CS2030 MySet Practice Task

Tags & Categories

Tags:

Categories: CodeCrunch, Java, Peer Learning

Related Tutorials

Task Content

CS2030 Practice MySet Exercise

Topics:

- OOP
- Generics
- Optional
- Stream

Introduction

A set as you have learned in high school is simply an unordered collection of items with no duplicates (with duplicates it's a multi-set). Some simple Set operations include the following:

- find(n) : retrieve item n from the set if it exist in the set
- insert(n) : insert item n into the set if it doesn't already exist in set
- remove(n) : remove item n from the set if it exist in the set
- union(s) : return the union of this set with another set s
- intersect(s) : return the intersection of this set with another set s

In this exercise, let's implement a simple set data structure that is supported by Generics, and able to perform the following functions:

- boolean add(E e): Adds the specified element to this set if it is not already present.
- void clear(): Removes all of the elements from this set.
- boolean contains(Object o): Returns true if this set contains the specified element.
- boolean isEmpty(): Returns true if this set contains no elements.
- boolean remove(Object o): Removes the specified element from this set if it is present.
- int size(): Returns the number of elements in this set.
- String toString(): For nicer formatting

You might have realized that the above methods match the methods in Java.util.HashSet. So you are not allowed to use Java.util.HashSet or any other implementation of Java.util.Set provided by default. Note that this exercise also forces you to use only implicit loops (think stream) and you are not allowed to use for/enhanced for/while loops etc.

Remember to:

- always compile your program files first before using jshell to test your program
- use checkstyle and javadoc comments to enhance code readability and facilitating code review
- compile frequently: `javac your_java_files`
- test whenever: `jshell -q your_java_files_in_bottom-up_dependency_order < test1.jsh`

Level 1

Bearing in mind principles of OOP, let's make a generic interface named MySet<E> that outlines the operations an implementation of MySet<E> must provide (See above for the list of operations). Implement MySet<E> using whatever underlying data structures of your choice (but not any existing implementation of Set!) and name it MySetImpl<E>. Note that we are not focusing on your ability to create the most efficient data structures, but to see if you can come up with a class design according to specification, under time constraint.

```
jshell> MySet<Integer> mySet = new MySetImpl<>();
jshell> mySet;
mySet ==> MySet contains: NOTHING!
jshell> mySet.size();
$25 ==> 0
jshell> mySet.isEmpty();
$26 ==> true
jshell> mySet.contains("test");
$27 ==> false
jshell> mySet.contains(1);
$28 ==> false
jshell> mySet.remove(1);
$29 ==> false
jshell> mySet.clear();
jshell> mySet.add(1);
$31 ==> true
jshell> mySet.add(2);
$32 ==> true
jshell> mySet.add(3);
$33 ==> true
jshell> mySet.size();
$34 ==> 3
jshell> mySet.isEmpty();
$35 ==> false
jshell> mySet.contains(1);
$36 ==> true
jshell> mySet.contains(0);
$37 ==> false
jshell> mySet.remove(0);
$38 ==> false
jshell> mySet.remove(1);
$39 ==> true
jshell> mySet.size();
$40 ==> 2
jshell> mySet;
mySet ==> MySet contains: 2, 3
jshell> /exit
```

Level 2 Varags & Exception Handling

Create a `DuplicateNotAllowedException` that extends the `Exception` class.DuplicateNotAllowedException should show the following error message when triggered: “Sets cannot have duplicates!”. Then create a static method

```
@SafeVarargs
static <E> MySet<E> of(E ...e) throws DuplicateNotAllowedException {
```

that can take in any number of variables. This makes use of Variable Arguments in Java and is specified by the three periods(...). An example input would be MySetImpl.<Integer>of(1,2,3,4,5). Note that if duplicates are found within the variable argument, the exception will be thrown.

```
jshell> MySet<Integer> mySet = new MySetImpl<>();
jshell> mySet.size();
$24 ==> 0
jshell> mySet.add(2);
$25 ==> true
jshell> mySet.add(2);
$26 ==> false
jshell> MySet<Integer> otherSet = MySetImpl.<Integer>of(1, 2, 3, 4);
jshell> MySet<Integer> anotherSet = MySetImpl.<Integer>of(1, 2, 3, 4);
jshell> try {
   ...>     MySet<Integer> lastSet = MySetImpl.<Integer>of(1, 1, 2, 3, 4);
   ...> } catch (DuplicateNotAllowedException ex) {
   ...>     System.out.println(ex.getMessage());
   ...> }
Sets cannot have duplicates!
jshell> otherSet.contains(2);
```

```
$30 ==> true
jshell> otherSet.add(5);
$31 ==> true
jshell> otherSet;
otherSet ==> MySet contains: 1, 2, 3, 4, 5
jshell> otherSet.remove(1);
$33 ==> true
jshell> /exit
```

