Lab 1 – Implementing and Testing Simple ADTs in Java

This lab is a review of creating Java programs and an introduction to *JUnit* testing. First, you will document a Java interface and complete a documented Java class, which together implement a simple abstract data type (ADT), the *Tank*. You will then complete a *JUnit* test class and use it to test your class implementation. Finally, you will repeat a similar process for another ADT, the *Fraction*.

Objectives:

- to become re-acquainted with *DrJava*
- to construct simple but complete ADTs
- to use *JUnit* to test Java classes

Preliminaries:

- For every lab, you'll download a zip file from Canvas and unzip it into your ***CS200/Labs*** folder; for today's lab, this should create a ***CS200/Labs/Lab01_ADTs*** folder.
- Move (i.e., cut/paste OR drag/drop – DO NOT simply copy/paste as this will create redundant copies) the ***fraction*** and ***tank*** folders from your unzipped ***Lab01_ADTs*** folder into your ***~/JavaPackages*** folder. Keep all other files in your lab ***Lab01_ADTs*** folder.
- Please note submission instructions at the very end of this document

A Reminder on Pair Programming:

You have been assigned to work in pairs for the next few labs following the "*Pair Programming*" technique where two of you work together using a SINGLE computer. One of you, "the driver, writes code while the other, the observer or navigator, dedicates and reviews each line of code as it is typed in"; you are expected to spend lab time talking to each other each step of the way, discussing problems, agreeing on solutions and switching roles often. Please plan to meet and continue working on the lab in pairs after lab is over; you will demo and explain your solution at the START of the next lab. Lab partners will receive the same grade, so you are expected to have produced it collaboratively as described here.

For your convenience, you may have the lab writeup open on the observer's computer provided you don't write any code on that computer. Also, make sure to have solutions to related class exercises (*JavadocExample* and *JUnit4Example*) accessible during lab (again, open them on the observer's computer); in general, you'll find these very handy often serving as a starting point for lab solutions.

Problem 1: The Tank ADT

Part 1.1: Completing the javadoc documentation for interface Tank

Read entire part before starting

Your task for this part is to complete the *javadoc* comments for the *Tank* interface in *~/JavaPackages/tank/Tank.java*, using the guidelines discussed in class. You can think of the *Tank* as simulating a large container that holds some unspecified contents, such as water, oil, gasoline, or salt pellets. The *Tank* has a *capacity* that specifies the maximum amount it may contain with the minimum amount being 0.0.

Using a UML diagram format for an interface, the *Tank* looks like below (you should know what "**+**" means)

```
<interface> Tank
   + isEmpty     (): boolean
   + isFull      (): boolean
   + getCapacity (): double
   + getLevel    (): double
   + add         (amount: double): void
   + remove      (amount: double): void
```

The operating specifications for the *Tank* interface are below:

- Methods ***isEmpty*** and ***isFull*** check and return a *boolean* value appropriate to the condition being checked; your *javadoc* should specify under what condition they return *true*
- ***getCapacity*** returns the maximum level the contents of the *Tank* can reach
- ***getLevel*** returns the current level of the contents of the *Tank*
- ***add*** increases the level of the tank by parameter *amount* unless resulting amount exceeds *capacity*. Use the *@throws* tag to specify that method throws an *IllegalArgumentException* if this (pre)condition is violated
- ***remove*** reduces the level in an analogous way to *add* and should have a similar *@throws* clause (but, obviously, not the same!)

Note that the *Tank.java* (and later on *DoubleTank.java*) file includes the statement `package tank` at the very top. Recall that packages are used to group together related classes in Java. Java classes that belong to a package must begin with a *package* statement that gives the name of the package and must be located in a folder structure that mirrors the package name. In Java, package names begin with a lower-case letter. Other programs can use the *tank* package by including an `import tank.*` statement at the top.

Part 1.2: Implementing class DoubleTank

Open Java class *DoubleTank* given to you inside folder *tank*; this class must be defined inside package *tank* (like interface *Tank*) and must implement the *Tank* interface given to you in file *Tank.java*. Complete these two tasks first; note that incomplete/missing pieces in this class have been marked for you with the keyword "*NOTE*".

Class *DoubleTank* currently contains a constant variable and method headers for all methods in interface *Tank*, but does not include needed instance variables, additional methods, or constructors. Discuss the design of the *DoubleTank* class with your pair partner then complete the class following the specifications below. The class is called *DoubleTank* because it uses two instance variables of type double *level* and *capacity*.

