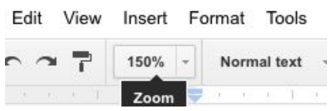
CMPSCI 187 (Fall 2018) Lab 01: JUnit and Debugging

The goal of this lab is to 1) meet your TA and get familiar with lab sessions and the Gradescope submission system; 2) get familiar with JUnit and debugging in Eclipse. Each lab assignment is created as a shared Google doc. To work on the assignment:

- Go to ***File -> Make a Copy*** to make an editable copy of this Google Doc for your account (the document shared with you is READ-ONLY, so you cannot edit it until you make a copy).
- Follow the instructions to complete the assignment. To zoom in, please use the zoom feature in your Google doc (see picture on the right). Do NOT use your browser's zoom feature.



- When you are done, go to ***File -> Download As -> PDF Document.*** You must select PDF, as Gradescope does not accept other formats.
- **Submit your PDF in** Gradescope **under Lab 01**. Note: submit to Lab 01, **not** to Project!

Section A: Java Exercise -- Fill in the Blanks (3 points)

Each of the following code has a problem / bug that causes either a compilation error or a run-time exception/error. Describe the problem for each case in **one sentence**.

(a)
```
int[] array = {1, 2, 3, 4, 5};
System.out.println(array[5]);
```

Index 5 in `array` does not exist, as it is out of bounds.

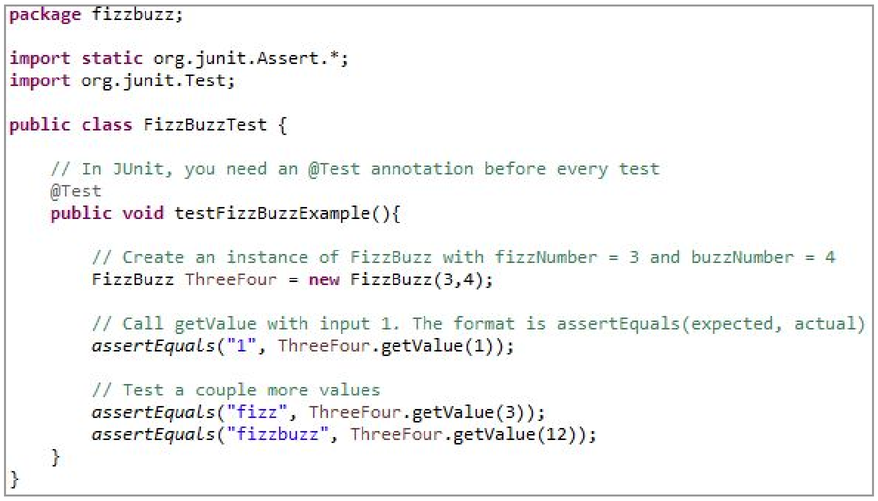
(b)
```
float a = 0.187;
System.out.println(a);
```

The default type of the constant 0.187 is `double`, which cannot fit into a `float` variable.

(c)
```
String name;
System.out.println(name.length());
```

The variable `name` is currently `null`, so the `length()` method cannot be called on it.

Section B: JUnit Tests [4 pts]
JUnit provides an easy way for you to test whether your program's output matches the expected output for each specific test case. Below is a sample JUnit test for the FizzBuzz project:



```java
package fizzbuzz;

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

public class FizzBuzzTest {

    // In JUnit, you need an @Test annotation before every test
    @Test
    public void testFizzBuzzExample(){

        // Create an instance of FizzBuzz with fizzNumber = 3 and buzzNumber = 4
        FizzBuzz ThreeFour = new FizzBuzz(3,4);

        // Call getValue with input 1. The format is assertEquals(expected, actual)
        assertEquals("1", ThreeFour.getValue(1));

        // Test a couple more values
        assertEquals("fizz", ThreeFour.getValue(3));
        assertEquals("fizzbuzz", ThreeFour.getValue(12));
    }
}
```

