CS2030S PE2
AY25/26 Sem 1

General Tips

- **Compile** with `javac -Xlint:unchecked -Xlint:rawtypes *.java`
- **Student number** on top of every file: `// @author STUDENT_NUMBER`
- **Increment coding**: check if your program compiles after every increment
- Know **argument types** and **return types** of the functions, to decide which operations to use
- If rewrite in functional style, **comment out** code instead of deleting
- **Readibility**: every operation should be on a new line
- Nested operations (e.g. nested `map`) to perform nested loops, or take out values from two objects
- **Check style**
- Remember to **bash submit** *all* tasks

Setup

vimrc: `set mouse+=a`, `set hidden`, `set wildmenu`, `set showcmd`

Generics

- `equals` method for `Pair`
  1. Check if `obj` is a `Pair`: `obj instanceof Pair<?, ?>`
  2. If yes, typecast `obj` to `Pair<?, ?>` (no need to suppress warning)
  3. Compare fields using fields' `equals` method
- Create **generic array**: cannot instantiate generic array `new T[]` directly, so first instantiate `Object[]`, then cast it to `T[]`

  Unchecked warning: suppress if you know typecast is safe

```java
class Seq<T> {
  private T[] array;
  public Seq(int size) {
    // Only way to put object into array is through set(),
         and we only put object of type T inside. Safe to
         cast Object[] to T[].
    @SuppressWarnings("unchecked")
    T[] tmp = (T[]) new Object[size];
    this.array = tmp;
  }
}
```

  If array contains `Pair` objects, use `new Pair<?, ?>[size]` instead

```java
class FruitStall<T extends Fruit> {
  private final List<T> fruits;
  public FruitStall(List<? extends T> fruits) { // flexible
    @SuppressWarnings("unchecked")
    List<T> tmp = (List<T>) fruits;
    this.fruits = tmp;
  }
}
```

- **Generic method**: `<T> boolean contains(T[] array, T obj)`
- **Bounded generic**: `<T extends GetAreable> T findLargest(T[] array)`
- To compare objects:

```java
class Seq<T extends Comparable<T>> {
  // :
  public T min() {
    if (this.array.length == 0) {
      return null;
    }
    T smallest = this.array[0];
    for (int i = 1; i < this.array.length; i++) {
      T current = this.get(i);
      if (current.compareTo(smallest) < 0) {
        smallest = current;
      }
    }
    return smallest;
  }
}
```

Wildcards: `A<?>`, `A<? extends T>`, `A<? super T>`

- **Raw types** are banned; instead, use wildcards

```java
new Comparable[10];   // avoid this
new Comparable<?>[10]; // good

a instanceof A<String> // doesn't work since type argument '
      String' isn't available during runtime due to erasure
a instanceof A        // avoid this
a instanceof A<?>     // good
```

- Make methods flexible: **Producer Extends, Consumer Super** (PECS)

  Producer: method produces instances of type `T` or its subtypes (producer is allowed to produce something more specific)

  Consumer: method accepts instances of type `T` or its supertypes (consumer is allowed to accept something more general)

Immutability

Immutable class: no visible changes outside abstraction barrier

- Make **fields `final`** to avoid re-assignment

  Remove any assignments to the fields (compilation error due to `final`)
- Make **class `final`** to disallow inheritance → avoid subclasses from overriding the methods
- Change **setter** from `void` to return a new instance → prevent mutating the current instance

Note: Within the scope of a class, you are able to access all private members of the class, including fields from other instances that are not `this`

Nested Classes

- **Static nested class**: only access static fields, methods of containing class
- **Inner class** (non-static): can access all fields, methods of containing class

  Qualified `this` reference: access instance fields in container class `<container>.this.<variable>`
- **Local class**: declared inside a method
- **Anonymous class**: `new X(arg) { body }`

```java
void sortNames(List<String> names) {
  names.sort(new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
      return s1.length() - s2.length();
    }
  }); // what is the name of this class?
}
```

Functional Interfaces

- Exactly one abstract method
- Annotation: `@FunctionalInterface`

```java
public interface BooleanCondition<T> { // predicate
  boolean test(T arg); // input T, output boolean
}
public interface Consumer<T> { // procedure
  void consume(T arg); // input T, output nothing
}
public interface Producer<T> { // constant
  T produce(); // input nothing, output T
}
public interface Transformer<T, R> { // unary function
  R transform(T arg); // input T, output R
}
public interface Combiner<T, S, R> { // binary function
  R combine(T arg1, S arg2); // input T, S, output R
}
```

Box

`Box<T>` is generic wrapper class used to store an item of any reference type.

