UECS2354 Software Testing
Lab 03: Various assert() methods and Parameterised Tests

Part A

1.1 Using other assertXXX() methods in testing (DemoAssertTest.java)

1. There are a few overloaded versions of the `assertXXX()` methods found in the `Assert` class. You can check the Javadocs offline from `junit-4.12-javadoc.jar` or check it online at (http://junit.sourceforge.net/javadoc/index.html?overview-summary.html).

2. The following class contains a variety of test methods that demonstrates the use of `assertArrayEquals()`, `assertTrue()`, `assertFalse()`, `assertNull()`, `assertNotNull()`,`assertSame()`,`assertNotSame()`.

   Go through all these methods and try and understand what is being checked for; comments are provided to explain.

   The functionality of the `assertSame()` method differs from the `assertEquals()` method. The **`assertSame()`** performs like the **==** operator, in that it checks whether two reference variables refer to the same object.
   The **`assertEquals()`** method calls the **`equals()`** method on its first object parameter, and the implementation of this method will return either true or false to determine equality between the two objects compared.

```
public class DemoAssertTest {

    @Test
    public void testAssertArrayEqualsPass() {

            int [] expectedResult = {1,2,3,4,5};
            int [] actualResult = {1,2,3,4,5};
            // Both arrays are same length and have exact contents
            // this test should pass
            assertArrayEquals("Arrays are not equal !!", expectedResult, actualResult);
    }

    @Test
    public void testAssertArrayEqualsFail() {

            int [] expectedResult = {1,2,3,4,6};
            int [] actualResult = {1,2,3,4,5};
            // Both arrays are same length but have different contents
            // this test should fail
            assertArrayEquals("Arrays are not equal !! ", expectedResult, actualResult);
    }

    @Test
    public void testAssertFalseAndAssertTruePass() {

            boolean firstCheck = (30 > 2); // results in true
            boolean secondCheck = (10 == 50); // results in false
            // firstCheck is true, so this test passes
            assertTrue("Error ! should be true", firstCheck);

            // secondCheck is false, so this test passes
            assertFalse("Error ! should be false", secondCheck );
    }
```

```
    @Test
    public void testAssertFalseAndAssertTrueFail() {

            boolean firstCheck = (6 != 10); // results in true
            boolean secondCheck = (100 <= 99); // results in false

            // secondCheck is false, so this test fails
            assertTrue("Error ! should be true", secondCheck );

            // firstCheck is true, so this test fails
            assertFalse("Error ! should be false", firstCheck);
    }

    @Test
    public void testAssertNullandAssertNotNull() {
            String str1 = null;
            String str2 = "hello there";

            // str1 is null, this test passes
            assertNull("Error ! should be null", str1);

            // str2 refers to an object (i.e. it is not null)
            // this test passes
            assertNotNull("Error ! should be not null", str2);
    }

    @Test
    public void testAssertSameAndAssertNotSame() {

            String str1 = "hello there";
            String str2 = str1;
            String str3 = new String("hello there");

            // both str1 and str2 refer to the same String object
            // so the test passes
            assertSame("reference variables do not refer to same object ! ", str1, str2);

            // str1 and str3 refer to different String objects
            // even though the String objects have the same literal value
            // therefore the test passes
            assertNotSame("reference variables refer to same object ! ", str1, str3);
    }
}
```

3. Notice that all these `assertXXX()` method calls contain a first parameter that is a String; this is a message that will be displayed as part of the test run output if that particular `assertXXX()` method fails. It will be shown in the failure trace in the JUnit view when the particular test class is executed. This first parameter is optional, but is usually included because this message can convey useful information on the nature of the failure that will aid in the debugging process.

4. Run the various test methods in `DemoAssertClass`. We should see 2 failures indicated in the JUnit view:
   - one in `testAssertFalseAndAssertTrueFail`
   - one in `testAssertArrayEqualsFail`.

   Highlight these two tests in the upper portion of the JUnit view. The corresponding failure trace in the lower portion of the view will show the error message that was included as the first parameter in the corresponding `assertXXX()` method call. This would be "`Error ! should be true`" for the case of `testAssertFalseAndAssertTrueFail` and "`Arrays are not equal !!` " for the case of `testAssertArrayEqualsFail`.

