**# TP10 : Interface, Vue et implantation de Stream

Exercice 1 - Reversible

1.
```java
public interface Reversible<E> {
int size();
E get(int index);

@SafeVarargs
static<T> Reversible<T> fromArray(T... elements) {
    Objects.requireNonNull(elements);
    Arrays.stream(element).forEach(Objects::requireNonNull);
    return new Reversible<>() {
        @Override
        public int size() {
            return elements.length;
        }

        @Override
        public T get(int index) {
            Objects.requireNonNull(elements[index]);
            return elements[index];
        }
    };
}
```

2.
```java
default Iterator<E> iterator() {
    return new Iterator<>() {
        private int index;

        @Override
        public boolean hasNext() {
            return index < size();
        }

        @Override
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return get(index++);
        }
    };
}
```

3.
```java
default Reversible<E> reversed() {
    return new Reversible<>() {
        @Override
        public int size() {
            return Reversible.this.size();
        }
```

```
        @Override
        public E get(int index) {
            return Reversible.this.get(size() - 1 - index);
        }
    };
}
```

4. Il nous suffit de rajouter une méthode reversed.

```
@Override
public Reversible<E> reversed() {
    return Reversible.this;
}
```

5.
```
public interface Reversible<E> extends Iterable<E> {
int size();
E get(int index);

@SafeVarargs
static<T> Reversible<T> fromArray(T... elements) {
    return fromList(Arrays.asList(elements));
}

static<T> Reversible<T> fromList(List<? extends T> list) {
    Objects.requireNonNull(list);
    list.forEach(Objects::requireNonNull);
    var size = list.size();
    return new Reversible<>() {
        @Override
        public int size() {
            return size;
        }

        /**
         * throws ISE if the size of the underlying data structure is
         * less than the size of the current Reversible.
         * @param index index
         * @return value from index
         */
        @Override
        public T get(int index) {
            Objects.checkIndex(index, size);
            if (list.size() < size) {
                throw new IllegalStateException();
            }
            Objects.requireNonNull(list.get(index));
            return list.get(index);
        }
    };
}
```

```
    default Iterator<E> iterator() {
        return new Iterator<>() {
            private int index;

            @Override
            public boolean hasNext() {
                return index < size();
            }

            @Override
            public E next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                try {
                    return get(index++);
                } catch (IllegalStateException e) {
                    throw new ConcurrentModificationException();
                }
            }
        };
    }

    default Reversible<E> reversed() {
        return new Reversible<>() {
            @Override
            public int size() {
                return Reversible.this.size();
            }

            @Override
            public E get(int index) {
                return Reversible.this.get(size() - 1 - index);
            }

            @Override
            public Reversible<E> reversed() {
                return Reversible.this;
            }
        };
    }
}
```

6.
```
    default 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) {
                    try {
```

```
                    consumer.accept(get(i++));
                } catch (IllegalStateException e) {
                    throw new ConcurrentModificationException();
                }
                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 | NONNULL | ORDERED;
        }
    };
}

default Stream<E> stream() {
    return StreamSupport.stream(spliterator(0, size()), false);
}
```

7.
```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(0, size()), true);
}
```

Exercice 2 - Reversible2

```
public abstract class Reversible2<E> extends AbstractList<E> implements Iterable<E>
{
   @SafeVarargs
   static<T> Reversible2<T> fromArray(T... elements) {
      return fromList(Arrays.asList(elements));
   }
   static<T> Reversible2<T> fromList(List<? extends T> list) {
      Objects.requireNonNull(list);
```

```
      list.forEach(Objects::requireNonNull);
      var size = list.size();
      return new Reversible2<>() {
         @Override
         public int size() {
            return size;
         }

         /**
          * throws ISE if the size of the underlying data structure is
          * less than the size of the current Reversible.
          * @param index index
          * @return value from index
          */
         @Override
         public T get(int index) {
            Objects.checkIndex(index, size);
            if (list.size() < size) {
               throw new IllegalStateException();
            }
            Objects.requireNonNull(list.get(index));
            return list.get(index);
         }
      };
   }
   public Iterator<E> iterator() {
      return new Iterator<>() {
         private int index;

         @Override
         public boolean hasNext() {
            return index < size();
         }

         @Override
         public E next() {
            if (!hasNext()) {
               throw new NoSuchElementException();
            }
            try {
               return get(index++);
            } catch (IllegalStateException e) {
               throw new ConcurrentModificationException();
            }
         }
      };
   }
   public Reversible2<E> reversed() {
      return new Reversible2<>() {

         @Override
         public int size() {
```

```
            return Reversible2.this.size();
        }

        @Override
        public E get(int index) {
            return Reversible2.this.get(size() - 1 - index);
        }

        @Override
        public Reversible2<E> reversed() {
            return Reversible2.this;
        }
    };
}
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) {
                try {
                    consumer.accept(get(i++));
                } catch (IllegalStateException e) {
                    throw new ConcurrentModificationException();
                }
                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 | NONNULL | ORDERED;
```

```
         }
      };
   }
   public Stream<E> stream() {
      return StreamSupport.stream(spliterator(0, size()), true);
   }
}
```

Steve Chen**