**Exercises on jUnit 5**

1. **MyMathSimple** : a class with a simple single method located in the package (main)

**public** **class** MySimpleMath {

*/\*\**

*\* A simple method that takes and input and returns*

*\* "positive" or "negative" depending on the input number*

*\*/*

**public** String checkSign(int number) {

**if**(number >= 0 ) {

**return** "positive";

} **else** {

**return** "negative";

}

}

*/\*\**

*\* Returns the division of numerator by the denominator.*

*\* If the denominator is zero, it throws an Exception*

*\*/*

**public** double divide(int num, int denom) {

**if**(denom == 0) {

**throw** **new** ArithmeticException("Cannot divide by zero");

} **else** {

**return** num/(double)denom;

}

}

}

In order to Test the functionality of our class and methods we used to add some code in a **main** method and compare the output of the execution against the expected one

**public** **class** MyMathSimpleMain {

**public** static void main(String args[]) {

MySimpleMath sm = **new** MySimpleMath();

*// Check a positive*

System.out.println("Input 10: " + sm.checkSign(10));

*// Check a negative*

System.out.println("Input -2: " + sm.checkSign(-2));

*// Check a positive*

System.out.println("Input 0: " + sm.checkSign(0));

*// Check a divide*

System.out.println("Input (10, 3): " + sm.divide(10, 3));

*// Check a divide*

System.out.println("Input (10, 0): " + sm.divide(10, 0));

}

}

The output of the execution should be

Input 10: positive

Input -2: negative

Input 0: positive

Write test class that implement two scenarios that returns positive or negative.

MyMathSimpleTest.java

@Test

**public** void testCheckSignShouldReturnPositive() {

}

@Test

**public** void testCheckSignShouldReturnNegative() {

}

@Test

**public** void testDivisionShouldReturnPositiveQuotient() {

}

@Test

**public** void testDivisionShouldReturnNegativeQuotient() {

}

@Test

**public** void testDivisionShouldThrowArithmeticException() {

}

**Solution**

MySimpleMathTest.java

**public** **class** MySimpleMathTest {

@Test

**public** void testCheckSignShouldReturnPositive() {

MySimpleMath sm = **new** MySimpleMath();

Assert.assertEquals("positive", sm.checkSign(5));

Assert.assertEquals("positive", sm.checkSign(0));

}

@Test

**public** void testCheckSignShouldReturnNegative() {

MySimpleMath sm = **new** MySimpleMath();

Assert.assertEquals("negative", sm.checkSign(-5));

}

@Test

**public** void testDivisionShouldReturnPositiveQuotient() {

MySimpleMath sm = **new** MySimpleMath();

Assert.assertEquals(2.0, sm.divide(10, 5), 0);

Assert.assertEquals(0.0, sm.divide(0, 5), 0);

}

@Test

**public** void testDivisionShouldReturnNegativeQuotient() {

MySimpleMath sm = **new** MySimpleMath();

Assert.assertEquals(-2.0, sm.divide(10, -5), 0);

}

@Test (expected = ArithmeticException.class)

**public** void testDivisionShouldThrowArithmeticException() {

MySimpleMath sm = **new** MySimpleMath();

sm.divide(10, 0);

}

}

1. Assume that we have a class that offers only two functionalities on an array, the findMin(int[] array) and the multiply(int[] array, int factor) where the first finds the min in an array, and the second multiplies each element of the array with a factor respectively

**public** **class** MySimpleArrayOperations {

**public** int findMin(int[] array) {

**if**(!(array.length > 0)) {

**throw** **new** IllegalArgumentException("Input array is empty");

}

int min = Integer.MAX\_VALUE;

**for**(int i=0; i<array.length; i++) {

**if**(array[i] <= min)

min = array[i];

}

**return** min;

}

**public** void multiply(int[] array, int factor) {

**if**(!(array.length > 0)) {

**throw** **new** IllegalArgumentException("Input array is empty");

}

**for**( int i=0; i<array.length; i++ ) {

array[i] = array[i] \* factor;