Level 3

Let's add in a `String join(String delimiter)` method that helps in printing out the values in MySet nicely. This method will return a string that combines the string representation of the items in MySet, separated by the delimiter supplied. Note that you are not allowed to use explicit loops such as for and while loops.

```
jshell> MySet<Integer> mySet = new MySetImpl<>();
jshell> mySet.add(1);
$24 ==> true
jshell> mySet.add(2);
$25 ==> true
jshell> mySet.add(3);
$26 ==> true
jshell> mySet.join(" -> ");
$27 ==> "1 -> 2 -> 3"
jshell> mySet;
mySet ==> MySet contains: 1, 2, 3
jshell> MySet<String> otherSet;
jshell> try {
   ...>     otherSet = MySetImpl.of("have", "fun", "doing", "it");
   ...> } catch (DuplicateNotAllowedException ex) {
   ...>     System.out.println(ex.getMessage());
   ...> }
jshell> otherSet.join(" ");
$31 ==> "have fun doing it"
jshell> otherSet.join(" - ");
$32 ==> "have - fun - doing - it"
jshell> otherSet;
otherSet ==> MySet contains: have, fun, doing, it
jshell> /exit
```

Level 4

Let's add in a method that returns an Optional! Create a method `get` that takes in a predicate and return an Optional containing the value, should the predicate be true. If the predicate is false, return Optional.empty instead. Assume that there will only be one item or none of items matching the predicate.

```
jshell> MySet<Integer> mySet = new MySetImpl<>();
jshell> mySet.add(1);
$24 ==> true
jshell> mySet.add(2);
$25 ==> true
jshell> mySet.add(3);
$26 ==> true
jshell> MySet<String> otherSet;
jshell> try {
   ...>     otherSet = MySetImpl.<String>of("have", "fun", "doing", "it");
   ...> } catch (DuplicateNotAllowedException ex) {
   ...>     System.out.println(ex.getMessage());
   ...> }
jshell> mySet.get(x -> x > 2);
$29 ==> Optional[3]
jshell> mySet.get(x -> x > 2).orElse(0);
$30 ==> 3
jshell> mySet.get(x -> x > 3);
$31 ==> Optional.empty
jshell> mySet.get(x -> x > 3).orElse(0);
$32 ==> 0
jshell> mySet;
mySet ==> MySet contains: 1, 2, 3
jshell> otherSet;
otherSet ==> MySet contains: have, fun, doing, it
```

```
jshell> otherSet.get(x -> x.length() == 2);
$35 ==> Optional[it]
jshell> otherSet.get(x -> x.length() == 2).orElse("not found");
$36 ==> "it"
jshell> otherSet.get(x -> x.length() > 10);
$37 ==> Optional.empty
jshell> otherSet.get(x -> x.length() > 10).orElse("not found");
$38 ==> "not found"
jshell> /exit
```

Level 5

Sometimes we create data structures in order to support some custom behaviours. In this level, let's again practice using Java Stream by adding the following methods. Note that your implementation should not break immutability. (You need to figure out the method signatures for the following methods!)

