CS2030 (2310) Lab #5: Class Roster

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Class Roster

Topic Coverage

- Generic Class
- Optional
- Functional Interface and Lambda

Problem Description

In each semester, a typical university student reads a number of courses, each comprising a number of graded assessments. As part of administrative purposes, it is useful to have an application where given a roster of students and some query consisting consisting of a triplet *(student, course, assessment)*, the application would proceed to retrieve the corresponding grade.

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

You are given the `ImmutableMap` generic class which makes use of the *immutable delegation pattern* by wrapping over a mutable `Map` from the Java Collections Framework. It is useful for storing a collection of items and retrieving an item by maintaining a map (aka dictionary) between keys (of type `K`) and values (of type `V`). The two core methods of `ImmutableMap` are:

- `put` which stores a (key, value) pair into the map, and
- `get` which returns an `Optional` containing the value associated with a given key if the key is found, or returns `Optional.empty` otherwise.

The following examples show how `ImmutableMap<K,V>` can be used.

```
jshell> ImmutableMap<String,Integer> map = new ImmutableMap<String,Integer>();
map ==> {}

jshell> map = map.put("one", 1).put("two", 2).put("three", 3);
map ==> {one=1, two=2, three=3}

jshell> map.get("one")
$.. ==> Optional[1]

jshell> map.get("four")
$.. ==> Optional.empty

jshell> for (Map.Entry<String,Integer> e: map) {
   ...>    System.out.println(e.getKey() + ":" + e.getValue());
   ...> }
```

```
one:1
two:2
three:3
```

Iterating the elements of a map is a little involved since Java's `Map` interface is not `Iterable`, and hence one cannot perform the usual looping by using the enhanced for loop as such:

```
for (Map<String, Integer> e : map) {
   System.out.println(e);
}
```

Instead, the `entrySet` method of `Map` is called to return a "view" of the map in terms of a `Set`, and since `Set` is `Iterable`, we loop the elements of the set instead.

The `equals` and `hashCode` Methods

Although not necessary for this exercise, it is interesting to know how key-value pairs behave when added to `ImmutableMap` (or any `Map`).

```
jshell> ImmutableMap<String,Integer> map = new ImmutableMap<String,Integer>()
map ==> {}

jshell> map = map.put("one", 1)
map ==> {one=1}

jshell> map = map.put("one", 11)
map ==> {one=11}
```

Notice the existing key-value pair `{one=1}` is replaced by `{one=11}`. This is because in the contract of a `Map`, a key should be associated to a single unique value. As such no duplicated keys is allowed.

Now let's create the following class `A` and store key-value pairs into the immutable map.

```
jshell> class A {
   ...>         private final int x;
   ...>         A(int x) { this.x = x;}
   ...>         public String toString() {
   ...>             return "A:" + this.x;
   ...>         }
   ...> }
|  created class A

jshell> ImmutableMap<A,Integer> m = new ImmutableMap<A,Integer>()m ==> {}

jshell> m = m.put(new A(1), 1)
m ==> {A:1=1}

jshell> m = m.put(new A(1), 11)
m ==> {A:1=1, A:1=11}
```

It seems like we are allowing duplicate keys to be stored. Adding a key-value pair requires "hashing" the key to compute the index of the hash map to which the key-value pair is stored. This hashing is provided by the `hashCode()` method which returns an integer.

Moreover, for each key-value pair, after the index is computed, the `equals` method is used to check if there is an existing key-value pair with the same key. Why is this necessary? Because different keys can be hashed to the same index!

As such, when considering an object to be used as the key for a map, we need to ensure that both the `hashCode` as well as the `equals` method are defined. The contract of a `hashCode` is that two objects which are deemed the same via the `equals` method must return the same value after calling `hashCode()`.

```
jshell> class A {
   ...>         private final int x;
   ...>         A(int x) { this.x = x;}
   ...>         public boolean equals(Object obj) {
   ...>             return (obj instanceof A a) && this.x == a.x; // simplified equals method
   ...>         }
   ...>         public int hashCode() {
   ...>             return Objects.hash(this.x); // using Objects.hash method
   ...>         }
   ...>         public String toString() {
   ...>             return "A:" + this.x;
```

```
   ...>         }
   ...> }
|  replaced class A

jshell> m = new ImmutableMap<A,Integer>()m ==> {}

jshell> m = m.put(new A(1), 1)
m ==> {A:1=1}

jshell> m = m.put(new A(1), 11)
m ==> {A:1=11}
```

Indeed, the `String` class has both `equals` and `hashCode` methods defined appropriately and hence we need not worry for the rest of this exercise.