5. Notice that in the test method `testAssertFalseAndAssertTrueFail`, the `assertTrue()` method call fails; which results in the test method throwing a `java.lang.AssertionError` exception and causing the method to halt execution at that point. Any remaining code (e.g. the other `assertFalse()` method call) is not executed. Comment out `assertTrue()` and rerun the test, the `assertFalse()` method call is executed and fails, resulting in an exception being thrown and failure indicated with a different error message.

6. When there are several `assertXXXX()` method calls within a single test method, bear in mind that failure in any one of the method calls will result in all other method calls following that failed call to be skipped. For example, assume that we have the following sequence of `assertXXX()` method calls within a single test method:

```
assertA();
assertB();
assertC();
assertD();
```

   If `assertB`() fails, the test method will immediately terminate with an exception and neither `assertC`() or `assertD`() will be invoked.

   Therefore we should NOT assume that because only a failure is reported in `assertB`(), that the remaining `assert`() methods completed without an issue. Since neither `assertC`() or `assertD`() was run, we have no idea whether they would have succeeded or not. To avoid confusion on this important point, either try to include only a single `assertXXX()` method in every test method; or ensure that all the `assertXXX()` methods included in a test method are verified to succeed before proceeding.

   Exercise: (`SomeMethodsClass.java`)
   The following class contains 4 methods whose functionalities are stated in comments above the respective method implementation. Write a series of test methods to test these 4 methods using:
   - `assertArrayEquals()`
   - `assertTrue()`
   - `assertFalse()`
   - `assertNull()`
   - `assertNotNull()`
   - `assertSame()`
   - `assertNotSame()`

```
public class SomeMethodsClass {

   // Given an array of Strings and a single String strToAdd,
   // strToAdd is appended to every String element in the array

   public void addSomeStrings(String[] strArray, String strToAdd) {
        for (int i = 0; i < strArray.length; i++)
             strArray[i] += strToAdd;
   }

   // Given an int parameter age, the method checks to see whether
   // age falls within a certain numeric range. If it does, the
   // method returns true; otherwise the method returns false

   public boolean checkHumanAge(int age) {
        if (age >0 && age < 130)
             return true;
        else
             return false;
   }

   // Given an array of Strings and an int parameter strLength,
   // the method returns the first String element in the array
   // whose length is larger than strLength. If there are no
   // elements whose length is larger than strLength, than a null
   // value is returned

   public String getAString(String[] strArray, int strLength) {
        String strToReturn = null;
        for (int i = 0; i < strArray.length; i++)
             if (strArray[i].length() > strLength) {
                  strToReturn = strArray[i];
                  break;
             }
        return strToReturn;
   }

   // Given an array of Strings and an int parameter pos,
   // the method returns the String element in the array
   // at index position pos

   public String getStringAtPos(String[] strArray, int pos) {
        return strArray[pos];
   }
}
```

1.2 Working with equals() and the == operator

1. Study the following program (`ComparisonWithoutEqualsDefined.java`)

```
class Student {
   private String name;
   private int age;

   public Student(String name, int age) {
      this.name = name;
      this.age = age;
   }
   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 class ComparisonWithoutEqualsDefined {

   public static void main(String[] args) {
      String s1 = new String("dog");
      String s2 = s1;
      String s3 = new String("dog");
      if (s1 == s2)
         System.out.println("s1 & s2 are referring to the same object");
      else
         System.out.println("s1 & s2 are NOT referring to the same object");

      if (s1 == s3)
         System.out.println("s1 and s3 are referring to the same object");
      else
         System.out.println("s1 and s3 are NOT referring to the same object");

      if (s1.equals(s2))
         System.out.println("s1 & s2 refer to objects with identical values");

      if (s1.equals(s3))
         System.out.println("s1 & s3 refer to objects with identical values");

      Student stu1 = new Student("Ah Beng", 25);
      Student stu2 = stu1;
      Student stu3 = new Student("Ah Beng", 25);

      if (stu1 == stu2)
         System.out.println("stu1 & stu2 are referring to the same object");
      else
         System.out.println("stu1 & stu2 are NOT referring to the same
   object");

      if (stu1 == stu3)
         System.out.println("stu1 & stu3 are refering to the same object");
      else
         System.out.println("stu1 & stu3 are NOT referring to the same
   object");
```

```
      if (stu1.equals(stu2))
         System.out.println("Student objects referred to by stu1 & stu2 have
   identical instance variable values");
      else
         System.out.println("Student objects referred to by stu1 & stu2 DO NOT
   have identical instance variable values");

      if (stu1.equals(stu3))
         System.out.println("Student objects referred to by stu1 & stu3 have
   identical instance variable values");
      else
         System.out.println("Student objects referred to by stu1 & stu3 DO NOT
   have identical instance variable values");
   }
}
```

2. The above program illustrates the difference between the concept of equality when the **==** operator is used and when the **`equals()`** method is used.

