Exercise 9

Aliasing, encapsulation of object structures, read only types

1) The invariant can be broken by exploiting the fact that `CList` captures and stores `Coordinates` objects.

```
CList list=new CList();
Coordinates c=new Coordinates(2, 1);
list.add(c);
c.x=0;
```

We can fix `CList` quite easily: we need to clone the `Coordinates` element before storing it.

```
  public void add(Coordinates el) {
    if(el.x>el.y) super.add((Coordinates) el.clone());
  }
```

The limit of such an approach is that we create a copy of all the elements stored in the list. On the other hand, it is not possible to make sure the invariant is preserved without creating objects that are only in the current `CList` object. The main benefit of using alias sharing in data structures is to minimize the consumption of memory. In addition, we may want to share aliases on data structures, for instance, in order to further update the content of an element in a list. The main drawback is that alias sharing does not allow us to reason locally on the values stored in the data structure, since the object may have been stored by the program that added elements, and so it may modify the content of the elements after they were stored.

2) We have to introduce a `ReadonlyHour` interface, let `Hour` extend it, and impose on class `Time` to return a `ReadonlyHour`.

```
public interface ReadonlyHour {
  public int getHour();
}

public class Hour implements ReadonlyHour {
  public int h=0;
  public int getHour() {return h;}
}

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

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

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

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

This solution is unsatisfactory, because we need to be able to assign to `h`, which makes it possible for outsiders to also assign to `h`. For example: (a) the constructor of `Time` takes an hour object as a parameter. This remains as an `Hour` object on the side of the client, which can change `h`. (b) The client can downcast a `ReadOnlyHour` reference to `Hour`.

3)

We can violate the claim by changing the target object `this` passing through the field `spouse`, for instance with `spouse->spouse->money=0;`
In order to do that, we have to suppose that the current object was initialized passing a value different from `null` as second argument of the constructor.

4)

- A method is `pure` if and only if:
  (1) It does not contain field updates
  (2) It does not invoke non-`pure` methods
  (3) It does not create objects

- Method `getMaxMin` is not `pure` because it allocates new objects. This is not allowed by the definition of pure method since it modifies the heap.
  In order to allow the `getMaxMin` method to be `pure`, we should relax such rules by allowing that `pure` methods create objects. So the proof obligation of `pure` methods will be relaxed in the following way: a method is `pure` if and only if:
  (1) It does not contain field updates
  (2) It does not invoke non-`pure` methods

  Note that we have to change the previous definition of `pure` methods (that is, a method is `pure` if it does not modify the heap) to the following one: a method is `pure` if and only if it does not modify the part of the heap it receives at the beginning of its execution.

- Method `getLessThan` does not change the behavior of other methods but it is not `pure` following neither the initial definition nor the relaxed one. In fact, it calls non-`pure` methods in order to add all the elements that are less than the given bound to the set returned by `getLessThan`. In order to accept method `getLessThan` as `pure`, we may relax the proof obligation of `pure` methods allowing calls of non-`pure` methods if they modify only newly allocated objects. This leads to the following definition: a method is `pure` if and only if:
  (1) It does not contain field updates
  (2) It invokes non-`pure` methods that modify only newly allocated objects

5) The general rules are:

- `readwrite T <: readonly T`
- when we access a field/method, we take the upper bound of the `readonly`/`readwrite` modifiers.

Program 1: it does not compile since `obj2` is `readonly`, and we try to assign to a `readwrite` variable the field of one of the objects contained in it.

Program 2: it does not compile since field `y` in `B` is readonly.
Program 3: it compiles!
Program 4: it does not compile since `obj` is `readonly` and it is passed to the constructor of `B` as first argument.
Program 5: it compiles!
Program 6: it compiles!

6) The general typing rules are `any >: peer` and `any >: rep` since `any` is more restrictive than `rep` and `peer`. Following these rules, we obtain that

- `peer Object foo(any String el)` overrides `any Object foo(peer String el)`
- `rep Object foo(any String el)` overrides `rep Object foo(peer String el)`, that overrides `any Object foo(peer String el)`
- `peer Object foo(any String el)` overrides `peer Object foo(rep String el)`

7)

- `readonly int[]` is more restrictive than `readwrite int[]`, so we could have `readonly int[] <: readwrite int[]`.

- Considering `y[1].f` as an access which goes first via `y`, and then `y[1]`, we would obtain that:

  i) If the first modifier is `readonly`, all the accesses to elements of the array will be treated as `readonly`, since the `readonly` modifier for the array will be considered first. Therefore, the only interesting combinations are:
  (1) `readonly readonly`
  (2) `readwrite readonly`
  (3) `readwrite readwrite`

  Note: The same approach is adopted when we have a `readonly` object variable and we access a `readwrite` field through it: the result would be `readonly`, since any access via a `readonly` reference is `readonly`.

  ii) (1) is more restricted than (2), and (2) is more restricted than (3). So the reasonable subtyping relations are (1) :> (2) :> (3)

- Considering `y[1].f` as a direct access, we would obtain that:

  i) All the four different combinations have different semantics. With respect to the previous example, we would have that `readonly readonly` will allow only read accesses both on the array and on the elements stored in it, while with `readonly readwrite` we have that we cannot assign elements in the array but we can write fields accessed via the array elements.

  ii) The subtyping relations already pointed out still work. In addition we could have
  (1) `readonly readonly :> readonly readwrite`

(2) `readonly readwrite :> readwrite readwrite`

- The second solution is more expressive than the first one, since it allows the developer to have more fine-grained control on the read and write accesses on arrays and on their elements. Thus, the second choice seems to be the best. However, it should be carefully considered whether such an approach (that would be different compared to the one adopted for objects and field accesses) may confuse the developers, and eventually create safety problems.