Exercise 10

1. 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
   a. `peer Object foo(any String el)` overrides `any Object foo(peer String el)`
   b. `rep Object foo(any String el)` overrides `rep Object foo(peer String el)`, that overrides `any Object foo(peer String el)`
   c. `peer Object foo(any String el)` overrides `peer Object foo(rep String el)`

2. The general rules are:
   a. `readwrite T <: readonly T`
   b. when we access a field/method, we take the upper bound of the `readonly`/`readwrite` modifiers.

   Program 1: 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: does not compile since field `y` in `B` is readonly.
   Program 3: compiles!
   Program 4: does not compile since `obj` is `readonly` and it is passed to the constructor of `B` as first argument.
   Program 5: compiles!
   Program 6: compiles!

3.
   a.
      i. `setElems` is capturing an external array
      ii. `getElems` is leaking the internal representation
   b.
      i.
```
void capturing(ArrayList al) {
  int[] myarr = new int[10];
  for(int i=0; i < myarr.length; ++i)
    myarr[i] = i;
  al.setElems( myarr );
  myarr[0] = 42;
}
```
      In this example, the last line changes the internal representation of the `ArrayList` because of the capturing problem.

      ii.
```
void leaking(ArrayList al) {
  int[] myarr = al.getElems();
  myarr[0] = 42;
}
```
      In this example, the last line changes the internal representation of the `ArrayList` because of the leaking problem.

c+d.

```
class ArrayList{
      protected rep int[] array;
      protected int next;

      public void add(int i) {
        if( next==array.length ){
          resize( );
        }
        array[ next ] = i;
        next++;
      }

      public peer int[] getElems() {
        peer int[] result =
          new peer int[array.length];
        System.arraycopy
        (array, 0, result, 0, array.length );
        return result;
      }

      public void setElems(any int[] ia){
        array = new rep int[ia.length];
        System.arraycopy
         (ia, 0, array, 0, ia.length );
        next = ia.length;
      }

      protected void resize() {
        if( next==array.length ) {
          int[] oa = array;
          array = new rep int[2*oa.length];
          System.arraycopy
           (oa, 0, array, 0, oa.length );
        }
      }

      Public peer String toString() {
        if( array.length == 0 ) return "[]";
        StringBuffer buf = new peer
          StringBuffer("[ " + array[0]);

        for( int i=1; i < next; ++i ) {
          buf.append(", " + array[i]);
        }
        buf.append(" ]");
        return buf.toString();
      }
}
```

4.

```
class Producer {
  rep int[] buf;
  int n;
  peer Consumer con;
  Producer() {
    buf = new rep int[10];
  }
  void produce(int x) {
    buf[n] = x;
    n = (n+1) % buf.length;
  }
}
```

```
class Consumer {
  any int[] buf;
  int n;
  peer Producer pro;
  Consumer(peer Producer p)
{
    buf = p.buf;
    pro = p;
    p.con = this;
  }
  int consume() {
    n = (n+1) % buf.length;
    return buf[n];
  }
}
```

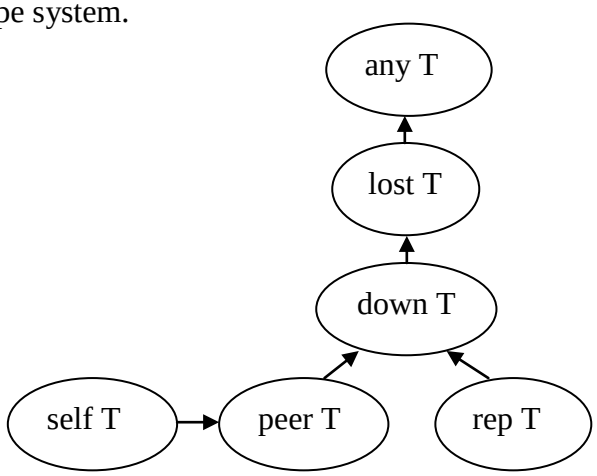
```
class Context {
  rep Producer p;
  rep Consumer c;

  Context(){
    p = new rep Producer();
    c = new rep Consumer(p);
  }

  public void run() {
    for(int i=-5; i <=5; ++i){
      p.produce(i);
      if(i%2 == 0)
        c.consume();
    }
  }
}
```

5.

a. Redraw the subtype relation diagram below to include the newly introduced type of the universe type system.



b.

►	peer	rep	lost	any	down
self	**peer**	**rep**	**lost**	**any**	**down**
peer	**peer**	**down**	**lost**	**any**	**down**
rep	**rep**	**down**	**lost**	**any**	**down**
lost	**lost**	**lost**	**lost**	**any**	**lost**
any	**lost**	**lost**	**lost**	**any**	**lost**
down	**down**	**down**	**lost**	**any**	**down**

c. The field write `exp.f = v;` is correctly typed if:

- `exp` is correctly typed
- `τ(v) <: τ(exp) ► τ(f)`
- `τ(exp) ► τ(f)not equal to` **`lost`** `and` **`down`**

6.

a. With the current system we cannot annotate this example to ensure encapsulation.

b. The problem is typing the result of clone – we want the caller of the method to be able to modify the result (hence view the returned reference as either peer or rep), but we cannot know where the caller is in the hierarchy and hence cannot type the result suitably, in general (the caller could potentially be a peer, owner, etc..).

