Exercise 9

More on Polymorphism, aliasing and encapsulation of object structures

1) Class `P1` can be instantiated with any type, while `P2` has to be instantiated with subtypes of `A`.

```
T1<:T2 <=> P1[T1] <: P1[T2], while P2 is invariant (T1=T2 <=> P2[T1] <:
P2[T2]).
Thus T1<:A <=> P1[T1] <: P1[A], while A=T1 <=> P2[T1] <: P2[A]
Then P1[A] is more general than P2[A].

val x : P1[A]=new P1[B] //correct

val y : P2[A]=new P2[B]
//wrong: found P2[B], required P2[A]
```

2) If we adopt a covariance annotation on `T` (that is, `Matrix[+T]`), the compiler rejects the program with the following error:

```
error: covariant type T occurs in contravariant position in type T of
value elem
          def set(i : int, j : int, elem : T) : Unit = {
                                               ^
```

This happens because covariance annotation is allowed only for types that do not occur on type parameters.
On the other hand, if we adopt a contravariance annotation (that is, `Matrix[-T]`), the compiler still rejects the program with the following error:

```
error: contravariant type T occurs in covariant position in type
(int,int)T of method get
          def get(i : int, j : int) : T  = {return m(i)(j);}
              ^
```

This happens because covariance annotation is allowed only for types that do not occur on result type.

The following assignments are rejected:

```
val z : Matrix[String, String, String]=
         new Matrix[Object, Object, Object](Array(Array(new Object())))
val y : Matrix[Object, Object, Object]=
         new Matrix[String, String, String] (Array(Array("foo")))
val x : Matrix[String, String, Object]=
         new Matrix[String, String, String] (Array(Array("foo")))
```

while the following ones are accepted

```
val y1 : Matrix[Object, Object, String]=
         new Matrix[Object, Object, Object] (Array(Array(new Object())))
val z1 : Matrix[String, Object, String]=
         new Matrix[String, String, String] (Array(Array("foo")))
```

In this way we can hide some information from the client (that is, `y1` can add only strings to a matrix of objects, while `z1` expects to receive Object values from method get).

3) 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.

4) 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`.

5)

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.

6)

- 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"
- A method is "weakly pure", if:
  - (1) It only contains field updates on newly allocated objects
  - (2) It only invokes non-weakly-pure methods that are known to modify only objects which have been allocated since the beginning of the weakly-pure method execution. Note that to check this concept statically (and modularly), one needs some kind of "modifies" clause or similar on methods, to be able to deduce which objects a non-weakly-pure method might modify.

- For constructors, we can make the same requirements. Note that we can also consider the object under construction as a newly-allocated object (i.e., it is ok to make modifications to the receiver object in the constructor, as well as any objects allocated during the constructor execution).