}

}

}

## Testing Arrays

* For comparing the equality of arrays we use the assertArrayEquals(...) method provided by the JUnit library
* The test class should have the following code

**public** **class** MySimpleArrayOperationsTest {

@Test

**public** void testFindMin() {

}

@Test (expected = IllegalArgumentException.class)

**public** void testFindMinShouldThrowException() {

}

@Test

**public** void testMultiply() {

}

@Test (expected = IllegalArgumentException.class)

**public** void testMultiplyShouldThrowException() {

}

}

Solution

**public** **class** MySimpleArrayOperationsTest {

@Test

**public** void testFindMin() {

MySimpleArrayOperations msao = **new** MySimpleArrayOperations();

int[] array = {10, 2, 3, 10, 1, 0, 2, 3, 16, 0, 2};

assertEquals(0, msao.findMin(array));

assertNotEquals(10, msao.findMin(array));

}

@Test (expected = IllegalArgumentException.class)

**public** void testFindMinShouldThrowException() {

MySimpleArrayOperations msao = **new** MySimpleArrayOperations();

msao.findMin(**new** int[]{});

}

@Test

**public** void testMultiply() {

MySimpleArrayOperations msao = **new** MySimpleArrayOperations();

int[] array = {10, 2, 3, 10, 1, 0, 2, 3, 16, 0, 2};

msao.multiply(array, 10);

assertArrayEquals(**new** int[]{100, 20, 30, 100, 10, 0, 20, 30, 160, 0, 20}, array);

}

@Test (expected = IllegalArgumentException.class)

**public** void testMultiplyShouldThrowException() {

MySimpleArrayOperations msao = **new** MySimpleArrayOperations();

msao.multiply(**new** int[]{}, 0); *//method call with dummy arguments*

}

}

## Simplify the test cases

* Make objects that are used in many test cases, instance variables
* Create a @BeforeAll method if you want to the perform the same operation (ex. initializing instance variables) before the execution of each test case

**Solution**

**public** **class** MySimpleArrayOperationsTest {

**private** MySimpleArrayOperations msao = **new** MySimpleArrayOperations();

**private** int[] array;

@BeforeAll

**public** void initInstanceVariables() {

System.out.println(**this**.getClass().getName() + " --> initializing fields");

**this**.msao = **new** MySimpleArrayOperations();

**this**.array = **new** int[] {10, 2, 3, 10, 1, 0, 2, 3, 16, 0, 2};

}

@Test

**public** void testFindMin() {

assertEquals(0, msao.findMin(array));

assertNotEquals(10, msao.findMin(array));

}

@Test (expected = IllegalArgumentException.class)

**public** void testFindMinShouldThrowException() {

msao.findMin(**new** int[]{});

}

@Test

**public** void testMultiply() {

msao.multiply(array, 10);

assertArrayEquals(**new** int[]{100, 20, 30, 100, 10, 0, 20, 30, 160, 0, 20}, array);

}

@Test (expected = IllegalArgumentException.class)

**public** void testMultiplyShouldThrowException() {

msao.multiply(**new** int[]{}, 0); *//method call with dummy arguments*

}

}

## Testing Lists

* Lists, Vectors, Stacks are Collections and collections can be represented as arrays by calling the method aList.toArray();
* Assume that we have a class that uses an ArrayList of Integers and provides the following functionalities: add, remove, clear, size, get (from an index)

