Notas sobre Collections y Stream

Miguel Toro

Contenido

Colecciones y Diccionarios

```
List<E> ls = new ArrayList<>();
List<E> lsc = new ArrayList<>(Collection<E> ls);  //copy
List<E> ls = Arrays.asList(e1,e2,…);
Set<E> s = new HashSet<>();
Set<E> sc = new HashSet<>(Collection<E> s); //copy
SortedSet<E> ss = new TreeSet<>(Comparator<E> cmp);
SortedSet<E> ssc = new TreeSet<>(SortedSet<E> ss); //copy
```

El tipo *Collection<E>* tiene como subtipos *List<E>*, *Set<E>, SortedSet<E>* con los métodos comunes:

```
boolean isEmpty();
boolean contains(E e);
int size();
boolean add(E e);
boolean remove(E e);
boolean containsAll(Collection<E> c2); this ⊂ c2
boolean addAll(Collection<E> c2);      this = this ∪ c2
boolean removeAll(Collection<E> c2);   this = this − c2
boolean retainAll(Collection<E> c2);   this = this ∩ c2
```

Ordenar listas:

```
Collections.sort(list,comparator);
```

Un diccionario se representa por el tipo *Map<K,V>*

```
Map<K,V> m = new HashMap<>();
Map<K,V> mc = new HashMap<>(Map<K,V> m);
boolean isEmpty();
boolean size();
boolean containsKey(K key);
Set<K> keyset();
Set<Map.Entry<K,V>> entrySet();
V get(K key);
V put(K key, V value);
```

```
Map.Entry<K,V>
      K getKey()
      V getValue()
```

Interfaces funcionales, lambda expresiones y tipos funcionales

Interface funcional

```
interface Comparator<E> {
```

```
        int compare(E e1, E e2);
}
```

Comparator<E>

int compare(E e1, E e2)

Tipo funcional: $E \times E \rightarrow int$

Lambda expresión: (e1,e2)->e1.getRadio().compareTo(e2-getRadio());

```
Comparator<Circulo2D> cmp =
      (e1,e2)->e1.getRadio().compareTo(e2-getRadio());
```

```
Comparator<E extends Comparable<E>> c =
       Comparator.naturalOrder();
Comparator<E extends Comparable<E>> c =
       Comparator.naturalOrder().reversed();
Comparator<E extends Comparable<E>> c =
       Comparator.naturalOrder();
Comparator<E> c = Comparator.comparing(Function<E,R> f);
```

Comparator<E>

int compare(E e1, E e2)

Tipo funcional: $E \times E \rightarrow int$

Comparator<Circulo2D> cmp = (e1,e2)->e1.getRadio().compareTo(e2-getRadio());

Predicate<E>

bool test(E e);

Tipo funcional: $E \rightarrow boolean$

Predicate<Integer> p = e->e%2==0;

Function<E,R>

R apply(E e);

Tipo funcional: $E \rightarrow R$

*Function<Integer,Integer> f = e->e*e;*

UnaryOperator<E>

E apply(E e);

Tipo funcional: $E \rightarrow E$

*UnaryOperator<Integer> f = e->e*e;*

BinaryOperator<E>

E apply(E e1, E e2);

Tipo funcional: $E \times E \rightarrow E$

BinaryOperator<Integer> sum = (e1,e2)->e1+e2;

Consumer<E>

void accept(E e);

Tipo funcional: $E \rightarrow void$

Consumer<Punto2D> cs = p->System.out.println(p.toString());

Supplier<E>

void accept(E e);

Tipo funcional: $() \rightarrow E$

Supplier<Punto2D> sp = ()-> Punto2D.of(8.9,5.6);

Stream

Factoría

IntStream IntStream.range(a,b);

LongStream LongStream.range(a,b);

IntStream string.chars();

Stream<E> collection.stream();

Stream<E> Arrays.stream(E[] elements);

Stream<E> Stream.iterate(E initialValue, Predicate<E> hashNext, UnaryOperator<E> next);

Stream<String> Files.lines(Path file);

```
class FileTools {
…
public static Stream<String> fromFile(String file) {
      Stream<String> r = null;
      try {
            r = Files.lines(Paths.get(file),
                        Charset.defaultCharset());
      } catch (IOException e) {
            throw new IllegalArgumentException(
                  "No se ha encontrado el fichero " + file);
      }
      return r;
}
```

Funciones de transformación

Stream<R> map(Function<E,R> f);

Stream<R> flatMap(Function<E,Stream<R>> f);

Stream<E> filter(Predicate<E> p);

Stream<E> distinct();

Stream<E> sorted(Comparator<E> cmp);

Acumuladores:

Long count()

Devuelve el número de elementos del stream que invoca.

Double/Integer/Long sum();

Sólo para streams numéricos (IntStream, LongStream, DoubleStream), devuelve la suma

Double average();

Sólo para streams numéricos (IntStream, LongStream, DoubleStream), devuelve la media de los valores del stream que invoca.

