EECS1021 Winter 2019
Guide to Lab Test 3
When: Week of March 25

Chen-Wei Wang

- This in-lab programming test accounts for 10% of your course grade.
- This lab test is **purely** a programming test, assessing if you can write **valid** Java programs free of syntax and type errors.
- **Stringent Requirements**:
  - It is absolutely critical that you submit Java code that compiles (i.e., no red crosses shown on the Eclipse editor). You receive a **zero** for the test if your code contains any compilation errors (i.e., syntax errors or type errors).
  - You are **only allowed** to use primitive arrays (e.g., `int[]`, `String[]`, *etc.*) for declaring attributes and implementing methods. Review the programming pattern that is covered in the **tutorial videos 40 to 46** and in **this written notes**. Any use of the Java collection library (e.g., `ArrayList`, `LinkedList`, *etc.*) will result an **immediate zero** of this lab test.
- In this lab test you will be required to code Java classes and methods in Eclipse. **Only** a tester class (similar to the `AccountTester` and `SMSTester` that you were given in Lab 5 and Lab 6) and its expected console output will be given to you. The Tester class illustrates how instances of classes are supposed to be created, and how methods may be invoked on their instances. From this tester class, which does **not compile** to begin with, you are required to:
  1. Identify the missing classes and methods (constructors, accessors, and mutators).
  2. Create all missing classes, as well as add headers of all required methods (i.e., name, return type, input parameters, and `return` default values for accessors).

     Completing the above two tasks should make everything **compile**.
  3. Implement the required methods (with any extra attributes or helper methods which you consider necessary), such that **executing the given tester produces the expected console output**.

Coverage

- Tutorial Videos for Lab 5 and Lab 6
- Exercises for Lab 5 and Lab 6
- Refer to the lecture materials (slides, recordings, and example codes) on Classes and Objects if necessary.
- The programming pattern (using a primitive array attribute and an integer counter attribute) is also discussed in **this written notes**. [click on link]

1 Rationale for the Grading Standard

The two most important learning outcome of this course are:

1. Computational thinking (for which you build through lab exercises and assessed by quizzes and exam)
2. Being able to write *runnable* programs (for which you are assessed through computer tests)

When you write an essay, if there are grammatical mistakes, it can still be interpreted by a human. Computer programs are unlike essays: when your program contains compile-time syntax or type errors, it just cannot be run, end of story. When a computer program cannot be run, its runtime behaviour is simply unknown; and this is particularly the case when your program contains if-statements and loops.

When you land a job upon graduation, you would not expect your supervisor or colleagues to read your code that does not run, because it does not even compile, would you? True, this is only your second programming course, and you're still learning. But it is exactly this mind set that restricts your potential of becoming a competent programmer. If we want to train you to be a competent programmer, NOW is the time to enforce the strict (but justifiable) standard. This is also why I make your Lab Test 1 weight less than others, so that if you really can't make it, it would not cause you to fail and you still have a good chance to obtain a decent grade, as long as you take proper action (e.g., more time spent on practicing to program) to strengthen your programming skills.

2 Rules

- The test takes place **in the beginning** of your registered lab session.
- You will be given **90 minutes** to complete the test.
- If your submitted classes **altogether compiles** with the given Tester class, then you already receive **10 marks** (out of 100) of the test.
- To determine the remaining 90% of your marks, we will run test cases on your submitted classes.

  Say we run 10 test cases (and say they happen to have equal weights) on your submitted code, and your submitted code complies and passes 6 of them, then your final marks are: $10 + 90 \times \frac{6}{10} = 64$.
- During the test, you are **required** to use Eclipse (but not any other programming tools) to develop your Java code.
- You are **allowed** to use a piece of paper (**blank** on both sides) for sketching your ideas. No sketch papers will be provided to you.
- You must show up for your **registered session only**.
- Wait outside WSC106/WSC108 when you arrive. The TAs will let you enter the room once the rooms are set up properly.
- Bring a piece of photo ID.
- A non-programmable calculator is allowed
- Put your mobile phone in your bag, and put your bag underneath the table.
- No data sheet will be allowed.

3 A Small Example

3.1 What You Are Given

3.1.1 A Tester

This tester class must not be modified.

```
1  public class CounterTester {
2    public static void main(String[] args) {
3      Counter c1 = new Counter();
4      Counter c2 = new Counter(10);
5      int c1Value = c1.getValue();
6      int c2Value = c2.getValue();
7      System.out.println("========(1)");
8      System.out.println("c1: " + c1Value);
9      System.out.println("c2: " + c2Value);
10
11     c1.increment();
12     c2.increment();
13     System.out.println("========(2)");
14     System.out.println("c1: " + c1.getValue());
15     System.out.println("c2: " + c2.getValue());
16
17     c1.increment(3);
18     c2.increment(3);
19     System.out.println("========(3)");
20     System.out.println("c1: " + c1.getValue());
21     System.out.println("c2: " + c2.getValue());
22   }
23 }
```

3.1.2 Expected Output of Executing the Tester

Executing the given tester and "completed" class(es) must produce the following console output:

```
========(1)
c1: 0
c2: 10
========(2)
c1: 1
c2: 11
========(3)
c1: 4
c2: 14
```

3.2 What You Are Required to Do

1. Look at the tester class (Section 3.1.1):

   1.1 Identify missing classes and Create empty classes.

   Principle 1: On the left-hand side of a variable assignment (=), if the type refers to the name of some non-existing class, then you must create that class.

