Exercise 10

Readonly and Ownership Types

2nd December

1. (The following multiple choice question is taken from a previous exam)
   In the `readonly`/`readwrite` type system, which of the following assignments is not type correct?
   1. `x=y;` where `x` is `readonly` and `y` is `readwrite`
   2. `x=y.f;` where `x` is `readwrite`, variable `y` is `readonly` and field `f` is `readwrite`
   3. `x=y.f;` where `x` is `readwrite`, variable `y` is `readwrite` and field `f` is `readwrite`
   4. `x=y.f;` where `x` is `readonly`, variable `y` is `readwrite` and field `f` is `readwrite`

2. 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.

3. Annotate the following program with appropriate ownership type modifiers to maximize the buffer, the producer, and the consumer encapsulation:

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

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

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

  Context(){
    p = new Producer();
    c = new 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) Consider two particular cases : e_2 is typed with the ownership modifier **any**, or e_2 is typed with the ownership modifier **lost**.

      Suppose that e_2 refers to an object (i.e., not null). Is there a difference between the information that these two modifiers convey about where this object is located in the

heap topology of ownership trees?

Can you find an example (by choosing the ownership modifiers for e_1 and `f`) when a field assignment would be typeable in one of the two cases (of e_2 being **any** or **lost**) but not the other? Explain briefly why this is the case.

b) Suppose instead that e_1 is typed with ownership modifier $\tau(e_1)$ and `f` has ownership modifier τ (`f`). We consider two different cases: $\tau(e_1) \blacktriangleright \tau$ (`f`) is the modifier **any**, or $\tau(e_1) \blacktriangleright \tau$ (`f`) is the modifier **lost**.

Is there a difference between the information that these two modifiers convey about *topological requirements* associated with the location e_1`.f` (i.e., what needs to be guaranteed before an object can be validly assigned to this location)?

Can you find an example (by choosing the ownership modifier for e_2) when a field assignment would be typeable in one of the two cases (of $\tau(e_1) \blacktriangleright \tau$ (`f`) being **any** or **lost**) but not the other? Explain briefly why this is the case.

c) Considering your answers above, explain why it makes sense that **rep►rep** is **lost** and not **any**. You may want to show an example.

5. (The following question is taken from a previous exam)
Consider the following declarations:

```
class A
{
  rep B first;
  rep B second;
}
class B
{
  any A obj;
  peer B sibling;
}
```

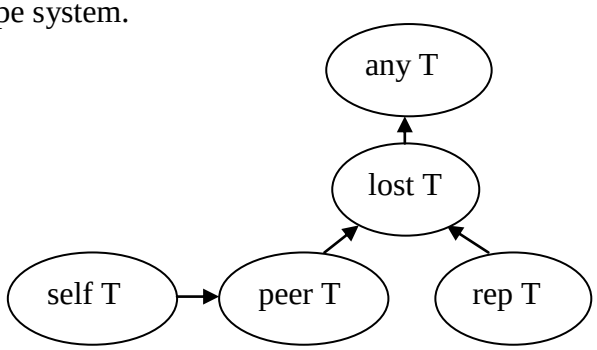
Which of the following programs are allowed in the topological ownership system? For any program that is accepted in the topological system, is it also accepted in the owner-as-modifier system? Assume that none of the objects involved are null. Briefly explain each of your answers.

Program (1)
`rep B b;`
…
`b = b.sibling;`
	Program (2)
	`peer A a; rep B b;`
	…
	`a = b.obj;`

Program (3)
`any A a;`
…
`a.first.obj = a;`
	Program (4)
	`peer A a;`
	…
	`a.first = a.first;`

6. The *Ownership type system* allows the following ownership modifiers: **peer**, **rep**, **self**, **lost**, and **any** - to structure the object store and to restrict how references can be passed and used. We want to extend the *Ownership type system* by adding one more modifier **down**. This modifier is introduced to denote references to objects in the same context as **this** or in the context (*transitively*) owned by an object in the same context as **this**.

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



   b) Define the most specific (in terms of the context information it conveys) viewpoint adaptation function ► by filling the table below (for a combination T_e►T_f the modifier T_e specifies the row, and the modifier T_f the column of the table used).
   Recall that the viewpoint adaptation function ► is used, in particular, to determine the owner of an object referenced by a field access. More exactly, if the ownership modifier of *e* is T_e and the ownership modifier of a field *f* is T_f, then the ownership modifier assigned to the field access *e.f* is determined as T_e►T_f. Note that this applies to field updates as well as field reads.

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

   c) Assuming that you only need to enforce the topological constraints of the type system (you do not need the owner-as-modifier property), how should the field update rule from lecture 7 slide 40 be adapted to the system extended with the **down** modifier. Do you need to make any changes? You might like to consider the following example code, in assessing your answers to b) and c):

```
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. (Harder!) Consider the following code:

```
public class List{
   rep Node head;

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

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

   private List(List other){
      head = null;
      Node p = null;
      for (Node n=other.head;n!=null;n=n.next){
            Node h = new Node(n.val,null);
            if (p!=null) {
                  p.next=h;
            } else {
                  head = h;
            }
            p = h;
      }
   }
   private class Node{
      Node next;
      int val;
      Node( int val, Node next){
            this.next = next;
            this.val = val;
      }
   }
}
```

```
class Client{

   rep List list;

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

}
```

a. Try annotating the code of the `List` with appropriate ownership annotations. You should find a problem – explain it. Does this indicate an aliasing issue in the code?
b. Can you think of a way to extend/modify the ownership type system to allow for this example to be typed? You might like to consider:
   i. What kind of topological property would you like to describe?
   ii. What rules do you need to preserve this property? Think about both field reads and field writes.
   iii. How does your approach relate to the existing modifiers (**rep**,**peer** etc)? Can you suggest rules for subtyping and casting?