Exercise 11 solutions

Non-null types and Initialisation

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 case 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.

- In both cases, it is guaranteed that the reference will contain a non-null value. For a reference of type `T!` it is safe to assume that any fields of the object accessible via the reference have been initialised – in particular, if the field type is a non-null type, it is safe to assume that the field stores a non-null value. This is unsafe if the original reference were of type `raw T!` – accessing a field of a raw type always gives a possibly-null type, even if the field is declared with a non-null type.
  When writing to a reference of type `T!` only a non-raw, non-null value may be assigned. However, for a reference of type `raw T!` it is allowed for a value of a (suitable) raw type to be assigned.
- The same differences as in the previous part. The "raw" annotation is essentially independent of the non-null annotation.

- Since fields persist in an object, and their *declared* types cannot change, if we were allowed to declare a field type as "raw" then it would never be possible to be sure that an initialised value were stored in the field. Given that "raw" is intended to express a temporary condition (partial initialisation) it doesn't seem to make intuitive sense either for a field to be declared as "raw". Note that according to the rules described in the lecture, we are still allowed to assign a raw reference to a non-raw field of a raw object, so we are able to flexibly initialise multiple objects at once, assigning them to each others' fields even while they are temporarily uninitialised – this seems flexible enough to allow field types to be as permissive as we need during execution of constructors.

4.

- We need a constructor which takes a possibly-null argument to get started – otherwise we could only create instances of this class when we had an existing one (which we wanted to marry...).
- The constructor can be called with a partially-initialised object, so its parameter needs to be raw. `setSpouse` can also be called when both its receiver and its argument are raw, hence the two annotations.

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

  public MarriedPerson(raw MarriedPerson? spouse)
    if(spouse!=null) {
      this.spouse = spouse;
    } else {
      this.spouse = new MarriedPerson(this);
    }
    this.spouse.setSpouse(this);
  }

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

- Because the types of the argument (and the receiver!) of `setSpouse` can only vary contravariantly in the subclass definition, we are required to keep the same raw annotations on this method signature as we had for the superclass. Now, the access to `this.childNames` will not type-check, since at that point, `this` is a raw reference type, and so accessing its field yields a possibly-null type. We cannot get around this problem by adding raw annotations. However, this restriction is actually correct – because `setSpouse` can be dynamically dispatched from the superclass constructor, it could be the case that `childNames` is not yet initialised, and so a null pointer exception would potentially be generated by this code.

5.

- 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();
}
```

- The implementations of the three methods above are all non-controversial in terms of the non-null annotations. In all cases, we assign references of exactly the right types to fields. For constructors, we need to be slightly careful because of the use of "raw" types. For the constructor of `AcyclicListNode`, we assign a value of type `X!` to a field whose type (at that point) is `raw X!` (since `this` is of a `raw` type, its fields are also). For the `CyclicListNode<X>` constructor, the first assignment sets a field of type `raw X?` to a value of type `X?`. This is allowed.

Finally, it assigns the value `this`, which is currently of type `raw CyclicListNode<X>` to the field `next` of `this` which is currently of type `raw CyclicListNode<X>`. This is also allowed.

6.

- ```
  public int length(ListNode<Integer>? l) {
    ArrayList seen = new ArrayList(); // track 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.
  ```