   The **==** operator checks to see whether two reference variables are **pointing to the same object** in memory.

   Since we set `s2` to point to the same object as `s1`, evaluating **`s1 == s2`** returns **true**.

   However, `s1` and `s3` are pointing to two different String objects in memory, even if these String objects have the same value "`dog`". Hence, comparing `s1` and `s3` using the `==` operator returns false.

3. The **`equals()`** method is found in the `Object` class, which is the root class for the class inheritance hierarchy in the Java API.
   Refer to http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html.

   This method is automatically inherited by all classes in a Java program, regardless of whether the class is part of the standard Java API (such as the String class) or whether the class is defined by the user (such as the `Student` class in the above program).

   This method is by default overridden in the String class so that if two **String objects** being compared have **identical values**, the method **returns true**. Thus `s1.equals(s2)` and `s1.equals(s3)` both return true, since `s1`, `s2` and `s3` all refer to String objects that all have identical content, which is "`dog`".

4. The program also performs the same sequence of comparisons with 3 objects (`stu1`, `stu2`, `stu3`) from the user defined class `Student`. The result of the comparisons are similar with the case of the String objects, except for the last comparison: `if(stu1.equals(stu3))`. This conditional statement returns **false**, instead of the expected true.
   The reason for this is that the `equals`() method is **NOT** automatically overridden in user defined classes such as `Student` to provide the kind of functionality as we have just seen in the String class.
   If we do not provide our own version of the **`equals()`** method, then the default `equals`() method implementation from the `Object` class is used. This **default implementation performs exactly like the == operator**; that is, it returns true only if both reference variables are pointing to the

same object in memory. Since `stu1` and `stu3` are pointing to two different objects, the result of evaluating `if(stu1.equals(stu3))` is false.

5. The `NewStudent` class in the following program (`NewStudent.java`) is a reworked version of the `Student` class, with an additional implementation of the `equals`() method that overrides the version inherited from the `Object` class. To override the `equals`() method properly, the signature of the method implemented here must be the same as its initial declaration in the `Object` class, which is:

   ```
   public boolean equals(Object obj)
   ```

```
public class NewStudent {
   private String name;
   private int age;

   public NewStudent(String name, int age) {
      this.name = name;
      this.age = age;
   }
   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 boolean equals(Object obj)
   {
      if (obj instanceof NewStudent)
      {
         NewStudent compareStudent = (NewStudent) obj;
         if(name.equals(compareStudent.getName())
               && age == compareStudent.getAge())
               return true;
      }
      return false;
   }

   public static void main(String[] args) {

      NewStudent stu1 = new NewStudent("Ah Beng", 25);
      NewStudent stu2 = stu1;
      NewStudent stu3 = new NewStudent("Ah Beng", 25);

      if (stu1 == stu2)
         System.out.println("stu1 & stu2 are referring to the same object");
      else
         System.out.println("stu1 & stu2 are NOT referring to the same
   object");

      if (stu1 == stu3)
         System.out.println("stu1 & stu3 are referring to the same object");
      else
         System.out.println("stu1 & stu3 are NOT referring to the same
   object");
```

```
      if (stu1.equals(stu2))
         System.out.println("Student objects referred to by stu1 & stu2 have
   identical instance variable values");
      else
         System.out.println("Student objects referred to by stu1 & stu2 DO NOT
   have identical instance variable values");

      if (stu1.equals(stu3))
         System.out.println("Student objects referred to by stu1 & stu3 have
   identical instance variable values");
      else
         System.out.println("Student objects referred to by stu1 & stu3 DO NOT
   have identical instance variable values");
   }
}
```

6. The first thing we do is to check that the object being passed to the `equals`() method is of the same class type as the class containing the `equals`() method (which in this case is `NewStudent`). If it is, we cast this object to the `NewStudent` class and then compare for equality between the instance variables of the current `NewStudent` object and the `NewStudent` object being passed in as a parameter. If both instance variables (`age` and `name`) in both objects being compared have the same value, return **true**. In all other cases, return false. When we run this class, the output in the Console view indicates that `if(stu1.equals(stu3))` now evaluates to the expected result of true.