   For example, Line 3 and Line 4 suggest that a new class `Counter` is needed for the declaration of variables `c1` and `c2`'s types. You should then start by first creating a new, empty class accordingly:

```
class Counter {

}
```

   The actual lab test might require you to create multiple new classes, but the same principle applies.

   1.2 Identify constructors.

   Principle 2: On the right-hand side of a variable assignment (=), if there is a `new` keyword, then the class name that follows indicates a call to a constructor of that class.

   For example, Line 3 and Line 4 suggest two versions of constructor for the `Counter` class (i.e., the constructor is *overloaded*): one version that takes no parameters, and the other that takes an integer parameter.

   Consequently, we should add these two constructor declarations (with no implementations) to the `Counter` class:

```
Counter () { }
Counter (int value) { }
```

   1.3 Identify accessors.

   Principle 3: If a method call appears on the right-hand side of a variable assignment (=), or as the input of a `System.out.println` call, then that method should be an accessor method.

   For example, Lines 5, 6, 8, 9, 14, 15, 20, 21 suggest that `getValue` is an accessor method with no parameters.

   Which class should `getValue` added to? Look at the **context objects** of the method calls: `c1` and `c2` are declared of type `Counter`, so the `getValue` method should be declared there.

   What should be the return type of `getValue`? Look at lines such as Line 5 and Line 6, which indicate the type of variable that stores the return value.

   Consequently, we should add this accessor method declaration (which only returns a **default value**) to the `Counter` class:

```
int getValue() {
  int result = 0; /* 0 is the default value of the return type int */
  return result;
}
```

   1.4 Identify mutators.

   Principle 4: If a method call appears as the entire line, then that method should be a mutator method.

   For example, Lines 11, 12, 17, 18 suggest that `increment` is a mutator method. More specifically, the `increment` is *overloaded*: Lines 11 and 12 suggest one version of `increment` that takes no parameters, whereas Lines 17 and 18 suggest a second version that takes an integer parameter.

Which class should `increment` added to? Look at the context objects of the method calls: `c1` and `c2` are declared of type `Counter`, so the `getValue` method should be declared there.

What should be the return type of `increment`? All mutator methods have the `void` return type.

Consequently, we should add these one accessor method declaration (with no implementations) to the `Counter` class:

```
void increment() { }
void increment(int value) { }
```

2. Modify the "empty class(es)" (e.g., `Counter`):

2.1 Add headers of the identified methods (just for compilation).

Based on the identification of the constructors, accessors, and mutators, we end up with:

```
class Counter {
  Counter () { }
  Counter (int value) { }
  int getValue() {
    int result = 0; /* 0 is the default value of the return type int */
    return 0;
  }
  void increment() { }
  void increment(int value) { }
}
```

Principle 5: The above expanded `Counter` class and the given `CounterTester` class now **compile**. However, executing the tester class will not produce the expected console output (Section 3.1.2).

2.2 Complete implementations of methods (for producing the expected output).

Principle 6: Complete implementations of all methods, by observing method calls in the tester class (Section 3.1.1) and their corresponding console output (Section 3.1.2). Additional attributes (class-level variables) and helper methods are allowed if considered necessary.

Consequently, here is the final working version of the `Counter` class:

```
class Counter {
  int value; /* attribute */
  Counter () {
    value = 0;
  }
  Counter (int value) {
    this.value = value;
  }
  int getValue() {
    return value;
  }
  void increment() {
    this.value ++;
  }
  void increment(int value) {
    this.value += value;
  }
}
```

Note. You can assume that in the test **all** classes will be placed inside the same `default package`, so there is no need to put the modifier `public` to the front of classes and methods.

4 Preparation Exercise for Lab Test 3

It is important that you apply principles discussed in Section 3 when solving the practice test.

4.1 Background

A coffee shop has a list of up to 100 members. Each member has a *unique* member id (i.e., `"mem1"`, `"mem2"`, `"mem3"`, and so on), a list of up to 30 current orders, and a balance of their account (e.g., `40.5`). Each order is characterized by the name of product (e.g., `"Americano"`, `"Cafe Lattee"`, *etc.*), its unit price (e.g., `4.7`), and the quantity (e.g., `4`).

Given an order, we may get its product name, unit price, or quantity. Given a member, we may add an order to their basket, or we may get their unique id, current balance, current list of orders, or amount to pay (based on the orders they have added so far). Given a shop, we may add a member, check if an id is an existing member id, or return the current list of members. Given a member id, if it exists, we may check out that member's current orders, by deducting their balance and clearing their charged orders accordingly.

Hint. To implement the unique member id's, you will need to use the `static` construct in Java. We will cover this later in the class, but you can find it here:

- Slides 67 to 74 in the Classes and Objects lecture. [click the link]
- Part of a previous lecture for EECS1022 (from 28:00 to 52:40). [click the link]

4.2 Starter Code

- Click on **this link** to download the tester and its expected output.
- You are required to write, in valid Java syntax, classes, attributes, and methods to implement the above (informal) system requirements. Study the `ShopTester` class and its expected output carefully. It indicates the classes and headers of methods that you need to define. You are forbidden to define additional classes, whereas you are free to declare extra attributes or helper methods as you find necessary.
- Here are requirements that you must follow **stringently** for the actual lab test:
  1. Use of any Java collection library classes is forbidden.
  2. **Nowhere** in all methods that you define can contain **any** print statements.
  3. **All** attributes declared in your classes should **not** be declared as `private` or `public`. For example, the following class

```
1 class A {
2   int i;
3   String s;
4 }
```

  is **acceptable** and there's no need to declare attributes `i` and `s` as `public` or `private`.

4.3 Solutions

Click on **this link** to download the solution project.