Task

By making use of the `ImmutableMap`, write an application to read in a roster of students, the courses they take, the assessments they have completed, and the grade for each assessment. Then, given a query consisting of a triplet: a student, a course, 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+`.

Level 1

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

```
interface Keyable {
    public 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") instanceof Keyable
$.. ==> true

jshell> new Assessment("Lab1", "B").getGrade()
$.. ==> "B"

jshell> new Assessment("Lab1", "B").getKey()
$.. ==> "Lab1"
```

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

```
jshell> new Course("CS2040")
$.. ==> CS2040: {}

jshell> new Course("CS2040") instanceof Keyable
$.. ==> true

jshell> new Course("CS2040").getKey();
$.. ==> "CS2040"

jshell> new Course("CS2040").put(new Assessment("Lab1", "B"))
$.. ==> CS2040: {{Lab1: B}}

jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+"))
$.. ==> CS2040: {{Lab1: B}, {Lab2: A+}}

jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+")).
   ...> get("Lab1")
```

```
$.. ==> Optional[{Lab1: B}]

jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+")).
   ...> get("Lab2")
$.. ==> Optional[{Lab2: A+}]

jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).put(new Assessment("Lab2","A+")).
   ...> get("Lab3")
$.. ==> Optional.empty
```

Level 2

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

```
jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> Optional[{Lab1: B}]

jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Tony: {CS2040: {{Lab1: B}}}

jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040")
$.. ==> Optional[CS2040: {{Lab1: B}}]

jshell> Student natasha = new Student("Natasha");
natasha ==> Natasha: {}

jshell> natasha = natasha.put(new Course("CS2040").put(new Assessment("Lab1", "B")))
natasha ==> Natasha: {CS2040: {{Lab1: B}}}

jshell> natasha.put(new Course("CS2030").put(new Assessment("PE", "A+")).
   ...> put(new Assessment("Lab2", "C")))
$.. ==> Natasha: {CS2040: {{Lab1: B}}, CS2030: {{PE: A+}, {Lab2: C}}}

jshell> Student tony = new Student("Tony");
tony ==> Tony: {}

jshell> tony = tony.put(new Course("CS1231").put(new Assessment("Test", "A-")))
tony ==> Tony: {CS1231: {{Test: A-}}}

jshell> tony.put(new Course("CS2100").put(new Assessment("Test", "B")).
   ...> put(new Assessment("Lab1", "F")))
$.. ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Test: B}, {Lab1: F}}}
```

Level 3

You will notice that the implementations of the `Student` and `Course` 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 an immutable map, but is also itself contained in another container (e.g. a student contains a map of courses but could be contained in a roster). The parameter type `V` is the type of the value of items stored in the immutable map; `V` must be a subtype of `Keyable`.

The class `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 immutable map. The `put` method returns a `KeyableMap`. 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 Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> Optional[{Lab1: B}]

jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B")))
$.. ==> Tony: {CS2040: {{Lab1: B}}}
```

```
jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040")
$.. ==> Optional[CS2040: {{Lab1: B}}]

jshell> Student natasha = new Student("Natasha");
natasha ==> Natasha: {}

jshell> natasha = natasha.put(new Course("CS2040").put(new Assessment("Lab1", "B")))
natasha ==> Natasha: {CS2040: {{Lab1: B}}}

jshell> natasha.put(new Course("CS2030").put(new Assessment("PE", "A+")).
   ...> put(new Assessment("Lab2", "C")))
$.. ==> Natasha: {CS2040: {{Lab1: B}}, CS2030: {{PE: A+}, {Lab2: C}}}

jshell> Student tony = new Student("Tony");
tony ==> Tony: {}

jshell> tony = tony.put(new Course("CS1231").put(new Assessment("Test", "A-")))
tony ==> Tony: {CS1231: {{Test: A-}}}

jshell> tony.put(new Course("CS2100").put(new Assessment("Test", "B")).
   ...> put(new Assessment("Lab1", "F")))
$.. ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Test: B}, {Lab1: F}}}

jshell> new Course("CS1231").put(new Assessment("Test", "A-")) instanceof KeyableMap
$.. ==> true

jshell> new Student("Tony").put(new Course("CS1231")) instanceof KeyableMap
$.. ==> true
```

Level 4

Notice that method chains below result in compilation errors:

```
jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").getGrade()
|  Error:
|  cannot find symbol
|    symbol:   method getGrade()
|  new Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").getGrade()
|  ^-------------------------------------------------------------------------^

jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040" ).get("Lab1")
|  Error:
|  method get in class java.util.Optional<T> cannot be applied to given types;
|    required: no arguments
|    found: java.lang.String
|    reason: actual and formal argument lists differ in length
|  get("CS2040" ).get("Lab1")
|  ^----------------^
```

This is because the `Optional` class does not have a `getGrade()` or `get(String)` method defined (although it does define a `get()` method which, other than for debugging purposes, should typically be avoided). Rather than chaining in the usual way, we do it through a `map` or `flatMap`. Let's start with `map`.

```
jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> Optional[{Lab1: B}]

jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").getGrade()
|  Error:
|  cannot find symbol
|    symbol:   method getGrade()
|  new Course("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").getGrade()
|  ^-------------------------------------------------------------------------^

jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).
   ...> get("Lab1").map(x -> x.getGrade())
$.. ==> Optional[B]
```

As expected, invoking `getGrade()` on an `Optional` results in a compilation error. However, we can perform a similar chaining effect by passing in the functionality of `getGrade` to `Optional`'s `map` method. Notice the return value is actually wrapped in another `Optional`. When using `map`, you can think of the operation as "taking the value out of the

`Optional` box, transforming it via the function passed to `map`, and wrap the transformed value back in another `Optional`".

Now this is where things start to get interesting! Look at the following:

```
jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040").map(x -> x.get("Lab1"))
$.. ==> Optional[Optional[{Lab1: B}]]
```

Observe that the return value is an `Optional` wrapped around another `Optional` that wraps around the desired value! Why is this so? The difference lies in the return type of `Assessment::getGrade` (read `getGrade` method of the `Assessment` class) and `Course::get`. The former returns a `String`, while the latter returns an `Optional`.

By passing `x -> x.getGrade()` to `map` in

```
jshell> new Course("CS2040").put(new Assessment("Lab1", "B")).
   ...> get("Lab1").map(x -> x.getGrade())
```

the transformed value is simply the grade, and this is wrapped in an Optional.

In contrast, passing `x -> x.get("Lab1")` to `map` in

```
jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040").map(x -> x.get("Lab1"))
```

results in a transformed value of `Optional`. And this transformed value is wrapped around another `Optional` via the `map` operation!

As such, we use the `flatMap` method instead. You may think of `flatMap` as flattening the `Optionals` into a single one.

```
jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040" ).flatMap(x -> x.get("Lab1"))
$.. ==> Optional[{Lab1: B}]

jshell> new Student("Tony").put(new Course("CS2040").put(new Assessment("Lab1", "B"))).
   ...> get("CS2040" ).flatMap(x -> x.get("Lab1")).map(x -> x.getGrade())
$.. ==> Optional[B]
```

Now you are ready to create a roster.

Define a `Roster` class that stores the students in an immutable 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 `Course`.

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 course 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 course, or the course does not have the corresponding assessment, then output `No such record` followed by details of the query. Here, you might find `Optional::orElse` useful.