   This seems however to be an implementation which would ideally be supported – intuitively there is no violation of encapsulation because the newly created list in the clone method is not aliased at any point during execution – its sole purpose is to be returned to the caller and manipulated from within their context. Intuitively, we would like to support the possibility of *postponing* the decision of what the owner of the cloned list will be, until after it has been returned to the calling context.

   If we are to allow objects whose owners are not immediately decided, these should naturally not be considered the rep or peer of objects which already have an owner. Furthermore, when we reach the point in execution when we wish to decide on the owner for the cloned list, we need to be careful – any aliases to the list which consider it to have an "unassigned" owner will be dangerous as soon as we introduce a reference which specifies its ownership. To avoid such aliasing issues before an owner is determined, we can try to enforce that only a single (*unique*) reference exists to an unowned object at any one time.

c. We would like to describe an object that has exactly one reference pointing to it (and no owner yet decided) – we shall call it a **unique** reference and add it as a new modifier in our type system. This modifier will not be used on field types. We need to enforce additional restrictions which guarantee that uniqueness is preserved (and hence, that we won't need to worry about aliasing when we choose an owner later).

   The obvious place to prevent aliasing is in assignments. A **unique** reference can only be assigned to from another **unique** reference, and the latter pointer must then be invalidated (to avoid violating the uniqueness guarantee) – it must be re-assigned with some other value before it can be used again. In this way we can be certain that only one (usable) reference exists at all times – we can enforce this requirement by doing the same kind of dataflow analysis as Java does for definite assignment of local variables (guaranteeing that such variables are initialized before being read).

   Argument passing can be treated similarly to assignment – the same rules apply. We must also be careful that a unique reference is not passed twice to the same call.

   In order to support the creation of objects without owners, we can add an extra **unique** annotation on a constructor declaration, whose meaning is to guarantee that the created object does not get aliased during the execution of the constructor. Within such a constructor body, we will treat the receiver as a **self** reference, as usual, but with the extra restrictions that it cannot be aliased, passed as an argument or have methods called on it (these restrictions can be relaxed, but it complicates the solution). A "unique" constructor, such as

```
public unique C(){…}
```
can be called by a new expression of the form `new unique C()` and returns a **unique** reference.

A **unique** reference can be cast to a rep, peer or any reference, and the cast invalidates the original reference (as above, the reference will have to be assigned a new value before it can be read from again, enforced by definite assignment checks). In this way we can "assign ownership" to such a reference, by an assignment such as

```
this.repField = (rep) new unique C();
```

Because the old reference is invalidated, and because it was guaranteed to be unique, we know that this "assignment" of ownership will be consistent – from now onwards the object will be treated as if it had always been a rep, peer, or whatever is chosen. Note that if we choose to cast the reference to "any" at this point, no one will ever know its owner (which is still sound, since no one will be able to modify its state either).

Here is how to extend the "viewpoint adapation" type combinator:

▶	peer	rep	lost	any	unique
self	peer	rep	lost	any	unique
peer	peer	lost	lost	any	unique
rep	rep	lost	lost	any	unique
lost	lost	lost	lost	any	unique
any	lost	lost	lost	any	unique
unique	lost	lost	lost	any	unique

Various extensions to this idea are possible. For example, one can allow **unique** to be a subtype of **any**, which would then permit **any** references to alias a **unique** reference. This is harmless, since **any** references do not guarantee any information about the heap/ownership topology.

We might also like to permit method calls on unique references. Here we need something similar to the "unique" annotation on constructors – we need to enforce that the receiver of the method is not aliased. Furthermore, we might like to be able to have receivers/arguments which cannot themselves be returned as "unique", but which are still forced not to be aliased – this is necessary if we want to call a method on "this" in a constructor, for example, since the reference to "this" cannot be invalidated as it cannot be assigned.

One could also allow more flexibility during the construction of such objects. For example, it is potentially ok for objects referred to by unique references to create their own peers and reps, and to mutually refer to each others fields. In this way we can build up an object structure in a "bubble", in which many objects mutually refer to each other, but the ultimate owner of the object structure is not yet decided. This concept is weaker than uniqueness (it is known as "external uniqueness" in research papers). This is also similar to the way in which object initialisation for non-null types is handled by the Construction Types later on in the course.

The annotation follows:
Notice that the clone method uses a unique constructor, and that no action can be done on the unique list before it is converted to **rep**.

```
public class List{

   …

   public void addFirst(int x) {
      head = new Node(x,head);
   }

   public unique List clone(){
      return new unique List(this);
   }

   private unique List(any List other) {
      head = null;
      rep Node h,p = null;
      for (any Node n=other.head;n!=null;n=n.next){
            h = new Node(n.val,null);
            if (p!=null)
                  p.next=h;
            else
                  head = h;
            p = h;
      }
   }

   rep Node head;
   private class Node{
      Node( int val, peer Node next){
            this.next = next;
            this.val = val;
      }
      peer Node next;
      int val;
   }
}
```

```
class Client{
 rep List list;

 void f(any List list){
  this.list = (rep) list.clone();
  this.list.addFirst(42);
 }

}
```