ADSOF: Ordinaria - 28/05/2020

Ejercicio 4 (1.5 punto)

Queremos construir una clase genérica `BiStream` que permita operar sobre los elementos de dos `Streams` simultáneamente (avanzando uno a uno en los elementos de los dos streams y aplicando la operación). En particular, estamos interesados en crear un método `map`, que pueda aplicar a cada par de elementos de los streams una función de dos argumentos y produzca un nuevo `Stream` con el resultado de la misma. El tipo de los elementos del `Stream` resultante vendrá determinado por el resultado de la función que aplica `map`. Ten en cuenta que los streams pueden tener distinta longitud, en cuyo caso `map` genera un `Stream` cuya longitud es la del `Stream` más corto.

Se pide implementar la clase `BiStream` para que el siguiente programa produzca la salida indicada más abajo. Ten en cuenta que:

1. Un `BiStream` debe poder construirse a partir de dos colecciones cualesquiera, por ejemplo un conjunto y una lista.
2. Debes procurar la máxima generalidad de tu clase, permitiendo funciones con tipos compatibles de argumentos y valor de retorno, como puedes observar en la segunda invocación a `map` del listado de más abajo.

Se valorará especialmente el uso de principios de orientación a objetos en el diseño, así como su generalidad, reusabilidad, extensibilidad y la concisión y claridad del código.

```
public class BiStreams {
  public static void main(String[] args) {
    List<String> aList   = Arrays.asList("a",  "list", "of", "words");
    Set<String>  aSet    = new LinkedHashSet<>(Arrays.asList("another", "collection", "with", "more", "Strings"));

    BiStream<String, String> bstr = new BiStream<>(aList, aSet);  // creamos un BiStream a partir de una lista y un conjunto

    System.out.println(bstr.map((x, y) -> x+" - "+y).             // aplicamos la operación map a cada par de strings
                            collect(Collectors.toList()));

    // Aquí creamos un BiStream con una lista de Strings, y otra de Integer, con las longitudes de los strings
    // La función del map está definida sobre pares de objetos, y debe poder ser aplicable a String e Integers
    System.out.println(new BiStream<String, Integer>(aList,
                                                     aList.stream().map(x -> x.length()).collect(Collectors.toList())).
                       map((Object x, Object y) -> x.toString()+" - "+y).collect(Collectors.toList()));
  }
}
```

Salida esperada:

```
[a - another, list - collection, of - with, words - more]
[a - 1, list - 4, of - 2, words - 5]
```

SOLUCIÓN Y PUNTUACIÓN, Ejercicio 4, Continua, 28 Mayo 2020, 1,5 puntos.

El reparto de puntos se refleja con la siguiente notación:

`[n] = valor aproximado sobre 30, a dividir por 20 para 1,5 puntos`

Además de las puntuaciones `[n]` asignadas a cada parte de la solución, se aplican penalizaciones por defectos relativos **principios de orientación a objetos en el diseño, así como su generalidad, reusabilidad, extensibilidad y la concisión y claridad,** como por ejemplo, código repetido innecesariamente, copiado innecesario de estructuras de datos, atributos no privados sin justificación válida, soluciones innecesariamente más complejas … y **especialmente falta de generalidad en los parámetros genéricos.**

```
import java.util.function.*;
import java.util.stream.*;
import java.util.*;

class BiStream<T1, T2> { [2]
      private Stream<? extends T1> str1; // Collection <? extends T1>
      private Stream<? extends T2> str2; // Collection <? extends T2>   [2]
                                                                  [4]
      public BiStream(Collection<? extends T1> str1, Collection<? extends T2> str2){
            this.str1 = str1.stream();
            this.str2 = str2.stream();   [1]
      }
                  [2]                          [6]
      public <R> Stream<R> map(BiFunction<? super T1, ? super T2, ? extends R> fun) {
            Collection<R> res = new LinkedList<R>();
            // this.parallelFoldWith((T1 x, T2 y) -> res.add(fun.apply(x, y)));
            Iterator<? extends T1> it1 = str1.iterator();
            Iterator<? extends T2> it2 = str2.iterator();
            while (it1.hasNext() && it2.hasNext())          [4]
              [4] res.add(  fun.apply(it1.next(), it2.next()) ); [4]
            return res.stream();   [1]
      }

      private void parallelFoldWith(BiConsumer<T1, T2> oper) {
            Iterator<? extends T1> it1 = str1.iterator();
            Iterator<? extends T2> it2 = str2.iterator();
            while (it1.hasNext() && it2.hasNext())
                        oper.accept(it1.next(), it2.next());
      }
}
```