Boolean allMatch(Predicate<E> p);

Boolean anyMatch(Predicate<E> p);

Optional<E> filter(p).findFirst();

Optional<E> min(Comparator<E>);

Optional<E> max(Comparator<E>);

Optional<E> reduce(BinaryOperator<E> bo);

List<E> collect(Collectors.toList());

Set<E> collect(Collectors.toSet());

String collect(Collectors.joining(sp,p,f)); // aplicado a Stream<String>

Map<K,List<E>> collect(Collectors.groupingBy(Function<E,K) key);

Map<K,Set<E>> collect(Collectors.groupingBy(Function<E,K) key,
Collectors.toSet()));

SortedMap<K,Set<E>> collect(Collectors.groupingBy(
Function<E,K) key,
()->new TreeMap<>(Comparator.reverseOrder()),
Collectors.toSet()));

Map<K,Long> collect(Collectors.groupingBy(Function<E,K) key,
Collectors.counting ()));

Map<K,Integer> collect(Collectors.groupingBy(Function<E,K) key,
Collectors.collectingAndThen(
Collectors.counting (),
Long::intValue)));

Map<K, Double > collect(Collectors.groupingBy(Function<E,K) key,
Collectors.summingDouble(Function<E,Double) f)));

Map<K, List<T>> collect(Collectors.groupingBy(Function<E,K) key,
Collectors.mapping(Function<E,T) f,
Collectors.toList())));

Acciones

void forEach(Consumer<E> c);

Optional<E>

E get();

E orElse(E value);

Métodos de factoría y operaciones adicionales de streams

<L,R,T> Stream<T> zip(Stream<L> left, Stream<R> right, BiFunction<L, R, T> combiner);

Similar al zip de Python

<E> Stream<Enumerate<E>> enumerate(Stream<E> stream, Integer start);

Similar al enumerate de Python

<E> Stream<Enumerate<E>> enumerate(Stream<E> stream);

Similar al enumerate de Python

Tipo Enumerate<E>:

Propiedades

- *Counter: Integer*
- *Value: E*

Código en Java sin streams

Factoría

IntStream IntStream.range(a,b);

```
for(Integer i=a, i<b; i++) {
      ….
}
```

IntStream string.chars();

```
for(Integer i=a, i< string.length();i++) {
      Character c = string.charAt(i);
      ….
}
```

Stream<E> collection.stream();

```
for(E e: collection){
      ….
}
```

Stream<E> Arrays.stream(E[] elements);

```
for(Integer i=a; i< elements.length; i++) {
      E e = elements[i];
      ….
}
```

Stream<E> Stream.iterate(E initialValue, Predicate<E> hasNext, UnaryOperator<E> next);

```
for(E e=initialValue; hasNext(e);e = next(e)) {
      ….
}
```

Stream<String> Files.lines(Path file);

```
List<String> lineas(String file){
	List<String> lineas = null;
	try {
		BufferedReader bufferedReader =
			new BufferedReader(new FileReader(file));
		lineas = bufferedReader.lines()
			.collect(Collectors.toList());
		bufferedReader.close();
	} catch (IOException e) {
		System.out.println(e.toString());
	}
	return lineas;
}


List<String> lineas = File
for(E String: lineas) {
	….
}
```

Funciones de transformación

Stream<R> map(Function<E,R> f);

```
for(E e: collection){
	R r = f(e);
	…
}
```

Stream<R> flatMap(Function<E,Stream<R>> f);

```
for(E e: collection){
	for(R r: f(e)){
		…
	}
}
```

Stream<E> filter(Predicate<E> p);

```
for(E e: collection){
	if(p(e)){
		…
	}
}
```

Acumuladores:

Long count()

```
Integer a = 0;
for(E e: collection){
      a = a+1;
}
```

Double/Integer/Long sum();

```
Integer a = 0;
for(E e: collection){
      a = a+e;
}
```

Double average();

```
Double a = 0;
Integer n = 0;
for(E e: collection){
      a = a+e;
      n = n+1;
}
return a/n;
```

Boolean allMatch(Predicate<E> p);

```
Boolean a = true;
for(E e: collection){
      a = p(e);
      if(!a) break;
}
return a;
```

Boolean anyMatch(Predicate<E> p);

```
Boolean a = false;
for(E e: collection){
      a = p(e);
      if(a) break;
```

```
}
return a;
```

Optional<E> filter(p).findFirst();

```
E a = null;
for(E e: collection){
      if(p(e)) {
            a = e;
            break;
      }
}
return Optional.ofNullable(a);
```

Optional<E> min(Comparator<E>); Optional<E> max(Comparator<E>);

```
E a = null;
for(E e: collection){
      if(a == null || e < a) {   // con el comparador
            a = e;
      }
}
return Optional.ofNullable(a);
```

