Libro

El objetivo es leer un libro de un fichero y hacer cálculos sobre las palabras que contiene, en que líneas aparecen, ordenarlas por frecuencias, etc. Se usará el fichero Quijote.txt para hacer los cálculos.

Se supone que las palabras están separadas una de otras por separadores que podemos definir. Los separadores que usaremos son `"[ ,;.\n()?¿!¡:\"]"` aunque podemos añadir o eliminar separadores.

Queremos implementar las siguientes funciones:

- `Integer numeroDeLineas(String file);`
- `Integer numeroDePalabrasDistintasNoHuecas(String file);`
- `Set<String> palabrasDistintasNoHuecas(String file);`
- `Double longitudMediaDeLineas(String file);`
- `Integer numeroDeLineasVacias(String file);`
- `String lineaMasLarga(String file);`
- `Integer primeraLineaConPalabra(String file, String palabra);`
- `String lineaNumero(String file,Integer n);`
- `SortedMap<String,Integer> frecuenciasDePalabras(String file);`

  Frecuencia de cada palabra. Ordenadas por palabras

- `SortedMap<Integer,Set<String>> palabrasPorFrecuencias(String file);`

  Palabras agrupadas por sus frecuencias de aparición. Ordenadas por frecuencias

- `SortedMap<String,Set<Integer>> lineasDePalabra(String file);`

  Grupos de líneas donde aparece cada palabra.

Notas:

Necesitamos emular los métodos zip y enumerate de Python

```
public class Enumerate<E> {

	public static <E> Enumerate<E> of(Integer num, E element) {
		return new Enumerate<E>(num, element);
	}

	public final Integer counter;
	public final E value;
	private Enumerate(Integer num, E element) {
		super();
		this.counter = num;
		this.value = element;
```

```
      }

      @Override
      public String toString() {
            return String.format("(%d,%s)",
                  counter,value.toString());
      }

}

public static <L, R, T> Stream<T> zip(
      Stream<L> leftStream,
      Stream<R> rightStream,
      BiFunction<L, R, T> combiner) {
        Spliterator<L> lefts = leftStream.spliterator();
        Spliterator<R> rights = rightStream.spliterator();
        return StreamSupport.stream(
            zip(lefts,rights,combiner),
            leftStream.isParallel() || rightStream.isParallel());
}

public static <E> Stream<Enumerate<E>> enumerate(
      Stream<E> stream, Integer start){
      Stream<Integer> st = Stream.iterate(start,e->e+1);
      return zip(st,stream,(n,e)->Enumerate.of(n, e));
}

public static <E> Stream<Enumerate<E>> enumerate(
      Stream<E> stream){
      return enumerate(stream,0);
}
```

```
public static <L, R, T> Spliterator<T> zip(
      Spliterator<L> lefts,
      Spliterator<R> rights,
      BiFunction<L, R, T> combiner) {
      return new AbstractSpliterator<T>(
        Long.min(lefts.estimateSize(),rights.estimateSize()),
        lefts.characteristics() & rights.characteristics()) {
        @Override
        public boolean tryAdvance(Consumer<? super T> action) {
            return lefts.tryAdvance(
                  left -> rights.tryAdvance(
                  right -> action.accept(
                        combiner.apply(left, right))));
        }
      };
}
```