Exercise 13

Non-null Types and Invariants

Don't forget the extra restriction for the non-null/raw type system, which was introduced in the previous sheet: "If e_2 is of a raw type, then a field assignment e_1`.f = `e_2 is only allowed when the field assignment is in the body of a constructor and e_1 is the expression `this`"

For this sheet, you can assume a slightly-stronger version of the type invariant given in the lectures (lecture 8.2, slide 25):

"If the static type of an expression e is a non-null type then e's value at run time is different from **null**"

1. Let's say that an object is "*locally initialised*" exactly when: every one of its fields which is declared with a non-null type contains a non-null value (which might or might not be raw). That is, when the local non-null constraints are satisfied for the object.

   Which of the following statements are always true in the raw/non-null type system? Justify your answers carefully.

   a. If expression `e` has a non-raw type, then `e` refers to an object.
   b. If expression `e` has a non-null type and `e` refers to an object then the object is locally initialised.
   c. If expression `e` has a non-raw type and `e` refers to an object then the object is locally initialised.
   d. If expression `e` has a raw type and `e` refers to an object then the object is not locally initialised.
   e. If expression `e` refers to an object which is locally initialised, and `e.f` refers to an object then the latter object is also locally initialised.
   f. If expression `e` has a non-raw type and `e` refers to an object which is locally initialised, then if `e.f` refers to an object then the latter object is also locally initialised.
   g. If expression `e` has a non-raw type and `e` refers to an object, then if `e.f` refers to an object then the latter object is also locally initialised.
   h. Once an object is locally initialised, it always remains locally initialised.
   i. Once a constructor for an object completes, the object is locally initialised.

2. Consider the following three classes:

```
public class DogOwner {
  Dog! dog;   // DogOwners must have a dog
  Bone! bone; // DogOwners must have a bone (for their dog)

  public DogOwner() {
    this.dog = new Dog(this);
    this.bone = dog.bone;
  }
}

public class Dog {
```

```
  DogOwner! owner; // Dogs must have an owner
  Bone! bone;      // Dogs must have a bone

  public Dog(DogOwner owner) {
    this.owner = owner;
    this.bone = new Bone(owner, this);
  }
}

public class Bone {
  DogOwner! owner; // Bones must have an owner
  Dog! dog;        // Bones must belong to a dog..

  public Bone(DogOwner owner, Dog dog) {
    this.owner = owner;
    this.dog = dog;
  }
}
```

- Annotate the code with non-null and raw annotations where they are necessary.
- The code will still not type-check according to the non-null rules. Why not?
- Use casting to fix the problem, so that the example type-checks.
- Add a post-condition ("ensures" clause) to one of the constructors, to provide sufficient information to guarantee that the cast(s) you have introduced are always safe.

3. An object o_2 is said to be *reachable* from an object o_1 if it is possible to reference o_2 via (possibly nested) field lookups starting from a reference to o_1.
For a recursive definition, we could state equivalently that o_2 is reachable from o_1 if and only if either:
   a) o_1 and o_2 are the same object, or,
   b) for some field `f` of o_1, the lookup o_1.`f` gives an object (not null) and o_2 is reachable from o_1.`f`

   Justify carefully why the following statements are true:
   - Any object is reachable from itself (reachability is reflexive).
   - If o_2 is reachable from o_1 and o_3 is reachable from o_2 then o_3 is reachable from o_1 (reachability is transitive).
   - If o_2 is reachable from o_1 it is not necessarily the case that o_1 is reachable from o_2 (reachability is not always symmetric).

     NOTE: During execution of any method/constructor, the only objects which are ever reachable must either have been originally reachable from the method receiver/arguments, or must have been created during the execution. (Justify?).

   Use your answers to question 1 to carefully justify why the following statement is always true in the raw/non-null type system:
   "If expression `e` has a non-raw type and `e` refers to an object then all objects reachable from the object are locally initialised."

4. Let's say that an object is "fully initialised" if all objects reachable from the object are locally initialised. In particular, all objects in a program should be "fully initialised" if

no constructors are currently executing. The point of "raw" types is to identify objects which **might** not be fully initialised.