MyIntegerList.java

**import java.util.ArrayList;**

**import java.util.Arrays;**

**public** **class** MyIntegerList {

**private** ArrayList<Integer> list;

**public** MyIntegerList(Integer[] array) {

**this**.list = **new** ArrayList<>(Arrays.asList(array));

}

**public** Integer[] getListAsAnArray() {

**return** (Integer[]) **this**.list.toArray();

}

**public** void add(int n) {

**this**.list.add(n);

}

**public** void remove() {

**if**(!**this**.list.isEmpty())

**this**.list.remove(**this**.list.size()-1);

}

**public** int get(int index) {

**if**(**this**.list.size() - 1 >= index)

**return** **this**.get(index);

**else**

**return** (Integer) **null**;

}

**public** int size() {

**return** **this**.list.size();

}

**public** void clear() {

**this**.list.clear();

}

}

**MyIntegerListTest.java**

**Write the test cases as shown below code**

**public** **class** MyIntegerListTest {

**private** MyIntegerList myList;

@BeforeAll

**public** void initInstanceVariables() {

}

@Test

**public** void testSize() {

}

@Test

**public** void testAdd() {

}

}

**Solution**

**public** **class** MyIntegerListTest {

**private** MyIntegerList myList;

@Before

**public** void initInstanceVariables() {

System.out.println(**this**.getClass().getName() + "--> initializing array");

Integer array[] = {1,2,3,4,5};

**this**.myList = **new** MyIntegerList(array);

}

@Test

**public** void testSize() {

assertEquals(5, **this**.myList.size());

assertEquals(0, **new** MyIntegerList(**new** Integer[]{}).size());

**this**.myList.add(6);

assertEquals(6, **this**.myList.size());

}

@Test

**public** void testAdd() {

**this**.myList.add(10);

assertArrayEquals(**new** Integer[]{1,2,3,4,5,10},**this**.myList.getListAsAnArray());

}

*// the other test cases are omitted to simplify the example*

}

1. Exploit JUnit to test the following programs

Product.java

public class Product {

    private String title;

    private double price;

    public Product (String t, double p) {

        this.title = t;

        this.price = p;

    }

    public String getTitle() {

        return title;

    }

    public double getPrice() {

        return price;

    }

    public boolean equals(Object o) {

        if (o instanceof Product) {

            Product p = (Product)o;

            return p.getTitle().equals(title);

        }

        return false;

    }

}

ProductNotFoundException.java

public class ProductNotFoundException extends Exception {

    public ProductNotFoundException() {

        super();

    }

}

ShoppingCart.java

import java.util.\*;

public class ShoppingCart {

    private ArrayList items;

    public ShoppingCart() {

        items = new ArrayList();

    }

    public double getBalance() {

        double balance = 0.00;

        for (Iterator i = items.iterator(); i.hasNext();){

            Product item = (Product)i.next();

            balance += item.getPrice();

        }

        return balance;

    }

    public void addItem(Product item) {

        items.add(item);

    }

    public void removeItem(Product item)

            throws ProductNotFoundException {

        if (!items.remove(item)) {

            throw new ProductNotFoundException();}

    }

    public int getItemCount() {

        return items.size();

    }

    public void empty() {

        items.clear();

    }

}

Write the test cases for the following specifications.

1. When created, the cart has 0 items

2. When empty, the cart has 0 items

3. When a new product is added, the number of items must be incremented.

4. When a new product is added, the new balance must be the sum of the previous balance plus the cost of the new product.

5. When an item is removed, the number of items must be decreased.

6. When a product not in the cart is removed, a ProductNotFoundException must be thrown

Hint: insert the call in a try block and put a fail() after the call to removeItem()

ShoppingCartTest.java

public class ShoppingCartTest {

  // before tests setUp method  cart  equals newshoppingCart() object

    @Before

    public void setUp(){

    }

    // testing adding one item to cart apple item is  added to cart.

    // assertEquals checks cart  for apple item on list

    @Test

    public void addOneItemToCart() {

    }

    // testing adding two items to cart apple and Orange item is  added to cart.

    // assertEquals checks cart  for apple item on list and Orange item and total items are 2

    @Test

    public void addTwoItemsToCart(){

    }

    // testing adding three items to cart Apple, Orange and Pear items is  added to cart.

    // assertThat checks cart  for apple item on list,

    //pear item and Orange item and total items are 3

    @Test

    public void addThreeItemsToCart(){

    }

    // testing adding same  item  twice to cart Apple is added to cart.