The UML diagram for the *DoubleTank* class is:

```
<class> DoubleTank   (what does "-" below mean?)
   - level: double
   - capacity: double
   + DoubleTank () // default no parameter constructor
   + DoubleTank (capacity: double)
   + toString   (): String
   //the following are methods from interface Tank
   + add        (amount: double): void
   + remove     (amount: double): void
   + getLevel   (): double
   + getCapacity(): double
   + isFull     (): boolean
   + isEmpty    (): boolean
```

Below are additional specifications for this class (*Do NOT include additional methods and/or instance variables*)

- The default constructor sets the *level* to 0.0 and the *capacity* to a reasonable default value specified in the class as a constant (named *DEFAULT_CAPACITY* using all caps and underscores following Java conventions for constants).
- The second constructor sets instance field *capacity* to parameter *capacity* and *level* to 0.0. Like methods *remove* and *add*, this constructor should throw new *IllegalArgumentException* (with a DESCRIPTIVE ERROR MESSAGE as a parameter) if given a negative parameter *capacity*.
- The overridden ***toString*** method (from class *Object*) should return a string containing the *level* and the *capacity* along with some appropriate descriptive text. For example, for a tank with level 20 and capacity 100, it might return the following: `DoubleTank--level: 20.00, capacity: 100.00`

Before you implement the methods in your *DoubleTank* class, be sure to complete the missing *javadoc* comments for the two instance variables, the second constructor and the *toString* method.

When you have finished implementing the *DoubleTank* class, check that it compiles without errors. Run *DrJava*'s *javadoc* utility for files *Tank.java* and *DoubleTank.java* by selecting the **Javadoc All Documents** item from the **Javadoc** submenu of the **Tools** menu in *DrJava* while the two files (AND ONLY THESE) are open. Verify that you have generated valid *javadoc* pages.

Part 1.3: Testing class DoubleTank with a JUnit test class

A good way to test a class is to write a simple program that runs a set of tests on class objects and then verifies whether those test results match the expected results. One popular tool for doing this is the *JUnit* testing suite. With *JUnit*, you just write a simple testing class consisting of test methods. Each method performs a few simple tests and after each test, asserts what the result should be. *DrJava* already has *JUnit* built in, so once you have the test class written, all you have to do is click the test button.

In practice, we should NEVER implement a class in its entirety and test it afterwards, like we are doing now. Instead, it is advisable to implement a method in our class, and then test it thoroughly before moving on to the next method. However, we are still learning the basics of *JUnit* so this is fine BUT only for now.

We have included the start for a *JUnit* test program called *TankTest* for you to use. Keep it inside your ***CS200/Labs/Lab01_ADTs*** folder–DO NOT MOVE it to package *tank* like we did with files *DoubleTank.java* and *Tank.java*. This class must access the tested class and interface (i.e., *DoubleTank.java* and *Tank.java* in package *tank*) which is accomplished via an `import tank.*` statement to be added at the very top of this test class; so, start by adding this import statement. The test class currently contains three instances of the class to be tested.

You will add appropriate *@Test* methods to test class *DoubleTank*. Test methods needed to test the two constructors of class *DoubleTank* under normal conditions as well as when the *add* method fails are already completed for you (shown in blue below); for the remaining tests, you will need to supply test methods that accomplish the strategy given below. The provided *init()* method is preceded with *@Before* which makes it re-execute before each test method; as a result, every test method starts with three brand new *Tank* objects as initialized in the *init()* method.

Note that you can run your tests by clicking the "*Test*" button with the test file open in the edit window. Do not forget to annotate each of your test methods with *@Test*; otherwise, *DrJava* will simply ignore them!

Before you write a *JUnit* test class, consider your testing strategy. Accessor (or getter) methods (*isEmpty*, *isFull*, *getLevel*, *getCapacity*, and *toString*) are simple enough, that they do not need separate testing. We can test them by using them in the tests of other methods. For the constructors and *add* and *remove* methods use the following strategy:

- Test the initial state of newly constructed tanks (*these test methods are already completed for you; please take time to understand them*):
  - ***testDefaultConstructor***: test that a tank created using the default constructor is empty (this implicitly tests a 0.0 level) and has the expected default capacity
  - ***testSecondConstructor***: test that a tank created using the non-default constructor is also empty (again, tests a 0.0 level) but has the provided capacity
- Test ***add*** under normal conditions (create a *separate, clearly named* test method for each case as shown below; when writing test methods here or elsewhere, please follow naming conventions shown in examples here. This way we do not have to document our test classes).
  - ***testAddToEmptyTank***: adding to an empty tank gives the expected level; tank should no longer be empty
  - ***testAddToNonEmptyTank***: adding to a non-empty tank gives the expected level
  - ***testAddUpToCapacity***: adding to capacity produces a full tank

- Test ***remove*** under normal conditions (create a *separate*, *clearly named* test method for each case)
  - ***testRemoveFromANonEmptyTank***: removing from a non-empty tank gives the expected level and removing from a full tank produces a tank that is no longer full
  - ***testRemoveToEmptyATank***: removing all contents of a non-empty or a full tank produces an empty tank
- Test failure conditions (see note below on how these are done) --- create a *separate*, *clearly named* test method for each operation that might throw an exception in order to determine which operation actually causes the exception. (*First one below is already completed for you; study it carefully*):
  - ***testAddFailsWhenExceedingCapacity***: adding beyond specified capacity throws an exception
  - ***testRemoveFailsWhenExceedingLevel***: removing more than current level throws an exception

Remember that to test the failure conditions, put the operation that should produce an exception in a test method preceded by a `@Test(expected=`*DesiredException*`.class)` annotation.

