Introduction to Java

CS9053

Tuesday 6 PM – 8:30 PM

Prof. Dean Christakos

November 4, 2020

Due: October 10, 2020

**Part I: Working with stacks and queues**

1. Write a method remAllStack(Stack<Object> stack, Object item) that takes a stack and an item, and that removes all occurrences of that item from the stack. After the removals, the remaining items should still be in the same order with respect to each other. For example, if you have a stack that contains (from top to bottom) {5, 2, 7, 2, 10}, if you use the method to remove all occurrences of 2, the resulting stack should contain (from top to bottom) {5, 7, 10}.

Important guidelines:

* + Your method may use either another stack or a queue to assist it. It may *not* use an array, linked list, or other data structure. When choosing between a stack or a queue, choose the one that leads to the more efficient implementation.
  + You should assume that the method does *not* have access to the internals of the collection objects, and thus you can only interact with them using the methods in the interfaces that we discussed in lecture.

1. Write a method remAllQueue(Queue<Object> queue, Object item) that takes a queue and an item, and that removes all occurrences of that item from the queue. After the removals, the remaining items should still be in the same order with respect to each other. For example, if you have a queue that contains (from front to rear) {5, 2, 7, 2, 10} and you use the method to remove all occurrences of 2, the resulting queue should contain (from front to rear) {5, 7, 10}. The same guidelines that we specified for remAllStack() also apply here.

**Part II: Sets**

The reason we like Sets in Java is because they help us think about Sets in a mathematical sense and we can easily implement the functions of Sets that exist in Math—eg, Set intersections and unions. In Python, these set functions are explicit. In Java, they are not, something which I briefly forgot during lecture.

Create a class MathSet which extends HashSet. It should have three methods:

public Set intersection(Set s2): Takes a Set, s2, and returns the intersection of the Set and s2—the elements that are in both sets.

public Set union(Set s2): Takes a Set, s2, and returns the union of the Set and s2—the combination of all elements.

public Set<Pair<T,S>> cartesianProduct(Set s2)

I have provided a Pair class for this. Return the Cartesian Product of the base set, s and s2: s × s2:

A **Cartesian product** of two sets A and B, written as A×B, is the set containing **ordered** pairs from A and B. That is, if C=A×B, then each element of C is of the form (x,y) where x∈A and y∈B:

A×B={(x,y)|x∈A and y∈B}.

For example, if A={1,2,3} and B={H,T}, then

A×B={(1,H),(1,T),(2,H),(2,T),(3,H),(3,T)}

Note that here the pairs are ordered, so for example, (1,H)≠(H,1). Thus A×B is **not** the same as B×A.

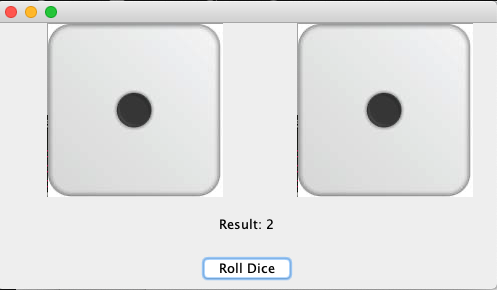
**Part III: Maps**

A Map creates Key -> Value relationships. Each Key is unique, but different Keys may map to the same Values.

Start with a Map<Object, Object> and implement a static method “getInverted” that takes a Map object and returns a Map that reverses the original Map object and maps the Values to Keys. Since a single Value could have multiple Keys associated with it, the reversed Map will be of the type Map<Object, Set<Object>>

**Part IV: Graphics**

You are going to create a “roll the dice” application. It will look like this:



The way it works is that when you click on the “Roll Dice” button, it will cycle through the dice images a random number of times until landing on the final one, which will have the total of the two dice. At that point, it will show the sum of the dice. If you click on ONE die, it will “roll” that one and then update the total.

Hints: Since there’s little “skeleton” code available, modify the “ImagePanel” code. The ActionListeners/MouseListeners will update the image in the ImagePanel and repaint() a random number of times before settling on the final value.