    // assertThat checks cart  for apple item on list is 2

    @Test

    public void addAnItemTwice(){

    }

   // test case checks for apple item and cart contains specific price is  1.0

    @Test

    public void checkOutOneItem(){

    }

     // test case checks for two items and cart contains specific  price is 1.60

    @Test

    public void checkoutTwoSeparateItems(){

    }

     // test case checks for three item and cart contains specific  price is 2.0

    @Test

    public void checkoutThreeSeparateItems(){

    }

    // test case checks for two same items and cart contains specific  price is 2.0

    @Test

    public void checkoutTwoSameItems(){

    }

     // test case checks for three same items and cart contains specific  price is 3.00

    @Test

    public void checkoutThreeItems(){

    }

     // test case checks for several items and cart contains specific  price is 3.60

    @Test

    public void checkoutManyItems(){

    }

     // test case checks for invalid item like carrot- not sold  in shop

    @Test

    public void checkOutAnInvalidItem(){

    }

    // test case checks removeItem method, addition of Apple and Orange, then Apple is removed

    // assertThat gets carts totalItems is 1

    @Test

    public void removeItem(){

    }

    //Two tests below test for offers 3 items for 2 deal and buy 2 items deal

    @Test

    public void offersCheckFor1(){

    }

    @Test

    public void offersCheckFor2(){

    }

    public String getTestCasesMessage() {

          return "test cases working!";

    }

}

**Solution**

ShoppingCartTest.java

public class ShoppingCartTest {

    ShoppingCart cart;

  // before tests setUp method  cart  equals newshoppingCart() object

    @Before

    public void setUp(){

        cart = new ShoppingCart();

    }

    // testing adding one item to cart apple item is  added to cart.

    // assertEquals checks cart  for apple item on list

    @Test

    public void addOneItemToCart() {

        cart.add("Apple");

        assertEquals(cart.doesContain("Apple"),(true));

    }

    // testing adding two items to cart apple and Orange item is  added to cart.

    // assertEquals checks cart  for apple item on list and Orange item and total items are 2

    @Test

    public void addTwoItemsToCart(){

        cart.add("Apple");

        cart.add("Orange");

        assertThat(cart.getTotalItems(), is(2));

        assertEquals(cart.doesContain("Apple"),(true));

        assertEquals(cart.doesContain("Orange"),(true));

    }

    // testing adding three items to cart Apple, Orange and Pear items is  added to cart.

    // assertThat checks cart  for apple item on list,

    //pear item and Orange item and total items are 3

    @Test

    public void addThreeItemsToCart(){

        cart.add("Apple");

        cart.add("Orange");

        cart.add("Pear");

        assertThat(cart.getTotalItems(), is(3));

        assertEquals(cart.doesContain("Apple"), (true));

        assertEquals(cart.doesContain("Orange"), (true));

        assertEquals(cart.doesContain("Pear"), (true));

    }

    // testing adding same  item  twice to cart Apple is added to cart.

    // assertThat checks cart  for apple item on list is 2

    @Test

    public void addAnItemTwice(){

        cart.add("Apple");

        cart.add("Apple");

        assertThat(cart.getTotalItems(), is(2));

    }

   // test case checks for apple item and cart contains specific price is  1.0

    @Test

    public void checkOutOneItem(){

        cart.add("Apple");

        assertThat(cart.checkout(),is(1.0));

    }

     // test case checks for two items and cart contains specific  price is 1.60

    @Test

    public void checkoutTwoSeparateItems(){

        cart.add("Apple");

        cart.add("Orange");

        assertThat(cart.checkout(),is (1.60));

    }

     // test case checks for three item and cart contains specific  price is 2.0

    @Test

    public void checkoutThreeSeparateItems(){

        cart.add("Apple");

        cart.add("Orange");

        cart.add("Pear");

        assertThat(cart.checkout(), is(2.0));

    }

    // test case checks for two same items and cart contains specific  price is 2.0

    @Test

    public void checkoutTwoSameItems(){

        cart.add("Apple");

        cart.add("Apple");

        assertThat(cart.checkout(), is(2.0));

    }

     // test case checks for three same items and cart contains specific  price is 3.00

    @Test

    public void checkoutThreeItems(){

        cart.add("Apple");

        cart.add("Apple");

        cart.add("Apple");

        assertThat(cart.checkout(), is(3.0));

