UNIVERSITÉ DU LUXEMBOURG

Pokedeck

Programming Fundamentals 2

Goals

- ✯ Object-oriented design and coding of a data management application.
- ✯ Read and manipulate JSON data in Java.
- ✯ Learn the functional API of Java.
- ✯ Composite, factory and iterator design patterns.
- ✯ The open-closed and single responsibility principles.
- ✯ **Relevant videos**:
  - Unified Modelling Language (UML).
  - Polymorphism in (more) depth.
  - Error Management.
  - Advanced Java Concepts.
  - Lambda Expressions.

Deliverables

1. The code on your Github repository generated by clicking here: `https://classroom.github.com/a/DHrvYXUX`
2. **Reviewer**: Pierre Talbot (ptal on Github).
3. **No automated review**

1 Pokedeck

The Pokémon TCG API is a great resource for all Pokémon cards collectors! You always wanted to program your own Pokémon cards collection software and this API seems like a great start. But what a dissapointment when you find out there is no SDK for your favorite programming language[1]. It is sad, but cry no more, because with what you know about Java, you will be able to program it yourself. "So cool", you think.

Exercise 1 – Reading a deck from a JSON file.

The JavaScript Object Notation (JSON) is a useful format to store and exchange data. Fortunately, some hardcore Pokémon lovers already encoded all cards in the JSON format. For instance, you find such a JSON file here.

1. Download the JSON file above and place it in the folder `src/main/resources/base1.json` of your project. To ease the testing of your application, it is worth to have a smaller file `src/main/resources/small-base1.json` containing only a few cards.
2. Read the file and extract its content into a `String`. You should find how to read a file in Java yourself (hint: Java has many ways of reading a file, some ways are easier than others—check out `Files.readAllBytes` and `Paths.get`).
3. Commit and push.
4. To parse the String into a JSON object, we will use the library `org.json`. This is not provided by default in the Java Standard Libraries, so you need to add this *dependency* to your Maven project. To do that, you can browse the *MVN repository* website, and it will give you an XML dependency that you can copy paste in the right place in your `pom.xml`. To check it works, try to import the library in your main file (i.e., `import org.json.*;`), and see if it compiles. Then you can read the documentation of this library to convert a String into a JSON structure (spoiler: it is easy, only one line).
5. Commit and push.
6. To get used to this library, read your small file `small-base1.json`[2] and print the name of the first Pokémon card in the deck.
7. Commit and push.

Exercise 2 – A view of the deck

Because the cards collection is huge, we would like to be able to filter the cards we are interested by. For instance, we would like to get the cards with "more than 50 HP" and of type "Basic" or "Colorless". An inefficient way to implement such filters would be to write methods iterating over the deck (a `JSONArray` object), and reconstructing a new object `JSONArray` containing only the desired cards. However, the cards are stored several times which is a waste of resources, so we are going to use a smarter technique.

1. Create a `DeckView` class with the JSON deck as an attribute, and a constructor taking this deck as parameter. Implement a method `String toStringID()` producing a comma-separated list of the cards IDs, e.g., `"base1-2,base1-3"`.
2. We are going to test this class using JUnit tests. You already encountered JUnit tests in the labs `dynamic_array` and `linked_list`. By copying how we achieve that in these labs, add a JUnit test file `DeckViewTest.java`. Using the small JSON file you created before, implement a test reading that file, and initializing a `DeckView` object. Then, use `assertEquals` to test your `toStringID()` method.

[1] `https://docs.pokemontcg.io/sdks/overview`

[2] Since it is just a test, you can directly write the path in your main function. Note that the path is `src/main/resources/small-base1.json`

3. Commit and push.
4. The key to avoid repeating cards among different "views" of the deck, is to use a "mask". Add an attribute `private ArrayList<Integer> view;` to your class. Imagine a deck initially with 3 cards, then this attribute is initialized to the array `[0,1,2]`. Later, when we filter the deck, we can create a new object `DeckView` using the same JSONArray deck attribute, but a different view of the deck.
5. Commit and push.
6. Write a method `public DeckView filterHPLessThan(int hp)` which returns a view of the deck with cards having only HP less than `hp`. Continuing the previous example, if you have three cards with respectively 10, 20 and 30 HPs, the statement `System.out.println(deck.filterHPLessThan(21).toStringID());` must print the first and second cards only. Note that after calling the filter, the initial variable `deck` must be unchanged.
7. Write JUnit tests to test your HP filter.
8. Commit and push.

