Introducción a la Programación

Tema 5. Lambda Expresiones y Tipos Funcionales. El tipo Stream

1. Introducción a los interfaces funcionales

En temas anteriores ya hemos visto el uso del tipo *Comparator<T>*. Este tipo es la abstracción de un orden. Como recordamos

```
package java.util;

public interface Comparator<T>{
      int compare(T o1, T o2);
}
```

Los interfaces como el anterior los denominaremos **interfaces funcionales**. Un interfaz funcional es aquél que sólo tiene un método. Posteriormente generalizaremos esta definición pero por ahora vemos que todo interfaz funcional, que puede ser genérico, tiene un solo método. Este método tiene una cabecera con un nombre, un conjunto de parámetros formales y un tipo del resultado. Si ignoramos el nombre del método y el nombre de los parámetros formales, el resto de la información de la cabecera constituye el **tipo funcional** asociado al *interfaz funcional*. Una forma de escribirlos es $\mathrm{T} \times \mathrm{T} \to \mathrm{int}$. Java en su versión 8 permite definir *interfaces funcionales* pero no tipos funcionales directamente.

Como hemos visto en capítulos anteriores al declarar un interfaz (funcional o no) estamos definiendo un nuevo tipo que podemos usar para declarar nuevos tipos de entidades. Los *interfaces funcionales* tienen asociado un *tipo funcional* y, por lo visto anteriormente, pueden existir muchos interfaces funcionales con el mismo tipo funcional.

Por ejemplo los dos interfaces funcionales *Predicate1* y *Predicate2* tienen el mismo tipo funcional $\mathrm{T} \to \mathrm{boolean}$.

```
interface Predicate1<T> {
       boolean apply(T input);
}
interface Predicate2<T> {
       boolean test(T input);
}
```

De forma similar un método *static* tomado de forma aislada tiene asociado un tipo funcional.

Como hemos visto en capítulos anteriores cuando declaramos una entidad de un tipo dado podemos inicializarla con una expresión que construya y devuelva una entidad de ese tipo: una expresión *new Constructor(..)* o un método de una factoría.

Las entidades que tienen asociados tipos funcionales pueden ser inicializadas mediante **referencias a métodos** o **expresiones lambda** con el tipo funcional adecuado. Veamos con un poco de detalle estas ideas. Este tipo de objetos actúan como una función sobre un conjunto de parámetros para producir un resultado.

Referencias a Métodos

En Java hay distintos tipos de métodos (static o no, constructores o simples métodos). Para referirnos a un método concreto usamos la notación $\boldsymbol{T :: m}$**.** Dónde **T** es el tipo del objeto y **m** el nombre del método. Las combinaciones posibles son:

Tipo de Método	*Ejemplo*
Referencia a un método *static*	*ContainingClass::staticMethodName*
Referencia a un método de instancia	*ContainingObject::instanceMethodName*
Referencia a un método de instancia de un objeto concreto de un tipo dado.	*ContainingType::methodName*
Referencia a un constructor	*ClassName::new*

Cada método tiene asociado un tipo funcional. Veamos algunos ejemplos. Sea el método de instancia *concat* del tipo *String*. Podemos declarar el tipo funcional *StringPlus,* inicializarlo con una referencia el método *concat* y usarlo posteriormente como se ve en el ejemplo siguiente. Podemos observar como un método de instancia tiene asociado un tipo funcional cuyo primer parámetro es el tipo del *this*. El tipo funcional de *concat* es $\boldsymbol{String \times String \rightarrow String}$

```
public String concat(String str);

public interface StringPlus {
     String plus(String s1, String s2);
}
```

```
public static void main(String[] args) {
          StringPlus ss = String::concat;
          System.out.println(ss.plus("hola","juan"));
}
```

Igualmente podemos inicializar un *Comparator<String>* haciendo referencia al método de instancia *compareToIgnoreCase.*

```
public int compareToIgnoreCase(String str);

Comparator<String> cc = String::compareToIgnoreCase;
```

Por último podemos ver la forma de referirnos a un constructor usan do *new*. Si hay varios constructores se escogerá el que ofrezca el tipo funcional compatible con la entidad a inicializar.

```
public interface ConstruyeSet<T> {
     SortedSet<T> nuevo(Comparator<T> c);
}

public static void main(String[] args) {
          Comparator<String> cc = String::compareToIgnoreCase;
          ConstruyeSet<String> cs = TreeSet::new;
          Set<String> st = cs.nuevo(cc);
          …
}
```

Expresiones Lambda

Una **expresión lambda** está formada por una serie de parámetros formales y un cuerpo que es una expresión construida sobre los parámetros formales. La forma general es

```
(T1 x1, T2 x2,…) -> { cuerpo }
```

Donde el cuerpo es una expresión o una secuencia de sentencias acabadas en un *return expresión.* Si **R** es el tipo de la expresión que forma el cuerpo, o de la expresión tras el *return*) entonces el tipo funcional de la expresión lambda es $\boldsymbol{T1 \times T2 \times \ldots \rightarrow R}$**.**

En muchos casos pueden obviarse los tipos de los parámetros formales si pueden ser deducidos por el compilador. Si sólo hay un parámetro formal entonces pueden quitarse los paréntesis y s el cuerpo es una expresión se pueden quitar las llaves. Una expresión lambda puede ser asignada a un objeto con el mismo tipo funcional.
Veamos el siguiente ejemplo dónde el tipo funcional del la expresión lambda y del interfaz funcional *IntegerBinary* es $\boldsymbol{Integer \times Integer \rightarrow Integer}$**.**

```
public interface IntegerBinary {
      Integer op(Integer e1, Integer e2);
}

IntegerBinary p = (x1, x2) -> x1+x2*x2;
System.out.println(p.op(2,3));
```