    }

     // test case checks for several items and cart contains specific  price is 3.60

    @Test

    public void checkoutManyItems(){

        cart.add("Apple");

        cart.add("Apple");

        cart.add("Orange");

        cart.add("Orange");

        cart.add("Pear");

        assertThat(cart.checkout(), is(3.60));

    }

     // test case checks for invalid item like carrot- not sold  in shop

    @Test

    public void checkOutAnInvalidItem(){

        cart.add("Carrot");

        assertThat(cart.checkout(),is(0.0));

    }

    // test case checks removeItem method, addition of Apple and Orange, then Apple is removed

    // assertThat gets carts totalItems is 1

    @Test

    public void removeItem(){

        cart.add("Apple");

        cart.add("Orange");

        cart.remove("Apple");

        assertThat(cart.getTotalItems(), is(1));

    }

    //Two tests below test for offers 3 items for 2 deal and buy 2 items deal

    @Test

    public void offersCheckFor1(){

        cart.add("Apple");

        cart.add("Apple");

        cart.add ("Apple");

        assertThat(cart.offers(),is(1.0));

    }

    @Test

    public void offersCheckFor2(){

        cart.add("Orange");

        cart.add("Orange");

        assertThat(cart.offers(),is(1.5));

    }

    public String getTestCasesMessage() {

          return "test cases working!";

    }

}

1. Write a class with a static method that converts a string into an integer value

Write the test cases for the following specifications.

1. The method must accept a string and convert it into an integer
2. Well-formed strings does not contain characters different from numbers, trailing spaces and minus.
3. The represented number must be in the range [-32768, 32767]
4. No real number are allowed

OK: “ -3”, “500”, “-10”, “32767”

Not Ok: “2 3”, “32768”, “A3”, “2.3”

5. Test also boundary conditions

6. Throw an exception in the converter method, and test if the exception has been thrown when the method is called with bad arguments

7. Exploit the Integer.parseInt() method for both the conversion and the check

## 6. Using ArrayList and HashMap

The basic Java data types have worked fairly well, but it is time to move beyond what they can do. Well the Java API provides classes like this, called the Collections classes. We will be working with two of them, the ArrayList and the HashMap.

* ArrayList
* HashMap

ArrayList is a wrapper around an array that allows you to grow the array at run time, so you no longer are required to know the maximum capacity of the array when you declare it.

HashMap is a data structure that you may have heard called a 'dictionary' or an 'associative array' in other languages. A HashMap is a conceptually simple data structure consisting of a set of keys and values. You might want to think of it as a type of array that you access with keys rather than by indexing. In this exercise you will be making a simple student 'Directory' application,

A Student is an object that has a

matriculation number,

name,

age,

mailbox,

gender and

department, and all the basic getters and setters to deal with this set of fields. Here is a table with the initial data:

| **MN** | **Name** | **Age** | **MailBox** | **Gnd** |
| --- | --- | --- | --- | --- |
| s0189034 | Peter | 17 | [peter@math](mailto:peter@math) | M |
| s0289125 | Michael | 21 | [mike@geo](mailto:mike@geo) | M |
| s0378435 | Helen | 28 | [helen@phys](mailto:helen@phys) | F |
| s0412375 | Mary | 18 | [mary@inf](mailto:mary@inf) | F |
| s0456782 | John | 22 | [john@inf](mailto:john@inf) | M |
| s0355689 | Dana | 33 | [dana@ling](mailto:dana@ling) | F |
| s0768633 | Lee | 36 | [lee@chem](mailto:lee@chem) | F |

The only slightly tricky task is to get the information about the student's department, since this is not given directly in the data above. Your task is to compute this from the student's MailBox. First, build a HashMap which encodes the following key-value pairs:

| **Domain** | **Department Name** |
| --- | --- |
| chem | Chemistry |
| geo | Geosciences |
| inf | Informatics |
| ling | Linguistics |
| math | Mathematics |
| phys | Physics |

Next, you have to parse the value of the student's MailBox attribute; you can use the String method split() to do this. Finally, you can use the domain part of the MailBox string to look up the department name in the HashMap that you have constructed.

