CS2030S Programming Methodology II (PE)

AY 24/25 Sem 1 — github/omgeta

1. PE1

Array<T>

```
class Array<T> {
  private T[] array;

  Array(int size) {
    // The only way we can put an object into the array is
   through
    // the method set() and we can only put an object of type
    T inside.
    // So it is type safe to cast ‘Object[]‘ to ‘T[]‘
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  public void copyFrom(Array<? extends T> src) {
    int len = Math.min(this.array.length, src.array.length);
    for (int i = 0; i < len; i++) {
      this.set(i, src.get(i));
    }
  }

  public void copyTo(Array<? super T> dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
      dest.set(i, this.get(i));
    }
  }
}
```

Implementing Comparable<T>

```
class Packet implements Comparable<Packet> {
  private String message;

  public Packet(String message) {
    this.message = message;
  }

  @Override
  public String toString() {
    return this.message;
  }

  public int compareTo(Packet other) {
    if (this.message.length() == other.message.length()) {
      return 0;
    } else if (this.message.length() < other.message.length()
   ) {
      return 1;
    } else {
      return -1;
    }
  }
}
```

Composing Comparable<T>

```
public class Buffer<T extends Comparable<T>> {
  private T[] messages;
  private int endIndex;

  public Buffer(int size) {
    // The only way to put an object into array is through
    // Buffer::send and we only put Object of type T inside.
    // Thus it is safe to cast ‘Object[]‘ to ‘T[]‘.
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Comparable<?>[size];
    this.messages = temp;
    this.endIndex = 0;
  }
}
```

2. PE2

Immutability

Checklist:

i. All fields are `final` (not necessary)
ii. All types in fields are immutable
iii. Arrays are copied before assignment
iv. No mutator (or return a new instance)
v. Class is `final`

Functional Interfaces

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

Equivalent Interfaces:

i. `BooleanCondition<T>::test` $\iff$ `Predicate<T>::test`
ii. `Producer<T>::produce` $\iff$ `Supplier<T>::get`
iii. `Consumer<T>::consume` $\iff$ `Consumer<T>::accept`
iv. `Transformer<T, R>::transform` $\iff$ `Function<T, R>::apply` / `UnaryOp<T>::apply`

Monads and Functors

Monad Laws:

i. `Monad.of(x).flatMap(x -> f(x))` $\equiv$ `f(x)` (Left Identity)
ii. `monad.flatMap(x -> Monad.of(x))` $\equiv$ `monad` (Right Identity)
iii. `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` $\equiv$ `monad.flatMap(x -> f(x).flatMap(y -> g(y)))` (Associative)

Functor Laws:

i. `functor.map(x -> x)` $\equiv$ `functor` (Identity)
ii. `functor.map(x -> f(x)).map(x -> g(x))` $\equiv$ `functor.map(x -> g(f(x))` (Composition)

Stream<T>

Creation:

i. `Stream::of(T...) : Stream<T>`
ii. `Stream::generate(Supplier<T>) : Stream<T>`
iii. `Stream::iterate(T, UnaryOp<T>) : Stream<T>`
iv. `Stream::iterate(T, Predicate<? super T>, UnaryOp<T>) : Stream<T>`
v. `List.stream() : Stream<T>`

Intermediate:

i. `filter(Predicate<? super T>)`
ii. `map(Function<? super T, ? extends R>)`
iii. `flatMap(Function<? super T, ? extends Stream<? extends R>>)`
iv. `takeWhile(Predicate<? super T>)`
v. `dropWhile(Predicate<? super T>)`
vi. `distinct()`
vii. `sorted()`
viii. `sorted(Comparator<? super T>)`
ix. `peek(Consumer<? super T>)`
x. `limit(long)`
xi. `skip(long)`

Terminal:

i. `anyMatch(Predicate<? super T>) : boolean`
ii. `allMatch(Predicate<? super T>) : boolean`
iii. `noneMatch(Predicate<? super T>) : boolean`
iv. `count() : long`
v. `findAny() : T`
vi. `findFirst() : Optional<T>`
vii. `forEach(Consumer<? super T>) : void`
viii. `forEachOrdered(Consumer<? super T>) : void`
ix. `min((x,y) -> x.compareTo(y)) : Optional<T>`
x. `max((x,y) -> x.compareTo(y)) : Optional<T>`
xi. `reduce(T, BinaryOperator<T>) : T`
xii. `reduce(U, BiFunction<U, ? super T, U>, BinaryOperator<U>) : U`
xiii. `toArray() : Object[]`
xiv. `toList() : List<T>`

Maybe<T>

Creation:

i. `Maybe::of(T) : Maybe<T>` (`Some<T>` if not `null` else `None<T>`)
ii. `Maybe::some() : Some<T>`
iii. `Maybe::none() : None<T>`

Intermediate:

i. `filter(BooleanCondition<? super T>)`
ii. `map(Transformer<? super T, ? extends R>)`
iii. `flatMap(Transformer<? super T, ? extends Maybe<? extends R>>)`

Terminal:

i. `orElse(Producer<? extends T>) : T`
ii. `ifPresent(Consumer<? super T>) : void`
iii. `toString() : String`
iv. `equals(Object) : boolean`

Lazy<T>

Creation:

i. `Lazy::of(T) : Lazy<T>` (pre-evaluated)
ii. `Lazy::of(Producer<? extends T>) : Lazy<T>`

Intermediate:

i. `filter(BooleanCondition<? super T>)`
ii. `map(Transformer<? super T, ? extends R>)`
iii. `flatMap(Transformer<? super T, ? extends Lazy<? extends R>>)`

Terminal:

i. `get() : T`
ii. `equals(Object) : boolean`

Parallelization

Parallelizing Streams:

i. `Collection::parallelStream()`
ii. `Stream::parallel()`

Conditions for Parallelization:

i. Non-interference with data source
ii. Avoid side-effects
iii. Stateless lambdas
iv. Prefer unordered (or use `.unordered()`)