TP9 : Structure de données persistante (fonctionnelle)

Exercice 1 - Seq

1.
```java
public class Seq<E> implements Iterable<E> {
    private final List<E> list;
    private Seq(List<E> list) {
        this.list = List.copyOf(list);
    }
    public static<T> Seq<T> from(List<? extends T> list) {
        Objects.requireNonNull(list);
        return new Seq<>(list);
    }
    public E get(int index) {
        Objects.checkIndex(index, list.size());
        return list.get(index);
    }
    public int size() {
        return list.size();
    }
}
```

2.
```java
@Override
public String toString() {
    return list.stream()
            .map(Object::toString)
            .collect(Collectors.joining(", ", "<", ">"));
}
```

3.
```java
@SafeVarargs
public static<T> Seq<T> of(T... elements) {
    return new Seq<>(List.of(elements));
}
```

4.
```java
public void forEach(Consumer<? super E> consumer) {
    Objects.requireNonNull(consumer);
    list.forEach(consumer);
}
```

5. La signature de la map doit être Seq ou T est propre à la méthode map. Les éléments de la liste doivent être des Object. La fonction stockée doit prendre un ? super E et renvoyer un ? extends T.

```java
public class Seq<E> implements Iterable<E> {
    private final List<Object> list;
    private final Function<? super Object, ? extends E> mapper;

    @SuppressWarnings("unchecked")
```

```
    private Seq(List<? extends E> list) {
        this(List.copyOf(list), o -> (E) o);
    }

    private Seq(List<Object> list, Function<? super Object, ? extends E>
mapper) {
        this.list = list;
        this.mapper = mapper;
    }

    public static<T> Seq<T> from(List<? extends T> list) {
        Objects.requireNonNull(list);
        return new Seq<>(list);
    }

    public E get(int index) {
        Objects.checkIndex(index, list.size());
        return mapper.apply(list.get(index));
    }

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

    @Override
    public String toString() {
        return list.stream()
            .map(mapper)
            .map(Object::toString)
            .collect(Collectors.joining(", ", "<", ">"));
    }

    @SafeVarargs
    public static<T> Seq<T> of(T... elements) {
        return new Seq<>(List.of(elements));
    }

    public void forEach(Consumer<? super E> consumer) {
        Objects.requireNonNull(consumer);
        list.forEach(o -> consumer.accept(mapper.apply(o)));
    }

    public<T> Seq<T> map(Function<? super E, ? extends T> fun) {
        Objects.requireNonNull(fun);
        return new Seq<T>(list, fun.compose(mapper));
    }
}
```

6.
```
public Optional<E> findFirst() {
    return list.stream()
            .<E>map(mapper)
```

```
            .findFirst();
}
```

7. Il faut implémenter l'interface Iterable avec le type paramétré E.

```
@Override
public Iterator<E> iterator() {
    return new Iterator<>() {
        private int i;
        @Override
        public boolean hasNext() {
            return i < list.size();
        }
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException("it has no next");
            }
            return mapper.apply(list.get(i++));
        }
    };
}
```

8.
```
@Override
public Spliterator<E> spliterator () {
    return new Spliterator<>() {
        private final Spliterator<Object> splitIterator = list.spliterator();

        @Override
        public boolean tryAdvance(Consumer<? super E> consumer) {
            Objects.requireNonNull(consumer);
            return splitIterator.tryAdvance(e ->
consumer.accept(mapper.apply(e)));
        }

        @SuppressWarnings("unchecked")
        @Override
        public Spliterator<E> trySplit() {
            return (Spliterator<E>) splitIterator.trySplit();
        }

        @Override
        public long estimateSize() {
            return splitIterator.estimateSize();
        }

        @Override
        public int characteristics() {
            return IMMUTABLE | NONNULL | ORDERED;
        }
    };
}
```

```java
public Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

Exercice 2 - Seq2 le retour (à la maison)

```java
public class Seq2<E> implements Iterable<E> {
    private final Object[] array;
    private final Function<? super Object, ? extends E> mapper;

    @SuppressWarnings("unchecked")
    private Seq2(Object[] array) {
        this(Arrays.copyOf(array, array.length), o -> (E) o);
    }

    private Seq2(Object[] array, Function<? super Object, ? extends E> mapper) {
        Objects.requireNonNull(array);
        Objects.requireNonNull(mapper);
        this.array = array;
        this.mapper = mapper;
    }

    public static<T> Seq2<T> from(List<? extends T> list) {
        Objects.requireNonNull(list);
        list.forEach(elem -> Objects.requireNonNull(elem));
        return new Seq2<>(list.toArray());
    }

    public E get(int index) {
        Objects.checkIndex(index, array.length);
        return mapper.apply(array[index]);
    }

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

    @Override
    public String toString() {
        return Arrays.stream(array)
                .map(mapper)
                .map(Object::toString)
                .collect(Collectors.joining(", ", "<", ">"));
    }

    @SafeVarargs
    public static<T> Seq2<T> of(T... elements) {
        Objects.requireNonNull(Arrays.stream(elements).findAny());
        return new Seq2<>(Arrays.copyOf(elements, elements.length));
    }
```

```
    public void forEach(Consumer<? super E> consumer) {
        Objects.requireNonNull(consumer);
        Arrays.stream(array).forEach(o -> consumer.accept(mapper.apply(o)));
    }

    public<T> Seq2<T> map(Function<? super E, ? extends T> fun) {
        Objects.requireNonNull(fun);
        return new Seq2<T>(array, fun.compose(mapper));
    }

    public Optional<E> findFirst() {
        return Arrays.stream(array)
                .<E>map(mapper)
                .findFirst();
    }

    @Override
    public Iterator<E> iterator() {
        return new Iterator<>() {
            private int i;
            @Override
            public boolean hasNext() {
                return i < array.length;
            }
            public E next() {
                if (!hasNext()) {
                    throw new NoSuchElementException("it has no next");
                }
                return mapper.apply(array[i++]);
            }
        };
    }

    public Spliterator<E> spliterator(int start, int end) {
        return new Spliterator<>() {
            private int i = start;

            @Override
            public boolean tryAdvance(Consumer<? super E> consumer) {
                Objects.requireNonNull(consumer);
                if (i < end) {
                    consumer.accept(get(i++));
                    return true;
                }
                return false;
            }

            @SuppressWarnings("unchecked")
            @Override
            public Spliterator<E> trySplit() {
                var middle = (i + end) >>> 1;
                if (middle == i) {
```

```
                    return null;
                }
                var spliterator = spliterator(i, middle);
                i = middle;
                return spliterator;
            }

            @Override
            public long estimateSize() {
                return end - i;
            }

            @Override
            public int characteristics() {
                return SIZED | IMMUTABLE | NONNULL | ORDERED;
            }
        };
    }

    public Stream<E> stream() {
        return StreamSupport.stream(spliterator(0, array.length), false);
    }
}
```

Steve Chen