SWEN20003
Object Oriented Software Development
Workshop 6

Eleanor McMurtry

Semester 2, 2020

Workshop

This week, we are learning all about interfaces.

- Interfaces allow an object to be represented by a set of **methods**, regardless of which class the object is an instance of.
- Interfaces contain methods with **no implementation**—only the method signatures.
- By upcasting to the interface type, we can operate on a much wider range of objects.
- Interfaces **are only useful with polymorphism.** Defining an interface that is implemented by only one class is incorrect, and you will lose marks in projects if you do this.
- Just because multiple classes share similar methods, does not necessarily mean defining an interface is useful.

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) throws IOException`
   This method should be used to write some textual representation of the object to the provided `BufferedWriter`. (This is a process called **serialisation**.)

   (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?

   (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.

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

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**

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**.)