Optional<E> max(Comparator<E>); Optional<E> max(Comparator<E>);

```
E a = null;
for(E e: collection){
      if(a == null || e > a) {   // con el comparador
            a = e;
      }
}
return Optional.ofNullable(a);
```

Optional<E> reduce(BinaryOperator<E> bo);

```
E a = null;
for(E e: collection){
      if(a == null) {
            a = e;
      } else {
            a = bo(a,e);
      }
```

```
}
return Optional.ofNullable(a);
```

List<E> collect(Collectors.toList());

```
List<E> a = new ArrayList<>();
for(E e: collection){
      a.add(e);
}
return a;
```

Set<E> collect(Collectors.toSet());

```
Set<E> a = new HashSet<>();
for(E e: collection){
      a.add(e);
}
return a;
```

String collect(Collectors.joining(sp,pf,sf)); // aplicado a Stream<String>

```
String a = pf;
Boolean primero = true;
for(E e: collection){
      if(primero){
            a = a+e;
            primero = false;
      } else {
            a = a +sp+e;
      }
}
return a+sf;
```

Map<K,List<E>> collect(Collectors.groupingBy(Function<E,K) fkey);

```
Map<K,List<E>> a = new HashMap<>();
for(E e: collection){
      K key = fkey(e);
      if(!a.keySet().contains(key)){
            a.put(key, new ArrayList<>());
      }
      m.get(key).add(e);
}
return a;
```

Map<K,Set<E>> collect(Collectors.groupingBy(Function<E,K) key, Collectors.toSet()));

```
Map<K,Set<E>> a = new HashMap<>();
for(E e: collection){
      K key = fkey(e);
      if(!a.keySet().contains(key)){
            a.put(key, new HashSet<>());
      }
      a.get(key).add(e);
}
return a;
```

SortedMap<K,Set<E>> collect(Collectors.groupingBy(
Function<E,K) key,
()->new TreeMap<>(Comparator.reverseOrder()),
Collectors.toSet()));

```
SortedMap<K,Set<E>> a = new TreeMap<>(Comparator.reverseOrder());
for(E e: collection){
      K key = fkey(e);
      if(!a.keySet().contains(key)){
            a.put(key, new HashSet<>());
      }
      a.get(key).add(e);
}
return a;
```

Map<K,Long> collect(Collectors.groupingBy(Function<E,K) key,
Collectors.counting ()));

```
Map<K,Long> a = new HashMap<<();
for(E e: collection){
      K key = fkey(e);
      Long r = 1L+ a.getOrDefault(key,0L);
      a.put(key,r);
}
return a;
```

Map<K,Integer> collect(Collectors.groupingBy(Function<E,K) key,
Collectors.collectingAndThen(
Collectors.counting (),
Long::intValue)));

```
Map<K,Integer> a = new HashMap<<();
for(E e: collection){
      K key = fkey(e);
      Integer r = 1+ a.getOrDefault(key,0);
```

```
      a.put(key,r);
}
return a;
```

Map<K, Double > collect(Collectors.groupingBy(Function<E,K) fkey,
Collectors.summingDouble(Function<E,Double) fsum)));

```
Map<K,Double> a = new HashMap<<();
for(E e: collection){
      K key = fkey(e);
      Integer r = fsum(e)+a.getOrDefault(key,0);
      a.put(key,r);
}
return a;
```

Map<K, List<T>> collect(Collectors.groupingBy(Function<E,K) fkey,
Collectors.mapping(Function<E,T) fm, Collectors.toList()));

```
Map<K,List<R>> a = new HashMap<<();
for(E e: collection){
      K key = fkey(e);
      R r = fm(e);
      if(!a.keySet().contains(key)){
            a.put(key, new ArrayList<>());
      }
      a.get(key).add(r);
}
return a;
```

Acciones

void forEach(Consumer<E> c);

```
for(E e: collection){
      c.accept(e);
}
```

Métodos de factoría y operaciones adicionales de streams

<L,R,T> Stream<T> zip(Stream<L> left, Stream<R> right, BiFunction<L, R, T> combiner);

```
List<L> ls1 = …
List<R> ls2 = …
Stream<L> s1 = ls1.stream();
Stream<R> s2 = ls2.stream();
Stream<T> se = zip(s1,s2,combiner);

      Equivalente a:

Integer counter = start;
for(Integer i = 0; i< min(ls1.size,ls2.size(); i++){
      L e1 = ls1.get(i);
      R e2 = ls2.get(i);
      T e = combiner(e1,e2;
      …
}
```

<E> Stream<Enumerate<E>> enumerate(Stream<E> stream, Integer start);

```
Stream<E> st = collection.stream();
Stream<Enumerate<E>> se = enumerate(st,start);

      Equivalente a:

Integer counter = start;
for(E e:collection){
      Enumerate<E> en = Enumerate.of(counter,e);
      counter = counter+1;
      …
}
```