CS2030 Lab#4: Class Roster

Topic Coverage

- Inheritance
- Overriding
- Interface
- Generic class
- HashMap

Problem Description

Task

Your task is to read in a roster of students, the modules they take, the assessments they have completed, and the grade for each assessment. Then, given a query consisting of a triplet: a student, a module, and an assessment, retrieve the corresponding grade.

For instance, if the input is:

```
Steve CS1010 Lab3 A
Steve CS1231 Test A+
Bruce CS2030 Lab1 C
```

and the query is `Steve CS1231 Test`, the program should print `A+`.

In our scenario, a roster has zero or more students; a student takes zero or more modules, a module has zero or more assessments, and each assessment has exactly one grade. Each of these entities can be uniquely identified by a string.

This lab also involves using the `HashMap` class from Java Collections.

Preamble

`Map`

`Map<K,V>` is a generic interface from the Java Collection Framework, the implementation of which is useful for storing a collection of items and retrieving an item. It maintains a map (aka dictionary) between keys (of type `K`) and values (of type `V`). The two core methods are (i) `put`, which stores a (key, value) pair into the map, and (ii) `get`, which returns the value associated with a given key if the key is found or returns `null` otherwise.

The following examples show how the `Map<K,V>` interface and its implementation `HashMap<K,V>` can be used.

```
jshell> Map<String,Integer> map = new HashMap<String,Integer>();
jshell> map.put("one", 1);
$.. ==> null
jshell> map.put("two", 2);
$.. ==> null
jshell> map.put("three", 3);
$.. ==> null
jshell> map.get("one")
$.. ==> 1
jshell> map.get("four")
$.. ==> null
jshell> map.entrySet()
$.. ==> [one=1, two=2, three=3]
jshell> for (Map.Entry<String,Integer> e: map.entrySet()) {
   ...>  System.out.println(e.getKey() + ":" + e.getValue());
   ...> }
one:1
two:2
three:3
```

Level 1

`Assessment` class and `Keyable` interface

We shall start by writing the `Assessment` class that implements the following `Keyable` interface.

```
interface Keyable {
    String getKey();
}
```

Include a `getGrade` method that returns the grade of an assessment.

```
jshell> new Assessment("Lab1", "B")
$.. ==> {Lab1: B}
jshell> new Assessment("Lab1", "B").getGrade()
$.. ==> "B"
jshell> new Assessment("Lab1", "B").getKey()
$.. ==> "Lab1"
jshell> /exit
```

Check the format correctness of the output by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q your_java_files_in_bottom-up_dependency_order < test1.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 2

`Module` class

Write the `Module` class to store (via the `put` method) the assessments of a module in a map for easy retrieval as part of answering queries. A module can have zero or more assessments, with each assessment having a title as a key — a unique identifier.

```
jshell> new Module("CS2040")
$.. ==> CS2040: {}
jshell> new Module("CS2040").getKey();
$.. ==> "CS2040"
jshell> new Module("CS2040").put(new Assessment("Lab1", "B"))
$.. ==> CS2040: {{Lab1: B}}
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+"))
$.. ==> CS2040: {{Lab1: B}, {Lab2: A+}}
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+")).get("Lab1")
$.. ==> {Lab1: B}
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+")).get("Lab2")
$.. ==> {Lab2: A+}
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+")).get("Lab3")
$.. ==> null
jshell> /exit
```

Check the format correctness of the output by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q your_java_files_in_bottom-up_dependency_order < test2.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 3

`Student` class

Write a `Student` class that stores the modules he/she reads in a map via the `put` method. A student can read zero or more modules, with each module having a unique module code as its key.

```
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> {Lab1: B}
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").getGrade()
$.. ==> "B"
jshell> new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Tony: {CS2040: {{Lab1: B}}}
jshell> new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B"))).get("CS2040")
$.. ==> CS2040: {{Lab1: B}}
jshell> Student natasha = new Student("Natasha");
jshell> natasha.put(new Module("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Natasha: {CS2040: {{Lab1: B}}}
jshell> natasha.put(new Module("CS2030").put(new Assessment("PE", "A+")).put(new Assessment("Lab2", "C")))
$.. ==> Natasha: {CS2030: {{Lab2: C}, {PE: A+}}, CS2040: {{Lab1: B}}}
jshell> Student tony = new Student("Tony");
jshell> tony.put(new Module("CS1231").put(new Assessment("Test", "A-")))
$.. ==> Tony: {CS1231: {{Test: A-}}}
jshell> tony.put(new Module("CS2100").put(new Assessment("Test", "B")).put(new Assessment("Lab1", "F")))
$.. ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Lab1: F}, {Test: B}}}
jshell> /exit
```

