## Questions

Bookmark this page

### Terrible

2/2 points (graded)

We wrote our `ImList` implementations without documenting the abstraction function and representation invariant, which makes us terrible people.

```
public class Empty<E> implements ImList<E> {
    public Empty() {
    }
    public ImList<E> cons(E e) { return new Cons<E>(e, this); }
    public E first() { throw new UnsupportedOperationException(); }
    public ImList<E> rest() { throw new UnsupportedOperationException(); }
}
```

Select an abstraction function for `Empty`:

- ( ) represents Empty
- (•) represents an empty list
- ( ) represents null

✔

And select lines to include in the rep invariant (let's include even the 6.005-implicit things):

- [x] nothing
- [ ] e != null
- [ ] e may be null
- [ ] e is not empty
- [ ] rest != null
- [ ] rest may be null
- [ ] rest is not empty
- [ ] this is empty
- [ ] this is not empty

✔

**Explanation**

AF: Remember that the abstraction function goes from representation to abstract value; the abstract value here is an empty list.
RI: Since `Empty` has an empty rep (no fields at all), it has an empty rep invariant.

Submit Show Answer

Answers are displayed within the problem

### Terrible II

2/2 points (graded)

```
public class Cons<E> implements ImList<E> {
    private E e;
    private ImList<E> rest;
    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }
    public ImList<E> cons(E e) { return new Cons<E> (e, this); }
    public E first() { return e; }
    public ImList<E> rest() { return rest; }
}
```

Select an abstraction function for `Cons` :

- ○ represents Cons
- ○ represents a non-empty list
- ○ represents a non-empty list of e and rest
- ○ represents a two-element list where the first element is e and the second element is rest
- ◉ represents a list where the first element is e and the remaining elements are rest

✔

And select lines to include in the rep invariant (let's include even the 6.005-implicit things):

- ☐ nothing
- ☑ e != null
- ☐ e may be null
- ☐ e is not empty
- ☑ rest != null
- ☐ rest may be null
- ☐ rest is not empty
- ☐ this is empty
- ☐ this is not empty

✔

**Explanation**
AF: Saying "a non-empty list" or "a non-empty list of e and rest" is not at precise as describing the structure that Cons defines, with a first element and remaining elements.
RI: We definitely want `rest != null` (we're being fully precise; normally this is implicit). What about `e` ? Again, `e != null` would be implicit. Do we want to allow null `e` ? Since our specs have implicitly disallowed `null` as the argument to `cons()` or the return from `first()` , there's no reason to allow it in the rep.

Submit

Show Answer

Answers are displayed within the problem

## The black hole at the center of my program

1/1 point (graded)

Alice and Ben are reading about `ImList` , and Ben suggests that since sharing immutable datatypes is safe, *all* empty lists should point to the same instance of `Empty` .

Which of the following is true:

- ☑ sharing the same instance of Empty throughout the entire program would be safe
- ☐ sharing the same instance of Empty throughout the entire program would not be safe
- ☐ we cannot implement this easily because empty() must return a new instance of Empty every time
- ☑ we cannot implement this easily because empty() must return instances of Empty with different element types E
- ☐ we cannot implement this easily because when we call a constructor (e.g. new Empty<>()) we must create a new object

✔

**Explanation**

Sharing the same `Empty` instance throughout the entire program would be safe: it's immutable, no incompetent or malicious client can disrupt another part of the program by messing with the shared Empty instance.

However, this is impossible to implement in a fully type-safe way in Java because `empty()` must return instances of `Empty` associated with different types `E`. Even though the empty list of, e.g., `Integer`s has no actual `Integer`s in it, the compiler will still check that we only try to `cons` on `Integer`s to that list, not elements of any other type. If we try to write down a `static ImList<SOMETHING> EMPTY` to share among the entire program, we can't fill in the `SOMETHING`.

It is also 100% true that calling a constructor *must* return a new object. But clients of `ImList` call `ImList.empty()`, not `new Empty()`, so that's not a barrier here: we do have the chance to return an already-existing `Empty` instance.

Submit

Show Answer

Answers are displayed within the problem

## Short layover

3/3 points (graded)

Given this code:

```
ImList<String> airports =
        ImList.empty().cons("SFO").cons("IAD").cons("BOS");
```

Which of the following is true?

- ○ airports.first() is "SFO"
- ◉ airports.first() is "BOS"
- ○ airports.first() is the list [ "SFO" ]
- ○ airports.first() is the list [ "BOS" ]

✔

Which of the following is true?

- ○ airports.rest().rest() is "SFO"
- ○ airports.rest().rest() is "BOS"
- ◉ airports.rest().rest() is the list [ "SFO" ]
- ○ airports.rest().rest() is the list [ "BOS" ]

✔

True or false: `Empty` and `Cons` both implementing `ImList` are analogous to `ArrayList` and `LinkedList` both implementing `List` in the Java library.

- ○ True
- ◉ False

✔

**Explanation**

They are variants that cooperate, rather than alternatives to pick from.

Submit

Show Answer

Answers are displayed within the problem

Previous Next