray()) {
            sum += letterScores[c - 'a'];
        }
        return sum;
    }
```

```
    public static void main(String[] args) {
        String word = new Scanner(System.in).nextLine();
        System.out.println(score(word));
    }
}
```