# Exercise Sheet 7

1. Look at the following program:

```
class Motor {
  boolean isOK(){...}
  boolean isRunning(){...}
  void start() {...}
  ...
}
```

```
class Wheel {
  void deflate(){...}
  boolean isOK(){...}
  ...
}
```

```
class MotorTrouble extends Exception {
  public Motor motor;
  public MotorTrouble( Motor m )
    {motor = m;}
}
```

```
public class Car {
  public Motor engine;
  public Wheel[] wheels;

  invariant engine.isRunning() ⇒
    (∀ int i; i ≥ 0 && i < wheels.length;
      wheels[i] ≠ null && wheels[i].isOK())

  public Motor getMotor() {return engine;}

  public Wheel[] getWheels() {return wheels;}

  public Car( Motor m, Wheel[] w ){
    engine = m;
    wheels = w;
  }

  public void start() throws MotorTrouble {
    if( engine == null || !engine.isOK() ) {
      throw new MotorTrouble( engine );
    } else {
      engine.start();
    }
  }
```

   a. What aliasing problems can arise in the example program?

   b. Write example code for every problem where the invariant can be violated.

   c. Change the code of `Car` in a way such that there are no more aliasing problems.

2. In lecture 5 we have seen what is needed to preserve the consistency of invariants in Java under assumption that the invariants of object **X** may only refer to private attributes of **X**. Now we want to allow invariants that can refer to both private and default access attributes.

   a. Strengthen the obligations in a way so that consistency is preserved again.

   b. For each strengthened obligations provide source code which demonstrates their necessity.

3. **This question was asked in a previous exam!**

   In this task, we assume that the invariants can be declared with any of the Java access modifiers. We only consider object invariants containing field accesses (but, no method calls). However, independent of its access modifier, an invariant has to hold in the pre- and post-state of each method execution of the program.

   Consider the following class skeleton, where A and B are access modifiers:

```
class T {
  A invariant f > 0; // invariant with modifier A
  B int f ; // field declaration with modifier B
  ...
}
```

   a. For the sake of maintenance, the following property is desired: If a class C has access to an invariant *I*, then C must also have access to the fields mentioned in *I*. To accomplish this goal, what conditions should the modifiers A and B satisfy?

   b. To enable sound modular verification, the following property is desired: If a class C does not have access to an invariant *I* and *I* changes, then one should not have to re-verify C.
   To achieve this goal, what conditions should the modifiers A and B satisfy? Give a code example for which the modular verification would not be possible without these conditions.