Programming 1

Christian Grévisse (christian.grevisse@uni.lu)
Gilles Neyens, Steffen Rothkugel (gilles.neyens@uni.lu,steffen.rothkugel@uni.lu)

Lab 9 – Interfaces, Exceptions, Enumerations & Javadoc

Exercise 1 – Music Player

#Interface

Based on the solutions from Lab 8 – Exercise 2, extend the small music player program. When playing a playlist, it shall be possible to indicate the way how the songs to be played shall be sorted. Therefore, implement the hierarchy indicated in figure 1.

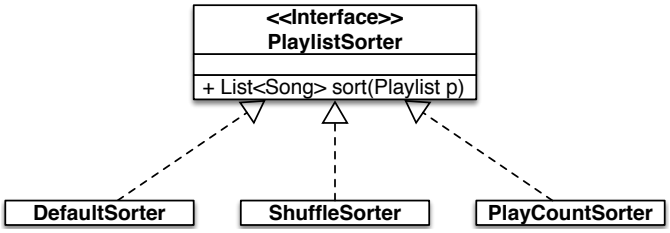

Figure 1 – `PlaylistSorter` interface and implementing classes

The `sort` method returns a list of the playlist's songs, ordered in a specific way:

Default Sorter Songs are played as they were inserted in the playlist

Shuffle Sorter The order of the songs gets shuffled. *Hint:* Use the `Collections.shuffle` method.

Play Count Sorter Songs are played in decreasing order of their play count. *Hint:* You might have the `Song` class implement the generic `Comparable` interface.

In order to "convert" the initial set into a list to be ordered, you may rely on the constructor of `ArrayList` that takes a `Collection` parameter.

Finally, extend the main program to run playlists with different sorters. Make sure the initial playlist is untouched by the sorters.

Exercise 2 – Space Combat

#Interface

In this task, you will develop a basic Star Trek-inspired space combat scenario with different game objects. These game objects may have different properties, such as:

Attackable Can be attacked and suffer damage.

Repairable Can be repaired.

Consumable Resources can be deducted and returned to the consumer. The type of resource shall be generic.

Cloaking Device Can become invisible.

The different game objects are:

Crystals have energy that can be consumed once.

Starships have an energy and shield level, as well as a name. They can be attacked and repaired. They can also attack other attackable game objects, if their energy level permits. The energy used in an attack is equal to its damage impact. Energy levels can be restored by consuming crystals. Obviously, a ship should not be able to attack itself. When attacked, the damage impact decreases the shield levels. Once the shields reach 0, the ship is destroyed. Repairs performed on a starship restore the shield levels.

Klingon starships While the above description is valid for Federation starships, ships of the Klingon empire have a cloaking device. They can only attack or be attacked when they are uncloaked (i.e. visible).

Outposts have a name. They can repair repairable objects (such as starships). Under attack, they are immediately destroyed.

Design and implement classes and interfaces to realize this scenario. Feel free to extend this scenario with your own game objects and properties. Also write a main program to test your implementation.

Exercise 3 – Access Control List #programming:CheckedException #Try #Catch #os:AccessControlList

In Operating Systems 1, we have seen the concept of *Access Control Lists (ACL)*, a common way to manage a list of subjects together with the permissions they have to access a certain object, such as for instance a file or a folder.

