EECS1022 Winter 2018
Guide to Lab Test 3

Chen-Wei Wang

In this lab test you will be required to **create** and **implement** Java methods in Android Studio.

Contents

1 Coverage

- Lab 5: this tutorial series and this tutorial document.
- Lecture notes on Classes and Objects

  No 2-D arrays will be covered in this lab test.

2 Format

- You will be given a tester class and several "empty" classes (with no attributes or methods). The tester class illustrates how instances of these classes are supposed to be created, and how methods may be invoked on these instances. These "empty" classes together with the tester class do not compile to begin with (due to the lack of implementation in the "empty" classes).
- Your tasks are then:

  1. Identify the constructors, accessors, and mutators from the tester
  2. Add signatures of all required methods and `return` default values for accessors.

     Completing the above two tasks should make everything **compile**.
  3. Add all other necessary attributes to these "empty classes" and complete the implementations of methods, so that running the tester class produce the **expected console output**.

     Your submitted solution **must** compile.

3 Rules

- This lab test is **purely** a programming test:

  - You are forbidden to use Java library classes and methods, e.g., `ArrayList`, `Math`, `Arrays.sort`, *etc.*.
  - If your submitted Java classes **altogether** do not compile due to any syntax or type errors, you receive **zero** marks for the test. There will be no partial marks rewarded to a class that does not even compile.

  This requirement will be imposed strictly, so your best preparation strategy is to familiarize yourself with all basic syntax of Java that's so far covered in the lectures and used in your labs.
- If your submitted class compiles, then you already receive **30 marks** (out of 100) of the test.
- To determine the remaining 70% of your marks, we will run test cases on your submitted class.

  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: $30 + 70 \times \frac{6}{10} = 72$.
- You will be given **80 minutes** for the lab test.
- You must show up for your registered session only.
- Bring a piece of photo ID.
- No mobile phone usage is allowed during the test.
- No data sheet will be allowed.
- You may bring pen/pencil and a piece of blank paper for sketching your solutions.

4 A Small Example

4.1 What You Are Given

4.1.1 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 }
```

4.1.2 Empty Class(es)

This "empty" class is expected to be expanded with attributes, constructors, accessors, and mutators.

```
class Counter {

}
```

4.1.3 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
```

4.2 What You Are Required to Do

Line 3 and Line 4 in the `CounterTester` class (Section 4.1.1) suggest that a new class `Counter` is needed for the declaration of variables `c1` and `c2`'s types. This is why an "empty" `Counter` class is given to you. When there are more new classes involved, you will be given more "empty" classes.

What you are given (tester in Section 4.1.1 and an empty class in Section 4.1.2) does not compile to start with. Here are the recommended steps for you to follow:

1. Look at the tester class (Section 4.1.1):

   1.1 Identify constructors.

   Principle 1: 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 (Section 4.1.2):

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

   1.2 Identify accessors.

   Principle 2: If a method call appears on the right-hand side of a variable assignment (=), or as the argument 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 these one accessor method declaration (which only returns a **default value**) to the `Counter` class (Section 4.1.2):

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

   1.3 Identify mutators.

   Principle 3: 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 (Section 4.1.2):

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

2. Modify the "empty class(es)" (Section 4.1.2):

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

   Based on the identification of the constructors, accessors, and mutators, we end up with an expanded version of the `Counter` class:

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

   Principle 4: 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 4.1.3).

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

   Principle 5: Complete implementations of all methods, by observing method calls in the tester class (Section 4.1.1) and their corresponding console output (Section 4.1.3). Additional attributes (class-level variables) might be 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;
  }
}
```

5 Preparation Exercise for Lab Test 3

5.1 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.

5.2 Starter Code

Click on **this link** to download the tester, "empty" classes, and 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 three classes (i.e., `Order`, `Member`, and `Shop`) and signatures of methods that you need to define. You are forbidden to define additional classes, whereas you are free to declare attributes or helper methods as you find necessary.

Here are two requirements that you must follow **stringently**:

1. **Nowhere** in all methods that you define can contain **any** print statements.
2. **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`.

If any of the above requirements are not followed, you will receive low marks for your answers.

6 Solutions

We will go over the solution to the preparation exercises during the remediation period.