7. One thing to note at this point is that concept of equality based on the implementation of the `equals`() method is essentially up to the developer. For example, we could have implemented the `equals`() method so that it checks for equality between only one of the instance variables in the objects being compared, for e.g.

```
public boolean equals(Object object){
   if (object instanceof NewStudent)
   {
      NewStudent compareStudent = (NewStudent) object;
      if (age == compareStudent.getAge())
         return true;
   }
   return false;
}
```

Or, define two objects as being equal if the name instance variable in these objects are of the same length, for e.g.

```
public boolean equals(Object object){
   if (object instanceof NewStudent)
   {
      NewStudent compareStudent = (NewStudent) object;
      if (name.length() == compareStudent.getName().length())
         return true;
   }
   return false;
}
```

8. The definition of equality between two objects using the `equals()` method is thus up to the developer to decide, and may vary depending on the context of the application. However, for the vast majority of situations, two objects are considered equal when their state (as represented by the values of all their instance variables) is exactly the same. This is the most commonly understood definition of equality between two objects, so you should stick to implementing your `equals()` method in this manner unless you have a good reason otherwise.

9. Refer to the following program (`TestsWithEqualsDefined.java`). The **`assertSame()`** method checks for equality in the same way as the **==** operator, and therefore will succeed when comparing `stu1` and `stu2`.
   The `assertEquals()` method invokes the `equals()` method on the first parameter (`stu1`), passing the second parameter (`stu3`) as an argument to it. This evaluates to true when the instance variables of both these objects have the same value.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestsWithEqualsDefined {
   @Test
   public void testCheckHumanAge() {
      NewStudent stu1 = new NewStudent("Ah Beng", 25);
      NewStudent stu2 = stu1;
      NewStudent stu3 = new NewStudent("Ah Beng", 25);

      assertSame(stu1, stu2);
      assertNotSame(stu1, stu3);
      assertEquals(stu1, stu3);
   }
}
```

UECS2354 Software Testing
Lab 03: Various assert() methods and Parameterised Tests

Exercise

Write a series of test methods to test the 2 methods in the `MethodsUsingNewStudent` class using the following methods:

- `assertEquals()`
- `assertArrayEquals()`
- `assertNull()`

```
import java.util.ArrayList;

public class MethodsUsingNewStudent {
   // Given an array of Student objects, stuArray, this method locates the
   // first Student object whose name variable starts with the string startStr
   // and returns this object.
   // If no such Student object is found, the value null is returned.

   public NewStudent findStudentWithName(NewStudent[] stuArray, String startStr) {

      NewStudent stud = null;

      for (int i = 0; i < stuArray.length; i++) {
         if (stuArray[i].getName().startsWith(startStr)) {
            stud = stuArray[i];
            break;
         }
      }
      return stud;
   }

   // Given an array of Student objects, stuArray, this method locates all objects
   // from this array whose age variable is more than the parameter ageLimit
   // and returns these objects in a second array.

   public NewStudent[] findOverAgedStudents(NewStudent[] stuArray, int ageLimit) {

      ArrayList<NewStudent> returnList = new ArrayList<NewStudent>();

      for (int i = 0; i < stuArray.length; i++) {
         if (stuArray[i].getAge() > ageLimit)
            returnList.add(stuArray[i]);
      }

      NewStudent[] arrReturn = new NewStudent[returnList.size()];
      arrReturn = returnList.toArray(arrReturn);
      return arrReturn;
   }
}
```

Part B

1.3 Creating parameterised tests

1. In many situations, we may wish to test a particular method with a large combination of different input parameter values, with the expectation of producing different results for each particular combination of input values. To do this, we can write a test method for every combination of input parameter values; but this leads to a large number of test methods with significant code duplication in all of these methods.

   To streamline the creation of test methods, we need a way of supplying a large number of input parameter combinations, along with the expected result for each combination, to a single test method. Such a test method is known as a **parameterised test**.

2. We will use an external package called **JUnitParams**, as it provides a simpler approach compared to the one employed in JUnit. We will need to add the jar file (`JUnitParams-1.0.2.jar`) for this package to the build path for our unit tests.

3. This following class contains the methods `addTwoNumbers()`.