```
jshell> Student natasha = new Student("Natasha");
natasha ==> Natasha: {}

jshell> natasha = natasha.put(new Course("CS2040").put(new Assessment("Lab1", "B")))
natasha ==> Natasha: {CS2040: {{Lab1: B}}}

jshell> natasha = natasha.put(new Course("CS2030").put(new Assessment("PE", "A+")).
   ...> put(new Assessment("Lab2", "C")))
natasha ==> Natasha: {CS2040: {{Lab1: B}}, CS2030: {{PE: A+}, {Lab2: C}}}

jshell> Student tony = new Student("Tony");
tony ==> Tony: {}

jshell> tony = tony.put(new Course("CS1231").put(new Assessment("Test", "A-")))
tony ==> Tony: {CS1231: {{Test: A-}}}

jshell> tony = tony.put(new Course("CS2100").put(new Assessment("Test", "B")).
   ...> put(new Assessment("Lab1", "F")))
tony ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Test: B}, {Lab1: F}}}

jshell> Roster roster = new Roster("AY1920").put(natasha).put(tony)
roster ==> AY1920: {Natasha: {CS2040: {{Lab1: B}}, CS2030: { ... : {{Test: B}, {Lab1: F}}}}
```

```
jshell> roster
roster ==> AY1920: {Natasha: {CS2040: {{Lab1: B}}, CS2030: {{PE: A+}, {Lab2: C}}},
Tony: {CS1231: {{Test: A-}}, CS2100: {{Test: B}, {Lab1: F}}}}

jshell> roster.get("Tony").flatMap(x -> x.get("CS1231")).flatMap(x -> x.get("Test")).
   ...> map(x -> x.getGrade())
$.. ==> Optional[A-]

jshell> roster.get("Natasha").flatMap(x -> x.get("CS2040")).flatMap(x -> x.get("Lab1")).
   ...> map(x -> x.getGrade())
$.. ==> Optional[B]

jshell> roster.get("Tony").flatMap(x -> x.get("CS1231")).flatMap(x -> x.get("Exam")).
   ...> map(x -> x.getGrade())
$.. ==> Optional.empty

jshell> roster.get("Steve").flatMap(x -> x.get("CS1010")).flatMap(x -> x.get("Lab1")).
   ...> map(x -> x.getGrade())
$.. ==> Optional.empty

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").put(new Student("Tony")) instanceof KeyableMap
$.. ==> true
```

Level 5

Include the `add` method in the `Roster` class that takes in the student id, the course code, the assessment title and the grade, so as to update the roster as shown in the sample run below:

```
jshell> Roster roster = new Roster("AY1920")
roster ==> AY1920: {}

jshell> roster = roster.add("Natasha", "CS2040", "Lab1" , "B")
roster ==> AY1920: {Natasha: {CS2040: {{Lab1: B}}}}

jshell> roster.add("Tony", "CS1231", "Test", "A-")
$.. ==> AY1920: {Natasha: {CS2040: {{Lab1: B}}}, Tony: {CS1231: {{Test: A-}}}}

jshell> roster.add("Natasha", "CS1231", "Test", "A-")
$.. ==> AY1920: {Natasha: {CS2040: {{Lab1: B}}, CS1231: {{Test: A-}}}}

jshell> roster.add("Natasha", "CS2040", "Test", "A-")
$.. ==> AY1920: {Natasha: {CS2040: {{Lab1: B}, {Test: A-}}}}

jshell> roster.add("Natasha", "CS2040", "Lab1", "A-")
$.. ==> AY1920: {Natasha: {CS2040: {{Lab1: A-}}}}

jshell> roster.getGrade("Natasha", "CS2040", "Lab1")
$.. ==> "B"

jshell> roster.getGrade("Natasha", "CS2040", "Test")
$.. ==> "No such record: Natasha CS2040 Test"

jshell> roster.getGrade("Natasha", "CS1231", "Lab1")
$.. ==> "No such record: Natasha CS1231 Lab1"

jshell> roster.getGrade("Tony", "CS2040", "Lab1")
$.. ==> "No such record: Tony CS2040 Lab1"
```

Level 6

Now use the classes that you have built and write a `Main` class to deal with program input and output.

The input comprises the following:

- 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 course code, the assessment title, and the grade, respectively.
- The final set of 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 course 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. User input is underlined.

```
$ cat 6.in
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

$ java Main < 6.in
B
C
A+
A+
A
No such record: Jason CS1010 Lab1
No such record: Jack CS2040 Lab5
No such record: Joel CS2040 Lab3
```

Submission (Course)

Select course: CS2030 (2023/2024 Sem 1) - Programming Methodology II

Your Files:

BROWSE

SUBMIT (only .java, .c, .cpp, .h, .jsh, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.