En los cuerpos de las expresiones lambda se puede hacer referencia a variables accesibles en el ámbito correspondiente pero siempre que estén declaradas *final* o sean efectivamente final. Por ejemplo.

```
Integer r = 5;
IntegerBinary p = (x1, x2) -> x1+r*x2*x2;
System.out.println(p.op(2,3));
```

Una variable es efectivamente final cuando su valor inicial no cambia. Este es el caso si se ha declarado final pero también si no se ha declarado pero su valor inicial no cambia. La interpretación de efectivamente final es que la identidad de la variable no cambie, si es de un tipo mutable, o el valor no cambie, si es de un tipo inmutable. O dicho de otra forma que noa haya una asignación a esa variable después de la incialización.

```
MutableBigInteger m = new MutableBigInteger(3L);
m.add(new BigInteger("4"));
Binary<BigInteger> bo = (BigInteger x, BigInteger y) ->
       {m.add(x.add(y)); return m.getValue();};
System.out.println(m.getValue()+"------"+bo.op(
        new BigInteger("7"), new BigInteger("10")));
```

Expresiones Lambda, métodos e interfaces funcionales

Ya en capítulos anteriores vimos los criterios para asignar una entidad de un tipo T a otra de tipo R. Esto se podía hacer si T es un subtipo de R. Esto se sigue aplicando objetos declarados de tipos funcionales.

Por otra parte una expresión lambda o la referencia a un método pueden ser asignadas a una entidad que ha sido declarada mediante un interfaz funcional si tienen el mismo tipo funcional o compatible.

Los mismos criterios que usamos para la asignación se usan para usar parámetros reales, en forma de expresiones lambda o referencias a métodos, en el lugar dónde se ha declarado un parámetro formal mediante un interface funcional.

Métodos default y definición de interfaz funcional

En Java 8 se permite que un interface pueda tener métodos con cuerpo. Estos métodos van precedidos par la palabra reservada **default**.

```
public interface IntegerBinary {
      Integer op(Integer e1, Integer e2);
      default Integer sum(Integer e1, Integer e2){
            return e1+e2;
      }
}
```

Como una clase puede implementar varios interfaces este mecanismo permite que la clase pueda heredar código de varios interfaces. Ahora una clase que implementa un interface sólo está obligada a implementar los métodos no declarados *default* o redefinir el cuerpo de los declarados *default*.

Con esta visión más general un interfaz es un interfaz funcional si tiene un solo método que no está etiquetado *default* aunque pueda tener otros métodos *default* y también otros métodos etiquetados con *static.*

2. Catálogo de interfaces funcionales

En la literatura se han ido consolidando nombres para algunos tipos interfaces funcionales. En la nueva versión se Java se proponen un conjunto de ellos que ya podemos considerar estandarizados. Por cada interfaces funcionales daremos presentaremos su interfaz, el conjunto de métodos default, su tipo funcional sus propiedades y en algunos casos factorías para los mismos.

Comparator<T>

Este tipo es adecuado para representar órdenes totales sobre los elementos de un tipo *T*. Su tipo funcional es $\boldsymbol{T \times T \rightarrow int}$**.** El interfaz funcional tiene el método compare y otros métodos *default* que veremos más adelante.

```
package java.util;

public interface Comparator<T>{
      int compare(T o1, T o2);
      …
}
```

El método compare devuelve un entero negativo, cero o positivo según que el primer argumento sea menor, igual o mayor que el segundo. Para describir sus propiedades usaremos las función *sgn(r),* la función signo, que devuelve -1, 0, 1 según r sea negativo, cero o positivo.

Los requisitos son:

- `sgn(compare(x, x)) == 0`.
- `sgn(compare(x, y)) == -sgn(compare(y, x))` para todo x e y. Es decir la relación definida es antisimétrica.
- `((compare(x, y)>0) && (compare(y, z)>0))` implica `compare(x, z)>0`. Para todo x, y, z. Es decir la relación definida es transitiva.
- `compare(x, y)==0` implica `sgn(compare(x, z)) == sgn(compare(y, z))` para todo x, y, `z`.
- Si se cumple que `(compare(x, y)==0) == (x.equals(y))` entonces decimos que el orden definido es consistente con la igualdad.

Cada relación de orden total define una relación de equivalencia. Según esta relación de equivalencia dos elementos son equivalentes si: `compare(x, y)) == 0`. Si el orden definido es consistente con la igualdad esta relación de equivalencia es la misma que la definida por la igualdad del tipo.

Métodos de factoría

- *static <T extends Comparable<? super T>> Comparator<T> naturalOrder()* Devuelve un *comparator* que usa el orden natural del tipo. El *comparator* devuelto dispara *NullPointerException* cuando se compara un objeto con null.
- *static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator).* Devuelve un orden que considera el valor null como menor que cualquier otro valor no null. Si ambos elementos a comparar son null los considera iguales y si son distintos de null usa el orden dado como parámetro.
- *static <T> Comparator<T> nullsLast(Comparator<? super T> comparator).* Devuelve un orden que considera el valor null como mayor que cualquier otro valor no null. Si ambos elementos a comparar son null los considera iguales y si son distintos de null usa el orden dado como parámetro.
- *static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor).* Compara dos elementos según el orden natural de los valores obtenidos al aplicarles la función que se pasa como parámetro.
- *static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)*. Compara dos elementos según el orden que se pasa como parámetro usado sobre los valores obtenidos al aplicarles la función que se pasa como parámetro.

