DD1339 Introduktion till datalogi 2013/2014

Uppgift nummer: 3

Namn: Marcus Larsson

Grupp nummer: 5

Övningsledare: Marcus Dicander

Betyg: ..... Datum: ..... Rättad av: .....

Exercise 10.71

Person

```
import java.util.Comparator;
/**
 * This is a class that represents a Person.
 * It's possible to compare Person objects. It will first compare by age and if same, names will be
compared.
 *
 * @author Marcus Larson
 * @version 2014-01-27
 */
public class Person{

  //Instance variables
   int age;
   String name;
   public static final Comparator COMP = new ComparePerson();

   private static class ComparePerson implements Comparator<Person>{
      /**
       * Compares one Person object to another Person object.
       * The age is compared
       * Returns negative if person p1.age<p2.age. Positive if p1.age>p2.age.
       * If age is same, the name is compared instead with String compareTo().
       * @param p1 The first Person object
       * @param p2 The second Person object.
       * @return Returns the difference between the two Persons age.
       */
      @Override
      public int compare(Person p1, Person p2) {
         if(p1.age==p2.age){
            return p1.name.compareTo(p2.name);
         }
         return p1.age-p2.age;
         }
   }
   /**
    * Constructor of this class.
    *
    * @param name Enter the persons name
    * @param age
    */
   public Person(String name, int age) {
      this.age = age;
      this.name = name;
   }
   /**
    *
    * @return The persons age.
    */
```

```
    public int getAge() {
        return age;
    }
    /**
     *
     * @return The persons name
     */
    public String getName() {
        return name;
    }
    /**
     *
     * @param age Enter the age to set.
     */
    public void setAge(int age) {
        this.age = age;
    }
    /**
     *
     * @param name Enter the name that should be set.
     */
    public void setName(String name) {
        this.name = name;
    }
    /**
     *
     * @return A string description of this Person object. The name of the Person is returned.
     */
    @Override
    public String toString(){
        return this.name;
    }
}
```

Test to insert Person to TreeSet

```
import java.util.TreeSet;

/**
 * This is a class that test the functionality of inserting Person objects to a TreeSet.
 *
 * @author Marcus
 * @version 2014-01-27
 */
public class Test {

    public static void main(String args[]){
            Test obj = new Test();
            obj.run();
    }

    /**
```

```
    * Runs the program. Adds Person objects to a TreeSet and prints it out to display that it is ordered
by age.
    */
   public void run(){
      Person p1 = new Person("p1", 10);
      Person p2 = new Person("p2", 12);
      Person p3 = new Person("p3", 9);
      Person p4 = new Person("p4", 10);

      TreeSet<Person> tree = new TreeSet<>(Person.COMP);
      tree.add(p1);
      tree.add(p2);
      tree.add(p3);
      tree.add(p4);

      for(Person p:tree){
         System.out.println(p);

      }
   }
}
```

Testet ger resultatet:
p3
p1
p4
p2
Vilket är förväntat resultat.

Exercise Stack

Stack Interface

```
/**
 * A Stack is a list with LIFO order (last-in-first-out).
 * All elements added to stack gets put in a pile where only the top element is accessible.
 * Classes that implements this interface has to provide the methods push(), pop(), top(), size() and
isEmpty().
 *
 * @author Marcus Larsson
 * @version 2014-01-24
 */
public interface Stack<T>
{
   /**
    * Pushes an item on to the stack. Element will be added in the top of the stack. (First in vector)
    * @param o Enter the element that you want to add to the stack.
    */
   void push(T o);
   /**
    * Returns the top element in the stack and removes it from the stack. (First in vector)
```

```
     * @return Returnds the element that currently is on top of the stack.
     */
    T pop();
    /**
     * Peeks on the top of the stack. Returns the top element in the stack, but the top element also
stays in the top of the stack.
     * (top is first in vector)
     * @return Returns the element that currently is on top of the stack.
     */
    T top();
    /**
     * Returns the number of items in the stack
     * @return The number of items in the stack
     */
    int size();
    /**
     * Gives an answer if the stack is empty or not.
     * @return true if the stack is empty and false if not.
     */
    boolean isEmpty();
}
```

Stack implemented

(size() och isEmpty() ärvs av LinkedList från förra veckan.)

```
import java.util.EmptyStackException;
/**
 * Write a description of class EventStack here.
 *
 * @author Marcus
 * @version 2014-01-24
 */
public class EventStack<T> extends LinkedList<T> implements Stack<T>
{
    /**
     * Pushes en element onto the top of the Stack. Same effect as addFirst() from LinkedList.
     * @param o The element to push onto the stack.
     */
    public void push(T o){
        addFirst(o);
    }
    /**
     * Removes and returns the top element from the stack.
     * @return The element in the top of the stack. (Same as the first element in LinkedList)
     * @throws EmptyStackException If the stack is empty.
     */
    public T pop() throws EmptyStackException{
        if(isEmpty()){
            throw new EmptyStackException();
        }
        return removeFirst();
    }
```

```
    /**
     * Keeps the stack as it is, just look at the top item.
     * @return The element in the top of the stack. (Same as the first element in LinkedList)
     * @throws EmptyStackException If the stack is empty.
     */
    public T top() throws EmptyStackException{
        if(isEmpty()){
            throw new EmptyStackException();
        }
        return getFirst();
    }
}
```

