Lab 2 - Unit Testing and Experimentation

due September 8th at 23:55

Objective

Today's lab will help you get familiar with

- Working with randomness
- Measuring execution time of a piece of code.
- Working with generics through using ArrayList
- The basics of unit testing

Random Data Generator

For this lab and subsequent labs and projects you need to generate arbitrarily large lists of numbers. The easiest way to do this is to use Java's `Random` class. An example is shown below:

```
import java.util.*;

public class RandomExplore {
  public static void main(String[] args) {
    RandomExplore r =new RandomExplore();
    r.run();
  }

  public void run()
  {
    long seed = 88087987;
    Random random = new Random(seed);
    for(int x = 0; x < 10; x++)
    {
      System.out.println(100 * random.nextDouble());
    }
  }
}
```

The code is simple, but it is important to note that there is a fixed random seed. If you do not set the seed you will have a different sequence each time you create an instance of **Random** (try to run this a few times to see that the sequence is exactly the same). This is important to consider for this lab, because using different lists of numbers will change your running time performance. On the other hand, if you use the same seed every time, it will always generate the same sequence of numbers and this sequence that you generate might be skewed in a particular way so any conclusion you reach using only one seed (and the sequence it generates) would be invalid.
The best practice is to always specify the seed when you are testing a program so you have predictable behavior and can determine if the methods/classes are working correctly. Note that when you actually want to use the program to gather data, you need to specify different seeds (e.g., the current time) so that you will get different data. If you always use the same seed, you will always get identical data.

Determining Execution Time

How do you determine how long a program takes to run? It is important to remember, if you run your code multiple times, then each will run for a slightly different amount of time because of what else is running on the machine at that time. This is the difference between the theoretical time complexity and actual runtime. To resolve this, you will want to run your code multiple times and average the result to get a good estimated time. You can compute time with the System method currentTimeMillis. Capture the time before and after you execute your code, calculate the difference, and you have the runtime. Below is an example. Try to run it multiple times, and change the number of times the loop is running to see your time increase/decrease.

```java
import java.util.*;

public class RandomExplore {
  public static void main(String[] args) {
    RandomExplore r =new RandomExplore();
    r.run();
  }

  public void run()
  {
    long seed = 88087987;
    Random random = new Random(seed);
    long startTime = System.currentTimeMillis();
    for(int x = 0; x < 10; x++)
    {
      System.out.println(100 * random.nextDouble());
    }
    long stopTime = System.currentTimeMillis();
    System.out.println("Execute time: " + (stopTime - startTime));
  }
}
```

ArrayList <E> Class

Up to this point you have been using arrays to store data, but these are static and inflexible. The **ArrayList** class is a dynamic array implementation that will grow the array as you add data. Plus it will allow you a variety of options. This class is also a particular type of class called a *generic*, which means that it will generically handle an array of any specified type. You can find the Class's API here.

```java
import java.util.ArrayList;

public class ArrayListTest {
  public static void main(String[] args) {
    ArrayListTest r =new ArrayListTest();
    r.run();
  }

  public void run()
  {
    ArrayList<String> array = new ArrayList<String> ();

           array.add("one");
           array.add("three");
           array.add("two");

           for(int i = 0; i < array.size(); i++) {
             System.out.println(array.get(i));
           }
  }
}
```

In the example above an **ArrayList** is created, three strings are added, and then printed. Note, the size of the arraylist is not set, and this strange syntax ("**<String>**") is used. **ArrayLists** like all the other container structures that we will study this semester are *generic* structures, i.e., they are not restricted to a specific type. We specify the type at the time that we declare the data structure. In the example, we are creating an **ArrayList** that will hold objects of the **String** class.

Note that the type cannot be a primitive since the **ArrayList** stores references and not primitives. Luckily Java provides wrapper classes for primitives which are specific classes that you can use like primitives. More information is here. Therefore, the following are all legal to specify for an **ArrayList**

- **`<String>`**
- **`<Boolean>`**
- **`<Integer>`**
- **`<Double>`**

Junit

JUnit is an automated unit testing implementation. Using this testing framework you can check that a method behaves in a specific way when given a specific set of parameter values.

The important idea behind unit testing is to ensure that your method will behave correctly for both valid and invalid parameter values. The biggest advantage of unit testing is that it automates the process of validating the behavior of methods whenever any part of the program changes. Correct program behaviour means well defined and expected behavior for valid and invalid parameters as described in the method specification. Therefore to ensure that the method is correct, you must check with all the range of parameter values from the method specification.

To use the Junit approach, **you create a test class for each class that you have to test**. Within each test class, you create methods to test each method of the class to be tested. Usually you create a few test methods to test the behavior of one method. Each test method will test the method (in the class being tested) with a particular set of values. What do you need to test?
Normally, for each set of inputs:

- identify the valid "normal" case and test at least two combinations,
- identify the valid boundary conditions and test each of them

- identify the invalid boundary conditions and test each of the ones that are handled by the method, either through an exception or the return of a flag or an error code

Testing With BlueJ and Junit

Creating Junit test cases with BlueJ is very easy. Use the following steps to set up for using unit testing in BlueJ.