Métodos defaults

- *default Comparator<T> reversed().* Devuelve un orden inverso al original.
- *default Comparator<T> thenComparing(Comparator<? super T> other).* Construye un orden compuesto. Primero compara los elementos con el orden original. Si se obtiene cero se usa other para determinar el resultado.

- *default <U extends Comparable<? Super U>> Comparator<T> thenComparing (Function<? super T,? extends U> keyExtractor).* Su implementación es equivalente a *thenComparing(comparing(keyExtractor)).*
- *default <U extends Comparable<? superU>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator).*
  Su implementación es equivalente a *thenComparing(comparing(keyExtractor, cmp)).*

Notas

Como hemos visto un orden total induce una relación de equivalencia. En algunos agregados de datos, como *Set<T>*, todos los elementos del agregado tienen que ser distintos. Pero más que distintos son no equivalentes con respecto a una relación de equivalencia. Si el conjunto se crea de la forma:

```
@SafeVarargs
public static <T> SortedSet<T> sortedSet(Comparator<T> cmp, T...
args) {
       SortedSet<T> result = new TreeSet<>(cmp);
       result.addAll(Arrays.asList(args));
       return result;
}

public static void main(String[] args) {
     Comparator<Integer> cmp = (x,y) -> x%2 - y%2;
     Set<Integer> s = sortedSet(cmp,5,2,0,4,7,9,11);
     s.stream().forEach(x-> System.out.println(x));
}
```

Si ejecutamos el método *main* anterior podemos observar que el resultado es 2, 5. El orden definido hace que cualquier entero impar sea mayor que otro par y que dos pares o dos impares sean equivalentes. Añadido un par y un impar el conjunto ordenado no acepta más elementos.

Ejemplos

Predicate<T>

Este tipo es adecuado para representar predicados sobre los elementos de un tipo *T*. Su tipo funcional es $\boldsymbol{T \rightarrow boolean}$. El interfaz funcional tiene el método test y otros métodos *default* que vemos abajo.

```
package java.util.function;

public interface Predicate<T>{
```

```
      boolean test(T o);
      default Predicate<T> and(Predicate<? super T> p)
      default Predicate<T> negate()
      default Predicate<T> or(Predicate<? super T> p)
}
```

Los métodos default combinan predicados con los operadores lógicos *not*, *and* y *or*. Los predicados resultantes se evalúan usando las propiedades de cortocircuito de los respectivos operadores. Es decir si queremos evaluar *p1 || p2* y p1 es true ese es el resultado y no se evalúa p2.

Para los tipos *int*, *double*, *long* existen las especializaciones *IntPredicate* ($\boldsymbol{int} \rightarrow \boldsymbol{boolean}$), *DoublePredicate* ($\boldsymbol{double} \rightarrow \boldsymbol{boolean}$), *LongPredicate* ($\boldsymbol{long} \rightarrow \boldsymbol{boolean}$).

Ejemplos

BiPredicate<T,U>

Este tipo es adecuado para representar predicados sobre los elementos de un tipo *T*. Su tipo funcional es $\boldsymbol{T \times U \rightarrow boolean}$. El interfaz funcional tiene el método test y otros métodos *default* que vemos abajo.

Es la versión de *Predicate<T>* con aridad 2.

```
package java.util.function;

public interface BiPredicate<T,U>{
      boolean test(T o1, U o2);
      default BiPredicate<T,U> and(
                  BiPredicate<? super T, ? super U > p)
      default BiPredicate<T,U> negate()
      default BiPredicate<T,U> or(
                  BiPredicate<? super T, ? super U > p)
}
```

Los métodos default combinan predicados con los operadores lógicos *not*, *and* y *or*. Los predicados resultantes se evalúan usando las propiedades de cortocircuito de los respectivos operadores. Es decir si queremos evaluar *p1 || p2* y p1 es true ese es el resultado y no se evalúa p2.

Para los tipos *int*, *double*, *long* existen las especializaciones *IntBiPredicate* ($\boldsymbol{int} \times \boldsymbol{int} \rightarrow \boldsymbol{boolean}$), *DoubleBiPredicate* ($\boldsymbol{double} \times \boldsymbol{double} \rightarrow \boldsymbol{boolean}$), *LongBiPredicate* ($\boldsymbol{long} \times \boldsymbol{long} \rightarrow \boldsymbol{boolean}$).

Ejemplos

Function<T,R>, UnaryOperator<T> y sus especializaciones

Este tipo es adecuado para representar funciones sobre los elementos de un tipo *T* que devuelven valores del tipo **R**. Su tipo funcional es $\boldsymbol{T \rightarrow R}$**.** El interfaz funcional tiene el método *apply* y otros métodos *default* que vemos abajo. Un subtipo importante de **Function<T,R>** es **UnaryOperator<T>** cuyo tipo funcional es $\boldsymbol{T \rightarrow T}$**.**

```
package java.util.function;

public interface Function<T,R>{
      R apply(T o);
      default <V> Function<T,V> andThen(
                  Function<? super R, ? super V> after)
      default <V> Function<V,R> compose(
                  Function<? super V, ? super T> before)
}
public interface UnaryOperator<T> extends Function<T,T> {}
```

El método *f1.andThen(f2)* construye una nueva función que aplica sucesivamente f1 y luego f2 a los resultados obtenidos. El método *f1.compose(f2)* construye una nueva función que aplica sucesivamente f2 y luego f1 a los resultados obtenidos.

