Week 9 Lab Session

CS2030S AY21/22 Semester 2

Lab 14B

17 Mar 2022

Yan Xiaozhi (David)
@david_eom
yan_xiaozhi@u.nus.edu

Admin

- Contact tracing & QR code
- Lab 5 marking finished
  - Do let me know if you have any question
  - 
  - Learn from mistakes!

Lab 5 Feedback

A Quick Revisit

- All classes & interfaces under `cs2030s/fp` package
- `Maybe<T>` needs to be `abstract`
- `Some<T>` and `None` nested classes to represent two different types
- `Maybe<T>` to be instantiated through factory methods `some`, `none` and `of`
- Factory methods are `public static`
- PECS correctly applied for all methods

Maybe<T>

- `private static final Maybe<?> NONE = new None();`
- `public static <T> Maybe<T> none() { … }`
- `public static <T> Maybe<T> some() { … }`
- `public static <T> Maybe<T> of() { … }`
- `protected abstract T get();`

Maybe<T>

- `protected abstract T get();`
- `public abstract Maybe<T> filter(BC<? super T> condition);`
- ```
  public abstract <U> Maybe<U> map(
      TF<? super T, ? extends U> transformer
  );
  ```
- ```
  public abstract <U> Maybe<U> flatMap(
      TF<? super T, ? extends Maybe<? extends U>> transformer
  );
  ```
- `public abstract <U extends T> T orElse(U u);`
- `public abstract T orElseGet(Producer<? extends T> producer);`

Some<T>

- ```
  @Override
  public <U> Maybe<U> map(
      TF<? super T, ? extends U> transformer) {
    return Maybe.<U>some(transformer.transform(this.get()));
  }
  ```

- ```
  @Override
  public <U> Maybe<U> flatMap(
      TF<? super T, ? extends Maybe<? extends U>> t) {
    @SuppressWarnings("unchecked")
    Maybe<U> m = (Maybe<U>) t.transform(this.get());
    return m;
  }
  ```

Grading Scheme

- 10 marks correctness, 2 marks style
- -1 / -0.5 for each bug
- -1 for each raw type usage
- -1 for abuse of `@SuppressWarnings`
- -1 for each missing PECS

Common Mistakes

- `Some<T>` and `None` becomes visible outside `Maybe<T>`
  - Inappropriate access modifiers for nested classes, as well as `some<T>`'s and `None`'s constructors
- Rewrapping `Maybe` in `flatMap`
  - i.e. `return Maybe.<U>of(tf.transform(this.get()).get());`
  - What if transformer returns a `Some` of `null`?
  - `Maybe.<String>some("hello").flatMap(strToSomeNull)`
  - Expected `Maybe.some(null)`, returned a `None`

Common Mistakes

- `NONE` has to be `private static final` and of type `Maybe<?>`
  - Some forgot `final`, some wrote it as `Maybe<Object>`
- Actually handling `null` in `map`/`flatMap`
  - “Note that, if the `transform` method does not handle the case where the input is `null`, a `NullPointerException` will be thrown.”
  - You’re not supposed to handle!
- Abuse of `@SuppressWarnings`
  - Unnecessary for `Some<?> other = (Some<?>) obj;`

“Not So Strict” (But idgi)

- PECS for `None::map`, `None::flatMap`, `None::filter`
  - It is hidden anyways and lambdas passed are not used
  - But should still apply as much as possible
- All non-public constructors for `None` & `Some<T>` are fine
  - By right (imo) it can and only can be `private`!
- `map`, `flatMap` and `filter` need not to be `abstract`
  - To take full advantage of polymorphism and dynamic binding, they should be implemented in `Some<T>` and `None` respectively!!

