SWEN20003
Object Oriented Software Development
Workshop 6 (Solutions)

Eleanor McMurtry

Semester 2, 2020

Workshop

Questions

1. Often when writing software, we would like to be able to save objects to a file.

   (a) Define a `FileWriteable` interface with a method `void writeToFile(BufferedWriter writer)`. This method should be used to write some textual representation of the object to the provided `BufferedWriter`.

   Solution:

```
import java.io.BufferedWriter;

public interface FileWriteable {
    void writeToFile(BufferedWriter writer) throws IOException;
}
```

   (b) Define the following classes, and implement the interface for them:

   - `Point`, with attributes `x` and `y`
   - `Student`, with attributes `name` and `id`
   - `Car`, with attributes `model` and `colour`

   Why does it not make sense for these classes to inherit from a base class?

   Solution:

```
import java.io.BufferedWriter;
import java.io.IOException;

public class Point implements FileWriteable {
    public final double x;
    public final double y;

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

    @Override
    public void writeToFile(BufferedWriter writer) throws IOException {
        String result = String.format("%f,%f", x, y);
        writer.write(result, 0, result.length());
        writer.newLine();
    }
}

public class Student implements FileWriteable {
    public final String name;
    public final int id;;
```

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

    @Override
    public void writeToFile(BufferedWriter writer) throws IOException {
        String result = String.format("%s,%d", name, id);
        writer.write(result, 0, result.length());
        writer.newLine();
    }
}

public class Car implements FileWriteable {
    public final String model;
    public final String colour;

    public Car(String model, String colour) {
        this.model = model;
        this.colour = colour;
    }

    @Override
    public void writeToFile(BufferedWriter writer) throws IOException {
        String result = String.format("%s,%s", model, colour);
        writer.write(result, 0, result.length());
        writer.newLine();
    }
}
```

(Note that for the sake of this simple example, the implementations are quite similar.)

(c) Define a class `Database`. It should store up to 100 `FileWriteable` objects. Objects can be added to and removed from the database.

(d) Add a method `void writeAll(String filename)` that opens a file called `filename`, and writes all of its objects to that file.

Solution:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Database {
    private final FileWriteable[] data = new FileWriteable[100];

    public void add(FileWriteable row) {
        // Obviously this code is a bit inefficient. In practice you'd use ArrayList.
        for (int i = 0; i < data.length; ++i) {
            if (data[i] == null) {
                data[i] = row;
                break;
            }
        }
    }

    public void remove(FileWriteable row) {
        for (int i = 0; i < data.length; ++i) {
            if (row.equals(data[i])) {
                data[i] = null;
                break;
            }
```

```
        }
    }

    public void writeAll(String filename) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            for (FileWriteable row : data) {
                if (row != null) {
                    row.writeToFile(writer);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

(e) Write a main method to test your `Database`.

Solution:

```
public static void main(String[] args) {
    Database db = new Database();
    db.add(new Student("Alice", 766050));
    db.add(new Point(4.2, 6.9));
    db.add(new Car("Corolla", "white"));
    db.writeAll("sample.txt");
}
```

Output:

```
Alice,766050
4.200000,6.900000
Corolla,white
```

2. The `Comparable<T>` interface is used to allowing sorting for arrays and other data structures. It defines a single method, `int compareTo(T other)`, which should return a negative number if `this` should come before `other`, a positive number if `this` should come after `other`, and 0 if they are equal. An array of any type that implements `Comparable<T>` can be sorted using the `Arrays.sort()` method:

```
import java.util.Arrays;

public class Student implements Comparable<Student> {
    public final String name;
    public final int number;

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

    public int compareTo(Student other) {
        if (name.compareTo(other.name) < 0) {
            return -1;
        }
        if (name.compareTo(other.name) > 0) {
            return 1;
        }
        return number - other.number;
    }

    public String toString() {
        return String.format("(%s, %d)", name, number);
    }
```