Para los tipos *int*, *double*, *long* existen especializaciones. Para cada una de ellas indicamos su tipo funcional. Estas especializaciones son: *IntFunction (*$\boldsymbol{int \rightarrow R}$*), DoubleFunction (*$\boldsymbol{double \rightarrow R}$*), LongFunction (*$\boldsymbol{long \rightarrow R}$*), IntToDoubleFunction (*$\boldsymbol{int \rightarrow double}$*), IntToLongFunction (*$\boldsymbol{int \rightarrow long}$*), LongToDoubleFunction (*$\boldsymbol{long \rightarrow double}$*), LongToIntFunction (*$\boldsymbol{long \rightarrow int}$*), IntUnaryOperator (*$\boldsymbol{int \rightarrow int}$*), DoubleUnaryOperator (*$\boldsymbol{double \rightarrow double}$*), LongUnaryOperator (*$\boldsymbol{long \rightarrow long}$*), ToIntFunction (*$\boldsymbol{T \rightarrow int}$*), ToDoubleFunction (*$\boldsymbol{T \rightarrow double}$*), ToLongFunction (*$\boldsymbol{T \rightarrow long}$*).*

Ejemplos

BiFunction<T,U,R>

Es la versión de aridad 2 del tipo *Function<T,R> con tipo funcional* $\boldsymbol{T \times U \rightarrow R}$. Un subtipo importante de **BiFunction<T,U,R>** es **BinaryOperator<T>** cuyo tipo funcional es $\boldsymbol{T \times T \rightarrow T}$**.**

```
package java.util.function;

public interface BiFunction<T,U,R>{
      R apply(T o1, U o2);
```

```
      default <V> BiFunction<T,U,V> andThen(
                  Function<? super R, ? super V> after)
}
interface BinaryOperator<T> extends BiFunction<T,T,T> {
      static<T> BinaryOperator<T> minBy(Comparator<? super T>c);
      static<T> BinaryOperator<T> maxBy(Comparator<? super T>c);
}
```

Los métodos *minBy* y *maxBy* obtienen un operador binario capaz de calcular el mínimo o el máximo de dos valores.

Para los tipos *int*, *double*, *long* existen especializaciones. Para cada una de ellas indicamos su tipo funcional. Estas especializaciones son: *IntBinaryOperator* ($\boldsymbol{int} \times \boldsymbol{int} \rightarrow \boldsymbol{int}$), *DoubleBinaryOperator* ($\boldsymbol{double} \times \boldsymbol{double} \rightarrow \boldsymbol{double}$), *LongBinaryOperator* ($\boldsymbol{long} \times \boldsymbol{long} \rightarrow \boldsymbol{long}$), *ToIntBiFunction* ($\boldsymbol{T} \times \boldsymbol{U} \rightarrow \boldsymbol{int}$), *ToDoubleBiFunction* ($\boldsymbol{T} \times \boldsymbol{U} \rightarrow \boldsymbol{double}$), *ToLongBiFunction* ($\boldsymbol{T} \times \boldsymbol{U} \rightarrow \boldsymbol{long}$).

Ejemplos

Consumer<T>, BiConsumer<T,U>

Consumer<T> es el tipo adecuado para representar operaciones que aceptan un elemento y no producen ningún resultado aunque si producen efectos laterales. Su tipo funcional es $\boldsymbol{T \rightarrow void}$. El interfaz funcional tiene el método *accept* y otros métodos *default* que vemos abajo. El tipo relacionado con aridad 2 es **BiConsumer<T,U>** cuyo tipo funcional es $\boldsymbol{T \times U \rightarrow void}$.

```
package java.util.function;

public interface Consumer<T>{
      void accept(T t);
      default Consumer<T> chain(Consumer<? super T> other);
}
public interface BiConsumer<T,U>{
      void accept(T t, U u);
      default BiConsumer<T,U> chain(
             BiConsumer<? super T, <? super U> other);
}
```

El método *chain* encadena dos consumidores de elementos. El consumidor resultante hace, consecutivamente las operaciones asociadas a cada consumidor.

Para los tipos *int*, *double*, *long* existen especializaciones. Para cada una de ellas indicamos su tipo funcional. Estas especializaciones son: *IntConsumer* ($\boldsymbol{int} \rightarrow \boldsymbol{void}$), *DoubleConsumer* ($\boldsymbol{double} \rightarrow \boldsymbol{void}$), *LongConsumer* ($\boldsymbol{long} \rightarrow \boldsymbol{void}$),

Ejemplo

Supplier<T>

Supplier<T> es el tipo adecuado para representar operaciones que producen un elemento como las factorías. Su tipo funcional es $() \rightarrow \boldsymbol{T}$**.** El interfaz funcional tiene el método *get*.

```
package java.util.function;

public interface Supplier<T>{
      T get();
}
```

Para los tipos *int*, *double*, *long* existen especializaciones. Para cada una de ellas indicamos su tipo funcional. Estas especializaciones son: *IntSupplier* ($() \rightarrow \boldsymbol{int}$), *DoubleSupplier* ($() \rightarrow \boldsymbol{double}$), *LongSupplier* ($() \rightarrow \boldsymbol{long}$).

