Compte Rendu TP7

BOURENNANE Amine

Exercice 1

1.

```
public class TimeSeries<E> {
    private final List<Data<E>> datas = new ArrayList<>();
    public record Data<E>(long timestamp, E element) {
        public Data {
            Objects.requireNonNull(element);
        }
    }
}
```

2.

```
public void add(long timestamp, E element) {
    Objects.requireNonNull(element);
    if (datas.isEmpty() || timestamp >= datas.get(size() - 1).timestamp())
{
        datas.add(new Data<>(timestamp, element));
    }
    else {
        throw new IllegalStateException(timestamp + " need higher of
precedently added");
    }
}

public int size() {
    return datas.size();
}

public Data<E> get(int index) {
    var select = datas.get(index);
    return new Data<>(select.timestamp(), select.element());
}
```

3.

```
public class Index {
    private int[] index;

    private Index(int capacity) {
```

```
            if (capacity < 0)
                throw new IllegalStateException("capacity must positive");
            index = new int[capacity];
            for (var i = 0; i < capacity; i++) {
                index[i] = i;
            }
        }

        public int size() {
            return index.length;
        }
    }

    public Index index() {
        return new Index(size());
    }
```

4.

```
public String toString() {
    return Arrays.stream(index).mapToObj(TimeSeries.this::get)
            .map(d -> d.timestamp() + " | " + d.element())
            .collect(Collectors.joining("\n"));
}
```

5. On utilise un *Predicate* qui est une interface fonctionnelle qui possède une fonction qui renvoie un booléen et qui prend un ? qui prend tous les super type de E.

```
private Index(Predicate<? super E> filter, int capacity) {
    Objects.requireNonNull(filter);
    if (capacity < 0)
        throw new IllegalStateException("capacity must positive");
    index = IntStream.range(0, capacity).filter(i ->
filter.test(datas.get(i).element)).toArray();
}
public Index index() {
    return index(e -> true);
}

public Index index(Predicate<? super E> filter) {
    return new Index(filter, size());
}
```

6. Le type du paramètre doit être un *Consumer<? super Data<E>>*

```
public void forEach(Consumer<? super Data<E>> consumer) {
    Arrays.stream(index).forEach(i -> consumer.accept(datas.get(i)));
```

```
}
```

7. Elle doit implémanter l'interface *Iterable*, on doit implanter la méthode `iterator()` sont type de retour doit être *`Iterator<Data<E>>`*

```
public final class Index implements Iterable<Data<E>> {
    ...

    @Override
    public Iterator<Data<E>> iterator() {
        return new Iterator<>() {
            private int i;

            @Override
            public boolean hasNext() {
                return size() != i;
            }

            @Override
            public Data<E> next() {
                if (!hasNext()) {
                    throw new NoSuchElementException("no next");
                }
                return datas.get(index[i++]);
            }
        };
    }
}
```

8. Car elle ne sera plus trié et on risque d'avoir les même éléments.

```
public Index or(Index indexOth) {
    if (!dis.equals(indexOth.dis)) {
        throw new IllegalArgumentException("It's not the same
TimeSeries");
    }
    return new Index(IntStream.concat(Arrays.stream(index),
Arrays.stream(indexOth.index))
            .sorted()
            .distinct()
            .toArray());
}
```

9.

```
public Index and(Index indexOth) {
    if (!dis.equals(indexOth.dis)) {
```

```
            throw new IllegalArgumentException("It's not the same
TimeSeries");
        }
        HashSet<Integer> indexSet = Arrays.stream(indexOth.index).boxed()
            .collect(Collectors.toCollection(HashSet<Integer>::new));
        return new
Index(Arrays.stream(index).filter(indexSet::contains).toArray());
    }
```