```
import java.util.StringTokenizer;

public class VariousMethodsClass {

   // accept two integers numbers as parameters and return the sum of them
   public int addTwoNumbers(int a, int b) {
      return a + b;
   }
}
```

4. Refer to the following code (`BasicParameterizedDemo.java`). The first 3 test methods (`testAddTwoNumbersV1`, `testAddTwoNumbersV2`, `testAddTwoNumbersV3`) call the `addTwoNumbers()` method in the `VariousMethodsClass` object, passing it various parameter combinations and checking the returned result using `assertEquals()`.

```
import junitparams.JUnitParamsRunner;
import junitparams.Parameters;

import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

@RunWith(JUnitParamsRunner.class)
public class BasicParameterizedDemo{

   @Test
   public void testAddTwoNumbersV1(){
      VariousMethodsClass vmc = new VariousMethodsClass();
      int result = vmc.addTwoNumbers(10, 15);
      assertEquals(25, result);
   }

   @Test
   public void testAddTwoNumbersV2(){
      VariousMethodsClass vmc = new VariousMethodsClass();
      int result = vmc.addTwoNumbers(-50, 10);
      assertEquals(-40, result);
   }
```

```
    @Test
    public void testAddTwoNumbersV3(){
        VariousMethodsClass vmc = new VariousMethodsClass();
        int result = vmc.addTwoNumbers(30, 100);
        assertEquals(130, result);
    }

    @Test
    @Parameters({"10, 15, 25","-50, 10, -40","30, 100, 130"})
    public void paramTestAddTwoNumbersV1(int num1, int num2, int expectedSum){
        VariousMethodsClass vmc = new VariousMethodsClass();
        int result = vmc.addTwoNumbers(num1, num2);
        assertEquals(expectedSum, result);
    }

    private Object[] getNumbersToAdd(){

        return new Object[] {
            new Object[] {10, 15, 25},
            new Object[] {-50, 10, -40},
            new Object[] {30, 100, 130}
        };
    }

    @Test
    @Parameters(method = "getNumbersToAdd")
    public void paramTestAddTwoNumbersV2(int num1, int num2, int expectedSum){
        VariousMethodsClass vmc = new VariousMethodsClass();
        int result = vmc.addTwoNumbers(num1, num2);
        assertEquals(expectedSum, result);
    }

    @Test
    @Parameters
    public void paramTestV3(int num1, int num2, int expectedSum){
        VariousMethodsClass vmc = new VariousMethodsClass();
        assertEquals(expectedSum, vmc.addTwoNumbers(num1, num2));
    }

    private Object[] parametersForParamTestV3(){
        return new Object[] {
                new Object[] {10, 15, 25},
                new Object[] {-50, 10, -40},
                new Object[] {30, 100, 130}
        };
    }
}
```

5. The `paramTestAddTwoNumbersV1()` test method is the first example of a parameterised test using the **JUnitParams** approach. In addition to the annotation `@Test`, this test also has the annotation **`@Parameters`**, which contains a series of strings nested with **`{}`** and **separated by commas**. Each string contains a series of values separated by commas.

   The test method below the `@Parameters` annotation is invoked once for every string declared in the annotation. For each invocation, each value in the string is passed as an argument to the corresponding parameter in the test method.

For example, on the first invocation of `paramTestAddTwoNumbersV1()`, its parameters `num1`, `num2`, and `expectedSum` will be passed the values 10, 15 and 25 respectively. On the second invocation, its parameters `num1`, `num2`, and `expectedSum` will be passed the values -50, 10 and -40 respectively, and so on.

We should be able to see that this results in `paramTestAddTwoNumbersV1` functioning as a short hand version of the first 3 test methods (`testAddTwoNumbersV1`, `testAddTwoNumbersV2` and `testAddTwoNumbersV3`).

6. The `paramTestAddTwoNumbersV2()` test method is an example of the 2nd approach to creating a parameterised test. The difference between `paramTestAddTwoNumbersV2()` and `paramTestAddTwoNumbersV1()` lies in how values are passed to the parameters of the test method. In this case, the `@Parameters` annotation specifies a method to call (`getNumbersToAdd()`) to provide values to the parameters of the test method, rather than explicitly listing these values itself, as in `paramTestAddTwoNumbersV1()`.

   `getNumbersToAdd()` returns a 2-dimensional array of objects from the class `Object`. Recall that there is a class `Object`, which is the root class for the class inheritance hierarchy in the Java API. Each element of this array of objects, is itself another array of objects.

   The first element of the array of objects that is returned is another array of objects containing the values 10, 15 and 25.
   The second element of the array of objects that is returned is another array of objects containing the values -50, 10 and -40, and so on.

   The test method `paramTestAddTwoNumbersV2()` is called for each element in the 2-dimensional array, and values of the inner array element are passed to the corresponding parameters of `paramTestAddTwoNumbersV2()` for each invocation. The end result is identical to that of `paramTestAddTwoNumbersV1()`.

7. The 3rd approach is used in the test method `paramTestV3()`. It is similar to the 2nd approach except that the parameters providing method is not specified in the `@Parameters` annotation. The parameters providing method will be named like the test method but prefixed with `parametersFor`. **Notice that the test method name start with uppercase after the `parametersFor` although the test method name does not start with uppercase.**

8. Regardless of which approach is used to create a parameterised test, there is a basic pattern involved in the parameters of the signature of the test method (`paramTestAddTwoNumbers`). These parameters can be roughly classified into 3 categories based on the intended usage of these values:
   - values that are used to instantiate an object from the class containing the method to be tested
   - values that are passed as parameters to the method being tested
   - values that are used in the `assertXXX` comparisons.

NOTE:

The first approach (which explicitly specifies the parameters as strings within the annotation `@Parameters`) is only able to specify values for test method parameters which are either primitive data types (such as `boolean`, `int`, `double`, `float`, `byte`, `char`, `long` and `short`) or Strings. If the test method parameters are reference variables (such as arrays or objects of user defined classes), then the second approach must be used.

9. Run the `BasicParameterizedDemo` class as a JUnit test. All the tests should pass. In the JUnit view, you can see that the three test methods `paramTestAddTwoNumbersV1()`, `paramTestAddTwoNumbersV2()` and `paramTestV3()` can be further expanded to show the various runs of these methods for different combinations of input parameter values.

Exercise:

Write parameterised tests for all other methods in the `VariousMethodsClass` using the 1st and 2nd approach.

NOTE: For the methods `findLargestNumber()` and `averageStringLength()`, only the 2nd approach is possible since these methods have a parameter that is an array.