Other Comments

- Over complication of `Lab5::getGrade`, use chaining instead
  - `return Maybe.of(map.get(student)).flatMap(getModule)`
    `.flatMap(getAssessment).orElse("No such entry");`
  - Okay as long as no explicit checks for `null` / `None`
- “PECS” for `orElse` is NOT needed ()
  - `public abstract <U extends T> T orElse(U u);`
  - Sorry for the confusion
- `Maybe<?> NONE` vs `None NONE`?
- Not deleting useless codes and files

Lab 6 Overview

Motivation

- Remember streams & lazy evaluation in CS1101S?
- Don’t evaluate an expression until we really need it
  - The evaluation might be expensive
  - You might not even use the value in the end, why bother evaluating?
- You’ll realise the meaning behind stuff that appears to be useless/stupid in Lab 5

Motivation

- Refer to `Lazy<T>` from lecture
- ```
  class Lazy<T> {
    private T value;
    private boolean evaluated;
    private Producer<T> producer;
    public Lazy(Producer<T> producer) {…}
    …
  }
  ```
- What if value may or may not be there
- We already have `Maybe<T>` as an abstraction for that!
  ```

Motivation

- ```
  class Lazy<T> {
    private Maybe<T> value;
    private Producer<? extends T> producer;
    …
  }
  ```
- Avoid checking if value is there, and avoid using `value.get()` since it could throw `NoSuchElementException`
- ```
  if (value.equals(Maybe.none())) { return -1; }
  else { return value.get() }
  ```
  ❌
- ```
  return value.orElse(-1);
  ```
  ✅
  ```

Lab 6: `Lazy<T>`

- Design `Lazy<T>` class that contains a `Maybe` and a `Producer`
- Imagine that calling `produce()` is going to be extremely expensive
- Avoid using `Maybe::get`
- Avoid accessing `Some<T>` and `None` directly

Lab 6: `Lazy<T>`

- Fundamentals:
  - Factory methods: `of(T v)` and `of(Producer<T> s)`
  - `get()`: if value available, return; otherwise, compute and return
    - Computation should only be done once for same value
  - `toString()`: ? if not available, string representation

Lab 6: `Lazy<T>`

- `map` and `flatMap`:
  - Lazy, do not compute the value unless is `get()` called
  - All values should only be computed once
  - Cache the result
- `filter`:
  - Lazily tests whether the value passes the test
  - Returns `Lazy<Boolean>`

Lab 6: `Lazy<T>`

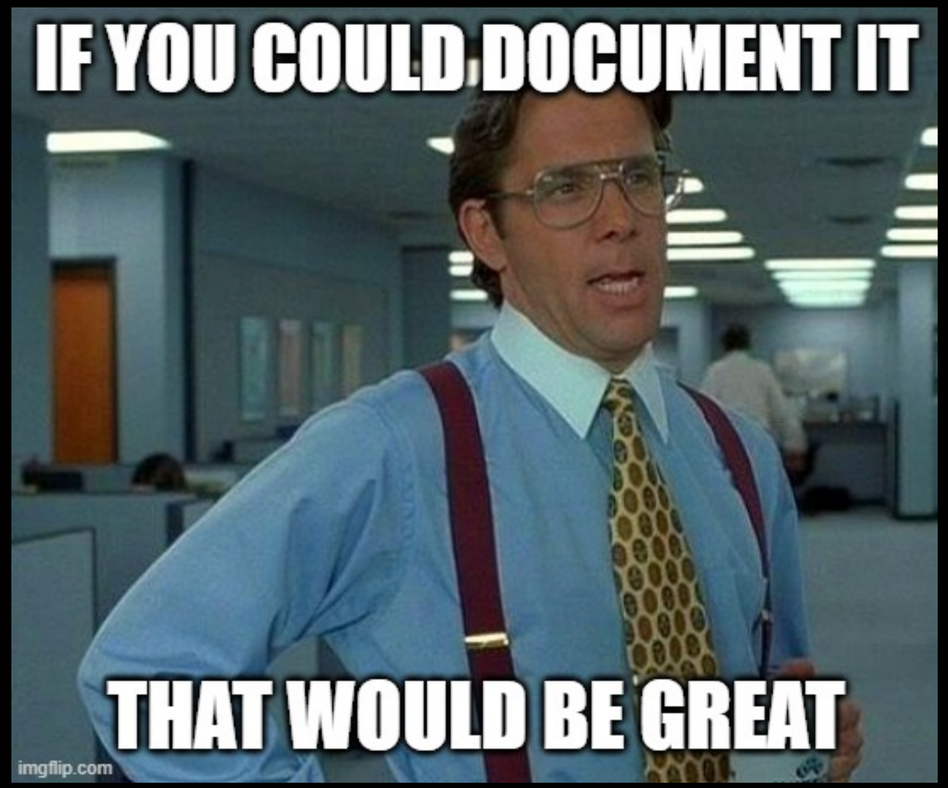
- `equals`:
  - Eager operation, compute the value immediately
  - Returns `true` iff both are lazy and value are equal
- `Combiner<S,T,R>`:
  - Combines two values (of types `S` and `T`) into a value of type `R`
  - Implement `Lazy::combine`