- The test is constructed as a Java class (`FizzBuzzTest` in the above example) and it contains one or more public methods, each a test case.
- Each public method is annotated with the directive `@Test` above the method name. You can optionally provide parameters for the directive. For example, `@Test(Timeout=1000)` specifies that the test must finish within 1000ms (1 second).
- Each method contains one or more calls to `assertxxx` which check if your program's outputs match the expected values. Common `assertxxx` methods that you will encounter are **`assertEquals, assertTrue, assertFalse`**. For example, **`assertEquals`** checks if two values are equal (these can be two integers, or two strings, etc. as long as the two values are of the same type). **`assertTrue`** and **`assertFalse`** check if a boolean value or a boolean statement is true or false. If the assertion fails, JUnit will report a test failure.
- After this lab, you should check the Appendix at the end of this Google Doc for more detailed JUnit information.

Based on the above information, complete the following exercise. Let's say you have written a `myMath` class that contains a method **`boolean isOdd(int x)`** which returns `true` if x is an odd integer, and `false` if x is an even integer. Complete the JUnit tests below to check whether your implementation is correct or not.

```java
@Test
public void test_isOdd() {
    myMath math = new myMath();
    assertTrue(math.isOdd(3));          // example test case for 3
    assertEquals(true, math.isOdd(3));  // this is an equivalent test

    // Write a test case for 0 (hint: is 0 an odd number or even number?)
    // you may choose to use any of the assertxxx methods as you see fit
    assertFalse(math.isOdd(0));
    // Write a test case for -2
    assertFalse(math.isOdd(-2));
}
```

Now, let's say you have written a `myList` class that keep track of a list of integer elements. You can add or delete elements from the list, and you have implemented **`int getNumElements()`** method that returns the number of elements currently in the list. Complete the JUnit tests below to check whether your implementation is correct or not.

```
@Test
public void test_getNumElements() {
    myList list = new myList();

    // a newly created list should have no element. Write a test case for it.
    assertEquals(0, list.getNumElements());

    list.add(10);
    list.add(20);
    list.add(30);

    // Write a test case to check the number of elements at this point.
    assertEquals(3, list.getNumElements());
}
```

Section C: Debugging

There are several debugging methods you can use to help diagnose your program and find out why it's not producing the correct output. It's crucial that you get familiar with the debugging tools -- for complex programs it's nearly impossible to eyeball the mistakes in the programs without using any debugging tool.

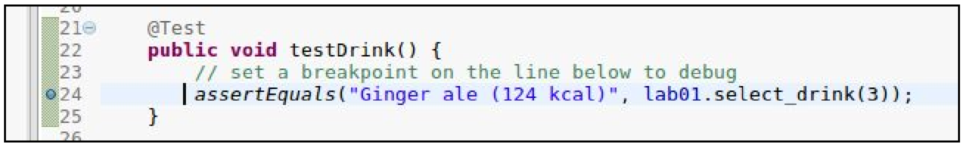
- To begin with, you can insert `System.out.println` at various places in your program to print out variable values. This way you can monitor the variable values as the program runs.
- You can place **breakpoints** in your code, and run the program in Debug mode. The program execution will stop as the breakpoints, allowing you to 1) check variable values by simply moving your mouse pointer to the variable names; 2) step through the code line by line, by using either Step Over, or Step Into; 3) resume the program execution until it encounters the next breakpoint, or the program terminates.
- 'Step Over' means when it encounters a method call, such as **`ThreeFour.getValue(3)`**, it will get the return value immediately without going into the method. 'Step Into' means it will jump into the **`getValue`** method and step through every line of code of that method.
- You can place breakpoints anywhere in the code, including the JUnit Tests.
- See Appendix at the end of this Google Doc for more detailed debugging information

In the next part of this lab, you will practise using breakpoints in Eclipse to help discover program bugs and fix these bugs.

Step 1: Download **lab01-debug.zip** from Piazza and import it into Eclipse just like how you import the first project.

