Tutorial 2: Type hierarchy (2)

Exercise 1: The `Counter` class mentioned in the previous tutorial has the following operations:

```
/**
 * @effects Makes this contain 0
 */
public Counter()

/**
*
* @effects Returns the value of this
*/
public int get()

/**
* @modifies this
* @effects Increments the value of this
*/
public void incr()
```

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

```
/**
 * @effects: Makes this contain 0.
 */
public Counter2()

/**
 * @modifies this
 * @effects Makes this 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). Discuss how each operation of `Counter2` either upholds or violates the substitution principle.

Exercise 2: Now consider another potential subtype of `Counter`, `Counter3`, with the following extra operations:

```
/**
 * @effects Makes this contain n
 */
public Counter3(int n)

/**
 * @modifies this
 * @effects If n > 0 adds n to this
 */
public void incr(int n)
```

Is `Counter3` 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). Discuss how each operation of `Counter3` either upholds or violates the substitution principle.

Exercise 3: Consider a type `IntBag`, with operations to `insert` and `remove` elements, as well as all the observers of `IntSet`. Bags are like sets except that elements can occur multiple times in a bag. Is `IntBag` a legitimate subtype of `IntSet`? Explain by arguing that either the substitution principle is violated (for a non-subtype) or that it holds (for a subtype).

Exercise 4: Inheritance (continue from previous tutorial)

1. Define in the class `Vehicle` a new operation named `travel()`, which travels from a given point A to another point B. This operation can simply print a message detailing the travelling (e.g. the type of vehicle, the two points, and the number of passengers). However, it must use a specialised symbol for the type of vehicle object that this operation is being invoked on. For instance, if you invoke it on a `Bus` object then the message must use a specialised symbol (for example: ♂♀↑↓→←∟▲▼ with *chcp 936* to change active code page in cmd) for `Bus`. This feature requires you to also update the two sub-types `Bus` and `Car` of `Vehicle`.
2. Define a two new subtypes of `Vehicle`: `Motorbike` and `Boat`. Use your practical understandings of these two types to add at least one attribute to each with suitable domain constraints.

Exercise 5: Method overriding

Improve the three classes `Vehicle`, `Bus`, and `Car` so that each class has a method named `validateRegistrationNumber` which validates the attribute `registrationNumber`. Override this method in `Bus` and `Car`.

Exercise 6: 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
   }
   ```