HARAMRIT SINGH KHURANA G01229319

SWE 619 Assignment 7

// TODO

Goal: Polymorphic Abstraction.

Assignment:

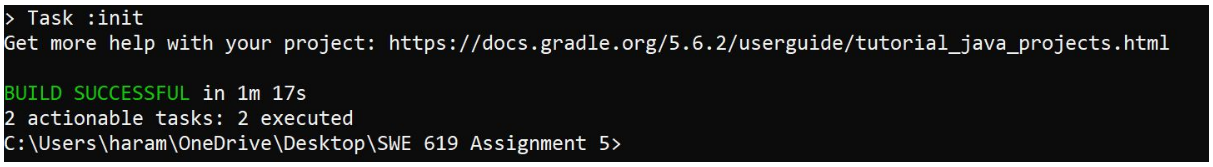
A Comparator based on absolute values is problematic. Code up the comparator and then write client code that illustrates the problem. Use a Java 8 *lambda function* implement the comparator. Explain what is wrong in a brief summary statement. Your explanation of the problem must be phrased in terms of a violation of the contract for Comparator.

To emphasize that this contract problem is real, your code should create two Java sets, one a HashSet, and the other a TreeSet. The TreeSet should order items with your absolute value comparator. Your example should add the same integers to both sets, yet still end up with sets that are different. Your summary statement should explain why.

As for other recent assignments, your deliverable is a clear, concise story that demonstrates completion of the assignment.

We need to create a new class **Assignment7** and write JUnit tests to test the corresponding class. Hence, I first created a **new gradle application** as follows:

- Open command prompt
- `cd` to the directory where you want to initialise your project
- Type `gradle init`
- Follow the onscreen instructions and Gradle should create a workspace for you



```
> Task :init
Get more help with your project: https://docs.gradle.org/5.6.2/userguide/tutorial_java_projects.html

BUILD SUCCESSFUL in 1m 17s
2 actionable tasks: 2 executed
C:\Users\haram\OneDrive\Desktop\SWE 619 Assignment 5>
```

Now, we have our project ready to roll!

LET'S GET STARTED!!!

1. Coding up the AbsoluteValueComparator:

We first start by implementing an absolute value comparator. In order to do this, we create a class AbsoluteValueComparator which implements the Comparator<> interface. Since the Comparator interface is being implemented, it is quintessential to implement the compare() method, which takes in as parameters, the two elements to be compared and returns an int value.

In order to compare two elements in this AbsoluteValueComparator, we simply subtract the absolute values of the two elements to be compared. This leads to 3 cases:

1. **Negative value**
   If the first element is smaller than the second element, the comparator returns a negative value.
2. **Zero**
   If the two elements are equal, the comparator returns zero.
3. **Positive value**
   If the first element is larger than the second element, the comparator returns a positive value.

```
// Classic Implementation of AbsoluteValueComparator
class AbsoluteValueComparator implements Comparator<Integer>{
    @Override
    public int compare(Integer o1, Integer o2) throws
NullPointerException, ClassCastException {
        return Math.abs(o1.intValue()) -
Math.abs(o2.intValue());
    }
}
```

2. Implementing the `compare()` method using lambda expression:

We implemented the AbsoluteValueComparator above in our classical Java method implementation. However, Comparator is a FunctionalInterface which means it has only one abstract method. This enables us to code up the comparator in a much more concise manner.

```
// Lambda Implementation for AbsoluteValueComparator
Comparator<Integer> comp = (o1, o2)->Math.abs(o1) - Math.abs(o2);
```

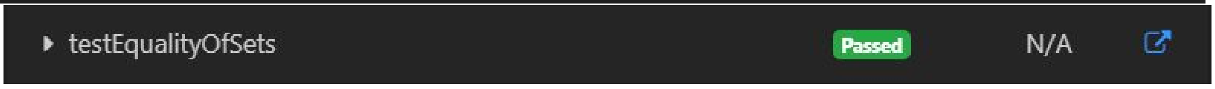
3. Client code to prove the fallacy:

Now we write the test cases to prove the fallacy in the AbsoluteValueComparator. To do this, we first write an @Before annotation to setup the client input code. In this, we create instances of a TreeSet and HashSet and add the exact same elements in those sets. Since we add the same elements, it should return the exact same set, however this is where the AbsoluteValueComparator comes into play and spoils things for us.

```
    Set<Integer> treeSet, hashSet;
    Comparator<Integer> comp;

    @Before
    public void beforeEachTest() {
        comp = (o1, o2) -> Math.abs(o1) - Math.abs(o2);

        treeSet = new TreeSet<Integer>((o1, o2) -> Math.abs(o1) -
Math.abs(o2));
        hashSet = new HashSet<Integer>();

        treeSet.add(3);
        treeSet.add(4);
        treeSet.add(-6);
        treeSet.add(5);
        treeSet.add(-5);
        treeSet.add(-15);

        hashSet.add(3);
        hashSet.add(4);
        hashSet.add(-6);
        hashSet.add(5);
        hashSet.add(-5);
        hashSet.add(-15);
    }
```