Here is the skeleton of the StudentList.java class.

import java.util.\*;

public class StudentList {

private ArrayList<Student> students = new ArrayList<Student>();

public void makeStudents () {

Student peter = new Student();

peter.setMn("s0189034");

peter.setName("Peter");

peter.setAge(17);

peter.setMbox("peter@math");

peter.setGender("M");

Student michael = new Student();

// complete this object

// add objects for all the other students

students.add(peter);

// add the other students

}

public ArrayList<Student> getStudents() {

return students;

}

public HashMap<String, Student> getMedianStudentsMap() {

// TODO: Return a HashMap with the student's name as

// the key and the student record as a value for any

// student aged over 18 but under 25.

}

public Student getStudentByName(String name) {

// TODO: Return a student record of the student whose

// name was passed in as argument to the method.

}

public String getStudentDepartment(String name) {

// TODO: Return the department of the student whose

// name was passed in as argument to the method.

}

public ArrayList<Student> getStudentsByAgeAndGender(int minAge, String gender) {

// TODO: Return an ArrayList of student that is older

// than the minAge, and of the Gender "M" or "F" given

// in gender

}

public void printStudents() {

for (Student s : students) {

System.out.println("Name: " + s.getName());

}

}

}

Implement the following methods:

* getMedianStudentsMap(),
* getStudentByName(),
* getStudentDepartment() and
* getStudentsByAgeAndGender().

Now that you know how to make your own unit tests, make a JUnit test to test that the four methods that you wrote are working as you expect. Write the following programs.

Student.java

StudentList.java

and StudentListTest.java.

**Sample solutions**

Student.java:

import java.util.HashMap;

public class Student {

private String mn;

private String name;

private int age;

private String mbox;

private String gender;

private HashMap<String, String> dept = new HashMap<String, String>();

private void setDept() {

dept.put("chem", "Chemistry");

dept.put("geo", "Geosciences");

dept.put("inf", "Informatics");

dept.put("ling", "Linguistics");

dept.put("math", "Mathematics");

dept.put("phys", "Physics");

}

public String getDept() {

setDept();

String[] mailDomain = mbox.split("@");

String domain = mailDomain[1];

return dept.get(domain);

}

public String getMn() {

return mn;

}

public void setMn(String mn) {

this.mn = mn;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getMbox() {

return mbox;

}

public void setMbox(String mbox) {

this.mbox = mbox;

}

public String getGender() {

return gender;

}

public void setGender(String gender) {

this.gender = gender;

}

}

StudentList.java:

import java.util.\*;

public class StudentList {

private ArrayList<Student> students = new ArrayList<Student>();

public void makeStudents() {

Student peter = new Student();

peter.setMn("s0189034");

peter.setName("Peter");

peter.setAge(17);

peter.setMbox("peter@math");

peter.setGender("M");

Student michael = new Student();

michael.setMn("s0289125");

michael.setName("Michael");

michael.setAge(21);

michael.setMbox("mike@geo");

michael.setGender("M");

Student helen = new Student();

helen.setMn("s0378435");

helen.setName("Helen");

helen.setAge(28);

helen.setMbox("helen@phys");

helen.setGender("F");

Student mary = new Student();

mary.setMn("s0412375");

mary.setName("Mary");

mary.setAge(18);

mary.setMbox("mary@inf");

mary.setGender("F");

Student john = new Student();

john.setMn("s0456782");

john.setName("John");

john.setAge(22);

john.setMbox("john@inf");

john.setGender("M");

Student dana = new Student();

dana.setMn("s0355689");

dana.setName("Dana");

dana.setAge(33);

dana.setMbox("dana@ling");

dana.setGender("F");

Student lee = new Student();

lee.setMn("s0768633");

lee.setName("Lee");

lee.setAge(36);

lee.setMbox("lee@chem");

lee.setGender("F");

students.add(peter);

students.add(michael);

students.add(helen);

students.add(mary);

students.add(john);

students.add(dana);

students.add(lee);

}

public ArrayList<Student> getStudents() {

return students;

}

/\*\*

\* Build a HashMap with the student's name as the key and the student record

\* as a value for any student aged over 18 but under 25.