Ejemplos

Runnable

Es el tipo adecuado para representar actividades que sin tomar parámetros ni devolver resultados producen efectos laterales. Su tipo funcional es $() \rightarrow \boldsymbol{void}$**.** El interfaz funcional tiene el método *run*.

```
package java.lang;

public interface Runnable {
      void run();
}
```

Ejemplos

3. El tipo Stream<T>: definición y operaciones terminales

Un objeto de tipo **Stream<T>** es un agregado de elementos que soporta operaciones secuenciales y en paralelo. Podemos traducirlo como *flujo de datos* pero mantendremos en la

mayoría de los casos mantendremos el término en inglés. En general podemos considerarlo como un agregado virtual de objetos en el sentido de que todos los objetos del *Stream* no tienen que estar a la vez en memoria o formando algún tipo de estructura de datos. Podemos pensar un *Stream* como un agregado de datos obtenidos de alguna fuente real o virtual. Los agregados típicos, *Set<T>, List<T> ,…,* se pueden convertir en *Stream* pero también podemos construir *Stream* formadas, por ejemplo, por los números primos menores que uno dado sin hacer una lista intermedia. Es importante aclarar que, por las razones anteriores, un Stream puede ser infinito. Por ejemplo al secuencia de los número primos. Veamos, en primer lugar, algunas propiedades de este tipo de objetos y los términos.

En general las operaciones sobre *Stream* serán *funcionales*. Es decir producirán un resultado sin modificar el *Stream* de partida. Las operaciones sobre estos flujos de datos las dividiremos en dos tipos: *operaciones terminales e intermedias*. Las operaciones intermedias producen, a partir de uno dado, un nuevo *Stream*. Las operaciones terminales producen entidades que no son de tipo *Stream*. A su vez las operaciones intermedias las dividiremos en sin estado y con estado. Las primeras no mantienen ningún valor acumulado de los elementos ya procesados. Las segundas si lo mantienen. A las operaciones intermedias se les exige una implementación *lazy*. Los detalles asociados a este concepto los veremos en capítulos siguientes. Pero en esencia lo que queremos decir es que la implementación debe retardar (*lazy*) el cálculo del elemento proporcionado por el nuevo *Stream* hasta que su valor sea estrictamente para evitar operaciones innecesarias. El concepto opuesto a *lazy* es *eager*. Estos conceptos los iremos aclarando con ejemplos. Otra característica de algunas operaciones es que pueden ser de *cortocircuito* en un sentido similar a la evaluación de cortocircuito de los operadores lógicos *and* y *or*.

Los *Stream* pueden ser secuenciales o paralelos. Esto será definido en el momento de la creación del *Stream*. Un *Stream* secuencial procesa sus elementos uno tras otro. Un *Stream* paralelo puede dividirse en *subStream*, obtener los resultados para cada uno de ellos en posiblemente diferentes procesadores y posteriormente combinar dichos resultados. En este capítulo veremos las operaciones a un nivel de abstracción adecuado para hacerlas independientes del carácter secuencial o paralelo del *Stream* correspondiente.

Otro atributo importante de un *Stream* es si está especificado el orden con el que se proporcionan los elementos del flujo de datos. En este tipo de *Stream* los elementos siempre aparecen en el mismo orden. Esto no quiere decir que los elementos están ordenados con respecto a algún orden. Los flujos de datos, dependiendo la fuente de donde toman los datos, pueden tener otras propiedades que dependen de las características de la fuente de datos. Estas propiedades son:

- ORDERED: Si el orden en que se encuentran los elementos está especificado
- DISTINCT: Los elementos del flujo de datos son distintos entre sí.
- SORTED: Los elementos del flujos del flujo de datos están ordenados según un orden definido por un *Comparator*.
- SIZED: Si el flujo de datos tiene esta característica es posible conocer a priori el número de sus elementos.

- NONNULL: Los elementos del flujo de datos no pueden ser *null*.
- IMMUTABLE: No se pueden añadir o eliminar elementos del flujo de datos. Es decir no se puede modificar la fuente de datos asociada al flujo.
- CONCURRENT: La fuente de datos puede ser modificada de forma concurrente. Es decir se pueden añadir o eliminar, de forma concurrente, elementos del flujo de datos
- SUBSIZED: Una característica que indica que si un flujo se divide en dos cada uno de ellos será SIZED y SUBSIZED. Es decir cuando dividimos el flujo de datos en partes es posible conocer a priori el tamaño de cada una de ellas.

Las anteriores son propiedades de los flujos de datos asociados a la fuente de los datos. Un flujo de datos puede ser además, según la forma en que se cree, *secuencial o paralelo*.

Veamos ahora las operaciones terminales de un flujo de datos. Estas son operaciones que acumulan los valores de los elementos en un *Stream* en un valor resultado. Operaciones típicas de este tipo son el cálculo de máximo, mínimo, número de elementos, comprobar si todos los elementos cumplen una propiedad, si existe alguno que la cumpla, etc. De estas operaciones hay algunas que aplicadas sobre determinados flujos de datos no devuelven ningún valor. Para tener en cuenta esa posibilidad usamos el tipo *Optional<T>*. Este tipo puede contener un valor o no. Si contiene un valor el método isPresent() devolverá verdadero y podremos obtener el valor mediante el método *get().*

Otra utilidad interesante son las clases *IntSummaryStatistics, LongSummaryStatistics y DoubleSummaryStatistics* que son adecuadas para el cálculo de estadísticos elementales sobre los tipos de datos básicos.

```
package java.util;

public class Optional<T>{
      public T get();
      public boolean isPresent();
      public static <T> Optional<T> of(T value);
      ...
}
public class IntSummaryStatistics implements IntConsumer {
      public void accept(int value);
      public final long getCount();
      public final long getSum();
      public final double getAverage();
      public final int getMin();
      public final int getMax();
      ...
}
public class LongSummaryStatistics implements LongConsumer {
      ...
}
public class DoubleSummaryStatistics implements DoubleConsumer {
```

```
      ...
}
```

