Exercise 8

Information hiding, encapsulation, aliasing

1) The subtyping relations are as follows:

   C <: B <: A

   Using structural subtyping we require that the methods and fields of subclasses are more accessible than those of superclasses. When dealing with access modifiers, this means that methods with more permissive modifiers may override methods with less permissive modifiers.

2) The 1st program does not compile because method `f` of class `Y` tries to access a field of the superclass with default access modifier (that is, it can be accessed only by classes in the same package) from an external package.

   The 2nd program does not compile because method `f` of class `Y` tries to access a `protected` field of an object instance of the superclass, but from a different package (`A2`, while the superclass belongs to `A1`). Note that Java does not allow subclasses to access `protected` fields of other objects instance of the superclass if they belongs to a different package.

   The 3rd program does not compile because method `f` of class `Y` tries to access a private field of the superclass.

   Finally, the 4th program compiles. In fact, method `f` of class `Y` is allowed to access `this.x` since it is a `protected` field of class `X`.

3) Assume that obj is of dynamic type B. The output of foo is:

   ```
   Class B:0
   ```

   This happens because in `obj.x=10` we have the static binding of the class whose field we are going to assign, that is, the compiler assigns field `x` of class `A` since the static type of `obj` is `A`. On the other hand, the method call `obj.print()` is dynamically bound to the dynamic type of `obj`, that is, it calls method `print()` of class `B`. Thus method `print()` of class `B` reads the value of field `x` of class `B`, that contains the initial value 0.

   Under the same assumptions, the output of bar is:

   ```
   Class B:10
   ```

   This happens because the call to `setX is bound dynamically`, thus we assign value 10 to the field `x` of class `B`. In addition, we have the same binding when calling `print()`, thus we invoke it when field `x` of class `B` is equal to 10 and we obtain that output.

In general, it is better to adopt setter and getter methods from the point of view of information hiding in order not to rely on the internal representation of the class. This example and the unexpected behavior obtained when executing the first program demonstrate this fact: if we rely on accesses to fields we may access fields that are different from the ones accessed using method calls, since in the first case we have static binding, while in the second case we have dynamic binding.

4)

The external interface is composed only of the method `public set(int)` since this is the only public element of class `Hour.`

The invariant can be broken easily by extending class `Hour`, and accessing the field `h` directly`.` For instance:

```
public WrongHour extends Hour {
    public WrongHour() {super.h=-1;}
}
```

5)

For the fields of class `BankAccount`, the most permissive access modifiers are:

- `importantCustomer`: default modifier. In this way, it would be accessible by other classes in the same package but not by subclasses. Otherwise, we may have a class that extends `BankAccount` and sets to true `importantCustomer` without being a `RichCustomer`.
- `maxDebit`: `public`, since it is final and it cannot be modified by other classes.
- `amount`: default, since we need to access it from other classes of this package, but we must prevent external attackers from modifing it.

Methods `withdraw` and `deposit` can be declared `public`, since they preserve the invariants.

If class `BankAccount` had been declared as `sealed`, we could choose `protected` as the access modifier of the `amount` and `importantCustomer` fields, since external classes would not be allowed to extend it and so would not be able to gain access to these fields.
More generally, if a class is `sealed`, the default and protected levels are equivalent, since it is not possible to extend the current class outside the current package.

6)

Drawback: we cannot check the consistency of an object considering only the current instance.

```
public Foo {
    int a=0; //invariant a>=0
    public Foo broken() {
        Foo result=new Foo();
        result.a=-1;
        return result;
```

```
    }
}
```

Another disadvantage is that with implementation inheritance we can break subclasses of the current class:

```
private C {
  protected int v = 0;
  public void f(C other){
       other.v = -1;
  }
}

public D extends C{
    //invariant v>=0;
}
```

Advantage: we can access the internal structure of other objects of the same class. Note that we already know their internal structure, since it is exactly the same as that of the current object.

```
public class List {
    private Object el;
    private List next;

    public removeSecondElement() {this.next=this.next.next;}
}
```

If we apply the same idea of private fields, we would expect that subclasses are allowed to access fields of superclasses of objects that are instance of the superclass . Instead this is not allowed, since we can access `protected` fields only of the current instance and not of other objects (except for the cases where a private field would be accessible – e.g. class C can access all protected fields of other objects that are of type C or a subtype of C).

7)

We can easily break the invariants through alias leaking. For instance, the following code breaks the invariant of class `Time`:

```
Time t=new Time();
Hour h=t.getHour();
h.h=-1;
```

We can fix in two ways. We have to avoid the alias leaking. We can reach this goal returning an integer value instead of an object, or a copy of the `Hour` object stored in the current `Time` object.

```
public int getHour() {return hour.h;}
public Hour getHour() {return (Hour) hour.clone();}
```

In general, it is simpler for reasoning, if possible, to return only primitive values, or to avoid

exposing aliases of the local state of the object, by instead returning copies of the stored objects. In this way, we can avoid alias leaking, thus no external code can modify the values contained in the current object.
Another option is to hide the h field of Hour and define a superclass `IMHour` of `Hour` with no mutator methods – and then return it. The client could still downcast from `IMHour` to `Hour` and break the invariant but aside from that the invariant is protected.

8)
The main advantage is that it is not possible to modify the strings. Thus, an external class cannot modify the content of a string contained in the current object even if we leak its reference. This allows all clients of the string to rely on it keeping its value always (a strong history invariant) and also, as a beneficial side effect, allows the compiler to make more space and time optimizations as it has more information on the string (e.g. constant propagation and sharing).
In this particular example, imagine that the `toString` method of the elements of the list returns a string contained in an internal field. When we return the string representing the list, the receiver of the result of method `prettyprint` will not be able to modify the string contained in those fields. For instance:

```
String s=prettyprint(list)
Strins s1=s.replace(“a”, “b”)
```

Affects neither the single string representing an element of the list, nor the string returned by `prettyprint`.

The main drawback is that we create a new string (a new object allocated on the heap) each time we concatenate two strings. Supposing that only `result+list.get(i).toString()+”,”` creates a new string (but it may be the case that it creates first `result+list.get(i).toString()` and then concatenates it with `“,”`), at the end of the method we have created 2*`list.size()`+1 strings (one with the initial value `“”` of result, and inside the loop the one returned by `list.get(i).toString()` and the one resulting from the concatenation). This can degrade the performance of the program, both in terms of space and time.