- Test additional missed conditions: What have we failed to consider in the above tests? **There are three specifications somewhere in the *Tank* interface and/or the *DoubleTank* class that we have not tested for yet**. You will not find these problems with the *JUnit* tests described above; you need to find them in the specifications or figure them out on your own. When you do, add new tests for these conditions as well.

Once you complete the implementation and testing of the *Tank* ADT, show it to the TA or lab instructor.

Problem 2: The Fraction ADT

Part 2.1: Building and testing a Fraction ADT

Read entire part before starting

You are given a fully documented *Fraction* Java interface file inside package *fraction*, which should be in your `~/JavaPackages` folder (move it there if not). Follow the *Tank*/*DoubleTank* example from problem 1 to create a fully *javadoc*-documented *IntFraction* class that implements the *Fraction* interface; place your *IntFraction* class in the same *fraction* package as interface *Fraction*.

In addition to methods specified in the *Fraction* interface, the *IntFraction* class has two *private int* instance variables (*numerator* and *denominator*) as well as three constructors: a default constructor that creates a zero *Fraction* (*numerator*=0, *denominator*=1), a one parameter constructor that creates a *Fraction* representing an integer (*numerator*=parameter, *denominator*=1) and a two-parameter constructor that sets the *Fraction's* fields to the specified *numerator* and *denominator*.

Note class *TestFraction* located in your lab folder; this class currently includes a few *Fraction* instances and a complete *init()* method to test your *IntFraction* class as well as instructions to help you complete the test class. **Once you fully implement a method/constructor in class *IntFraction* and it compiles properly, complete corresponding test methods in class *TestFraction*. You may only proceed to the next method/constructor once all tests for current method run correctly**. As before, accessor methods `getNumerator` and `getDenominator` are simple enough, that they don't need separate testing.

Recall that fractions cannot have a zero denominator and division by zero is not allowed. Thus, the third constructor should throw an *IllegalArgumentException* if the denominator is 0. Similarly, the *divideThisBy* method throws an *ArithmeticException* if the divisor is a zero *Fraction*.

Consult with your lab partner, other students in the lab and/or the Internet to get the correct formulas for arithmetic on fractions. Note that two fractions may be equal even if they have different numerators and denominators (e.g., 2/3 == 4/6), so your *equals* method must take this into account. Again, use appropriate resources to determine the correct way to test fractions for equality.

Here is a shell for implementing the *equals* method:

```
public boolean equals(Object other) {
  try {
        Fraction otherFraction = (Fraction) other;

        // rest of your code here …

  } catch (ClassCastException cce) {
        return false;
  }
}
```

Once you complete the implementation and testing of the *Fraction* ADT, show it to the TA or lab instructor.

Part 2.2: Improving our Fraction ADT

Make a copy of your complete class *IntFraction* and rename it as *IntFraction2*. Class *IntFraction2* will improve things by putting fractions in simplified form; that is, the numerator and denominator should have no common divisor greater than one, and the denominator should always be positive. (If you have already done some of the simplification you have a head start.)

The first two constructors already put fractions in simplified form, but the third constructor does not, so that is where the main changes in *IntFraction2* need to go.

Factoring out the greatest common divisor will be easier if you have a method to compute it, so add a helper method to *IntFraction2* with the prototype: `private int gcd(int m, int n)`

Use Euclid's recursive algorithm to complete the function as follows:

1. if m < 0, return `gcd(-m,n)`
2. if n < 0, return `gcd(m,-n)`
3. if m = 0, return n
4. otherwise, return `gcd(n%m,m)`

Now, the third constructor can just compute the greatest common divisor and divide both the *numerator* and *denominator* by that value. If the *denominator* is negative, it changes the sign of both the *numerator* and *denominator*.

In simplified form, there is less likely to be an overflow on the arithmetic operations, although it can still occur. Also, simplifying fractions eliminates multiple forms for each fraction. (For example, -4/-6 gets simplified to 2/3.) In simplified form, two fractions are equal if and only if they have the same numerator and denominator, so the *equals* method can be simpler.

Repeat your testing process on your *IntFraction2* class. Note that the same test class should still work. Why? Simply use constructors from *IntFraction2* instead of *IntFraction* in the *init()* method.

Once you have completed the implementation and testing of *IntFraction2*, show it to the TA or lab instructor.

Submission Instructions

To submit (individual submissions!) --- please note that grading is based on the rubric sheet available in your lab folder

1. place a copy of your *~/JavaPackages* folder inside your *CS200/Labs/Lab01_ADTs* lab folder,
2. right-click on your lab folder,
3. select *Compress*
4. choose *.zip* from the file extension drop down menu after *Filename*
5. choose a meaningful *Location* to save the compressed zip file so that you may easily find it when you are ready to submit / upload to Canvas (I recommend saving it inside your lab folder)