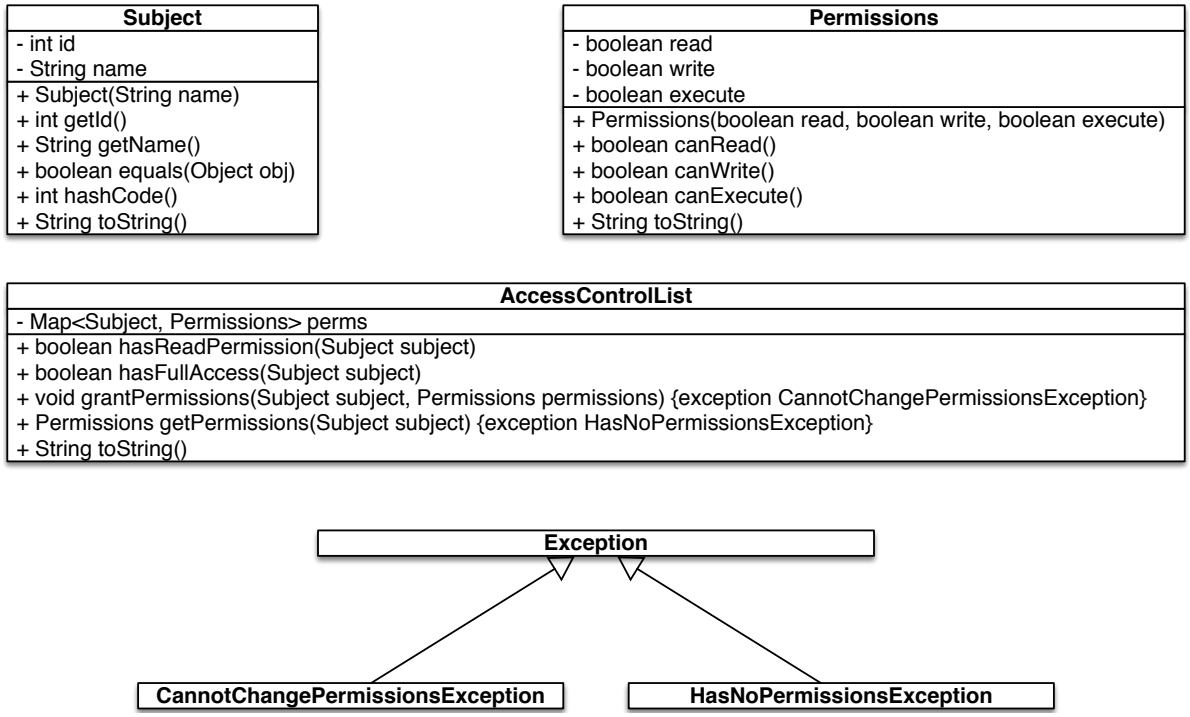

Figure 2 – `AccessControlList` and related classes

Implement the concept of ACL according to figure 2 and the guidelines below.

- The attribute `id` of a `Subject` should be auto-generated, assigning a unique number, starting with the value 1001. Implement the method `equals` such that two `Subjects` are considered equal if they have the same `id`. Use the `id` as return value of the method `hashCode`.
- Use the type `Map<Subject, Permissions>` for the attribute `perms` of the class `AccessControlList`. Do not forget to create an empty map in the constructor, using the class `HashMap`. Add and throw exceptions as specified in the class diagram above. For instance, `grantPermissions` should throw a `CannotChangePermissionsException` in case the subject has already some permissions assigned. Similarly, the method `getPermissions` should throw a `HasNoPermissionsException` in case no permissions are assigned to the subject. Please note that the convenience methods `hasReadPermission` and `hasFullAccess` should simply return `false` in case the subject has no permissions assigned.

You may find the methods `containsKey`, `get`, and `put` of the class `Map` particularly useful. Please feel free to consult the official Java 8 API documentation for details.

- Create exception classes for `CannotChangePermissionsException` and `HasNoPermissionsException`. Both classes should represent *checked exceptions*, i.e. they should extend the class `Exception`. For reasons of simplicity, let Eclipse create a `@SuppressWarnings("serial")` annotation to suppress a compiler warning related to a missing `serialVersionUID` field.
- Write a main program to test your implementation, particularly the exception handling. Insert matching `try-catch` clauses with the help of Eclipse. For the sake of simplicity, simply print a message in case some exception is thrown.

Exercise 4 – Parsing Command Line Arguments

#programming:UncheckedException #Try #Catch

In Lab 5 – Exercise 8, we learned how to use command line arguments in the `main` method. We used `Integer.parseInt()` to parse a string as an integer.

