Exercise 11

Non-null types and Initialisation

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,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?

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. In the non-null type system, a class type T can be annotated to explicitly declare non-nullity (`T!`) and possible-nullity (`T?`). A "raw" variant of each of these types is also used in the type system (`raw T!` and `raw T?`) to represent partial initialisation.
   - Give at least two examples showing differences between what can be done with a reference of type `T!` and a reference of type `raw T!`
   - Give at least two examples showing differences between what can be done with a reference of type `T?` and a reference of type `raw T?`
   - "raw" types are not allowed to be used in field types. Does this restriction seem reasonable? What would be the consequences of relaxing the restriction?

4. Suppose we want a class to represent a `MarriedPerson`, who always has a spouse:

```
public class MarriedPerson {
  private MarriedPerson! spouse;
  // invariant: this.spouse.spouse == this;

  public MarriedPerson(MarriedPerson? spouse)
    if(spouse!=null) {
      this.spouse = spouse;
    } else {
      this.spouse = new MarriedPerson(this); // invent one!
    }
    this.spouse.setSpouse(this);
  }
```

```
    // requires: spouse.spouse == this;
    // ensures: this.spouse == spouse;
    protected setSpouse(MarriedPerson! spouse) {
      this.spouse = spouse;
    }
  }
```

- Why do we need to allow the constructor to take a possibly-null argument?
- Add "raw" annotations where necessary, for this code to type-check.

Consider a subclass `MarriedParent` which has the definition:

```
public MarriedParent extends MarriedPerson {
  String! childNames;

  public MarriedParent(MarriedPerson? spouse, String! names) {
    super(spouse);
    this.childNames = names;
  }

  // requires: spouse.spouse == this;
  // ensures: this.spouse == spouse;
  protected setSpouse(MarriedPerson! spouse) {
    this.spouse = spouse;
    if(spouse instanceof MarriedParent) {
      this.childNames += ((MarriedParent!)spouse).childNames;
    }
  }
```

- If possible, add appropriate "raw" annotations to make this code type-check. If not possible, explain clearly why not.

5. 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;
  }

  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.

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; // default – maybe changed later
  }

  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.
- Now consider how to annotate the method signatures in `ListNode<X>` so that both implementations can be accommodated.
- Can the code be type-checked according to the rules of non-null types? In particular, are the bodies of the constructors acceptable, according to the rules regarding raw types? Justify your answers, explaining carefully what is checked for constructors.

6. 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?