Exercise 3 – Filtering the deck

1. If we want to add more filters, it is going to be painful, for instance imagine adding `filterHPEqualsTo`, `filterHPNotEqualsTo`, `filterHPMoreThan`, etc. Therefore, we need a better abstraction. To achieve that, we are going to use lambda expressions and the `java.util.function` library. Write a method `public filterHP(Predicate<Integer> hpFilter)` (don't forget to add `import java.util.function.*;`). The class `Predicate` encapsulates a logical predicate returning true or false when passed an integer. Therefore, the user will be able to call our method like this: `deck.filterHP(hp -> hp < 21)` or `deck.filterHP(hp -> hp == 50)`. It makes the code very clear, and we do not need to program all methods ourselves.
2. Adapt and add new JUnit tests to verify your implementation.
3. Commit and push.
4. For now, we have only a single kind of filter over the HP characteristic of the cards. Add a new method `filterName(Predicate<String> nameFilter)` filtering on the name of the Pokemon.
5. Test, commit and push.
6. Annoyingly, `filterHP` and `filterName` share a very similar code, and as you know, *we should not repeat ourselves.* Therefore, we need a better abstraction, again. One problem of the class `DeckView` is that it fails to satisfy two principles:
   - Open-closed principle.
   - Single-responsibility principle.

   You will explain why in the report (see last question). Modify your design to have instead a generic filtering function `public DeckView filter(Predicate<JSONObject> filter)`.
7. A problem with our newest design is that we lose a bit in terms of functionalities: the users of our class need to create themselves the predicates to filter HP and name. Create a new class `FilterFactory`, and add two static methods:

```
public static Predicate<JSONObject> hp(Predicate<Integer> filter);
public static Predicate<JSONObject> name(Predicate<String> filter);
```

8. Modify your tests to use the new filter function and the filters constructed using `FilterFactory`.

9. Commit and push.
10. Add a new filter functionality to filter according to the "subtype". Be careful that a card can have several subtypes, and thus the subtype is a JSON array of string values. For this filter, we consider that the filter is satisfied if the predicate is satisfied by *one of the subtype*.
11. Test this new filter, commit and push.
12. This filter design allows us to straightforwardly compose different filters, thus allowing to model the query mentioned above: *get the cards with "more than 50 HP" and of type "Basic" or "Colorless"*. Add three *filter combinators* for conjunction, disjunction and negation of filters, e.g., `static Predicate<JSONObject> and(Predicate<JSONObject> pred1, Predicate<JSONObject> pred2)`.
13. Test all combinators, commit and push.

Exercise 4 – Iterator design pattern

We can successfully filter the cards, but we have no way to iterate over (and for instance, print) a view of the deck. It would be so nice to be able to write something like:

```
for(JSONObject card : deckView) {
  System.out.println(card.getString("name"));
}
```

Fortunately, it is possible to use this kind of for-loop on our own objects, if they implement the `Iterable` interface.

1. Add an inner class (so a class inside `DeckView`):

   ```
   private class ViewIterator implements Iterator<JSONObject> {
     private int currentIndex;
     //...
   }
   ```

   An iterator is like a cursor over a collection. You need to override the methods `hasNext` and `next` of the class `Iterator`, see the documentation. Note that inside an inner class, you have access to the attribute of the parent class.

2. Implement the interface `Iterable<JSONObject>` in the class `DeckView` and override the method `iterator()` (spoiler: this is only one line of code).
3. Rewrite the method `toStringID()` to use the for-loop mentioned above (and thus iterators).
4. Test, commit and push.

Exercise 5 – Report

Write the report directly in the file `README.md` using the Markdown format. For each of your explanations, you must cite your sources, using proper quoting if needed. To explain something, it is always best to illustrate it with concrete examples, but you must base your examples on this laboratory (e.g., don't illustrate a principle or design pattern using a generic "Cat and Dog" class diagram extracted from the web). Also, don't write something you don't understand, use your own words. Add three sections to the README:

- "Open-closed and single responsibility principles": briefly explain these principles and why the modifications we made in 3.7 improves our design regarding these principles. Is there a limit to our design regarding these principles?
- "Composite design pattern": Study this design pattern and explain where it occurs in our application (hint: this is connected to our usage of the `Predicate` class). Draw a UML diagram of this pattern using our classes and/or the classes we use.
- "Iterator design pattern": Same as previous question for this pattern.