1° In the Java 8 API documentation, lookup which exception can be thrown by this method. Is it a checked or an unchecked exception? What implications does this have wrt. the responsibility of the developer?

2° Reimplement this program by not assuming anymore that only integer numbers would be passed. In case any of the passed arguments is not an integer, stop calculating the sum and output an error message.

Exercise 5 – Zero Division

#programming:UncheckedException #Try #Catch

Write a method that takes two `double` parameters and returns the quotient of both numbers. Remember that division by zero is not allowed. Handle this case by throwing an appropriate exception. Write a `main` method to test your implementation.

Exercise 6 – Multiple Exceptions

#programming:UncheckedException #Try #Catch

Consider the following code snippet:

```
1  public class MultipleExceptions {
2
3    public static void main(String[] args) {
4
5      String[] teachers = new String[] { "Denis", "Steffen", "Gilles", "Christian" };
6
7      int m = 4;
8      int n = 10;
9
10     System.out.println(teachers[m].charAt(n));
11   }
12 }
```

1° Depending on the values of *m* and *n*, what exceptions are susceptible to be thrown at line 10? What happens if this code is executed like that?

2° Rewrite this code with a `catch`-block for each exception that might be thrown.

3° Rewrite this code with a single `catch`-block by benefiting from polymorphism. *Hint:* Lookup which common superclass the exceptions caught in the previous step have.

4° Rewrite this code with the same set of different exceptions as in the second step, but in a single `catch`-block.

✎ **Multi-catch**

You can catch several exceptions in a single `multi-catch`-block:

```
1 catch (Type1 | Type2 | Type3 e) { ... }
```

Exercise 7 – Recipe #Enum #programming:CheckedException #Try #Catch

Write a class `Ingredient` which has a name, a quantity and a unit. The unit can be modeled as an `enum`. A `Recipe` takes a name as well as a list of ingredients. It shall be possible to print all the ingredients of a recipe with their name, quantity and unit.

A possible output could be:

```
Recipe for Pancakes:
- Flour: 0.2 kg
- Sugar: 2.0 tbsp.
- Eggs: 3.0
- Milk: 0.5 l
```

In addition, provide a method which takes the name of an ingredient potentially included in the recipe and returns the corresponding `Ingredient` object. Do not forget to implement the `equals` and `hashCode` method of `Ingredient`. If the ingredient was not found, throw an `IngredientNotFoundException`. This exception shall encapsulate the name of the ingredient that was not found, such that you can use it when catching the exception.

Finally, write a main program to test your implementation.

Exercise 8 – Javadoc

Write and generate the Javadoc for the classes in exercise 7.

✎ Javadoc

Javadoc is a generator for API documentation for the Java language. Based on specially formatted Javadoc comments in code, HTML pages are generated. In fact, the official Java 8 API documentation is also based on Javadoc! There exists a command line tool, but many IDEs, such as Eclipse, also integrate the generation of Javadoc.

For instance, before an already written method stub, you can write `/**` and press ⏎, and a set of Javadoc tags will be listed, such that you only have to complete the description.

```
1  package lu.uni.programming1.lab9.javadoc;
2
3  /**
4   * This interface defines ...
5   * @author Christian
6   * @version 1.0
7   * @since 0.1
8   */
9  public interface JavadocExample {
10
11   /**
12    * This method is responsible for ...
13    * @param bar Indicates whether or not ...
14    * @return The value of ...
15    * @throws IllegalArgumentException This may happen when ...
16    * @deprecated Should not be used anymore because ...
17    */
18   public int foo(boolean bar) throws IllegalArgumentException;
19 }
```

To finally generate the HTML pages, Project > Generate Javadoc... and follow the instructions in the wizard.

The generated HTML pages for the above example can be found at `https://coast.uni.lu/teaching/programming1/javadocexample/`.

For further information on Javadoc, the available tags and options, as well as documentation styles, please refer to the following resources:

- Wikipedia (is a good starting point, but nothing more)
- How to Write Doc Comments for the Javadoc Tool
- Javadoc Tool Reference