Check the format correctness of the output by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q your_java_files_in_bottom-up_dependency_order < test3.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 4

Generic class `KeyableMap`

You will notice that the implementations of the `Student` and `Module` classes are very similar. Hence, by applying the *abstraction principle*, write a generic class `KeyableMap` to reduce the duplication.

Hint: `KeyableMap<V>` is a generic class that wraps around a `String` key (i.e. implements `Keyable`) and a `Map<String,V>`. `KeyableMap` models an entity that contains a map, but is also itself contained in another container (e.g. a student contains a map of modules but could be contained in a roster). The parameter type `V` is the type of the value of items stored in the map; `V` must be a subtype of `Keyable`.

The class `KeyableMap` is a mutable class -- we made this decision since the `Map` implementation in Java Collection Framework is mutable. `KeyableMap` provides two core methods:

- `get(String key)` which returns the item with the given `key`;
- `put(V item)` which adds the key-value pair (`item.getKey()`,`item`) into the map. The `put` method returns a `KeyableMap`. To avoid type mismatch when chaining `put` method calls together, each sub-class of `KeyableMap` should override the `put` method from `KeyableMap` and change the return type to the corresponding sub-classes. E.g., `Student` should override `put` to return a `Student` through a type-cast (which is guaranteed to be safe). Moreover, how do we restrict the classes bound to type `V` to be able to invoke the `getKey` method? The trick is to define the type parameter of `Keyable` as follows:

  ```
  class KeyableMap<V extends Keyable> implements Keyable {
      ...
  }
  ```

```
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> {Lab1: B}
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").getGrade()
$.. ==> "B"
jshell> new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Tony: {CS2040: {{Lab1: B}}}
jshell> new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B"))).get("CS2040")
$.. ==> CS2040: {{Lab1: B}}
jshell> Student natasha = new Student("Natasha");
jshell> natasha.put(new Module("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Natasha: {CS2040: {{Lab1: B}}}
jshell> natasha.put(new Module("CS2030").put(new Assessment("PE", "A+")).put(new Assessment("Lab2", "C")))
$.. ==> Natasha: {CS2030: {{Lab2: C}, {PE: A+}}, CS2040: {{Lab1: B}}}
jshell> Student tony = new Student("Tony");
jshell> tony.put(new Module("CS1231").put(new Assessment("Test", "A-")))
$.. ==> Tony: {CS1231: {{Test: A-}}}
jshell> tony.put(new Module("CS2100").put(new Assessment("Test", "B")).put(new Assessment("Lab1", "F")))
$.. ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Lab1: F}, {Test: B}}}
jshell> new Module("CS1231").put(new Assessment("Test", "A-")) instanceof KeyableMap
$.. ==> true
jshell> new Student("Tony").put(new Module("CS1231")) instanceof KeyableMap
```

```
$.. ==> true
jshell> /exit
```

Check the format correctness of the output by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q your_java_files_in_bottom-up_dependency_order < test4.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 5

`Roster` class

Now you are ready to create a roster. Define a `Roster` class that stores the students in a map via the `put` method. A roster can have zero or more students, with each student having a unique id as its key. Once again, notice the similarities between `Roster`, `Student` and `Module`.

Define a method called `getGrade` in `Roster` to answer the query from the user. The method takes in three `String` parameters, corresponds to the student id, the module code, and the assessment title, and returns the corresponding grade.

