Tutorial 1: OOP review

Exercise 1. Define and implement a subtype of `java.util.ArrayList` called `MaxMinIntList` that provides methods to return the smallest (`min()`) and largest (`max()`) elements of the list. You need to code `main` function to test.

Exercise 2:

A. Consider a type `Counter` with the following operations:

```
Public class Counter {

   int count;
  /**
   * Effects: Makes count contain 0
   */
  public Counter()

  /**
  *
  * Effects: Returns the value of count
  */
  public int get()

  /**
  * Effects: Increments the value of count
  */
  public void incr()

}
```

Define and implement this `Counter` class. You need to code `main` function to test. For example: when you loop from 1 to 100, in each iteration you will use `Counter` to count the number of iterations that you have gone through.

B. Consider a potential subtype of `Counter`, `Counter2`, with the following extra operations:

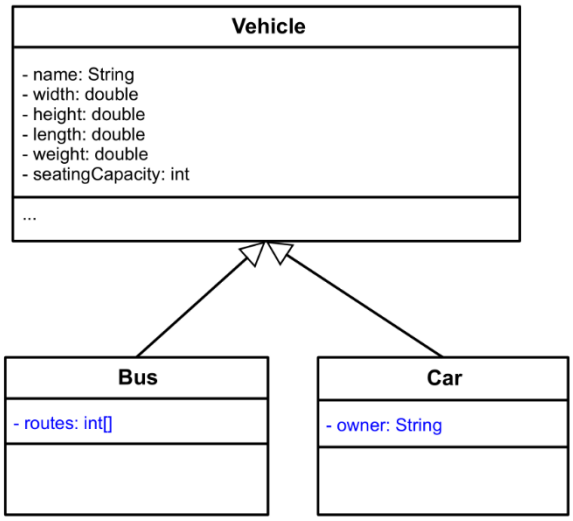
```
/**
 * Effects: Makes count contain 0.
 */
public Counter2()

/**
 *
 * Effects Makes count contain twice its current value.
 */
public void incr()
```

Is `Counter2` a legitimate subtype of `Counter`? Explain by arguing that either the substitution principle is violated (for a non-subtype) or that it holds (for a subtype)

Exercise 3. Inheritance

This exercise uses the `Vehicle` type hierarchy.

Attributes	Formal type	Mutable	Optional	Min	Max	Length
`name`	String	T	F	-	-	100
`width`	Double	T	F	0+	-	-
`height`	Double	T	F	0+	-	-
`length`	Double	T	F	0+	-	-
`weight`	Double	T	F	0+	-	-
`seating Capacity`	Integer	T	F	0+	-	-



1. Update the two classes `Bus` and `Car` so that their weight constraints are as follow:

   1.1. `Bus.weight` is in the range [5000.0, 20000.0] (kgs)

   1.2. `Car.weight` is in the range [1000.0, 2000.0] (kgs)

2. Update the two classes `Bus` and `Car` so that they now have the following constraints on the length dimension:

   2.1. `Bus.length` is in the range [4.0, 10.0] (meters)

   2.2. `Car.length` is in the range [1.5, 3.5] (meters)

3. Update class `Vehicle` to have a new attribute called `registrationNumber`. Based on your practical understanding of this attribute, decide a suitable data type and restrictions for it. *Note:* you must update and/or define the operations that are relevant to the new attribute.

4. Update the two classes `Bus` and `Car` so that they each have different restrictions for the attribute `registrationNumber` from the restrictions defined in the class `Vehicle`. For example, if `Vehicle.registrationNumber` can contain up to 12 alpha-numerical characters then `Bus.registrationNumber` and `Car.registrationNumber` could only contains up to 8 and (respectively) 6 such characters.

5. Update the three classes `Vehicle`, `Bus` and `Car` so that the `toString()` method can be removed from `Bus` and `Car`, and that the inherited `toString()` method from the class `Vehicle` now provides the accurate class label for not only `Vehicle` but also for `Bus` and `Car`. *Hint:* In Java, you can use the following statement in a method to get the actual

(run-time) type of the object that carries that method: `this.getClass().getSimpleName()`.

You need to code `main` function to test.

Exercise 4: Sub-type with **additional attributes**

1. Design and implement a new sub-type of `Vehicle` called `IronSuit`, which gives superhuman powers, such as flying, to the one wearing it. Class `IronSuit` must have at least **one additional attribute, along with necessary operations**. One essential operation, for instance, is `fly()`, which should carry the person wearing the suit from point A to point B. Operation `fly()` should simply print a message stating the two points and the distance.
2. Update the operation `IronSuit.fly()` so that it can simulate the flying progress from point A to point B with a real-time progress bar. The longer the distance is, the longer the progress bar becomes. A finished flight may look like so:

   ```
   Hanoi . . . . . . . . . . . . . . . Da Nang
   ```

   The method needs to slowly output one dot at a time so that the user can have a sense of moving from point A to point B.

   (*) Hint: In Java, you can use the following code to cause a program to pause for a given number of milli-seconds:

   ```
   int millies = 300; // 0.3 second
   try {
       Thread.sleep(millis); // pause
       // wake up: do something
   } catch (InterruptedException e) {
       // Ignore Exception handling
   }
   ```

Submission

Submit a **zip** file containing all Java programs to this tutorial's submission box in the course website on FIT Portal.