```
import java.util.StringTokenizer;

public class VariousMethodsClass {

   // Given a String words containing a sequence of smaller strings,
   // count the number of occurrences of the String strFind in words
   public int countWordInString(String words, String strFind) {

      int posStrToFind = words.indexOf(strFind);
      int wordCount = 0;
      while (posStrToFind != -1) {
         wordCount++;
         posStrToFind = words.indexOf(strFind, posStrToFind+1);
      }
      return wordCount;
   }

   // Given a String words containing a sequence of smaller strings and
   // an integer x, return a String which contains the smaller strings
   // from words whose length is greater than x.
   public String combineStrings(String words, int x) {

      StringTokenizer st = new StringTokenizer(words);
      String returnStr = "";

      while (st.hasMoreElements()) {
         String currentWord = (String) st.nextElement();
         if (currentWord.length() > x)
            returnStr = returnStr + currentWord + " ";
      }
      return returnStr.trim();
   }




   // Given an integer array, find the largest number contained in that
```

```
   // array
   public int findLargestNumber(int[] numArray) {

      int bigNum = numArray[0];
      for (int i = 0; i < numArray.length; i++)
         if (bigNum < numArray[i])
            bigNum = numArray[i];
      return bigNum;
   }

   // Given an array of Strings and an integer x, calculate the
   // average of the length of the String elements in the array
   // whose length is greater than x.
   public double averageStringLength(String[] strArray, int x) {

      int countWords = 0;
      int sumLength = 0;
      for (int i = 0; i < strArray.length; i++) {
         if (strArray[i].length() > x) {
            sumLength += strArray[i].length();
            countWords++;
         }
      }
      if (countWords == 0) return 0;
      double average = (double) sumLength / countWords;
      return average;
   }
}
```

JUnitParams provides a useful **$()** (dollar sign) method which allows us to write data-providing methods in a less verbose way. We need to import statically to use this method:

import static junitparams.JUnitParamsRunner.$;

However, the $() version DOES NOT allow for accepting NULL values.
Example:

```
private Object[] getNumbersToAdd(){

      return new Object[] {
         new Object[] {10, 15, 25},
         new Object[] {-50, 10, -40},
         new Object[] {30, 100, 130}
      };
   }
```

Written in `$()` method:

```
private Object[] getNumbersToAdd_V2(){

   return $(
         $(10, 15, 25),
         $(-50, 10, -40),
         $(30, 100, 130)
   );
}
```