Test implemented Stack

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.util.Random;

/**
 * The test class EventStackTest.
 *
 * @author  Marcus Larsson
 * @version 2014-01-25
 */
public class EventStackTest
{
    /**
     * Default constructor for test class EventStackTest
     */
    public EventStackTest()
    {
    }

    /**
     * Sets up the test fixture.
     *
     * Called before every test case method.
     */
    @Before
    public void setUp()
    {
    }

    /**
     * Tears down the test fixture.
     *
     * Called after every test case method.
     */
    @After
    public void tearDown()
```

```
{
}

/**
 * Tests to pop element from an empty Stack. This should throw EmptyStackException.
 */
@Test(expected=java.util.EmptyStackException.class)
public void testPopEmptyStack(){
   EventStack testStack = new EventStack();
   testStack.pop();
}
/**
 * Test to peek on the top element of an empty Stack. This should throw EmptyStackException.
 */
@Test(expected=java.util.EmptyStackException.class)
public void testTopEmptyStack(){
   EventStack testStack = new EventStack();
   testStack.top();
}

/**
 * Test the constructor of EventStack. Tests so that Stack is healthy after creation.
 */
@Test
public void testStackConstruct(){
   EventStack testStack = new EventStack();
   assertTrue(testStack.isHealthy());

   EventStack<String> testStack2 = new EventStack<>();
   assertTrue(testStack2.isHealthy());
}

/**
 * Test to push elements onto the Stack. Checks so that element is pushed on the the right position
and that Stack is still healthy after.
 */
@Test
public void testPush(){
   EventStack<String> testStack = new EventStack<>();
   //add element to empty list
   String s1 = "test";
   testStack.push(s1);
   assertSame(s1, testStack.top());
   assertTrue(testStack.isHealthy());

   //add element to list with 1 element
   String s2 = "test2";
   testStack.push(s2);
   assertSame(s2, testStack.top());
   assertTrue(testStack.isHealthy());
```

```
        //add element to list with 2 elements(more than 2 will behave the same since there are no more
variables changed.)
        String s3 = "test3";
        testStack.push(s3);
        assertSame(s3, testStack.top());
        assertTrue(testStack.isHealthy());

        //test to add NULL (list should still be healthy since only the value null is added in the first node.)
        testStack.push(null);
        assertNull(testStack.top());
        assertTrue(testStack.isHealthy());

    }

    /**
     * Tests to pop an element from the Stack.
     * Adds null element to Stack and makes sure it's still healthy and pops the null element as well.
     */
    @Test
    public void testPop(){
        EventStack<String> stack = new EventStack<>();
        //creates Strings to thest with
        String s1 = "test";

        //Push and pop.
        stack.push(s1);
        assertSame(s1, stack.pop());
        assertTrue(stack.isHealthy());

        //test to add null object and pop that.
        stack.push(null);
        stack.push(s1);
        stack.pop();
        assertTrue(stack.isHealthy());
        assertNull(stack.pop());
        assertTrue(stack.isHealthy());
    }

    /**
     * Test to peek on the top element of the Stack. Pushes on String elements and makes sure it's the
correct element on top.
     * Also test to push null and makes sure element null is in the top.
     */
    @Test
    public void testTop(){
        EventStack<String> stack = new EventStack<>();

        //creates Strings to thest with
        String s1 = "test";

        //Push and check top.
        stack.push(s1);
```

```
        assertSame(s1, stack.top());
        assertTrue(stack.isHealthy());

        //test to add null object and pop that.
        stack.push(null);
        assertTrue(stack.isHealthy());
        assertNull(stack.top());
        assertTrue(stack.isHealthy());

    }
    /**
     * Tests so that the size variable is correct in the list. Adds a random number of elements and
checks so that it matches with the size variable in the list.
     * Computes at time complexity O(n) where n is the number of elements randomly chosen from 1
to 100000.
     */
    @Test
    public void testSize(){
        EventStack<String> stack = new EventStack<String>();

        //test so that empty stack has size 0.
        assertEquals(0, stack.size());
        stack.removeFirst();
        assertEquals(0, stack.size());

        //test a random number of elements is correct.
        int numOfElements = (new Random().nextInt(100000))+1; //random between 1 and 100000
        int count=0;
        while(count<numOfElements){
            stack.push("test"+count);

            count++;
        }
        assertEquals(numOfElements, stack.size());

        //test to remove elements and makes sure the size is still correct.
        stack.pop();
        assertEquals(numOfElements-1, stack.size());