\*

\* @return The constructed HashMap.

\*/

public HashMap<String, Student> getMedianStudentsMap() {

HashMap<String, Student> result = new HashMap<String, Student>();

for (Student s : students) {

if (s.getAge() > 18 && s.getAge() < 25) {

result.put(s.getName(), s);

}

}

return result;

}

/\*\*

\* Return the Student record of student with a given name.

\*

\* @param name

\* The name of the student to look-up.

\* @return The instance of Student with that name.

\*/

public Student getStudentByName(String name) {

for (Student s : students) {

if (s.getName().equals(name)) {

return s;

}

}

return null;

}

/\*\*

\* Return the department of the student whose name was passed in as argument

\* to the method.

\*

\* @param name

\* The name of the student to look-up.

\* @return The student's department.

\*/

public String getStudentDepartment(String name) {

for (Student s : students) {

if (s.getName().equals(name)) {

return s.getDept();

}

}

return null;

}

/\*\*

\* Build an ArrayList of Student that is older than the minAge, and of the

\* Gender given in gender

\*

\* @param minAge

\* The minimum age to be considered.

\* @param gender

\* The required gender "M" or "F"

\* @return

\*/

public ArrayList<Student> getStudentsByAgeAndGender(int minAge,

String gender) {

ArrayList<Student> result = new ArrayList<Student>();

for (Student s : students) {

if (s.getAge() > minAge && s.getGender().equals(gender)) {

result.add(s);

}

}

return result;

}

/\*\*

\* Print out the roster of students.

\*/

public void printStudents() {

for (Student s : students) {

System.out.println("Name: " + s.getName());

}

}

}

StudentListTest.java:

import java.util.\*;

import junit.framework.TestCase;

public class StudentListTest {

private StudentList r;

private String[] names;

private HashMap<String, String> studentDepts;

@BeforeEach

protected void setUp() {

r = new StudentList ();

r.makeStudents();

names = new String[7];

names[0] = "Peter";

names[1] = "Michael";

names[2] = "Helen";

names[3] = "Mary";

names[4] = "John";

names[5] = "Dana";

names[6] = "Lee";

studentDepts = new HashMap<String, String> ();

studentDepts.put("Peter", "Mathematics");

studentDepts.put("Michael", "Geosciences");

studentDepts.put("Helen", "Physics");

studentDepts.put("Mary", "Informatics");

studentDepts.put("John", "Informatics");

studentDepts.put("Dana", "Linguistics");

studentDepts.put("Lee", "Chemistry");

}

@Test

public final void testGetMedianStudentsMap() {

HashMap<String, Student> studentMap = r.getMedianStudentsMap();

assertTrue(studentMap.containsKey("Michael"));

assertTrue(studentMap.containsKey("John"));

}

@Test

public final void testGetStudentByName() {

for (String name : names) {

assertNotNull(r.getStudentByName(name));

}

}

@Test

public final void testGetStudentDepartment() {

for (String s : studentDepts.keySet()) {

assertEquals(studentDepts.get(s), r.getStudentDepartment(s));

}

}

@Test

public final void testGetStudentsByAgeAndGender() {

ArrayList<Student> s1 = r.getStudentsByAgeAndGender(21, "M");

assertEquals("John", s1.get(0).getName());

ArrayList<Student> s2 = r.getStudentsByAgeAndGender(36, "M");

assertTrue(s2.isEmpty());

ArrayList<Student> s3 = r.getStudentsByAgeAndGender(0, "F");

assertEquals(4, s3.size());

}

}

## Cryptography with a One-time Pad

The task in this exercise is to implement a class Crypto which contains methods for encrypting and decrypting a message using a one-time pad. A one-time pad is an encryption technique where the plaintext message is combined with a random key or "pad" that is as long as the plaintext and used only once. Modular addition is used to combine the plaintext with the pad. If the key is truly random, never reused, and kept secret, the one-time pad provides perfect secrecy. The pad is sometimes called a "keystream".