- `duplicate` : takes in an integer, returns a List of all the items in MySet, duplicated by x copies each.
- `filter` : takes in a predicate, returns a MySet with all the items that return true for the predicate.
- `reduce` : takes in a seed and a binary operator, returns the reduced value. (Note again that you are not allowed to use a for/while loop)
- `map` : returns a MySet of all the values transformed by the mapping function provided.
- `sort` : sorts the items in MySet with the provided comparator and returns a List of them. (Set and MySet are inherently not ordered, for the purpose of this practice, we want to provide the `sort` method to return an ordered list of items for the cases in which items in MySet can be compared to each other.

```
jshell> MySet<Integer> mySet = MySetImpl.<Integer>of(1,2,3,4,5,6);
jshell>
jshell> mySet;
mySet ==> MySet contains: 1, 2, 3, 4, 5, 6
jshell>
jshell> mySet.duplicate(3).size() == 18;
$25 ==> true
jshell>
jshell> mySet.filter(x -> x % 2 == 0);
$26 ==> MySet contains: 2, 4, 6
jshell>
jshell> mySet;
mySet ==> MySet contains: 1, 2, 3, 4, 5, 6
jshell>
jshell> mySet.reduce(0, (subtotal, element) -> subtotal + element)
$28 ==> 21
jshell>
jshell> mySet.map(number -> number * 3)
$29 ==> MySet contains: 3, 6, 9, 12, 15, 18
jshell>
jshell> mySet.map(number -> number + 2)
$30 ==> MySet contains: 3, 4, 5, 6, 7, 8
jshell>
jshell> mySet.sort((x, y) -> y - x)
$31 ==> [6, 5, 4, 3, 2, 1]
jshell>
jshell> MySet<String> otherSet = MySetImpl.<String>of("a", "b", "c", "d", "e");
jshell>
jshell> otherSet.duplicate(2).size() == 10;
$33 ==> true
jshell>
jshell> otherSet.filter(x -> x == "a");
$34 ==> MySet contains: a
jshell>
jshell> otherSet.filter(x -> true);
$35 ==> MySet contains: a, b, c, d, e
jshell>
jshell> otherSet.reduce("", (partialString, element) -> partialString + element)
$36 ==> "abcde"
jshell>
jshell> otherSet.map(letter -> letter + letter)
$37 ==> MySet contains: aa, bb, cc, dd, ee
jshell>
jshell> otherSet.map(String::toUpperCase)
$38 ==> MySet contains: A, B, C, D, E
jshell>
jshell> otherSet.sort((x, y) -> y.compareTo(x))
$39 ==> [e, d, c, b, a]
```

```
jshell>
jshell> otherSet;
otherSet ==> MySet contains: a, b, c, d, e
jshell>
jshell> /exit
```

Level 6 Union and Intersect

Sets have the Union and Intersect methods that help to compare and organize the items in them. Implement the

```
MySet<E> union(MySet<E> otherSet)
MySet<E> intersect(MySet<E> otherSet)
```

methods. Union returns a new MySet which is a combination of the two (without duplicates) and intersect returns a new MySet with only the common items.

```
jshell> MySet<Integer> mySet = MySetImpl.<Integer>of(1,2,3,4,5,6);
jshell>
jshell> MySet<Integer> otherSet = MySetImpl.<Integer>of(6,2,7,4,5,8);
jshell>
jshell> mySet.union(otherSet);
$25 ==> MySet contains: 1, 2, 3, 4, 5, 6, 7, 8
jshell>
jshell> mySet;
mySet ==> MySet contains: 1, 2, 3, 4, 5, 6
jshell>
jshell> mySet.intersect(otherSet);
$27 ==> MySet contains: 2, 4, 5, 6
jshell>
jshell> mySet;
mySet ==> MySet contains: 1, 2, 3, 4, 5, 6
jshell>
jshell> mySet.union(mySet);
$29 ==> MySet contains: 1, 2, 3, 4, 5, 6
jshell>
jshell> mySet.intersect(mySet);
$30 ==> MySet contains: 1, 2, 3, 4, 5, 6
jshell>
jshell> MySet<String> anotherSet = MySetImpl.of("a", "b", "c", "d", "e");
jshell>
jshell> MySet<String> lastSet = MySetImpl.of("a", "d", "e", "q", "r");
jshell>
jshell> anotherSet.union(lastSet);
$33 ==> MySet contains: a, b, c, d, e, q, r
jshell>
jshell> anotherSet.intersect(lastSet);
$34 ==> MySet contains: a, d, e
jshell>
jshell> lastSet.union(lastSet);
$35 ==> MySet contains: a, d, e, q, r
jshell>
jshell> lastSet.intersect(lastSet);
$36 ==> MySet contains: a, d, e, q, r
jshell>
jshell> /exit
```

Good luck for your PA2!!

Submission (Practice)

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.