Las operaciones finales que soporta el tipo *Stream* son las que se muestran abajo. También se incluyen especializaciones de *Stream* para cada uno de los tipos básicos junto con algunas operaciones específicas de cada una de ellas.

```
package java.util.stream;

public interface Stream<T>{
      T reduce(T identity,BinaryOperator<T> accumulator);
      Optional<T> reduce(BinaryOperator<T> accumulator);
      <U> U reduce(U identity,
                   BiFunction<U,? super T,U> accumulator,
                   BinaryOperator<U> combiner);
      <R> R collect(Supplier<R> resultFactory,
                    BiConsumer<R,? super T> accumulator,
                    BiConsumer<R,R> combiner);
      <R,A> R collect(Collector<? super T,A,R> collector);
      boolean allMatch(Predicate<? super T> predicate);
      boolean anyMatch(Predicate<? super T> predicate);
      boolean noneMatch(Predicate<? super T> predicate);
      Optional<T> min(Comparator<? super T> comparator);
      Optional<T> max(Comparator<? super T> comparator);
      long count();
      Optional<T> findFirst();
      Optional<T> findAny();
      void forEach(Consumer<? super T> action);
      void forEachOrdered(Consumer<? super T> action);
      ...
}
public interface IntStream{
      ...
      Stream<Integer> boxed();
      OptionalDouble average();
      IntSummaryStatistics summaryStatistics();
      int[] toArray();
      int sum() ;
}
public interface LongStream{
      ...
      Stream<Long> boxed();
      OptionalDouble average();
      LongSummaryStatistics summaryStatistics();
      long[] toArray();
      long sum() ;
```

```
}
public interface DoubleStream{
      ...
      Stream<Double> boxed();
      OptionalDouble average();
      DoubleSummaryStatistics summaryStatistics();
      double[] toArray();
      double sum() ;
}
```

Veamos en primer ligar las operaciones que devuelven un tipo *boolean.* Son

- *boolean allMatch(Predicate<? super T> predicate):* El resultado es verdadero si todos los elementos del flujo de datos cumplen el predicado.
- *boolean anyMatch(Predicate<? super T> predicate):* El resultado el verdadero si alguno de los elementos del flujo de datos cumplen el predicado.
- *boolean noneMatch(Predicate<? super T> predicate):* El resultado es verdadero si ninguno de los elementos del flujo de entrada cumplen el predicado.

En el ejemplo siguiente la variable b tomará el valor *true.*

```
List<Integer> s = DslAggregates.list(5,2,4,0,7,9,11);
boolean b = s.stream().allMatch(x->x>=0);
```

Los siguientes métodos obtienen el máximo, el mínimo, el número de elementos y devuelven un elemento del flujo de datos.

- *Optional<T> min(Comparator<? super T> comparator):* Obtiene el valor mínimo de los valores en el flujo de datos. Si el flujo es vacío no existe valor.
- Optional<T> max(Comparator<? super T> comparator): Obtiene el valor máximo de los valores en el flujo de datos. Si el flujo es vacío no existe valor.
- *long count():* Cuenta el número de elementos del flujo de datos.
- *Optional<T> findFirst():* Devuelve el primer elemento del flujo de datos.
- *Optional<T> findAny():* Devuelve un elemento cualquiera del flujo de datos.
- *IntSummaryStatistics summaryStatistics():* Obtiene un resumen estadístico del flujo de entrada. Este método está disponible para las especializaciones sobre los tipos básicos del tipo *Stream (IntStream, LongStream, DoubleStream).*

Veamos el ejemplo siguiente:

```
List<Integer> s = DslAggregates.list(34,52,52,3,-35,46,0,67);
Integer a =s.stream()
      .min(Comparator.<Integer>naturalOrder())
```

```
      .get();
Long b = s.stream()
      .count();
Integer c = s.stream()
      .findFirst()
      .get();
IntSummaryStatistics sm = s.stream()
      .mapToInt(x->x)
      .summaryStatistics();
System.out.println("Min="+a+", Count="+b+",
      Primero="+c+", Cualquiera="+d);
System.out.println(sm);
```

```
Min=-35, Count=8, Primero=34
IntSummaryStatistics{count=8, sum=219,
       min=-35, average=27.375000, max=67}
```

Si la fuente de datos es un conjunto se obtienen resultados diferentes.

```
Set<Integer> s = DslAggregates.set(34,52,52,3,-35,46,0,67);
Integer a = s.stream()
      .min(Comparator.<Integer>naturalOrder()).get();
Long b = s.stream()
      .count();
Integer c = s.stream()
      .findFirst()
      .get();
IntSummaryStatistics sm = s.stream()
      .mapToInt(x->x)
      .summaryStatistics();
System.out.println("Min="+a+", Count="+b+", Primero="+c);
System.out.println(sm);
```

```
Min=-35, Count=7, Primero=3, Cualquiera=3
IntSummaryStatistics{count=7, sum=167, min=-35,
      average=23.857143, max=67}
```

Por último el método *forEach* es adecuado para realizar una acción que tomando como entrada cada uno de los elementos del flujo produzca algún efecto lateral como por ejemplo imprimirlos en la consola.

