SWE 619 Assignment 10

// TODO

Goal: Object class contracts.

As it happens, Liskov's implementation of clone() for the IntSet class (see figure 5.10, page 97) is wrong.

Use the version of IntSet from the in-class exercise. Implement a subtype of IntSet to demonstrate the problem. Your solution should include appropriate executable code in the form of JUnit tests.

Provide a correct implementation of clone() for IntSet. Again, give appropriate JUnit tests.

Correctly override hashCode() and equals(). As discussed in the class exercise, the standard recipe is not appropriate in this (unusual) case.

In addition to code and tests, your deliverable is a story. Explain what is going on at each stage of the exercise. The GTA will primarily grade your story.

We need to create a new class **IntSet** 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 IntSet:

The IntSet class is already outlined for us. We need to implement correct versions of `equals()`, `hashCode()` and `clone()` methods. IntSet contains a List of type Integer.

2. Implementing the `hashCode()` method:

There are various implementations of hashCode() method. There is one wherein we calculate the hashcode of all important fields in that class to find hashcode of the object. There is also one implementation where the method returns a constant value (e.g. 42). This is a perfectly valid implementation and also supports the argument that unequal objects don't necessarily mean unequal hashcodes. We choose the method of computing hashcode for each element to find the final object hashcode.

```
    @Override
    public int hashCode() {
        int hashVal = 0;
        for (Integer i : els) {
            hashVal = hashVal * 31 + i;
        }
        return hashVal;
    }
```

We also write JUnit tests to check if our hashCode() method functions properly.

```
    // Testing hashcodes -> same hashcode for equal objects
    @Test public void testHashCode(){
        IntSet set1 = new IntSet();
```

```
        set1.insert(1);
        set1.insert(2);
        set1.insert(3);
        IntSet set2 = new IntSet();
        set2.insert(1);
        set2.insert(2);
        set2.insert(3);
        IntSet set3 = new IntSet();
        set3.insert(1);
        set3.insert(2);
        set3.insert(3);
        set3.insert(4);
        //Equal objects -> equal hashcodes
        assertTrue(set1.equals(set2));
        assertTrue(set1.hashCode() == set2.hashCode());
        // Unequal objects -> not necessarily unequal hashcodes
        // The hashcode function can also simply return 42 which
        //  makes both the hashcodes equal.
        assertFalse(set1.equals(set3));
        assertFalse(set1.hashCode() == set3.hashCode());
    }
```

3. Implementing the `equals()` method

The equals() method is used to check for equality between two objects. Here, quite unusually, the standard recipe doesn't work. Hence, we use getClass() to check for instance-class validation instead of instanceOf().

```
    @Override
    public boolean equals(Object obj) {
        if (obj == null || obj.getClass() != getClass()) {
            return false;
        }

        IntSet s = (IntSet) obj;
        return s.els.containsAll(els) && s.els.size() == els.size();
    }
```

The equals method also needs to satisfy three main properties:

A. Reflexivity
B. Transitivity
C. Symmetry

We write JUnit tests to test the same.

```
    // Test properties of equals (symmetry, reflexivity,
transitivity)
    @Test public void testEquals() {

        IntSet set1 = new IntSet();
        set1.insert(1);
        set1.insert(2);
        set1.insert(3);

        IntSet set2 = new IntSet();
        set2.insert(1);
        set2.insert(2);
        set2.insert(3);

        IntSet set3 = new IntSet();
        set3.insert(1);
        set3.insert(2);
        set3.insert(3);

        IntSet set4X = new IntSet();
        set4X.insert(1);
        set4X.insert(2);
        set4X.insert(3);
        set4X.insert(4);

        //Check basic implementation
        assertFalse(set1.equals(set4X));

        //Check symmetry
        assertTrue(set1.equals(set2));
        assertTrue(set2.equals(set1));
```

```
        //Check reflexivity
        assertTrue(set1.equals(set1));

        //Check transitivity
        assertTrue(set2.equals(set3));
        assertTrue(set1.equals(set3));
    }
```

4. Implementing the `Subclass` extending IntSet

We also add the Subclass class extending the parent class, IntSet.

5. Implementing the `clone()` method for parent and child class

The clone method as in Liskov's implementation is good when only the parent class exists. However, on extending the parent class to form child classes, this clone method starts creating problems.
In order to fix it, we use the super's clone method which in case of IntSet is the Object class. This call is then explicitly typecast to IntSet to return a clone of the object. We create clones of all mutable fields explicitly too.
The subclass' clone method simply calls the clone method of the superclass class typecasting it alongside.

```
     @Override
    public IntSet clone() throws CloneNotSupportedException {
        try {
            IntSet clonedSet = (IntSet) super.clone();
            clonedSet.els = new ArrayList<Integer>(els);
            return clonedSet;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
class SubClass extends IntSet {
    @Override
    public SubClass clone() throws CloneNotSupportedException {
        SubClass sc = (SubClass) super.clone();
        return sc;
    }
```

6. Implementing the helper methods: `insert()` and `isPresentIn()`

In order to perform testing, we need some elements in our IntSet, which compels us to implement these two imperative methods as part of our IntSet implementation.

```
void insert(Integer e) {
    if (!isPresentIn(e)) {
        els.add(e);
    }
}

boolean isPresentIn(Integer e) {
    return (els.indexOf(e) >= 0);
}
```

Conclusion:

Thus we implemented a IntSet class and wrote method implementations for common operations such as checking object equality, object hashCode and object cloning. We also added a subclass extending IntSet and showed how Liskov's implementation of clone was problematic when inheritance came into the picture.