1. If you have not already done so, set the preferences to show the unit testing tools - Tools -> Preferences -> Interfaces and then select show unit testing tools
2. Create a unit test class - New Class -> Test Class. This will create a class for testing with the basic template filled out.
3. Create the unit tests. You can do this by directly adding the code or you can use the recording tool. There is a tutorial for testing in BlueJ. It was written for an older version of BlueJ but it is still useful.

You can find an example of JUnit which will be helpful for your lab on the assignment's Moodle page

The following resources might also be helpful to you in understanding and using Junit.

- Unit Testing in BlueJ
- Video (Youtube) for Unit Testing in BlueJ (uses Mac)

Assignment:

1. Create a class **RandomStringContainer**. This class should have an private **ArrayList** to store data.
    a. Create a constructor for the class to create an empty array list.
    b. Create a public method (*addToFront*) that takes one parameter, a String. The String will be inserted as the first element of the ArrayList, moving all the other elements up by one position.

    Hint: Before you start writing code, read the descriptions of the relevant methods in the API for ArrayList.

   c. Create a public method (*addToBack*) that takes one parameter, an String. The String will be inserted as the last element of the ArrayList.
   d. Create a public method (*addSorted*) that takes one parameter, a String. Assuming the ArrayList is sorted, the String should be inserted in the correct location that would keep the array sorted (you do not need to check if the array is sorted. Just assume it is).
   e. Create a public method (*prependSorted*) that takes one parameter, a String. Assuming the ArrayList is sorted, the String should be prepended to the first String in the ArrayList, which is then moved to the correct location that would keep the array sorted (you do not need to check if the array is sorted. Just assume it is).
   f. Create a public method (*selectionSort*) that sorts the ArrayList using the selection sort algorithm that is in your textbook (Section 4.2).
   g. Return a String array whose entries are equal to the ArrayList. This array should be used to perform your JUnit test using the assertArrayEquals method.

2. Create a class **RandomIntegerContainer**. This class should have an private **ArrayList** to store data.
   a. Create a constructor for the class to create an empty array list.
   b. Create a public method (*addToFront*) that takes one parameter, an Integer. The Integer will be inserted as the first element of the ArrayList, moving all the other elements up by one position.
   c. Return an Integer array whose entries are equal to the ArrayList. This array should be used to perform your JUnit test using the assertArrayEquals method.
3. Create an **ExperimentController** class. Create a private static String array (*twoLetterWords*) initialized to contain the following words, one

per element: aa ab ad ae ag ah ai al am an ar as at aw ax ay ba be bi bo by da de do ed ef eh el em en er es et ew ex fa fe gi go ha he hi hm ho id if in is it jo ka ki la li lo ma me mi mm mo mu my na ne no nu od oe of oh oi ok om on op or os ow ox oy pa pe pi po qi re sh si so ta te ti to uh um un up us ut we wo xi xu ya ye yo za

The class will have the following methods:

a. timeAddToFront(int numberOfItems, int seed):
    i. create an instance of a RandomStringContainer
    ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToFront()* method.
    iii. The method will return the time taken to add all the items to the container using *addToFront()*.
b. timeAddToBack(int numberOfItems, int seed):
    i. create an instance of a RandomStringContainer
    ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToBack()* method.
    iii. The method will return the time taken to add all the items to the container using *addToBack()*.
c. timeAddSorted(int numberOfItems, int seed):
    i. create an instance of a RandomStringContainer
    ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addSorted()* method.
    iii. The method will return the time taken to add all the items to the container using *addSorted()*.
d. timePrependSorted(int numberOfItems, int seed):
    i. create an instance of a RandomStringContainer
    ii. for the specified *numberOfItems*, prepend random words from *twoLetterWords* to the first element of the container using the *prependSorted()* method.

        iii. The method will return the time taken to add all the items to the container using *prependSorted()*.

    e. timeSortofUnsortedList(int numberOfItems, int seed):
        i. create an instance of a RandomStringContainer
        ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToBack()* method.
        iii. The method will call `selectionSort()`.
        iv. The method will return the time it took to sort the array.
    f. timeSortOfSortedList(int numberOfItems, int seed):
        i. create an instance of a RandomStringContainer
        ii. for the specified *numberOfItems*, insert random words from *twoLetterWords* to the container using the *addToBack()* method.
        iii. The method will call `selectionSort()`.
        iv. The method will call `selectionSort()` again.
        v. The method will return the time it took to sort the already sorted array.

4. Unit test the RandomStringContainer and RandomIntegerContainer classes.
5. Run your program through ExperimentController so that you test your program for **various sizes of input**. For each amount of data you should run multiple trials with different seeds. You then can create graphs where the y axis signifies the amount of time, and the x axis is the number of elements. More specifically:
    a. Compare the average run time of inserting from the front, back and sorted for different amounts of data.
    b. Compare the average run time of adding the elements and the sorting the unsorted list vs. inserting the entries using the addSorted for different amount of data.
    c. Compare the average run of sorting an unsorted array and sorting a sorted array for different amount of data.

Submission:

In addition to your code you must submit a report (in PDF format). Save your report in the project folder before you compress it and upload it. Details about the report can be found here.

Grading:

1. **ExperimentController** design and implementation. (2 pt)
2. **RandomStringContainer** and **RandomIntegerContainer** design and implementation (2 pt)
3. Unit testing (2 pt)
4. commenting+style(1 pt)
5. Report (3 pt)