Step 2: Use the Debugger to find bugs in Lab01Debug.java
When you run your JUnit tests (`Lab01Test.java`), the tests will fail because there are bugs in `Lab01Debug.java`. Your job is to use breakpoints and debugging tools to find out the bugs and fix them:

- If a JUnit test fails, you can **double click on the test** and Eclipse will jump to the specific line of test that failed. On the lower portion of the JUnit test sub-window, you will see messages such as "***expected value is xxx, but actual value was yyy***."
- Set a breakpoint by double clicking the area left to the line number. A **circular marker** will appear to indicate a breakpoint has been set.

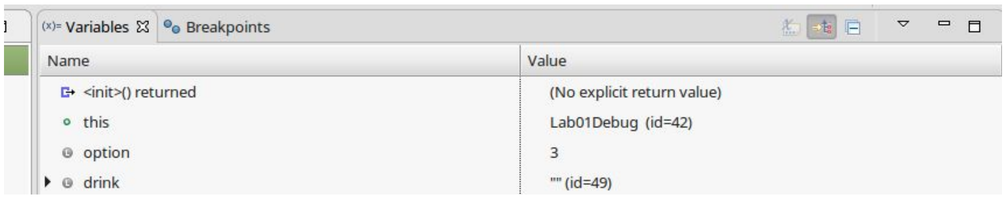


```
    @Test
    public void testDrink() {
        // set a breakpoint on the line below to debug
        assertEquals("Ginger ale (124 kcal)", lab01.select_drink(3));
    }
```

- Click on the **Debug button** (the button with a little bug icon, or go to Menu Run -> Debug, or use hotkey F11). Select **Yes** if Eclipse prompt you to change to Debug Perspective. The program execution will stop at the breakpoint, then you can click on **Step Into** (see the icon below, or go to Menu Run -> Step Into, or use hotkey F5). In the above example, the execution will now continue into the `select_drink` method, where you can click on **Step Over** (or go to Menu Run -> Step Over, or use hotkey F6) to step through the code and see why the method is returning the wrong value. Remember, Step Into will cause the execution to continue into a method call, whereas Step Over won't.

  (**Debug**) (**Step Into**) (**Step Over**) (**Stop**) (**Perspective**)

- While debugging, you can **move your mouse pointer onto any variable** and the debugger will show you the value of that variable. In the Debug perspective, the variables will also show up automatically in the Variables sub-window (as shown below). This allows you to monitor the variables and examine exactly when the variable values have changed.



Variables | Breakpoints

Name	Value
<init>() returned	(No explicit return value)
this	Lab01Debug (id=42)
option	3
drink	"" (id=49)

- You can stop the Debugging process by pressing the **Stop** button (red square). Change back to the Java perspective by clicking on the Java Perspective button (see above).

Step 3: Find the bug in each method and explain how to fix the bug. [3 points]

Explain the bug in **`select_drink()`** method and how to fix it:

The `if` statements are multiple lines long but aren't bracketed. As a result, only the first line's execution is dependent on the condition, while the following lines execute regardless. In this case, the executed code for each `if` statement can be consolidated into one line, but the general solution would be to place curly brackets around the code to be executed in the `if` statement.

Explain the bug in **`select_food()`** method and how to fix it:

The `switch` statement doesn't stop after one `case` - instead, it executes every subsequent `case`, ending with `default`. Thus, `food` is always assigned "`Invalid`", since that is the last statement to be executed. To fix this, `break` statements should be placed after each case.

Explain the bug in **`circular_left_shift()`** method and how to fix it:

The first value of `a` is overwritten with the second value of `a` at the beginning of the `for` loop. The last value of `a` is later overwritten with the first value of `a`, which is no longer what is originally was, but rather what it was overwritten with earlier. The solution is to store the first value of `a` in a temporary variable and replace the last value of `a` with the stored value.

