CodeCrunch

CS2030 Lab #6: Making Things Optional

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Making Things Optional

Topic Coverage

- Optional
- Functional Interfaces

Problem Description

This lab is a continuation of the class roster lab.

So far we have been dealing with invalid retrievals such as

```
new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab2")
```

which would result in a `null` value. Furthermore, chaining methods like

```
new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B"))).get("CS203
```

would result in a `NullPointerException` being thrown due to calling `get("Lab1")` on a `null` value.

Take a look at the Java documentation of Optional to familiarize yourself with the APIs available.

Note

On CodeCrunch, we will check for any use of `null` in your code. Any occurence of the string `null` would fail the CodeCrunch test. Thus, avoid variable or method names that contain the substring `null`.

Further, we will disallow the use of methods `Optional::get`, `Optional::isPresent` and `Optional::isEmpty`, as the former could cause `NullPointerException`, while the latter is essentially the same as checking for `null`.

Level 4+

Avoiding `null` with `Optional`

Modify the `KeyableMap` generic class such that each call to `get` returns an `Optional` where `V` is a subtype of `Keyable`.

```
jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> Optional[{Lab1: 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"))).ge
$.. ==> Optional[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 Assessmer
$.. ==> 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
$.. ==> Tony: {CS1231: {{Test: A-}}, CS2100: {{Lab1: F}, {Test: B}}}
jshell> /exit
```

As you can see, the only difference is that each value returned from an invocation of the `get` method is wrapped in an `Optional`.

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+

"Chaining" `Optionals`

Chaining `Optionals` together as such

```
new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B"))).get("CS204
```

will now result in a compilation error instead. This is because the `Optional` class does not have a `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 Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1")
$.. ==> Optional[{Lab1: B}]

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

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

jshell> new Module("CS2040").put(new Assessment("Lab1", "B")).get("Lab1").map(Assessment:
$.. ==> 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` (either in the form of a lambda or method reference) 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 Module("CS2040").put(new Assessment("Lab1", "B"))).ge
$.. ==> 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 `Module::get`. The former returns a `String`, while the latter returns an `Optional`.

In `x -> x.getGrade()` (or `Assessment::getGrade`), the transformed value is simply the grade, and this is wrapped in an Optional. However, passing `x -> x.get("Lab1")` in the above code snippet 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 Module("CS2040").put(new Assessment("Lab1", "B"))).ge
$.. ==> Optional[{Lab1: B}]

jshell> new Student("Tony").put(new Module("CS2040").put(new Assessment("Lab1", "B"))).ge
$.. ==> Optional[B]
```

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. Here, you might find `Optional::orElse` useful.

```
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 Assessmer
$.. ==> 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
$.. ==> 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:
jshell> roster.get("Tony").flatMap(x -> x.get("CS1231")).flatMap(x -> x.get("Test")).map(
$.. ==> Optional[A-]
jshell> roster.get("Natasha").flatMap(x -> x.get("CS2040")).flatMap(x -> x.get("Lab1")).ma
$.. ==> Optional[B]
jshell> roster.get("Tony").flatMap(x -> x.get("CS1231")).flatMap(x -> x.get("Exam")).map(
$.. ==> Optional.empty
jshell> roster.get("Steve").flatMap(x -> x.get("CS1010")).flatMap(x -> x.get("Lab1")).map
$.. ==> 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> /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
```

Note:

- You might find `Optional::ifPresentOrElse` useful.

Compile and check your styling by issuing the following

```
$ javac -Xlint:rawtypes *.java
$ checkstyle *.java
```