In cases where there are no such student, or the student does not read the given module, or the module does not have the corresponding assessment, then output `No such record` followed by details of the query.

```
jshell> Student natasha = new Student("Natasha");
jshell> natasha.put(new Module("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Natasha: {CS2040: {{Lab1: B}}}
jshell> natasha.put(new Module("CS2030").put(new Assessment("PE", "A+")).put(new Assessment("Lab2", "C")))
$.. ==> Natasha: {CS2030: {{Lab2: C}, {PE: A+}}, CS2040: {{Lab1: B}}}
jshell> Student tony = new Student("Tony");
jshell> tony.put(new Module("CS1231").put(new Assessment("Test", "A-")))
$.. ==> Tony: {CS1231: {{Test: A-}}}
jshell> tony.put(new Module("CS2100").put(new Assessment("Test", "B")).put(new Assessment("Lab1", "F")))
$.. ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Lab1: F}, {Test: B}}}
jshell> Roster roster = new Roster("AY1920").put(natasha).put(tony)
jshell> roster
roster ==> AY1920: {Natasha: {CS2030: {{Lab2: C}, {PE: A+}}, CS2040: {{Lab1: B}}}, Tony: {CS1231: {{Test: A-}}, CS2100: {{Lab1: F}, {Test: B}}}}
jshell> roster.get("Tony").get("CS1231").get("Test").getGrade()
$.. ==> "A-"
jshell> roster.get("Natasha").get("CS2040").get("Lab1").getGrade()
$.. ==> "B"
jshell> roster.get("Tony").get("CS1231").get("Exam")
$.. ==> null
jshell> roster.get("Steve")
$.. ==> null
jshell> roster.getGrade("Tony","CS1231","Test")
$.. ==> "A-"
jshell> roster.getGrade("Natasha","CS2040","Lab1")
$.. ==> "B"
jshell> roster.getGrade("Tony","CS1231","Exam");
$.. ==> "No such record: Tony CS1231 Exam"
jshell> roster.getGrade("Steve","CS1010","Lab1");
$.. ==> "No such record: Steve CS1010 Lab1"
jshell> new Roster("AY1920") instanceof Keyable
$.. ==> true
jshell> new Roster("AY1920").put(new Student("Tony")) instanceof Keyable
$.. ==> true
jshell> /exit
```

Check the format correctness of the output by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ jshell -q your_java_files_in_bottom-up_dependency_order < test5.jsh
```

Check your styling by issuing the following

```
$ checkstyle *.java
```

Level 6

The `Main` class

Now use the classes that you have built and write a `Main` class to solve the following:

Read the following from the standard input:

- The first token read is an integer *N*, indicating the number of records to be read.
- The subsequent inputs consist of *N* records, each record consists of four words, separated by one or more spaces. The four words correspond to the student id, the module code, the assessment title, and the grade, respectively.
- The subsequent inputs consist of zero or more queries. Each query consists of three words, separated by one or more spaces. The three words correspond to the student id, the module code, and the assessment title.

For each query, if a match in the input is found, print the corresponding grade to the standard output. Otherwise, print "No Such Record:" followed by the three words given in the query, separated by exactly one space.

See sample input and output below. Inputs are underlined.

```
$ java Main
12
Jack   CS2040 Lab4    B
Jack   CS2040 Lab6    C
Jane   CS1010 Lab1    A
Jane   CS2030 Lab1    A+
Janice CS2040 Lab1    A+
Janice CS2040 Lab4    A+
Jim    CS1010 Lab9    A+
Jim    CS2010 Lab1    C
Jim    CS2010 Lab2    B
Jim    CS2010 Lab8    A+
Joel   CS2030 Lab3    C
Joel   CS2030 Midterm A
Jack   CS2040 Lab4
Jack   CS2040 Lab6
Janice CS2040 Lab1
```

```
Janice CS2040 Lab4
Joel   CS2030 Midterm
Jason  CS1010 Lab1
Jack   CS2040 Lab5
Joel   CS2040 Lab3
B
C
A+
A+
A
No such record: Jason CS1010 Lab1
No such record: Jack CS2040 Lab5
No such record: Joel CS2040 Lab3
```

Check the format correctness of the output by running the following on the command line:

```
$ javac -Xlint:rawtypes *.java
$ java Main < test6.in
```

Check your styling by issuing the following

```
$ checkstyle *.java
```