- `of`: factory method
- `ofNullable`: behaves just like `of` if input is non-null, and returns an empty box if input is null
- `filter`: check if item passes/fails the test
- `map`: transform item

Maybe

`Maybe<T>` is an option type: a wrapper around a value that might be missing. In other words, it represents either some value (`Some`), or none (`None`.

- `of`, `some`, `none`: factory method
- `filter`: check if the value passes/fails the test
- `map`: transform the value
- `flatMap`: transform the value, then flatten
- `orElse`: store the given value if the value inside is missing
- `orElseGet`: produce the value if it is missing
- `ifPresent`: consume the value if it is present

* `get` is protected, so cannot use

Lazy

Lazy value: the expression that produces a lazy value is not evaluated until the value is needed. Lazy value is useful for cases where producing the value is expensive, but the value might not eventually be used.

- `of`: factory method, accepts a value or a producer that produces the value ONLY WHEN it is needed
- `filter`: lazily test if the value passes the test or not
- `map`:
- `flatMap`:
- `combine`: lazily combine two `Lazy` objects (which may contain values of different types), and return a new `Lazy` object
- `get`: if the value is already available, return it; otherwise, compute the value and return it

Lazy should not evaluate anything until `get()` is called

InfiniteList

`InfiniteList<T>` is constructed using a lazy evaluation. The list is a **recursive** structure, containing a head and a tail, with the tail being a list itself. Head and tail are lazy producers: store a `Producer`.

- `generate`: construct a new `InfiniteList` given a producer `[V, V, V, ...]`
- `iterate`: construct a new `InfiniteList` given a seed and transformer `[seed, next(seed), next(next(seed)), ...]`
- `sentinel`: create a `sentinel`
- `filter`: keep only elements that pass the test
- `map`: transform each element to a new element
- `flatMap`: transform each element to an `InfiniteList`, then flatten the resulting nested `InfiniteList`
- `takeWhile`: truncate the `InfiniteList` as soon as an element fails the test
- `append`: append the given infinite list to the end of this (finite) list
- `limit`: limit the list to `n` elements
- `head`, `tail`: retrieve head and tail lazily
- `count`: return number of elements
- `reduce`: use all the elements in the list, and return a single value

Piecewise functions: `if (a) { b } else { c }` is equivalent to

```java
Maybe.filter(a).map(b).orElse(c)
```

To get the n-th element:

```java
lst.limit(n).reduce(0, (x, y) -> y);
```

Functors

Functor: class that holds a value (no context)

- **Identity**: `fct.map(x -> x)` ≡ `fct`
- **Composition**: `fct.map(x -> f(x)).map(x -> g(x))` ≡ `fct.map(x -> g(f(x)))`

Monads

Monad: class that holds a value & context

- **Left identity**: `Monad.of(x).flatMap(x -> f(x))` ≡ `f(x)`
- **Right identity**: `mn.flatMap(x -> Monad.of(x))` ≡ `mn`
- **Associative**: `mn.flatMap(x -> f(x)).flatMap(y -> g(y))` ≡ `mn.flatMap(x -> f(x).flatMap(y -> g(y)))`

List and Stream

`List` API (Abridged)

- `of`: construct a `List` containing the specified elements
- `size`: return the number of elements
- `isEmpty`: check if list contains no elements
- `contains`: check if this list contains the specified element
- `add`: append the specified element to the end of the list
- `remove`: remove first occurrence of the specified element from the list
- `get`: return element at the specified position
- `set`: replace the element at the specified position with the given element
- `indexOf`: return index of the first occurrence of the given element
- `stream`: convert `List` to `Stream`