```
    public static void main(String[] args) {
        Student[] students = new Student[] {
                new Student("Alice", 753285),
                new Student("Charlie", 913571),
                new Student("Bob", 832572),
                new Student("Bob", 632564)
        };
        System.out.println(Arrays.toString(students));
        Arrays.sort(students);
        System.out.println(Arrays.toString(students));
    }
}
```

Implement `Comparable<T>` (where `T` is itself) for each of the following. Sort the objects in the attributes' listed order.

- `AuctionBid`, which has a **name**, **item name**, and **amount** (in dollars)
- `ZooAnimal`, which has a **name** and a **species**
- `Javamon`, which has a **Javadex number** and a **name**

Solution:

```
public class AuctionBid implements Comparable<AuctionBid> {
    public final String name;
    public final String itemName;
    public final int amount;

    public AuctionBid(String name, String itemName, int amount) {
        this.name = name;
        this.itemName = itemName;
        this.amount = amount;
    }

    public int compareTo(AuctionBid other) {
        int result = name.compareTo(other.name);
        if (result != 0) {
            return result;
        }
        result = itemName.compareTo(other.itemName);
        if (result != 0) {
            return result;
        }
        return amount - other.amount;
    }
}

public class ZooAnimal implements Comparable<ZooAnimal> {
    public final String name;
    public final String species;

    public ZooAnimal(String name, String species) {
        this.name = name;
        this.species = species;
    }

    public int compareTo(ZooAnimal other) {
        int result = name.compareTo(other.name);
        if (result != 0) {
            return result;
        }
        return species.compareTo(other.species);
```

```
    }
}

public class Javamon implements Comparable<Javamon> {
    public final int number;
    public final String name;

    public Javamon(int number, String name) {
        this.number = number;
        this.name = name;
    }

    public int compareTo(Javamon other) {
        if (number != other.number) {
            return number - other.number;
        }
        return name.compareTo(other.name);
    }
}
```

3. You are tasked with improving the design of a software called +Etacolla. This software allocates students class times for their enrolled subjects. Here is the current core of +Etacolla.

```
public class Etacolla {
    private final MonsterUniService mUniService;

    public Etacolla() {
        this.mUniService = new MonsterUniService();
    }

    public void generateTimetable(Student student) {
        List<String> subjectNames = mUniService.getEnrolledSubjectCodes(student);
        List<Subject> subjects = new ArrayList<>();
        for (String subjectName : subjectNames) {
            Subject subject = mUniService.getSubject(subjectName);
            subjects.add(subject);
        }
        // allocate activities to student ...
        List<Activity> allocated = allocatePreferences(student.getPreferences(), subjects);
        for (Activity activity : allocated) {
            mUniService.registerStudentInActivity(student, activity);
        }
    }
}
```

+Etacolla's first client was Monster University, but more universities are hopping on board. By creating an interface `UniversityService` and changing the `Etacolla` constructor to take an instance of this interface as an argument, generalise the design to support other universities. (This kind of approach is sometimes called **dependency injection**.)

Solution: We create an interface to reduce the tight dependency on the MonsterUniService.

```
public interface UniversityService {
    List<String> getEnrolledSubjectCodes(Student student);
    Subject getSubject(String name);
    void registerStudentInActivity(Student student, Activity activity);
}
```

We use the interface to generalise the class design:

```
public class Etacolla {
    private final UniversityService universityService;
```

```
    public Etacolla(UniversityService service) {
        this.universityService = service;
    }

    public void generateTimetable(Student student) {
        List<String> subjectNames = universityService.getEnrolledSubjectCodes(student);
        List<Subject> subjects = new ArrayList<>();
        for (String subjectName : subjectNames) {
            Subject subject = universityService.getSubject(subjectName);
            subjects.add(subject);
        }
        // allocate activities to student ...
        List<Activity> allocated = allocatePreferences(student.preferences, subjects);
        for (Activity activity : allocated) {
            universityService.registerStudentInActivity(student, activity);
        }
    }
}
```