CS2030 Lab #5: Immutable Map

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Immutable Map

Topic Coverage

- Generic class
- Map and LinkedHashMap

Problem Description

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 `LinkedHashMap<K,V>` can be used. We use a `LinkedHashMap` in order to preserve the order of insertions into the `Map`.

```
jshell> Map<String,Integer> map = new LinkedHashMap<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
```

As expected, the `Map` implementation in the Java Collection Framework is mutable. You will also notice that unlike `List` or `Set`, `Map` does not implement `Iterable`, and hence one cannot perform the usual looping by using the enhanced `for` loop as shown below.

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

Instead, the `entrySet` method 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.

Task

In this task you will build your own immutable version of `Map` that implements various methods to manipulate this map, by wrapping an immutable API over a mutable Map object. This pattern is sometimes known as the immutable delegation class which is simple to implement, although it comes with a performance overhead.

You are encouraged to familiarize yourself with the `Map` interface as specified by the Java API, as well as the `ImList` class that implements an immutable version of `List`.

Level 1

Create a class `ImmutableMap` with the type parameters `K` and `V` that "wraps around" an internal `java.util.Map` object.

Write a constructor `ImmutableMap()` that creates an empty map and define a `toString()` method that simple returns the string `"{}"` for an empty map. You should delegate implementation of `toString` to the internal `Map`.

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

jshell> new ImmutableMap<String, Integer>().toString()
$.. ==> "{}"
```

Level 2

Include the method `put` that takes as argument the key of type `K` followed by the value of type `V`, and adds the key-value association into the map. You should delegate the implementation of `put` to the internal map's `put` method. It is worth noting that the `put` method of `java.util.Map` returns the the previous value associated with key, or `null` if there was no mapping for the key. In our implementation, however, we shall just return the resulting map with the new mapping.

Include a `isEmpty()` method that outputs `true` if the map is empty, or `false` otherwise.

Moreover, if you have delegated the `toString()` method in the earlier level, then there is no need to modify the method further.

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

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

jshell> map.put("one", 1).isEmpty()
$.. ==> false

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

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

jshell> map
map ==> {}

jshell> map.isEmpty();
$.. ==> true

jshell> map.put("one", 1).put("two", 2).put("three", 3).put("one", 11)
$.. ==> {one=11, two=2, three=3}
```

Level 3

The `get` method in `java.util.Map` is specified to take in an `Object`, and returns the value to which the specified key is mapped, or `null` if the map contains no mapping for the key. In our implementation, we shall let

`java.util.Optional` deal with the `null` value. Specifically our `get` method should now return `Optional.empty` is there is no mapping, or the value wrapped within an `Optional` otherwise. It is now a good time to study the specifications of the `Optional` class in the Java API.

Implement additional methods to obtain the keys, values, and key-value associations in the map. Study the `keySet`, `values` and `entrySet` methods of `java.util.Map` in the Java API.

```
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> map.get(1)
$.. ==> Optional.empty

jshell> map.keySet()
$.. ==> [one, two, three]

jshell> map.values()
$.. ==> [1, 2, 3]

jshell> map.entrySet()
$.. ==> [one=1, two=2, three=3]
```

Level 4

We would like to make our `ImmutableMap` iterable by iterating through the key-value associations. Study how we made `ImList` iterable and modify `ImmutableMap` accordingly.

```
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> // iteration via enhanced for loop

jshell> for (Map.Entry<String, Integer> e : map) {
   ...>     System.out.println(e);
   ...>     System.out.println(e.getValue() + " is mapped to " + e.getKey());
   ...> }
one=1
1 is mapped to one
two=2
2 is mapped to two
three=3
3 is mapped to three

jshell> // old-style iteration

jshell> Iterator<Map.Entry<String, Integer>> iter = map.iterator()
iter ==> java.util.LinkedHashMap$LinkedEntryIterator@8bd1b6a

jshell> while (iter.hasNext()) {
   ...>     System.out.println(iter.next());
   ...> }
one=1
two=2
three=3
```

Level 5

Let's write an application for our immutable map. 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") instanceof Keyable
$.. ==> true

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

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

Next, write the `Module` class to store (via the `put` method) the assessments of a module in an immutable 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") instanceof Keyable
$.. ==> true

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+")).ge
$.. ==> Optional[{Lab1: B}]

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

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