    }

    /**
     * Tests if method isEmpty is working correctly.
     * Computes at constant time.
     */
    @Test
    public void testIsEmpty(){
        EventStack<String> stack = new EventStack<String>();
        assertTrue(stack.isEmpty());
        stack.push("a");
        assertFalse(stack.isEmpty());
        stack.pop();
```

```
        assertTrue(stack.isEmpty());
        assertTrue(stack.isHealthy());
    }
}
```

Postfix

```
import java.util.StringTokenizer;

/**
 * The Postfix class implements an evaluator for integer postfix expressions.
 *
 * Postfix notation is a simple way to define and write arithmetic expressions
 * without the need for parentheses or priority rules. For example, the postfix
 * expression "1 2 - 3 4 + *" corresponds to the ordinary infix expression
 * "(1 - 2) * (3 + 4)". The expressions may contain decimal 32-bit integer
 * operands and the four operators +, -, *, and /. Operators and operands must
 * be separated by whitespace.
 *
 * @author  Marcus Larsson
 * @version 2014-01-29
 */
public class Postfix {

                    /**
                     * Evaluates the given postfix expression.
                     *
                     * @param expr  Arithmetic expression in postfix notation
                     * @return      The value of the evaluated expression
                     * @throws      A subclass of RuntimeException if the expression is wrong
                     */
                    public static int evaluate(String expr) throws RuntimeException {

            EventStack<Integer> stack = new EventStack<>();
            StringTokenizer s = new StringTokenizer(expr);
            while(s.hasMoreTokens()){
               String t = s.nextToken();
               if(isInteger(t)){
                  stack.push(new Integer(t));
               }else if(isOperator(t)){
                  //calculate
                  int result = 0;
                  int element2 = stack.pop();
                  int element1 = stack.pop();
                  switch(t){
                        case("+"):
                           result = element1 + element2;
                        break;
                        case("-"):
                           result = element1 - element2;
                        break;
                        case("*"):
```

```
                result = element1 * element2;
            break;
            case("/"):
                result = element1 / element2;
            break;
        }
        stack.push(result);
    }else{
        throw new RuntimeException();
    }
  }
  if(stack.size()!=1){
    throw new RuntimeException();
  }
                            return stack.pop();
        }

        /**
         * Returns true if s is an operator.
         * An operator is one of '+', '-', '*', '/'.
         */
        private static boolean isOperator(String s) {

  if(s.matches("([+\\-*/])")){
  return true;
}
                            return false;
        }

        /**
         * Returns true if s is an integer.
         *
         * We accept two types of integers:
         *
         * - the first type consists of an optional '-'
         *   followed by a non-zero digit
         *   followed by zero or more digits,
         *
         * - the second type consists of an optional '-'
         *   followed by a single      '0'.
         */
        private static boolean isInteger(String s) {

  if(s.matches("^\\-\\d+")){
    if(s.substring(1).matches("^0\\d+")){
      return false;
    }else {
      return true;
    }
  }

  if(s.matches("(\\D)") || s.matches("^0\\d+")){
```

```
   return false;
}else {
   return true;
}

    }

    /**
     * Unit test. Run with "java -ea Postfix".
     */
    public static void main(String[] args) {
                        assert evaluate("0") == 0;
                        assert evaluate("-0") == -0;
                        assert evaluate("1234567890") == 1234567890;
                        assert evaluate("-1234567890") == -1234567890;
                        assert evaluate("1 23 +") == 1 + 23;
                        assert evaluate("0 1 /") == 0 / 1;
                        assert evaluate("1 2 + -3 *") == (1 + 2) * -3;
                        assert evaluate("12 34 - 56 -78 + *") == (12 - 34) * (56 + -78);
                        assert evaluate("1 2 + 3 * 4 - 5 /") == (((1 + 2) * 3) - 4) / 5;
                        assert evaluate("2 3 4 -0 + - *") == 2 * (3 - (4 + -0));
                        assert evaluate("                                    1                    -2
    + ") == 1 - 2; // tabs and spaces

                        assert explodes("");
                        assert explodes("+");
                        assert explodes("--1");
                        assert explodes("-1-0");
                        assert explodes("-0-1");
                        assert explodes("1 +");
                        assert explodes("1 2 ,");
                        assert explodes("1 2 .");
                        assert explodes("1 2 3 +");
                        assert evaluate("4") == 4;
                        assert explodes("1 2 + +");
                        assert explodes("017");
                        assert explodes("0x17");
                        assert explodes("-03");
                        assert explodes("x");
                        assert explodes("1234L");
                        assert explodes("9876543210"); // larger than maxint
                        assert explodes("1 0 /");
                        assert explodes("1 2+");
                        assert explodes("1 2 3 +*");
    }

    /**
     * Returns true if <code>evaluate(expr)</code> throws
     * a subclass of RuntimeException.
     */
    private static boolean explodes(String expr) {
                        try {
```

```
                    evaluate(expr);
                } catch (RuntimeException e) {
                    return true;
                }
                return false;
            }
}
```