Exercise 9

Aliasing, encapsulation of object structures, read only types

1) Data structures 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 class? Is it possible to fix it without allocating directly or indirectly new objects, that is, without consuming additional memory? Discuss the benefits and the drawbacks of using alias sharing in data structures.

2) In Question 7 of the previous week 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 readonly interfaces. Find an example that still breaks the

invariant, and explain why such a solution is not satisfactory in terms of safety.

3) 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.

4) 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.
- Class `Pair` represents a pair of integers. Class `Set` represents a set of integers. Method `Pair getMaxMin()` returns a freshly-allocated instance of class `Pair` that contains the minimal and maximal values of the set.
  - Even though the method `getMaxMin` 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 `getMaxMin` as pure, without violating the intuition above?
  - Provide proof obligations that guarantee purity of a method according to your relaxed definition.
- Class `Set` has also method `Set getLessThan(int bound)` which returns all elements of the set that are smaller than `bound`. Answer the previous questions for `getLessThan`.

5) Consider the following classes:

```
class A {
  readwrite StringBuffer n1=…;
  readonly StringBuffer n2=…;
}

class B {
  readwrite A x;
  readonly A y;
```

```
    public B(readwrite A x, readonly A y) {
      this.x=x;
      this.y=y;
    }
  }
```

Check which programs are correct and explain why.

Program 1	**Program 2**	**Program 3**

`readwrite A obj=new A();`
`readonly B obj2=`
`  new B(obj, obj);`
`readwrite StringBuffer v=`
`  obj2.y.n1;`
	`readwrite A obj=new A();`
	`readwrite B obj2=`
	`  new B(obj, obj);`
	`readwrite StringBuffer v=`
	`  obj2.y.n1;`
		`readwrite A obj=new A();`
		`readwrite B obj2=`
		`  new B(obj, obj);`
		`readwrite StringBuffer v=`
		`  obj2.x.n1;`

Program 4	**Program 5**	**Program 6**

`readonly A obj=new A();`
`readonly A obj2=new A();`
`readwrite B obj3=`
`  new B(obj, obj2);`
`readwrite StringBuffer v=`
`  obj3.y.n1;`
	`readwrite A obj=new A();`
	`readonly A obj2=new A();`
	`readwrite B obj3=`
	`  new B(obj, obj2);`
	`readonly StringBuffer v=`
	`  obj3.y.n1;`
		`readwrite A obj=new A();`
		`readonly A obj2=new A();`
		`readwrite B obj3=`
		`  new B(obj, obj2);`
		`readonly StringBuffer v=`
		`  obj3.y.n2;`

6) Consider the following method signatures:

```
peer Object foo(any String el);
peer Object foo(rep String el);
rep Object foo(any String el);
any Object foo(peer String el);
rep Object foo(peer String el);
```

Find all the valid pairs of signatures such that one overrides the other.

7) Consider how we might extend `readonly` types to handle arrays. For an array of primitive types, one might want to declare that the array cannot be written to, using a declaration such as:

```
readonly int[] x;
```

which, similarly to the rules for accessing fields of objects, would forbid an update such as

```
x[2] = 2; // error – x is declared with a readonly type
```

Should there be a subtyping relationship (in either direction) between types `readwrite int[]` and `readonly int[]`?

For arrays of reference types, there are *two* reasonable questions to consider for `readonly` typing. Firstly, just as for an array of primitive types, whether or not the array reference can be used for modifications. Secondly, whether the array *elements* can be used for modifications.

```
y[1] = y[2];     // is this allowed?
y[1].f = y[2].f; // is this allowed?
```

In order to express all possibilities, consider having *two* `readonly/readwrite` modifiers for an array type – the first to denote access via the array, and the second to denote access via its elements, e.g., `readonly readonly T[] y;` is the most restrictive.

Consider what the semantics of `readonly readwrite T[] y;` could be. There are two reasonable choices here, depending on whether we regard accesses of the form `y[1].f` as accesses which go first via `y`, and then `y[1]`, or consider them as accesses directly from `y[1]` (in a sense "skipping" `y`). For each of these two possible semantics, consider the following:

Do all four combinations of modifiers express something different from one another?
Can you think of an example in which each might be potentially useful?
What subtyping relationships (if any) would be reasonable between the four possible variants of a `T[]` type?

In the light of these questions, which of the two semantics seems the best choice?