Using your answers to questions 1(i) and 3, justify the following two statements:

- "If expression `e` refers to an object and the object is not fully initialised, then `e` cannot possibly have a non-raw type."
- "Once an object is fully initialised *and the constructors of all objects reachable from it have been executed*, it will always remain fully initialised." (NOTE: the text in italics has been added, since the original version of the sheet).

Note that the first of these statements is equivalent to the following:

"If expression `e` has a non-raw type and refers to an object, then the object is guaranteed to be fully initialised."

Now consider the classes from Question 2. Consider an execution of a statement:
`new DogOwner();`
Complete the following table, describing (at the specified points during execution) how many reachable objects (reachable from the references available in the method currently executing) are considered "raw", in the sense that all existing references to the objects must have a raw type. In the second column, show how many of these objects are currently locally initialised. It's probably easiest to give the objects "names" – say "O" for the owner which gets created, "D" for the dog, and "B" for the bone. Note that when the first constructor begins, the only reachable object is "O" (no arguments are passed).

Point during execution:	Reachable "raw" objects...	...which are locally initialised
Beginning of executing `DogOwner` constructor	O	none
Beginning of executing `Dog` constructor	O, D	none
Beginning of executing `Bone` constructor		
End of executing `Bone` constructor		
End of executing `Dog` constructor		
End of executing `DogOwner` constructor		

Using the table, or otherwise, explain carefully why the following properties hold:

a) Once the constructor for an object has finished executing, the object is guaranteed to be locally initialised.

b) Once a constructor has finished executing, all objects which were created during execution of the constructor are guaranteed to be locally initialised.

c) If raw references are passed to a constructor *then, at the very end of the execution of the constructor, there might be objects reachable via references currently in scope (variables, parameters, etc.) which are not yet locally initialised.*(NOTE: the text in italics has been clarified since the original version)

d) If raw references are passed to a constructor, then when it finishes executing, the object being constructed might not be fully initialised.

e) If raw references are passed to a constructor, the object returned from the constructor call should only be allowed to be given a raw type.

f) If no references are passed to a constructor, then when it finishes executing, all objects reachable from the constructed object will also be locally initialised.

g) If no references are passed to a constructor, then when it finishes executing, all objects reachable from the constructed object will be fully initialised.

h) If no **raw** references are passed to a constructor, then when it finishes executing, all objects reachable from the constructed object will also be locally initialised.

i) If no raw references are passed to a constructor, then when it finishes executing, all objects reachable from the constructed object will be fully initialised.

j) If no raw references are passed to a constructor, it makes sense to give the object returned from the constructor call can a non-raw type.

5. A technique to represent a complete binary tree T using an array A, is:
   - store the root in A[0]
   - for any node N stored in A[i], store the children of N to A[2i+1] and A[2i+2].

   The size of the array should be equal to 2^{h+1}-1, where *h* is the height of the tree.

   Consider the following invariant on a complete binary tree of integers: *any non-leaf node stores the sum of the integers stored in its two children*.

   The following class uses the above-mentioned representation.

```
class CompleteBinaryTree
{
 private int[] theTree;

 public CompleteBinaryTree (int h)
 {
       theTree = new int [Math.pow(2,h+1)-1];
       for (int i=0; i<theTree.length; i++)
             theTree[i]=0;
 }

 // requires 0 ≤ i < theTree.length
 public int getNode (int i) { return theTree[i]; }

 // requires theTree.length/2 ≤ i < theTree.length
 // this means i must be a leaf
 // attention: this precondition was wrong in the initial
 // version of the exercise
 public void addToLeaf (int i, int s)
 {
       addToNode (i, s);
 }

 private void addToNode (int i, int s)
 {
       theTree[i]+=s;
       if (i>0) addToNode ((i-1)/2, s);
```

```
  }
}
```

a) Write formally the invariant described above

b) Show that the invariant is always preserved by the public methods of the class. Hints:

(a) does the method `addToNode` preserve the invariant?

(b) what expectations do you have of the state of the object when `addToNode` is called?

(c) write a precondition for `addToNode` expressing these expectations

(d) prove that the precondition is always satisfied when the method is called

(e) is it true that the invariant holds after every call to `addToNode` (assuming your precondition was satisfied when the call was made)?