Lab 6: `LazyList<T>`

- Supposed we have 1 bn elements in the list
- `EagerList<T>` will immediately compute 1 bn times
- What if we just want to get the first item in the list?
- Your task: change `EagerList<T>` into `LazyList<T>`

Documentation… Why?

- Programming is about communication of ideas
- Preparation for real world
- Not only for other people
- You yourself might also get blur in the near future



class **range**(*stop*)

class **range**(*start*, *stop*[, *step*])

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, `ValueError` is raised.

For a positive *step*, the contents of a range `r` are determined by the formula `r[i] = start + step*i` where `i >= 0` and `r[i] < stop`.

For a negative *step*, the contents of the range are still determined by the formula `r[i] = start + step*i`, but the constraints are `i >= 0` and `r[i] > stop`.

A range object will be empty if `r[0]` does not meet the value constraint. Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than `sys.maxsize` are permitted but some features (such as `len()`) may raise `OverflowError`.

Range examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
```

add

```
public void add(int index,
                E element)
```

Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Specified by:

`add` in interface `List<E>`

Overrides:

`add` in class `AbstractList<E>`

Parameters:

`index` - index at which the specified element is to be inserted

`element` - element to be inserted

Throws:

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index > size())

Documentation

- Javadoc are required from now onwards, refer to CS2030S Javadoc guide
  - Happy to see many already started
  - No need for simple methods e.g. getter, setter
  - Do not need to generate and host HTML files
- Write `WHAT`, `WHY` but not `HOW`
  - Your code should already show how
- Users/clients should treat your documentation as an “API”
  - How to call, any exception/error to be thrown, what will be returned

Javadoc Tags

- Most used:
  - `@param`: method parameter and type parameter
  - `@return`: return value (omit if void)
  - `@throws`: checked exception thrown
- Also:
  - `@author`: author of a class
- Will probably be used in CS2103T:
  - `{@code}`, `{@link}`, `@version`…

Documentation

- For a method
- ```
  /**
   * Create an instance of Maybe with a given value t
   *
   * @param <T> The type of the value in the Some instance
   * @param t The value to be wrapped within this Maybe
   * @return A new Maybe instance initialized with value t
   */
  public static <T> Maybe<T> some(T t) {
    return new Some<T>(t);
  }
  ```
  ```

Documentation

- For a class:
- ```
  /**
   * Represents a location in a 2D space
   *
   * @author yourName
   */
  public class Point {
    ...
  }
  ```
  ```

Grading Scheme

- Documentation: 2 marks
  - Honestly free marks…
- Everything else: 10 marks
- Usual deductions for violations regarding `@SuppressWarnings` and raw type
- Note: no more style marks
  - You should already be following the recommended style
  - -2 marks for serious violations

Happy coding!