Tutorial Section:
☐ Monday at 8:30 B1
☐ Friday at 2:30 B2

COMP 1406B In-Class Test #2 (W2018)

Name:_____ **Student#:** _____

(1) [5 marks] Consider a **Student** class as follows:

```
public class Student {
      private double costPerCourse;    // amount of $$$ per course

      public double computeTuition(int numberOfCourses) {
            return costPerCourse * numberOfCourses;
      }
      ... // Some methods have been omitted
}
```

Now assume we have a subclass of **Student** called **SeniorStudent**. The method below is defined in the **SeniorStudent** class and it *overrides* the one in **Student**. Complete the method so that **SeniorStudents** pay for courses as follows: costPerCourse for each of the first 2 courses and then 1/2 times costPerCourse for any additional course(s). For example, a senior student will pay 2*costPerCourse to take two courses, but pay 3*costPerCourse to take four courses.

Your method **MUST** make use of the superclass method computeTuition. You MAY **NOT** access the costPerCourse attribute within your method since it is **private** and you MAY **NOT** use a ***get*** method for the costPerCourse value.

```
@Override
public double computeTuition(int numCourses) {

 if(numCourses <= 2){                          // 1 mark check lower 3
  return super.computeTuition(numCourses);     // 1 mark return normal cost
 }

 return super.computeTuition(numCourses)       // 1 mark compute normal part
     + 0.5*super.computeTuition(numCourses-2);// ½ mark for multiply by 0.5
 }                                             // ½ mark numCourses-2

                      // 1 mark using super.computeTuition
}
```

(2) [9 marks ... 1/2 each] For each statement below, circle **T** if the statement is true, otherwise circle **F**:

[T F] Every object that we define will be a direct or indirect subclass of **Object**.

[T F] Subclasses always have direct access to all attributes and methods from their superclasses..

[T F] The **super** keyword can be used to call a specific constructor in the superclass.

[T F] **super.super.x()** will allow you to call a method **x()** two superclasses up in the hierarchy.

[T F] **Abstract classes** are just like concrete classes except that they can have **abstract methods**.

[T F] The keyword **interface** in java defines a class that can have regular methods, but not attributes.

[T F] Both **this()** and **super()** can be called in a single constructor.

[T F] Java has eight primitive data types. String is one of them.

[T F] Every object has a **toString()** method that returns a String.

[T F] Both **abstract classes** and **interfaces** are blueprints that define “what” an inheriting class is.

[T F] Arrays in Java can be extended (made bigger) by calling their **append()** method.

[T F] When you call **new Student()** at least two constructors are actually executed.

[T F] If the MP3 class directly extends the Music class, then **MP3 song = new Music();** is valid.

[T F] A **TextField** can have its text color changed.

[T F] **TextField t1** has an integer in it. The following adds one to that number: ((**int**) t1.getText()) +1;

[T F] **setDisable(true)** will hide a component from the window.

[T F] A **Pane** can contain another **Pane**.

[T F] The same **Pane** code can be used in more than one application window.

(3) Consider the **Smellable** interface and the concrete **Fruit**, **Flower** and **Dog** classes shown below.

```
public interface Smellable {
    public int smell();
}
```

```
public class Fruit implements Smellable{
    public int calories;
    // ... code omitted ... //
}
```

```
public class Flower implements Smellable {
    public String colour;
    // ... code omitted ... //
}
```

```
public class Dog implements Smellable {
    public int weight;
    // ... code omitted ... //
}
```

(a) **[1 mark]** Write **a single line of code** that creates an array that can hold 50 things that have the ability to smell (i.e., Fruit, Flower or Dog objects) and stores its reference in a variable called **smellyThings**.

```
Smellable[] smellyThings = new Smellable[50];  // 1 mark

//   Object[] smellyThings = new Object[50]; // also acceptable
```

(b) **[3 marks]** Assume that the array, from (a), is always full and has no **null** items. Fill in the missing line(s) of code in the two methods below that take this array as input. The top method must return the total amount of calories of all food in the array. The bottom method must return the number of flowers with the specified input colour. In both cases, write the code as efficiently/short as possible to get your marks. The **...** depends on the type of array you created in part (a)

```
public int totalCalories(... smellyThings) {
  int total = 0;
  for (int i=0; i<smellyThings.length; i++) {

     if( smellyThings[i] instanceof Fruit){     // ½ instance of
        total += 1;                             // ½ check if Fruit
     }

  }
  return total;
}
```

```
public int flowersWithThisColour(... smellyThings, String colour) {
  int count = 0;
  for (int i=0; i<smellyThings.length; i++) {

    if( smellyThings[i] instanceof Flower ){
        if( ((Flower)smellyThings[i]).colour.equals(colour) ){
            count += 1;
        }                                       // ½ check if Flower
    }                                           // ½ cast to Flower
  }                                             // ½ check is colour is same
  return count;                                 // ½ use .equals in check
}
```

(c) **[2 marks]** Complete the method below as efficiently as possible so that it prints out all the weights of the **Dogs** in the array. Again, the array from part (a) will be passed as input to the method. Each weight must be printed on a separate line.

```
public void weightsOfDogs(... smellyThings) {

  for (int i=0; i<smellyThings.length; i++) {     // ½ instanceof
    if( smellyThings[i] instanceof Dog ){
      System.out.println( ((Dog)smellyThings[i]).weight );
                                                  // 1 typecast to Dog
                                                  // ½ access weight properly
    }
  }
}
```