`Stream` API (Abridged)

- `of`: construct stream whose elements are the specified values
- `generate`: construct stream where each element is generated by the provided `Supplier`
- `iterate`: construct stream by iteratively applying `f` to initial element `seed` `[seed, f(seed), f(f(seed)), ...]`
- `filter`: return stream of elements that pass the predicate
- `map`: returns stream of results of applying the given function to elements
- `flatMap`: apply a one-to-many transformation to the elements of the stream, then flatten into a single stream
- `takeWhile`: return stream of elements that satisfy the predicate, terminating when predicate becomes false
- `dropWhile`: drop elements that satisfy the predicate, terminating when predicate becomes false, return resulting stream
- `forEach`: perform an action for every element
- `anyMatch`: true if at least one element satisfies the predicate
- `allMatch`: true if all elements satisfy the predicate
- `noneMatch`: true if no elements satisfy the predicate
- `peek`: return a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
- `limit`: truncate stream
- `distinct`: return distinct elements
- `sorted`: sort according to natural order, or the provided `Comparator`
- `count`: return number of elements
- `reduce`: perform reduction on elements, then return the reduced value

```java
T result = identity;
for (T element : this stream)
  result = accumulator.apply(result, element)
return result;
```

- `toList`: convert `Stream` to `List`
- `concat`: concatenate the two input streams
- `min`: return minimum element according to the provided `Comparator`

Parallel stream: `.parallel()`

`<U> reduce(U e, BiFunction<U, ? super T, U> f, BinaryOperator<U> g)`

- identity: starting value of each sub-stream
- accumulator: accumulator for each sub-stream
- combiner: combines the result of each sub-stream's accumulation in the encounter order of elements

If the following properties are satisfied, then the call to `reduce` is **parallelisable**. In other words, the sequential and parallel execution always produce the same result.

- `e` is identity
- `f` and `g` are pure functions (no side-effect and deterministic)
- `f` and `g` are **associative**:

  `g(g(x, y), z)` ≡ `g(x, g(y, z))`

  `f(f(x, y), z)` ≡ `f(x, f(y, z))`
- `f` and `g` are **compatible**:

  `g(x, f(e, y))` ≡ `f(x, y)`

Implementations

```java
public class Box<T> {
  private final T content;
  private static final Box<?> EMPTY_BOX = new Box<>(null);

  private Box(T content) {
    this.content = content;
  }

  public static <T> Box<T> of(T content) {
    if (content == null) {
      return null;
    }
    return new Box<>(content);
  }

  public static <T> Box<T> empty() { // empty box
    @SuppressWarnings("unchecked")
    final Box<T> result = (Box<T>) EMPTY_BOX;
    return result;
  }

  public boolean isPresent() {
    return (this.content != null);
  }

  public static <T> Box<T> ofNullable(T content) {
    if (!isPresent()) { return empty(); }
    return of(content);
  }

  public Box<T> filter(BooleanCondition<? super T> condition)
       {
    if (!isPresent() || !(condition.test(this.content))) {
        return empty(); }
    return this;
  }

  public <U> Box<U> map(Transformer<? super T, ? extends U>
       transformer) {
    if (!isPresent()) { return empty(); }
    return Box.ofNullable(transformer.transform(this.content))
  }
  }
}
```

```java
public abstract class Maybe<T> {
  private static final Maybe<?> NONE = new None();

  public static <T> Maybe<T> none() {
    @SuppressWarnings("unchecked")
    Maybe<T> temp = (Maybe<T>) NONE;
    return temp;
  }

  public static <T> Maybe<T> some(T t) {
    return new Some<T>(t);
  }

  public static <T> Maybe<T> of(T val) {
    if (val == null) { return none(); }
    return new Some<T>(val);
  }

  protected abstract T get();
  public abstract Maybe<T> filter(BooleanCondition<? super T>
       pred);
  public abstract <U> Maybe<U> map(Transformer<? super T, ?
       extends U> mapper);
  public abstract <U> Maybe<U> flatMap(Transformer<? super T,
       ? extends Maybe<? extends U>> mapper);
  public abstract T orElse(T obj);
  public abstract T orElseGet(Producer<? extends T> obj);
  public abstract void ifPresent(Consumer<? super T> obj);

  private static class None extends Maybe<Object> {
    @Override
    protected Object get() {
      throw new NoSuchElementException();
    }

    @Override
    public Maybe<Object> filter(BooleanCondition<Object> pred)
         {
      return none();
    }

    @Override
    public <U> Maybe<U> map(Transformer<Object, ? extends U>
         mapper) {
      return none();
    }

    @Override
    public <U> Maybe<U> flatMap(Transformer<Object, ? extends
         Maybe<? extends U>> mapper) {
      return none();
    }

    @Override
    public Object orElse(Object obj) {
      return obj;
    }

    @Override
    public Object orElseGet(Producer<?> obj) {
      return obj.produce();
    }

    @Override
    public void ifPresent(Consumer<Object> obj) {
      // do nothing
    }
  }

  private static final class Some<T> extends Maybe<T> {
    private final T val;

    private Some(T t) {
      this.val = t;
    }

    @Override
    protected T get() {
      return this.val;
    }

    @Override
    public Maybe<T> filter(BooleanCondition<? super T> pred) {
      if (this.val != null && !pred.test(this.val)) {
        return none();
      }
      return this;
    }

    @Override
    public <U> Maybe<U> map(Transformer<? super T, ? extends U
         > mapper) {
      return some(mapper.transform(this.val));
    }

    @Override
    public <U> Maybe<U> flatMap(Transformer<? super T, ?
         extends Maybe<? extends U>> mapper) {
      // transform can only convert a value into a subclass of
           Maybe<? extends U>. Safe to cast it to Maybe<U>.
      @SuppressWarnings("unchecked")
      Maybe<U> temp = (Maybe<U>) mapper.transform(this.val);
      return temp;
    }

    @Override
    public T orElse(T obj) {
      return this.val;
    }

    @Override
    public T orElseGet(Producer<? extends T> obj) {
      return this.val;
    }

    @Override
    public void ifPresent(Consumer<? super T> obj) {
      obj.consume(this.val);
    }
  }
}
```

```java
public class Lazy<T> {
  private Producer<? extends T> producer;
  private Maybe<T> value;

  private Lazy(T v) {
    this.value = Maybe.some(v);
  }
  private Lazy(Producer<? extends T> p) {
    this.producer = p;
    this.value = Maybe.none();
  }
  public static <T> Lazy<T> of(T v) {
    return new Lazy<>(v);
  }
  public static <T> Lazy<T> of(Producer<? extends T> s) {
    return new Lazy<>(s);
  }

  public Lazy<Boolean> filter(BooleanCondition<? super T> pred
       ) {
    Producer<Boolean> t = () -> pred.test(this.get());