Assume your cleartext is the string:

DONOTUSEPC

(i.e. "do not use PC"). Note that this technique only represents the 26 letter of the alphabet, and doesn't do anything with punctuation or spaces. Assume your keystream string is:

KDWUPONOWT

### [Encryption](https://www.inf.ed.ac.uk/teaching/courses/inf1/op/Labs/2008/doc/lab5_with_solutions.html#id14)

Step 1. Convert the cleartext message from letters to numbers: A = 1, B = 2, etc:

4 15 14 15 20 21 19 5 16 3

Step 2. Convert the keystream letters similarly:

11 4 23 21 16 15 14 15 23 20

Step 3. Add the cleartext number stream to the keystream numbers, modulo 26. (I.e., if the sum is more than 26, substract 26 from the result.) For example, 1 + 1 = 2, 26 + 1 = 27, and 27 - 26 = 1, so 26 + 1 = 1:

15 19 11 10 10 10 7 20 13 23

Step 4. Convert the numbers back to letters:

OSKJJJGTMW

This string of letters is the ciphertext.

### [Decryption](https://www.inf.ed.ac.uk/teaching/courses/inf1/op/Labs/2008/doc/lab5_with_solutions.html#id15)

Assume you have a ciphertext message like the string above. To decrypt, you have to use the same keystream that way used for encryption.

Step 1. Convert the ciphertext message into numbers:

15 19 11 10 10 10 7 20 13 23

Step 2. Convert the keystream letters similarly:

11 4 23 21 16 15 14 15 23 20

Step 3. Subtract the keystream numbers from the ciphertext numbers, modulo 26. For example, 22 - 1 = 20, 1 - 22 = 5 (If the first number is less than the second number, add 26 to the first number before subtracting. So 1 - 22 = ? becomes 27 - 22 = 5.):

4 15 14 15 20 21 19 5 16 3

Step 3. Convert the numbers back to letters:

DONOTUSEPC

For an alternative source of information, you can look at these Wikipedia articles:

* <http://en.wikipedia.org/wiki/One-time_pad>
* <http://en.wikipedia.org/wiki/Keystream>

In the Crypto class, you should implement methods corresponding to the following declarations:

public String encrypt(String clearText, String keyStream) {

...

}

public String decrypt(String cipherText, String keyStream) {

...

}

These should satisfy the following pair of equations:

1. cipherText = encrypt(clearText, keyStream)
2. clearText = decrypt(cipherText, keyStream)

Here is a ciphertext:

CUFNFLRQMLWBLARVJDALMURNGSHILXXMVDUMFPYG

which I encrypted using this keystream:

ACWZYRJYHGVNHUCARNVXJPUIFACBWOJFBOTIEBVBBZHANSLYOP

If your decrypt() method works, you should be able to figure out the original cleartext message.

Write CryptoTest class with the possible test cases.

8. Create a class MyCalculator that implements a calculator and provides the following functionalities (methods) for any pair of **positive integers** :

addition

multiplication

division

Consider checking the input numbers for their illegibility.

For example:

1. the denominator of the division cannot be zero,
2. input numbers should not result to overflow (ex. the result of adding two *int* numbers should fit in a int variable)
3. think of more cases if exist..
4. In cases that the input values violate your constraints, you should throw an IllegalArgumentException with a corresponding message

9. Create a class named MyAdvancedMath and implement the following methods

* int factorial(int n) : Calculates and returns the factorial of a given non-negative number n. If n < 0 throw an IllegalArgumentException("n cannot be < 0"). Also investigate which is the largest factorial that can fit in an integer variable and do not allow the user (throw an IllegalArgumentException) to give an n larger than the value that causes an overflow.
* double power(int b, int n) : Calculates and returns the power of a number, where b is the base and n is the exponent. The exponent (n) can be any integer between [0, 20]. If the input is larger than that, an IllegalArgumentException("n should be 0 <= n <= 20") should be thrown
* int[] reverse(int[] array) which should return an array which is the reversed of the one you gave as an input

Write a test class and test all the methods.