Concepts of Object-Oriented Programming

Exercise 9

More on Polymorphism, aliasing and encapsulation of object structures
26th November

In-class Assessment: One or more questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

1) Consider the following Scala classes:

```
class A
class B extends A
class P1[+T]
class P2[T <: A]
```

- What are the possible instantiations of `P1` and `P2`?
- What is the difference between `P1[A]` and `P2[A]` from the perspective of a client? Provide an example to show which class is more restrictive.

2) You have the following Scala class:

```
class Matrix[T] (val m : Array[Array[T]]) {
    def get(i : int, j : int) : T  = {return m(i)(j);}
    def set(i : int, j : int, elem : T) : Unit = {
          m(i)(j)=elem;
    }
}
```

- What happens if we try to use covariant type arguments on `T` (that is, `Matrix[+T]`)? Why?
- And what about contravariance (`Matrix[-T]`)?
- We can add two generics (one with covariant and the other one with contravariant annotation) to class `Matrix` (one for positive and the other one for negative positions).

  ```
  class Matrix[T, +T1 >: T, -T2 <: T] (val m : Array[Array[T]]) {
      def get(i : int, j : int) : T1  = {return m(i)(j);}
      def set(i : int, j : int, elem : T2) : Unit = {
        m(i)(j)=elem;
      }
  }
  ```

  In practice, is this more flexible than the initial solution?

3) Data structures often intentionally share aliases. For instance, consider the following `ArrayList` class:

```
class ArrayList<T> {
  private T[] elements=…;
  private int LastEl=0;
  public T get(int i) {return elements[i];}
```

```
  public void add(T el) {elements[intLastEl++]=el;}
}
```

Imagine that this class is extended as follows

```
class Coordinates {
  int x, y;
  public Coordinates(int xx, int yy) {x=xx; y=yy;}
}

class CList extends ArrayList<Coordinates> {
  // invariant \forall el \in elements : el.x>el.y
  public void add(Corrdinates el) {
    if(el.x>el.y) super.add(el);
  }
}
```

Write a program that breaks the invariant of `CList`. How can we fix this problem? Is it possible to fix it without allocating new objects (either directly or indirectly), that is, without consuming additional memory? What new problems might arise from your changes? Discuss the benefits and the drawbacks of using alias sharing in data structures.

4) In Question 7 of last week’s sheet we were able to break the invariant because of alias leaking.

```
public class Hour {
    public int h=0;
}

public class Time {
    private Hour hour;
    //invariant hour.h>=0 && hour.h<24

    public Time(Hour hour) {this.hour=hour;}

    public void setHour(int h) {
        if(h>=0 && h<24) this.hour.h=h;
    }

    public Hour getHour() {return hour;}
}
```

Modify class `Hour` by using read-only interfaces. Find an example that still breaks the invariant, and explain why such a solution is not satisfactory in terms of safety.

5) Consider the following C++ class:

```
class Person
{
      int money;
      Person *spouse;

public:
      void f () const;
      Person (int m, Person *s)
```

```
          { if (!s) spouse = 0;
            else { spouse = s; s->spouse = this; }
            money = m;
          }
    };
```

Method `f` promises not to make any changes to its target object. Show that this claim can still be violated by a definition of `f`, without using casting and without introducing any local variables.

6) The intuition behind a `pure` method is that its execution effects are not observable by the client. This essentially means that the result of any other method call or field read inside client code wouldn't be affected by a pure method execution. One way to formalize this property is to require that the execution of a pure method doesn't change the program heap.

- Provide proof obligations that guarantee the purity of a method, according to this requirement. Can you define an analogous notion for constructors?
- Class `Set` represents a set of integers. Method `Set allLessThan(int bound)` (in class `Set`) returns a freshly-allocated instance of class `Set` that contains all elements of the original set that are smaller than `bound`..
    - Even though the method `allLessThan` does not change the behavior of other methods, it is not pure, according to our definition. Why?
    - How can the provided definition of purity be relaxed to allow declaration of the method `allLessThan` as pure, without violating the intuition above?
    - Provide proof obligations that guarantee purity of a method according to your relaxed definition.
    - Can you define an analogous notion for constructors?