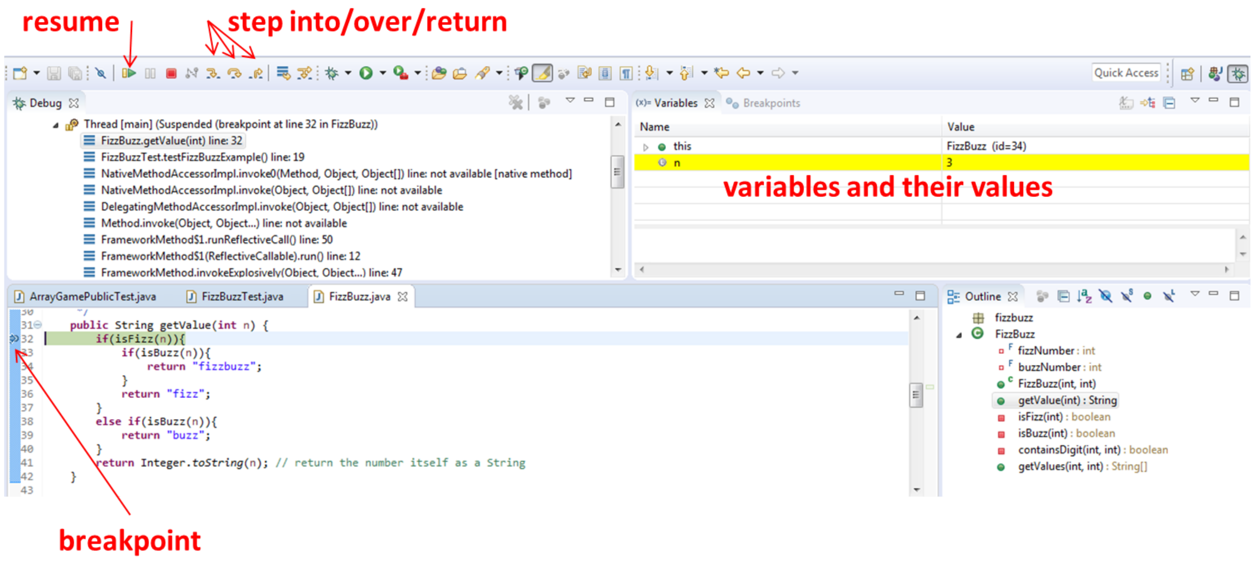
Once you finish exercises above, you can fix the bugs in the code and run JUnit test again. Now all tests should pass.

(The END)

Appendix

- More detailed explanation of JUnit:
  - http://www.javatpoint.com/junit-tutorial

- More detailed explanation of Eclipse's Debugger:
  - http://www.mscs.mu.edu/~frahman/COSC/Eclipse_Debug.pdf
  - Eclipse debugger buttons (in FizzBuzz example):



```
Thread [main] (Suspended (breakpoint at line 32 in FizzBuzz))
    FizzBuzz.getValue(int) line: 32
    FizzBuzzTest.testFizzBuzzExample() line: 19
    NativeMethodAccessorImpl.invoke0(Method, Object, Object[]) line: not available [native method]
    NativeMethodAccessorImpl.invoke(Object, Object[]) line: not available
    DelegatingMethodAccessorImpl.invoke(Object, Object[]) line: not available
    Method.invoke(Object, Object...) line: not available
    FrameworkMethod$1.runReflectiveCall() line: 50
    FrameworkMethod$1(ReflectiveCallable).run() line: 12
    FrameworkMethod.invokeExplosively(Object, Object...) line: 47
```

Name	Value
this	FizzBuzz (id=34)
n	3

ArrayGamePublicTest.java FizzBuzzTest.java FizzBuzz.java

```
30      */
31      public String getValue(int n) {
32          if(isFizz(n)){
33              if(isBuzz(n)){
34                  return "fizzbuzz";
35              }
36              return "fizz";
37          }
38          else if(isBuzz(n)){
39              return "buzz";
40          }
41          return Integer.toString(n); // return the number itself as a String
42      }
43
```

```
fizzbuzz
FizzBuzz
    fizzNumber : int
    buzzNumber : int
    FizzBuzz(int, int)
    getValue(int) : String
    isFizz(int) : boolean
    isBuzz(int) : boolean
    containsDigit(int, int) : boolean
    getValues(int, int) : String[]
```