Lab 1 – Implementing and Testing Simple ADTs in Java

This lab is a review of creating programs in Java and an introduction to JUnit testing. You will document a Java interface and create a fully 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. You will then 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:

- If you don't have a ***CS200*** folder in your Linux *home* (not your *desktop*), create one and place two new sub-folders inside of it; name them ***examples*** and ***labs***. Create two sub-folders in the ***examples*** folder; name them ***complete*** and ***incomplete***. Use the sub-folders inside ***CS200/examples*** to store complete examples and incomplete exercises from class, respectively.

- Copy folder ***/usr/people/classes/CS200/labs/Lab01_ADTs*** into your ***CS200/labs*** folder. For each new lab, copy the folder for that lab from ***/usr/people/classes/CS200/labs*** into your ***CS200/labs***.

- If you do not have a ***JavaPackages*** folder in your Linux *home* directory, create one now. Be sure to spell the folder name exactly as specified here (case sensitive). We will use this folder throughout the course to store classes and packages we develop that can be reused in multiple applications.

- Move the *fraction* and *tank* folders from your ***Lab01_ADTs*** folder into your *JavaPackages* folder

- Launch DrJava from *Applications→CSBSJU→Computer Science* to configure it properly:
  - ✓ Choose the "*Preferences*" option from the "*Edit*" pull-down menu
  - ✓ Select the "*Resource Locations*" category on the left. This allows us to identify resources that DrJava needs to compile and run our programs
  - ✓ Enter (or select) the ***tools.jar*** location to let DrJava know where to find it: ***/opt/java-jdk/lib/tools.jar*** if there is not already an entry there or if the entry is Java 1.6 or lower. When you enter this file, it will change to the current default on the system, which will be somewhat longer and more mysterious
  - ✓ If not there already, add each of the following paths to your "*Extra ClassPath*" by clicking the "*Add*" button just below the area next to "*Extra Classpath*":
    (1) ***/usr/people/classes/Java/***
    (2) ***~/JavaPackages*** (~ in Linux represents your home directory)
  - ✓ Click the "*Javadoc*" category and set "*Access Level*" to "*package*"
- Dismiss the "*Preferences*" window by pressing *Apply* and then *OK*

You will be assigned to work in pairs for the next 4 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, reviews each line of code as it is typed in"; switch roles after *every part*. Please continue to work in pairs after lab is over; you will demo and explain your solution AT THE START OF THE next time. Lab partners will receive the same grade.

Problem 1: The Tank ADT

Part 1.1: Completing the javadoc documentation for an interface

Read this entire section before starting

Your task for this part is to complete the javadoc comments for the *Tank* interface in the *tank/Tank.java* file, 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; of course, the minimum amount it may contain is 0.0.

Using a UML diagram format for an interface, the *Tank* looks like:

```
<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:

- `isEmpty` and `isFull` check and return a `boolean` 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 `amount`, unless the resulting amount would exceed the *capacity*.
  - ✓ Use the precondition part of the comment to specify the corresponding restriction on the value of `amount`.
  - ✓ Use the *@throws* tag to specify that the method throws an *IllegalArgumentException* if the precondition is not met
- `remove` reduces the level in an analogous way to *add()* and has similar preconditions and *@throws* clause

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 put together groups of related classes in Java. Java files 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 the statement "`import tank.*;`" with the other import statements at the top.

Please show the complete javadoc documentation for *Tank.java* to your lab TA or instructor before you proceed.

Part 1.2: Implementing the DoubleTank class

Create a new *DoubleTank* Java class inside folder *tank* ; this class must be defined inside package *tank* (like *Tank.java*, you should include a "`package tank;`" statement at the top of the file) and must implement the *Tank* interface given to you in file *Tank.java*. You can use the interface file as the starting point in writing the *DoubleTank* class. Copy the body of the interface file, *Tank.java*, and paste it inside *DoubleTank.java* to get started. The interface file contains method headers for most methods, but does not include instance variables or constructors. Discuss the design of the *DoubleTank* class with your pair partner then complete the class design and put in the appropriate javadoc comments using the UML diagram of the *DoubleTank* class and the specifications above (for *Tank.java*) and below to assist you. 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
- level      : double
- capacity   : double

+ DoubleTank ( )
+ DoubleTank (capacity : double)
+ toString   ( )  : String
+ add        (amount : double) : void
+ remove     (amount : double) : void
+ getLevel   ( )  : double
+ getCapacity( )  : double
+ isFull     ( )  : boolean
+ isEmpty    ( )  : boolean
```