Since we use AbsoluteValueComparator in the TreeSet, it will treat a negative and positive value of the same magnitude as the same element. For example, according to AbsoluteValueComparator, 5 and -5 are exactly the same elements. So for instance, if we first add 5 to the TreeSet and then attempt to add -5, we will not be able to do so because they are both treated the same element. Thus, TreeSet and HashSet will end up being two different sets.

```
    @Test
    public void testEqualityOfSets() {
        assertNotEquals(treeSet, hashSet);
        System.out.println("TreeSet is "+treeSet);
        System.out.println("HashSet is "+hashSet);
    }
```



```
testEqualityOfSets    Passed    N/A
```

```
TreeSet is [3, 4, 5, -6, -15]
HashSet is [3, 4, -5, -6, 5, -15]
```

Also, if we look closely at the contents of TreeSet, we can see that the order imposed on the set is not a total order. -15 should have been at the very start of the set followed by -6 and so on. But comparing the absolute values has messed up the total ordering property of the set.

4. Checking for violation of Comparator contract

The AbsoluteValueComparator, gives wrong output, because it violates the Comparator contract which states that:

“ It follows immediately from the contract for `compare` that the quotient is an *equivalence relation* on `S`, and that the imposed ordering is a *total order* on `S`. When we say that the ordering imposed by `c` on `S` is *consistent with equals*, we mean that the quotient for the ordering is the equivalence relation defined by the objects' `equals(Object)` method(s):

`{(x, y) such that x.equals(y)}`.”

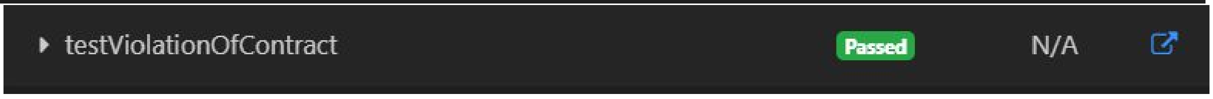
This means that a custom Comparator abides by its contract if it satisfies the equivalence relation between compare(x,y) and x.equals(y).

In mathematical terms,

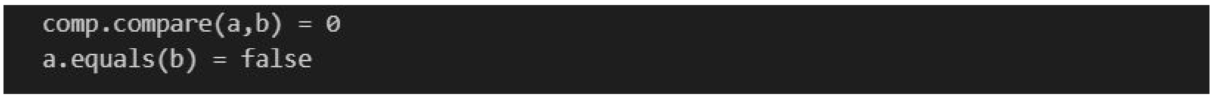
compare(x,y) == 0 ⇔ x.equals(y)

However, our AbsoluteValueComparator returns 0 even after comparison between unequal elements.

```
    @Test
    public void testViolationOfContract() {
        Integer a = 3;
        Integer b = -3;
        assertEquals(0, comp.compare(a, b));
        assertFalse(a.equals(b));
        System.out.println("comp.compare(a,b) = "+comp.compare(a,
b));
        System.out.println("a.equals(b) = "+a.equals(b));
    }
```



testViolationOfContract Passed N/A

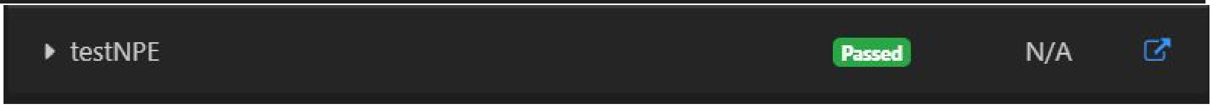


```
comp.compare(a,b) = 0
a.equals(b) = false
```

5. Checking if NPE is thrown on addition of null elements

The compare method does not allow null elements to be compared and throws NullPointerException if it encounters any null elements passed as its parameters.

```
    @Test(expected = NullPointerException.class)
    public void testNPE(){
        Integer a = null;
        Integer b = 3;
        comp.compare(a, b);
    }
```



testNPE Passed N/A

Conclusion:

Thus we implemented an AbsoluteValueComparator in both the classical format as well as using lambda expressions. We also performed a series of tests to come to the conclusion that AbsoluteValueComparator does not abide by the contract.