- *void forEach(Consumer<? super T> action);*

Veamos el ejemplo siguiente:

```
Set<Integer> s = DslAggregates.sortedSet(
     Comparator.<Integer>naturalOrder(),34,52,52,3,-35,46,0,67);
s.stream().forEach(x->System.out.print(x+","));
```

Cuyo resultado es:

```
-35,0,3,34,46,52,67,
```

Las operaciones anteriores del tipo *Stream* (o de sus especializaciones) podemos considerarlas como casos particulares de la *operación de reducción*. Esta operación consiste en inicializar un acumulador, recorrer los elementos del flujo de datos, combinar sucesivamente los valores del flujo con los del acumulador y finalmente devolver el valor acumulado. Hay dos tipos básicos de operaciones de reducción: **reducción inmutable** y **reducción mutable**. En el primer caso el acumulador es de un tipo inmutable y se lleva a cabo mediante el método reduce cuya signatura en el caso más general es:

```
<U> U reduce(U identity,
                  BiFunction<U,? super T,U> accumulator,
                  BinaryOperator<U> combiner);
```

Si el flujo de datos es secuencial y asumimos que se puede recorrer con un *for* entonces la implementación del método es equivalente a:

```
U result = identity;
for (T element : toIterable(stream))
         result = accumulator.apply(result, element);
return result;
```

Pero la operación de reducción se podría realizar en paralelo. Para ello se proporciona el operador binario *combiner*. Este operador es el encargado de combinar los valores acumulados de cada una de las partes en que se ha divido el flujo de datos. La ventaja del método *reduce* frente a la implementación secuencial anterior es que el método reduce puede usar el paralelismo posible.

```
Set<Integer> s = DslAggregates.sortedSet(
     Comparator.<Integer>naturalOrder(),34,52,52,3,-35,46,0,67);
Double a = s.s.parallelStream()
     .reduce(0.,(x,y)->x+y*y,(x,y)->x+y);
System.out.println(a);
```

Con la reducción anterior llevamos a cabo la suma de los cuadrados de los elementos del flujo. Vemos que el acumulador es de tipo *Double*, la función acumuladora tiene tipo funcional $\boldsymbol{Double \times Integer \rightarrow Double}$ y una función lambda asociada *(x,y)->x+y*y*. Es decir al acumulador de tipo *Double* se suma el cuadrado del elemento del flujo de satos que va

llegando. El operador binario para combinar los resultados de las partes en que se ha podido dividir el flujo de datos tiene el tipo funcional $\boldsymbol{Double} \times \boldsymbol{Double} \rightarrow \boldsymbol{Double}$ y la función lambda *(x,y) ->x+y.*

Si los tipos *T* y *U* son los mismos y el operador unario es asociativo existen formas más reducidas del método reduce tal como se puede ver en los métodos del tipo *Stream<T>* arriba. Más adelante veremos formas más sencillas de obtener resultados similares.

La ***reducción mutable*** se lleva a cabo sobre un acumulador de un tipo mutable y se implementa mediante el método *collect* cuya signatura en el caso más general es:

```
<R,A> R collect(Collector<? super T,A,R> collector);
```

Siendo un *Collector* (recolector) de la forma :

```
public interface Collector<T,A,R>{
      BiConsumer<A,T> accumulator();
      BinaryOperator<A> combiner();
      Function<A,R> transformer();
      Supplier<A> supplier();
      ...
}
```

Donde:

- T - es el tipo de los elementos del flujo de entrada
- A - es el tipo de acumulador mutable
- R – es el tipo del resultado de la operación de reducción.

La implementación del método *collect* usando un *Collector* es de la forma:

```
R container = collector.supplier().get();
     for (T t : toIterable(stream))
         collector.accumulator().accept(container, t);
     return collector.transformer().apply(container);
```

La implementación anterior es adecuada si el flujo de datos es secuencial y asumimos que se puede recorrer con un *for* pero la operación de reducción se podría realizar en paralelo. Para ello se proporciona el consumidor binario *combiner*. Este es el encargado de combinar los valores acumulados de cada una de las partes en que se ha divido el flujo de datos. Como en el caso de la reducción inmutable la ventaja del método *collect* frente a la implementación secuencial anterior es la posible paralelización de los cálculos en el caso del método *collect*.

Como ejemplo veamos una implementación de un recolector (*Collector*) con el objetivo de contar cuantos elementos distintos hay en un flujo de datos. Para ello se escoge como acumulador un *Set<T>* y como tipo del resultado *Integer.*

```
public class SetCollector<T> implements
              Collector<T, Set<T>, Integer> {
      public Supplier<Set<T>> supplier() {
            return ()-> new HashSet<T>();
      }
      public BiConsumer<Set<T>, T> accumulator() {
            return (x,y) -> x.add(y);
      }
      public BinaryOperator<Set<T>> combiner() {
            return (x,y)->{x.addAll(y); return x;};
      }
      public Function<Set<T>, Integer> transformer() {
            return (Set<T> x)->x.size();
      }
      public Set< Collector.Characteristics> characteristics() {
            return EnumSet.of(
                     Collector.Characteristics.UNORDERED);
      }
}
```

Ahora podemos aplicar una reducción mutable para obtener número de elementos distintos en un flujo dado.

```
List<Integer> s = DslAggregates.list(
       34,52,52,3,-35,46,0,52,67,52,-35);
Integer n = s.parallelStream().collect(
      new SetCollector<Integer>());
System.out.println(n);
```

Los objetos de tipo *Collector* son muy reutilizables. Más adelante veremos una factoría de los mismos.