Additional specifications for the *DoubleTank* are listed below (*Do not add/remove any methods and/or fields*)

- the default constructor sets the `level` to 0.0 and the `capacity` to a reasonable default value which is specified in the class as a `static final double` value, and following Java conventions is named using all caps and underscores (e.g.: `DEFAULT_CAPACITY`)
- the second constructor sets the internal `capacity` to parameter `capacity` and the `level` to 0.0.
- the overridden `toString` method should return a string containing both the `level` and the `capacity` along with some appropriate labelling text, beginning with the class name. For example, for a tank with level 20 and capacity 100, it might return "`DoubleTank--level: 20.00, capacity: 100.00`"
- the second constructor as well as methods `remove` and `add` should throw new *IllegalArgumentException ("INCLUDE A DESCRIPTIVE ERROR MESSAGE HERE FOR THE USER") if* their preconditions aren't met

Be sure to modify the header javadoc comments in your *DoubleTank* class appropriately and add similar comments to the two constructors and the *toString* method.

When you have finished implementing the *DoubleTank* class, be sure that it compiles without errors or warnings. 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 THESE ONLY) are open. Verify in a web browser that you have generated valid javadoc pages.

Part 1.3: Testing the DoubleTank class with a JUnit test class

One of the best ways 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 tool for doing this that has become very popular 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.

We have included the start for a JUnit test program called *TankTest* for you to use. Keep it in your lab folder OUTSIDE package *tank*. This class must access the tested class i.e., *DoubleTank.java* and *Tank.java*, which is accomplished via the "`import tank.*`" statement at the very top. It contains three instances of the class to be tested. You will add appropriate methods to test the *DoubleTank*. Test methods needed to test the two constructors of class *DoubleTank* are already given for you; 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, each test method starts with two brand new *Tank* objects in the initialized states.

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*.

Before you writing a JUnit test class, consider your testing strategy. Accessor methods (`isEmpty`, `isFull`, `getLevel`, `getCapacity`, and `toString`) are simple enough, that they don't 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 (*NOTE: test methods here are already completed for you*):
  - test that a tank created using the default constructor is empty (this implicitly tests a 0 level) and has the expected default capacity
  - test that a tank created using the non-default constructor is also empty (this implicitly tests a 0 level) but has the provided capacity

- Test the `add` method under ordinary conditions (create a ***separate, clearly-named*** test method for each case; when writing test methods here or elsewhere, please follow naming conventions used below)
  - `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
  - `testAddToCapacity`: adding to capacity produces a full tank

- Test the `remove` method under ordinary 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 not 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
  - `testAddFailsWhenExceedingCapacity`: adding more than remaining 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 an `@Test(expected=DesiredException.class)` annotation.

- Test unanticipated conditions: These are the tricky ones. What have we failed to consider in the above tests? Are there any situations that will fail that we haven't tested? Are there any situations that should fail, but don't? There are three specification deficiencies (errors) somewhere in the *Tank* interface and the *DoubleTank* class (unless you had already discovered and corrected them). You won't find these problems with the JUnit tests described above; you need to figure out what the unanticipated conditions are. Consider what kind of values you haven't tested and add tests for what happens if you use those values. (There's a hint at the end of this writeup, but don't cheat and look at the hint before you try to figure it out for yourselves.) When you find the errors, go back and correct the interface and class specifications and add code to handle these situations. After you've made your corrections, revise the test class so that it adds 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

Create a fully javadoc-documented *IntFraction* class that implements the *Fraction* interface, found in the *fraction* folder. (Think about why it is in a folder with that name.) Use the *Fraction* interface as a template for the *IntFraction* class and place your class in the same *fraction* folder.

In addition to the 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 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 your *IntFraction* class compiles properly, complete class *TestFraction* located OUTSIDE PACKAGE *fraction*.

Part 2.2: Improving our Fraction ADT

IntFraction class will perform better if we put the 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've 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 doesn't, so that's where the main changes 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 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.) With 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 revised *IntFraction* class. Note that the same test class should still work. Why?

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

(Hint for the testing: the problems are in the *add()* and *remove()* methods and in the second constructor when used with negative values.)