Exercise 10

Ownership type system

1.
   -
     - `setElems` is capturing an external array
     - `getElems` is leaking the internal representation
   -
     -
       ```
       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.
     -
       ```
       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.

   -

class ArrayList

```
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();
}
```

```
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;
}
```

2.

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

3.

```
class Entry {
  any Object element;
  peer Entry previous, next;
  Entry(any Object o, peer Entry p, peer Entry n )
   {element=o; previous=p; next=n;}
```

```
class ReadIterator {
  protected any Entry current, header;

  public ReadIterator( any Entry h )
    {current = h; header = h;}

  public pure boolean hasNext()
    {return current.next != header;}

  public void moveNext()
    {current = current.next;}

  public pure any Object element()
    {return current.element;}
}
```

```
public class DeleteIterator extends ReadIterator {
  private peer LinkedList list;
  invariant current.owner == list;

  requires h.owner == l;
  public DeleteIterator( peer LinkedList l, any Entry h ) {
    super( h );
    list = l;
  }

  public void delete() {
    list.delete(current);
    current = current.next;
  }
}
```

```
public class LinkedList {
  private rep Entry header;
  private int size;

  public LinkedList() {
    Init();
  }

  public Init() {
    // the header is a dummy that is never null
    header = new rep Entry(null, null, null);
    header.next = header;
    header.previous = header;
    size = 0;
  }

  public void add(any Object o){
    rep Entry newE = new rep Entry(o,header,header.next );
    header.next.previous = newE;
    header.next = newE;
    ++size;
  }

  public pure any Object get( int idx ) {
    if( idx > size ) return null;
    any Entry e = header.next;
    for( int i=0; i<idx; ++i ) {
      e = e.next;
    }
   return e.element;
  }

  public pure peer ReadIterator getReadIterator() {
   return new peer ReadIterator( header );
  }

  public pure peer DeleteIterator getDeleteIterator() {
   return new peer DeleteIterator( this, header );
  }

  // precondition that the node belongs to us
  requires e.owner == this;
  protected void delete( any Entry e ) {
    rep Entry re = (rep Entry) e;
    if( re.previous != null )
      re.previous.next = re.next;
    if( re.next != null )
      re.next.previous = re.previous;
    --size;
  }
}
```

In the provided solution the LinkedList owns its elements. The read iterator uses a list's element only for reading but not for writing, that is why it is enough to have only an "any" reference into the list representation. On the other hand the DeleteIterator requires write access to the list representation which is not possible via an "any" reference. One way to deal with this problem is to delegate removal of an element to the list. Method `delete( any Entry e )` does it. To make this method accessible outside of the list we annotate the input parameter as **any** rather than **rep**. Inside the method we use casting to get a valid **rep** reference. To guarantee validity of the casting we add the

precondition of the method. To satisfy the precondition of the `delete` method call we add the invariant to the DeleteIterator class. To satisfy the invariant we add the precondition to the constructor of the DeleteIterator.

4.

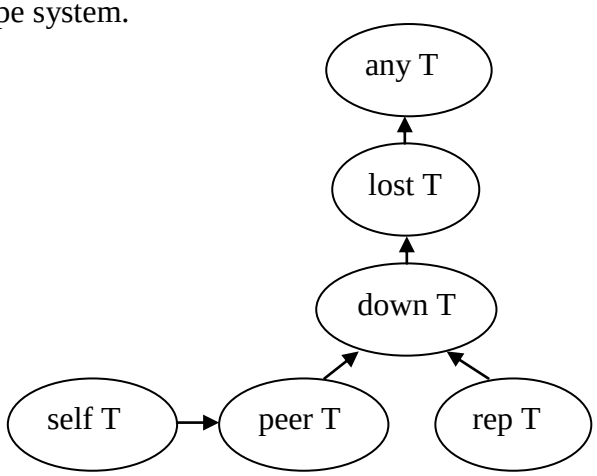
```
public void merge(LinkedList other) {
  if (other.header.next == other.header)
    return;

  any Entry entry = other.getHeader().next;
  while(entry != other.header){
    add(entry.element);
    entry = entry.next;
  }
  other.Init();
}
```

To implement the initial version of the proposed method we need ownership transfer. Namely we need transfer elements from the representation of one list to the representation of another. Since we don't have such an operator we need to redesign the `merge` method. In the proposed implementation we copy the representation of one list to another. This requires only read-only access. As an obvious drawback complexity of the method increase from constant to linear by size of the second list.

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`**