HARAMRIT SINGH KHURANA G01229319
ADITYA RAGHUNATH SAWANT G01177792

SWE 619 Assignment 13

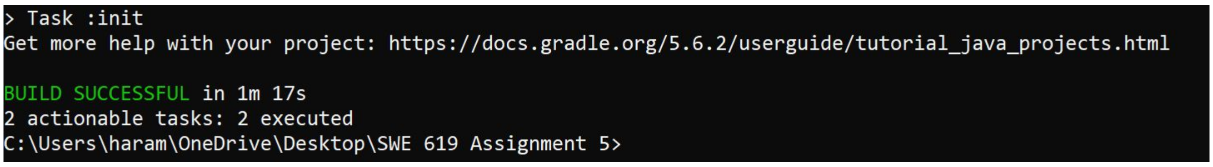
// TODO

Goal: Applying lessons learned.

Convert an existing JUnit test set to JUnit Theories. If you wish to construct JUnit from scratch for this, that's fine too.

We need to create a few classes and write JUnit tests and theory for the corresponding classes. 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 ImmutableStack class:

We will be writing theories and tests related to ImmutableStack class, so let's first code up the ImmutableStack class

ImmutableStack.java:

```
/**
 * Bloch's Stack example from page 60, 3rd edition CONVERTED to
ImmutableStack
 * @author Haramrit Singh Khurana, Aditya Sawant
 */

package StackTheory;

// import java.util.*;

public class ImmutableStack {

    private Object[] elements;
    int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public ImmutableStack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    private ImmutableStack(int size) {
        this.elements = new Object[size];
    }

    public ImmutableStack push(Object e) {
        ImmutableStack newStack = new ImmutableStack(size + 1);
        System.arraycopy(elements, 0, newStack.elements, 0, size);
        newStack.size = size + 1;
        newStack.elements[size] = e;
        return newStack;
    }

    public ImmutableStack pop() {
        if (size == 0)
            throw new IllegalStateException("ImmutableStack is
empty!");

        ImmutableStack newStack = new ImmutableStack(size + 1);
        System.arraycopy(elements, 0, newStack.elements, 0, size);
        newStack.elements[size - 1] = null;
        newStack.size = size - 1;
        return newStack;
    }

    public Object viewTopElement() {
        if (size == 0)
            throw new IllegalStateException("ImmutableStack is
empty");
        return elements[size - 1];
    }

    @Override
    public String toString() {
        String aF = "";
        for (int i = size-1; i >= 0; i--) {
            aF= aF+" | "+elements[i];
            if(i == size - 1)
                aF+= " <- TOP";
            aF+= "\n";
        }
        // System.out.println(stack);
        return aF;
    }
}
```

2. Immutable Stack Theory and Tests:

DataPoints are as follows:

```
@RunWith(Theories.class)
public class ImmutableStackTest {

    @DataPoints("stack")
    public static ImmutableStack[] im = {
        new ImmutableStack().push(1).push(2),
        new ImmutableStack().push("ELEPHANT"),
        new ImmutableStack(),
        null
    };

    @DataPoints("items")
    public static Object[] items(){
        return new Object[]{1, 2, 3, 3.14, 42.99, "CAT", "DOG", "MOUSE",
false, true, null };
    }
```

- We have a stack of integers, stack of string, empty stack and null as our stack data points.
- Items to be pushed in stack are taken from the item data points.

HARAMRIT SINGH KHURANA G01229319
ADITYA RAGHUNATH SAWANT G01177792

We will be testing **3 theories** pertaining to the ImmutableStack class:

a. **The last element pushed on the stack is the top element**

```
//This theory will hold when the item from datapoint pushed is same as stack
top
    @Theory
    public void Theory_LastElementPushedIsStackTop(@FromDataPoints("items")
Object item,
            @FromDataPoints("stack") ImmutableStack stack) {
        System.out.println("Stack is " + stack);
        System.out.println("Item is " + item);
        System.out.println();

        assumeTrue(stack != null);
        stack = stack.push(item);
        assertEquals(stack.viewTopElement(), item);
    }
```

- This theory assumes that stack object is not null. Hence when the code comes across the null stack in the stack data points, it does not go ahead of the precondition (assumeTrue condition).
- It will hold true when the top of stack is the same as the item pushed from datapoint.

b. **ImmutableStack creates a new object on push operation**

```
    //This theory holds true when stack operation push performed returns new
updated stack object
    //Theory will fail if the stack objects are same
    @Theory
    public void
Theory_NewObjectReturnedAfterPushingItemOnStack(@FromDataPoints("items")
Object item,
            @FromDataPoints("stack") ImmutableStack stack) {
        assumeTrue(stack != null);
        assertNotSame("The two stack objects are SAME", stack.push(item),
stack);
    }
```

- This theory validates the immutability of stack.
- Assume that stack is not null.

- Push operation performed on the stack will create new stack object.
- It will thus push item in the new stack and return the updated version as a different object
- Thus the original stack is kept intact and the object returned is different from the original stack.

c. <u>ImmutableStack creates a new object on pop operation</u>

```
    //This theory holds true when stack operation pop performed returns new
updated stack object
    //Theory will fail if the stack objects are same
    @Theory
    public void
Theory_NewObjectReturnedAfterPoppingItemOffStack(@FromDataPoints("stack")
ImmutableStack stack) {
        assumeTrue(stack != null && stack.size > 0);
        assertNotSame("The two stack objects are SAME", stack.pop(), stack);
    }
```

- This theory validates the immutability of stack.
- Assume that stack is not null as well as not empty.
- Pop operation performed on the stack will create new stack object.
- The new stack will have its top element removed and we get this new stack as a new immutable object.
- Thus the original stack is kept intact and the object returned is different from the original stack.

Contribution:

Haramrit:
Coded up the top element theory and responsible for stack and item data points creation

Aditya:
Coded the two theories pertaining to push and pop operations

Conclusion:

Thus we revisited the ImmutableStack class (from Assignment 3) and reviewed all its tests. Then we tried to convert those tests into theories by supplying them with suitable data points for testing.