SWE 619 Assignment 6

// TODO

Goal: Type Abstraction.

Liskov 7.11.

Instead of building an IntBag, complete the implementation of a generic version Bag.java and compare it to this generic version of IntSet: LiskovGenericSet.java.

Note that the rep has been chosen for you. Your code should be pretty simple. It won't be quite as simple as the set version, but it will be close. You should also implement repOk().

To answer Liskov's question about subtype relationships, find a relevant "Properties" rule. Your should code this up as an appropriate JUnit test.

Make your write-up very easy to understand. In particular, you should be clear in your analysis of the subtyping question. In other words, you should again submit a story, not just code.

I expect to see a robust discussion on Piazza about the relevant property.

Earn one bonus point for coding the relevant property as JUnit theory and clearly explaining how your theory answers the question. (We'll talk about JUnit theories later in the semester.)

We need to create a new class **Bag** 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. Creating the class Bag:

The Bag class is already outlined for us. We need to implement functionalities of the various methods it contains. Bag uses Map data structure.

2. Implementing the `insert()` method:

Since we are using Map data structure, we know that a key cannot be repeated within the map. Hence, before inserting, we need to check whether the element we are trying to insert already exists within the map or not. If it does, we simply increment its value by 1. Else we create a map entry with the element as the key and value 1.

```
    public void insert(E e) {
        if (isIn(e)) {
            int val = getValue(e);
            map.put(e, val + 1);
        } else {
            map.put(e, 1);
        }
    }
```

3. Implementing the `remove()` method

If element that we want to remove is found in the map, decrement the value by 1.
Else if it is not found, throw NoSuchElementException()

```
    public void remove(E e) {
        if (isIn(e)) {
            map.put(e, map.get(e) - 1);
        } else
            throw new NoSuchElementException("Element "+e+" not
found");
    }
```

4. Implementing the `isIn()` method

We check if a particular element exists in the map or not.
Returns true if element exists, else returns false.

```
    public boolean isIn(E e) {
        if (map.containsKey(e))
            return true;
        else
            return false;
    }
```

5. Implementing the `size()` method

Returns the number of elements inside the map

```
    public int size() {
        return map.size();
    }
```

6. Implementing the `choose()` method

Choose next element from the map and return it.
Throws IllegalStateException if size of map ≤ 0

```
    public E choose() throws IllegalStateException {

        if (size() <= 0) {
            throw new IllegalStateException("No elements to choose
from");
        } else {
            return map.keySet().iterator().next();
        }
    }
```

7. Implementing the `getValue()` method

Check if the element exists in the map. If it does, return its value.
Else return 0

```
    public int getValue(E e) {
        if (isIn(e)) {
            return map.get(e);
        } else {
            return 0;
        }
    }
```

8. Implementing the `repOk()` method

The value of any of the key/elements cannot be < 0. Hence we check all keys for their values. If all values are > 0, we return true. Else if even one value turns out to be < 0, we break immediately and return false.

```
public boolean repOk() {
    boolean repValid = true;
    for (Integer i : map.values()) {
        if (i <= 0) {
            repValid = false;
            break;
        }
    }
    return repValid;
}
```

9. Implementing the `toString()` method

Returns the String representation of the map.

```
public String toString() {
    return map.toString();
}
```

10. Creating LiskovGenericSet class

In the same directory where Bag class was created, we create another class named LiskovGenericSet. The implementation for this class is already provided to us, hence we fill in the details inside the class.

TESTING:

1. Test `insert()`

```
@Test
public void testInsert()
{
    Bag<String> myBag = new Bag<>();

    myBag.insert("Haramrit");
    assertEquals(1,myBag.getValue("Haramrit"));
}
```

2. Test `remove()`

```
@Test
public void testRemove()
{
    Bag<String> myBag = new Bag<>();
    myBag.insert("Haramrit");

    myBag.remove("Haramrit");
    assertEquals(0,myBag.getValue("Haramrit"));
}
```

3. Test `isIn()`

```
@Test
public void testisIn()
{
    Bag<String> myBag = new Bag<>();

    myBag.insert("Haramrit");
    assertTrue(myBag.isIn("Haramrit"));
}
```

4. Test `getValue()`

```
@Test
public void testgetValue()
{
    Bag<String> myBag = new Bag<>();

    myBag.insert("Haramrit");
    assertEquals(1 ,myBag.getValue("Haramrit"));
}
```

5. Test `size()`

```
@Test
public void testSize()
{
    Bag<String> myBag = new Bag<>();

    myBag.insert("Haramrit");
    myBag.insert("Nabeel");

    assertEquals(2,myBag.size());
}
```

6. Test `choose()`

```
@Test
public void testChoose() {
    Bag<String> myBag = new Bag<>();

    myBag.insert("Haramrit");

    assertTrue(myBag.isIn(myBag.choose()));
}
```

7. Test `repOk()`

```
@Test
public void repOk() {
    Bag<String> myBag = new Bag<>();

    myBag.insert("Haramrit");
    myBag.insert("Nabeel");

    assertTrue(myBag.repOk());
}
```

8. Test properties rule

Validating the condition - (continuing from the discussion in 7.11 - Liskov) - Is Bag a legitimate subtype of Set ?

This test case FAILS. This is because the Bag allows duplicates but LiskovGenericSet does NOT.
This causes violation of the properties-rule (i.e.) the super-type defines a property that is lost in its sub-type.

```
@Test
public void testPropertiesRule() {
    Bag<String> myBag = new Bag<>();
    LiskovGenericSet<String> mySet = new LiskovGenericSet<>();

    //Adding an element "Haramrit" to LiskovGenericSet & Bag
    mySet.insert("Haramrit");
    myBag.insert("Haramrit");

    //Adding the same element "Haramrit" to LiskovGenericSet &
Bag
    mySet.insert("Haramrit");
    myBag.insert("Haramrit");

     // Attempting to remove the element "Haramrit": the bag as
it allows duplicates, does NOT completely remove the element
     // while the set completely removes the element
```

```
        myBag.remove("Haramrit");
        mySet.remove("Haramrit");

        assertFalse(mySet.isIn("Haramrit"));
        assertFalse(myBag.isIn("Haramrit"));
    }
```

Conclusion:

Thus we implemented a Bag class which utilized the Map data structure. We wrote method implementations to perform basic operations such as insert, remove, check if element present, etc. Then we compared Bag and LiskovGenericSet which according to Liskov share a supertype and subtype relationship. However after performing tests, we came to the conclusion that there was a violation of the subtype relationship since sets and maps behaved differently for removal of elements.