Exercise 11

Non-null types and (a little) initialisation

10th December

In-class Assessment: One or more questions from this sheet will be used for the in-class assessment. No notes are allowed during the assessment.

1. Consider a Java class `Coordinates`, storing two numerical values:

```
public class Coordinates {
  public Number x; // Remark: Number is a class which
  public Number y; //      Integer, Double, etc. all extend

  public Coordinates (Number x, Number y) {
    this.x = x;
    this.y = y;
  }
}
```

   Suppose that in some other class we write the following method to calculate the length of the vector represented by a `Coordinates` object:

```
public double vectorLength(Coordinates c) {
  double x = c.x.doubleValue();
  double y = c.y.doubleValue();
  return Math.sqrt(x * x + y * y);
}
```

   - This implementation is unsafe – when executed it may throw exceptions. Why?
   - It is unreasonable to expect an implementation of the intended method to execute safely in all situations. Why?
   - Add a pre-condition for the method, specifying what is required to be safe.
   - Suppose you are allowed to modify the signature of the method to include non-null type annotations. To what extent can you weaken the necessary pre-condition?
   - Suppose that you are also allowed to upgrade the class `Coordinates` to include reasonable non-null type annotations. How does this affect your previous answer? Do these changes to the class seem reasonable?

2. Suppose that we add a subclass `TripleCoordinates` which has a third `Number` field `z` and a new method `volume()`:

```
public class TripleCoordinates extends Coordinates {
  public Number! z;

  double volume() {
    return x.doubleValue()*y.doubleValue()*z.doubleValue();
  }
}
```

Which of the following method definitions compile (assuming that the data-flow analysis for non-null types doesn't consider the semantics of `instanceof`)? Which would always run safely?

```
double getVolume1(Coordinates? c) {
  if(c instanceof TripleCoordinates) {
    return c.volume();
  } else { return 0.0; }
}

double getVolume2(Coordinates? c) {
  if(c instanceof TripleCoordinates) {
    return ((!) c).volume();
  } else { return 0.0; }
}
double getVolume3(Coordinates? c) {
  if(c instanceof TripleCoordinates) {
    return ((TripleCoordinates!) c).volume();
  } else { return 0.0; }
}
double getVolume4(Coordinates? c) {
  if(c!=null && (c instanceof TripleCoordinates)) {
    return c.volume();
  } else { return 0.0; }
}
double getVolume5(Coordinates? c) {
  if(c!=null && (c instanceof TripleCoordinates)) {
    return ((!) c).volume();
  } else { return 0.0; }
}
double getVolume6(Coordinates? c) {
  if(c!=null && (c instanceof TripleCoordinates)) {
    return ((TripleCoordinates!) c).volume();
  } else { return 0.0; }
}
```

3. **(question from a previous exam)**
   This question is about extending the non-null type system to handle arrays (ignoring initialisation). Array types can have *two* type modifiers, declaring independently the nullity expectations for the array itself and the array elements. For any array type `T[]` the corresponding variants are `T?[]?`, `T?[]!`, `T![]?`, `T![]!` (the first modifier applies to the type of objects stored in the array, while the second modifier concerns the reference to the array object itself).
   Assuming that we want to guarantee a statically sound approach to subtyping (that is, we want to enforce safety at compile time, without using runtime checks), explain whether or not the following subtype relations are safe. For each relation you consider unsafe, provide a code snippet illustrating that allowing such a subtype relationship would break the safety guarantees of the type system. For these unsafe cases, explain also what runtime checks could be made to restore safety.

- `T?[]! <: T?[]?`
- `T![]! <: T![]?`
- `T![]? <: T?[]?`
- `T![]! <: T?[]!`

4. Consider the following abstract class, representing a node of a singly-linked list:

```
public abstract class ListNode<X> {
  public abstract void setItem(X x);
  public abstract X getItem();
  public abstract ListNode<X> getNext();
}
```

Consider now the following implementation using a simple (acyclic) list:

```
public class AcyclicListNode<X> extends ListNode<X> {
  protected X item;
  protected AcyclicListNode<X> next;

  public AcyclicListNode<X> (X item) {
    this.item = item;
    this.next = null;
  }

  public void setItem(X x) { item = x; }
  public X getItem() { return item; }
  public AcyclicListNode<X> getNext() { return next; }
}
```

In this implementation, suppose that an empty list is represented simply by a null reference. Suppose that a further design intention of this implementation is that each node is *guaranteed* to store an `X` object in its `item` field.

- Annotate the class `AcyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).

Now consider an alternative implementation using a cyclic list:

```
public class CyclicListNode<X> extends ListNode<X> {
  protected X item;
  protected CyclicListNode<X> next;

  public CyclicListNode<X> (X item) {
    this.item = item;
    this.next = this;
  }

  public void setItem(X x) { item = x; }
  public X getItem() { return item; }
  public CyclicListNode getNext() { return next; }
}
```

In this implementation, the design intention is that every node will always have a `next` object in the list (sometimes itself). In this design, we choose to represent an empty list by a single node whose `next` field points to itself, but whose `item` field is null. All non-empty lists will be represented using only nodes whose `item` fields are non-null.

- Annotate the class `CyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).
- Now consider how to annotate the method signatures in `ListNode<X>` so that both implementations can be accommodated. Your solution should be compatible with the usual co/contra-variance rules for subclass method signatures.

5. Consider a client method `length` which takes a `ListNode<Integer>` argument and calculates the length of the represented list (i.e., the length of the sequence of `Integer` items stored in the sequence of nodes starting from the method argument, and reachable by successive calls to `getNext()`)
   - Write an implementation of this method. Make sure that your method cannot throw null-pointer exceptions. It must work for both of the two implementing classes from the previous question, and for any valid list representation according to those two designs. Do not assume in your implementation that there are *only* these two implementing classes (i.e., don't use `instanceof` to distinguish them). Use non-null types in the signature of your method as appropriate.

- **For fun only**: is your method guaranteed to terminate, considering either of the two implementing classes? If not, write a version which is. Can you think of an interesting alternative implementation extending `ListNode<X>` for which your method does not terminate?