    return Lazy.of(t);
  }

  public <U> Lazy<U> map(Transformer<? super T, ? extends U>
       mapper) {
    Producer<U> t = () -> mapper.transform(this.get());
    return Lazy.of(t);
  }

  public <U> Lazy<U> flatMap(Transformer<? super T, ? extends
       Lazy<? extends U>> mapper) {
    Producer<U> t = () -> mapper.transform(this.get()).get();
    return Lazy.of(t);
  }

  public <S, R> Lazy<R> combine(Lazy<S> obj, Combiner<? super
       T, ? super S, ? extends R> combiner) {
    return Lazy.of(() -> combiner.combine(
        this.get(), obj.get()));
  }

  public T get() {
    T obj = this.value.orElseGet(this.producer);
    this.value = Maybe.some(obj);
    return obj;
  }
}
```

```java
public class InfiniteList<T> {
  private Lazy<Maybe<T>> head;
  private Lazy<InfiniteList<T>> tail;

  private static final InfiniteList<?> SENTINEL = new Sentinel
       ();

  private static class Sentinel extends InfiniteList<Object> {
    private Sentinel() {
      super();
    }
    :
  }

  private InfiniteList() {
    this.head = null;
    this.tail = null;
  }

  private InfiniteList(Lazy<Maybe<T>> head, Lazy<InfiniteList<
       T>> tail) {
    this.head = head;
    this.tail = tail;
  }

  public static <T> InfiniteList<T> generate(Producer<T> prod)
       {
    return new InfiniteList<T>(
        Lazy.of(() -> Maybe.some(prod.produce())),
        Lazy.of(() -> InfiniteList.generate(prod)));
  }

  public static <T> InfiniteList<T> iterate(T seed,
       Transformer<? super T, ? extends T> next) {
    return new InfiniteList<T>(
        Lazy.of(Maybe.some(seed)),
        Lazy.of(() -> InfiniteList.iterate(next.transform(seed)
            , next)));
  }

  public static <T> InfiniteList<T> sentinel() {
    @SuppressWarnings("unchecked")
    InfiniteList<T> tmp = (InfiniteList<T>) SENTINEL;
    return tmp;
  }

  public T head() {
    return this.head.get()
        .orElseGet(() -> this.tail.get().head());
  }

  public InfiniteList<T> tail() {
    return this.head.get()
        .map(x -> this.tail.get())
        .orElseGet(() -> this.tail.get().tail());
  }

  public <R> InfiniteList<R> map(Transformer<? super T, ?
       extends R> func) {
    return new InfiniteList<R>(
        head.map(x -> x.map(func)),
        tail.map(l -> l.map(func)));
  }

  public InfiniteList<T> filter(BooleanCondition<? super T>
       pred) {
    return new InfiniteList<T>(
        head.map(x -> x.filter(pred)),
        tail.map(l -> l.filter(pred)));
  }

  public InfiniteList<T> limit(long n) {
    Lazy<InfiniteList<T>> lazyTail =
        Lazy.of(() -> this.head.get()
            .map(ignr -> this.tail.get().limit(n - 1))
            .orElseGet(() -> this.tail.get().limit(n)));
    // If head exists, decrement n, call limit on the tail.
    // Else don't decrement n, call limit on the tail.

    return Maybe.of(n)
        .filter(x -> x > 0)
        .map(ignr -> new InfiniteList<>(this.head, lazyTail))
        // If n > 0, construct a new InfiniteList with
        // this.head as the head, lazyTail as the tail
        .orElseGet(() -> sentinel());
        // Else return sentinel to mark end of the list.
  }

  public InfiniteList<T> takeWhile(BooleanCondition<? super T>
       predicate) {
    Lazy<Maybe<T>> lazyHead = Lazy.of(() -> this.head.get().
         filter(predicate));

    Lazy<InfiniteList<T>> lazyTail =
        Lazy.of(() -> this.head.get()
            .map(x -> lazyHead.get()
                .map(y -> this.tail.get().takeWhile(predicate)).
                    orElseGet(() -> sentinel()))
            .orElseGet(() -> this.tail.get().takeWhile(
                predicate)));

    return new InfiniteList<>(lazyHead, lazyTail);
  }

  public InfiniteList<T> append(InfiniteList<T> list) {
    return new InfiniteList<>(
        this.head,
        this.tail.map(tail -> tail.append(list)));
  }

  public <R> InfiniteList<R> flatMap(Transformer<? super T, ?
       extends InfiniteList<R>> mapper) {
    return this.head.get()
        .map(h -> mapper.transform(h)
        .append(this.tail.get().flatMap(mapper)))
        .orElseGet(() -> this.tail.get().flatMap(mapper));
  }

  public <U> U reduce(U identity, Combiner<U, ? super T, U>
       accumulator) {
    return this.tail.get().reduce(
      this.head.get().map(tail -> accumulator.combine(
          identity, tail)).orElse(identity),
      accumulator);
  }

  public long count() {
    return this.reduce(0, (x, y) -> x + 1);
  }
}
```