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.

A possible solution would be to have readonly fields in class `Coordinates`. This would ensure that the invariant cannot be broken, but it requires the allocation of new objects each time we want to modify the fields. For instance, the following code:

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

should be re-written to

```
Coordinates c=new Coordinates(2, 1);
c=new Coordinates(0,1);
```

which allocates a new object even though this is not necessary (since the object pointed by `c` is not shared, and so changing its fields cannot break the invariants of other objects).

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

We cannot reasonably provide an analogous notion for constructors, since a constructor call is guaranteed to modify the heap.

-
  - Method `allLessThan` is not `pure` because it allocates new objects. Furthermore, it must either make field updates or call non-`pure` methods in order to add all the elements that are less than the given bound to the set returned by `allLessThan`. Nonetheless, it seems likely it does not change the behavior of other methods, and we would like to consider it as pure.
  - We need to allow "pure" methods to allocate new objects, and to perform modifications on those newly-allocated objects. In this case, we say that the method is "weakly pure"
  - We shall use the readonly type system - a method is "weakly pure", if:
    (1) All its arguments are readonly
    (2) The receiver is treated as readonly – we can annotate as in C++:

```
Set allLessThan(int bound) readonly{
  Set result = new Set();
  for (readonly Node n = head;n!=null;n=n.next)
       if (n.val<bound)
           result.append(n.val);
  return result;
}
```

We assume the set is implemented as a linked list.
We can access this.head as **readonly** as **this** is **readonly**.
The disadvantage here is that we cannot store read/write references to our arguments (e.g. we could not have that the new Set includes a read/write pointer to the old one, or it has read/write access to members if it did not clone them).

- For constructors, we can make the same requirements except that the **this** pointer can be read/write (but again could not store read/write pointers to arguments).

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`, so `obj2.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.
Program 2: it does not compile since field `y` in `B` is `readonly`, so `obj2.y.n1` is `readonly`, and we try to assign it to a `readwrite` variable.
Program 3: it compiles! `obj2` is `readwrite`, `x` is `readwrite` (so `obj2.x` is `readwrite`), `n1` is readwrite (so `obj2.x.n1` is `readwrite`), and we assign `obj2.x.n1` to a `readwrite` variable.
Program 4: it does not compile since `obj` is `readonly` and it is passed to the constructor of `B` as first argument, while the constructor expects a `readwrite` variable.
Program 5: it compiles! We can always assign something to a `readonly` variable.
Program 6: it compiles! We can always assign something to a `readonly` variable.

In addition: for all the programs expect 4, the first argument passed to the constructor of `B` is `readwrite`, and the second argument can be `readwrite` or `readonly` since a `readonly` argument is expected.