Exercise 10

1. Number 2 is not allowed – it casts from a readonly reference to a readwrite reference.
2. 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)`
3.

```
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();
    }
  }
}
```

4. Consider the typing rules for a field update $e_1.f = e_2$ (lecture 7, slide 40)
   a) There is no difference between the information that these two modifiers convey about where this object is located in the heap topology; something referred to by either **any** or **lost** could have any owner. There is no example where we could use **lost** and not **any** as the type for e_2 or vice versa; in fact, the only time either would be acceptable is if the field `f` was typed with the **any** modifier.

   b) In the case where $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier **any**, this indicates that there are *no* requirements that need to be satisfied in order for such a field update to preserve topological information. On the other hand, if $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier **lost**, then this indicates that there *are* requirements that need to be satisfied, but that the type system is not able to describe them precisely (for example, we assign to the **rep** field of a **rep** reference; there is no ownership modifier to describe the requirements here). For this reason, such field updates are never allowed.

   If e_2 is any reference with an appropriate class type (it doesn't make a difference what the ownership modifier is), then the field update $e_1.f = e_2$ will be allowed when $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier **any**, and disallowed if $\tau(e_1) \blacktriangleright \tau(f)$ is the modifier **lost**.

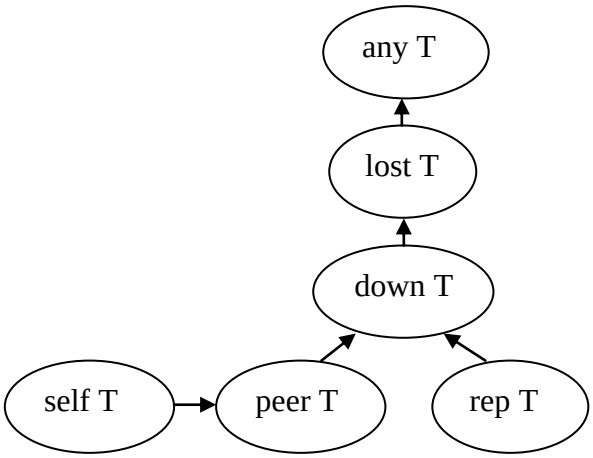
c) As mentioned above, when we try to assign to the **rep** field of a **rep** reference there is no ownership modifier to describe the topological requirements that the assigned value needs to satisfy. **rep►rep** is **lost** for this reason; this indicates that the type system cannot express when such a field update would be safe. **any** would indicate that such a field update is always safe, which would allow us to break the guarantees that a **rep** field is supposed to make.

5.

Program 1 is accepted in both systems.
Program 2 is not accepted in the topological system (and neither in the "ownership as modifier" system). It attempts the assignment of an **any** reference to a **peer** reference. **peer** is not a super-type of **any**.
Program 3 is accepted in the topological system (it assigns **any** to **any**). However, it assigns to the field of a **lost** reference, which means that it is not accepted in the "owner as modifier" system.
Program 4 is not accepted in the topological system (and neither in the "ownership as modifier" system), because it assigns to a **lost** location.

6.

a.



b.

There are two reasonable approaches to defining viewpoint adaptation. One (and perhaps the most intuitive) is to define it as describing the most precise information possible about where such a reference may belong in the heap topology (possibly over-approximating, in cases where we cannot describe precisely what we want). For example, **rep►rep** can be **down** in this approach, because down over-approximates the objects which can actually be stored in such a field. Note that this is a true approximation - **rep►rep** is not allowed to store *all* objects which can be referred to via **down**, only some of them. This means that we need to add extra restrictions on field assignment in the cases where we use **down** to over-approximate in this way; otherwise the examples in part c) would type-check, which would not be safe. Here is the appropriate table, taking this approach:

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

The alternative approach is not to allow this kind of over-approximation; the modifier chosen has to reflect *precisely* the requirements for a reference to be allowed to be stored in such a location, and thus avoid the need for extra requirements on the field assignment rule. Here is the table, in this approach:

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

In this case, perhaps surprisingly, cases such as **rep►rep** and **down►down** result in **lost**. This is because, choosing the answer **down** is not restrictive enough. In general, we have no way to express what is safe to assign to the **down** field of a **rep** receiver (**down** from our viewpoint includes objects above the **rep**, which should not be included), and similarly for a **down** receiver. See also the code in the next part. As you can see, this second approach is not very flexible; only **rep** and **peer** objects can ever be types as **down** (via subtyping).

c. Depends on the answer to the previous part. In the first case, we need to require that the result of the viewpoint adaption is not **down,** except in the special case of the receiver being **self** or **peer**, and the field type being **down** (in these cases, the **down** result expresses precisely what is safe to assign to the location; it is not an over-approximation).

With the second approach, we do not need to make any changes to the field assignment rule, to guarantee the topological constraints of the type system (as long as we define viewpoint adaptation as above).

The example code shows two cases where the field updates should *not* be allowed, because we would allow a **down** field to point upwards (to `this`) in the ownership topology, and in the second, because we would allow a **down** field to point to some object which is considered **down** from the viewpoint of `this`, but not necessarily from the viewpoint of `this.c`

These would be disallowed by both of our solutions above (check this!):

```
public class Node{
  rep Node c;
  down Node d;

  public void foo() {
```

```
        this.d.d = this;   // does this/should this type-check?
        this.c.d = this.d; // does this/should this type-check?
      }
    }
```

7.

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.

Concepts of Object-Oriented Programming

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The annotation follows:
Notice that the clone method uses a unique constructor, and that no action can be performed 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);
 }

}
```