Exercise 11 solutions

Non-null types and (a little) initialisation

9th December

1.

- If `c` were `null`, the field dereferences `c.x` and `c.y` would generate exceptions. Furthermore, if `c.x` were `null` then method call `c.x.doubleValue()` would generate an exception. Similarly if `c.y` were `null`.
- There is no reasonable answer for the method to return if it encounters `null` values – any attempt to deal with these cases would have to return some arbitrary value, since the question the method is meant to answer is undefined in these cases.
- `requires: c≠null ∧ c.x≠null ∧ c.y≠null`
- `public double vectorLength(Coordinates! c)` would make the following pre-condition sufficient: `requires: c.x≠null ∧ c.y≠null`
- By changing the types of the fields `x` and `y` to be `Number!` we could guarantee that no pre-condition would be required. This seems a reasonable change, since a null coordinate doesn't seem to be meaningful anyway.

2. `getVolume1` won't compile for two reasons – Java will complain that `c` is of (class) type `Coordinates` for which method `volume` is not defined, and a non-null type checker would complain that it cannot determine that `c` is non-null when the call is made. However, the program would run safely – the if-condition not only guarantees that the method is defined for the call, but implicitly that the expression `c` is non-null when the call is made (because Java defined that `(null instanceof T)` always evaluates to `false`.
`getVolume2` won't compile for the first reason above - Java will complain. The code would still be safe.
`getVolume3` will compile - the cast satisfies all the necessary constraints to be checked. The code will still be safe (in particular, the cast always succeeds).
`getVolume4` and `getVolume5` won't compile for the first reason above - Java will complain. The code would be safe though. Note that the non-null type checker won't complain in either case, because of the new if-condition.
`getVolume6` will compile and run safely.

3.

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

```
Object![]? x = new Object![1]?;
Object?[]? y = x;
if(y!=null)
     y[0]=null;
if(x!=null)
     x[0].toString();
```

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

```
Object![]! x = new Object![1]!;
Object?[]! y = x;
y[0]=null;
x[0].toString();
```

In both the last two cases, we need to check at runtime if a value stored in an array with dynamic non-null type for the elements stored in the array is not the null value.
Alternatively, we can check at runtime if a value read from an array with dynamic non-null type is not the null value.

4. (Side note: the interaction of generic types and non-null types, e.g., the interpretation of a type `X!` if `X` can be instantiated with types that themselves include non-nullity expectations, is beyond the scope of the course, but in case you are worried, you can assume that the explicitly visible annotation ! overrides any annotation in the instantiation for X, i.e., `X!` can still be safely assumed to always store a non-null value)

- The following class definitions express the design expectations:

```
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; }
}
```

-

```
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.

- We have to pick suitable method signatures so that the implementing methods have valid overriding signatures in both classes above. This typically means strengthening the argument types and weakening the return types:

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

5.

-

```
public int length(ListNode<Integer>? l) {
  ArrayList! seen = new ArrayList(); // nodes seen
  ListNode<Integer>? current = l;
  int count = 0;

  while(current!=null && !seen.contains(current)) {
    if(current.getItem()!=null) { // skip null items
      count++;
    }
    seen.add(current); // termination in cyclic lists
    current = current.getNext();
  }

  return count;
}
```

Note that the method argument may reasonably be `null`, since this is the `AcyclicListNode<Integer>` representation of an empty list.

- The method will terminate for the two implementation classes in the previous question because we keep track of the nodes we have already inspected – this is necessary in the case of cyclic lists. It seems likely that the methods would terminate for all "usual" implementations of `ListNode<X>`. However, if the behaviour of `getNext()` was not to return an existing reference, but to create a new list node every time and return that, we couldn't guarantee termination. Incidentally, if we were to declare `getNext()` to be pure, and impose the strictest definition of purity checks, this awkward case could be avoided.