Section B

Attempted questions: 3

Attached question: 3

a.

i. Call the constructor of the parent class

ii. The current instance of the `Element` class being constructed

iii. It is a sanity check so that if there was a typo/error in the code such that no method is actually being overridden (e.g. if the programmer had written `toStrign` rather than `toString`), the compilation will fail, in order to let you know.

iv.

```
class Element {
    final int item;
    final Element next;

    Element(int item, Element next) {
        super();
        this.item = item;
        this.next = next;
    }

    @Override
    public String toString() {
        return item + " " + (next == null ? "" : next);
    }
}
```

b.

```
class ListEmptyException extends RuntimeException {}

class FuncList {
    private Element myHead;

    public FuncList() {
    }

    private FuncList(Element head) {
        this.myHead = head;
    }

    public int head() {
        if (myHead == null) {
            throw new ListEmptyException();
        } else {
            return myHead.item;
        }
    }

    public FuncList tail() {
        if (myHead == null) {
            throw new ListEmptyException();
        } else if (myHead.next == null) {
            return new FuncList();
        } else {
            return new FuncList(myHead.next);
        }
    }

    public void cons(int x) {
        Element oldHead = myHead;
        myHead = new Element(x, oldHead);
    }

    @Override
    public String toString() {
        if (myHead == null) {
            return "[]";
        }
        return "["+myHead+"]";
    }
}
```

c.

i.

This is because whichever class is used for `T` might not be immutable. The programmer could hold a reference to one of the instances of `T` held in the list and modify it from there, thus modifying the list contents. This could be remedied by requiring `T` to implement some interface with a method called `copy` which would create a copy of the object (with a different reference) and then the `cons` method coud call `copy` on the object before passing it to the `Element` constructor.

ii.

This is necessary in this case. Imagine we have some class `B` which extends `A`. If covarience of generic types was allowed, `FuncList<B>` would be a child class of `FuncList<A>`. Suppose we had some object of type `FuncList<A>` whose head is an instance of `Element<A>` and we passed to its `cons` method an instance of `B`. The `myHead` object created would be of type `Element<B>`, whose `next` attribute would be of type `Element<A>` which is contravariant.