4. El tipo Stream<T>: operaciones intermedias

Las operaciones intermedias construyen un nuevo flujo a partir otro. Son operaciones funcionales a las que se les exige una implementación *lazy*. Veamos las que tenemos disponibles

```
package java.util.stream;

public interface Stream<T>{
```

```
      Stream<T> filter(Predicate<? super T> predicate);
      <R> Stream<R> map(Function<? super T,? extends R> mapper);
      IntStream mapToInt(ToIntFunction<? super T> mapper);
      LongStream mapToLong(ToLongFunction<? super T> mapper);
      DoubleStream mapToDouble(
           ToDoubleFunction<? super T> mapper);
      <R> Stream<R> flatMap(
           Function<? super T,
               ? extends Stream<? extends R>> mapper);
      IntStream flatMapToInt(Function<? super T,? extends
      IntStream> mapper);
      LongStream flatMapToLong(Function<? super T,? extends
           LongStream> mapper)
      DoubleStream flatMapToDouble(Function<? super T,
           ? extends DoubleStream> mapper);
      Stream<T> distinct();
      Stream<T> sorted();
      Stream<T> sorted(Comparator<? super T> comparator);
      Stream<T> peek(Consumer<? super T> consumer);
      Stream<T> limit(long maxSize);
      Stream<T> substream(long startInclusive);
      Stream<T> substream(long startInclusive,
                    long endExclusive);
      ...
]
```

Veamos una descripción de las operaciones intermedias más relevantes.

- *Stream<T> filter(Predicate<? super T> predicate):* Se devuelve un nuevo flujo formado por todos aquellos elementos que satisfacen el predicado.
- *<R> Stream<R> map(Function<? super T,? extends R> mapper):* Se devuelve un nuevo flujo formado por los elementos obtenidos al aplicar la función *mapper* a cada uno de los elementos del flujo original.
- *<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper):* Se devuelve un nuevo flujo formado por la concatenación de los flujos obtenidos al aplicar la función *mapper* a cada uno de los elementos del flujo original.
- *Stream<T> distinct():* Se devuelve un nuevo flujo formado por todos los elementos distintos del flujo orioginal.
- *Stream<T> sorted(Comparator<? super T> comparator):* Se devuelve un nuevo flujo formado por mismos elementos del flujo original pero ordenados.
- *Stream<T> peek(Consumer<? super T> consumer):* Se devuelve el mismo flujo original y se aplica la acción correspondiente (consumer) a cada uno de ellos.
- Stream*<T> substream(long startInclusive, long endExclusive):* Se devuelve el flujo formado por los elementos que van de la posición *startInclusive* hasta la *endExclusive* sin incluir la misma.
- Stream<T> limit(long maxSize): Se devuelve un flujo del tamaño máximo indicado.

Veremos ejemplos más adelante.

5. El tipo Stream<T>: métodos de factoría

Existen mecanismos en Java para crear flujos de datos a partir de las fuentes de datos más usuales. Por una parte todos los tipos derivados de *Collection<T>* ofrecen un método para crear objetos de tipo *Stream<T>* secuenciales o paralelos. Los arrays a través de la clase *Arrays* también pueden ser convertidos en flujos de datos.

```
package java.util;

public interface Collection<T>{
   default Stream<E> stream();
   default Stream<E> parallelStream();
   ...
}
public class Arrays{
   public static <T> Stream<T> stream(T[] array);
   public static <T> Stream<T> stream(T[] array,
            int startInclusive, int endExclusive);
   public static IntStream stream(int[] array);
   public static IntStream stream(int[] array,
            int startInclusive, int endExclusive);
   public static LongStream stream(long[] array);
   public static LongStream stream(long[] array,
            int startInclusive, int endExclusive);
   public static DoubleStream stream(double[] array);
   public static DoubleStream stream(double[] array,
            int startInclusive, int endExclusive);
   ...
}
```

Otra fuente de datos muy importante son los ficheros de texto. La clase BufferedReader ofrece el método lines para crear un Stream<String> formado por las líneas del fichero. Las cadenas de texto, todos los objetos de tipos que derivan de CharSequence, ofrecen el método chars para construir un IntStream.

```
package java.io;

public class BufferedReader{
   public BufferedReader(Reader in);
   public Stream<String> lines();
   ...
}
```

```
public class FileReader extends Reader{
   public FileReader(String fileName)
               throws FileNotFoundException ;
   ...
}
```

```
package java.lang;

public class CharSequence{
    default IntStream chars();
    ...
}
```

Por comodidad de uso es adecuado crear métodos que puedan ser reutilizados para crear flujos de datos a partir de ficheros de texto, de la consola de entrada, o los de las partes en que se divide una cadena de caracteres con delimitadores especificados por una expresión regular. Todos esos métodos los agrupamos en la clase Streams2 del paquete us.lsi.stream. También incluimos un método para guardar como líneas de un fichero de texto los elementos de un flujo de datos de tipo String.

```
package us.lsi.stream;

@SuppressWarnings("resource")
public static Stream<String> streamFromFile(String file) {
      BufferedReader f = null;
      try {
            f = new BufferedReader(new FileReader(file));
      } catch (FileNotFoundException e) {
            throw new IllegalArgumentException(
                  "No se ha encontrado el fichero "+file);
      }
       return f.lines();
}

public static Stream<String> streamFromConsole() {
      BufferedReader f =
           new BufferedReader(new InputStreamReader(System.in));
      return f.lines();
}

public static void  streamToFile(Stream<String> s,
           String file) {
      try {
            final PrintWriter f = new PrintWriter(
               new BufferedWriter(new FileWriter(file)));
            s.forEach(x-> {f.println(x);});
```

```
            f.close();
      } catch (IOException e) {
            throw new IllegalArgumentException(
                "No se ha podido crear el fichero " + file);
            }
}

public static Stream<String> streamFromString(String s,
            String delim) {
        String[] r = s.split(delim);
        return Arrays.<String>stream(r)
               .<String>map((String x)->x.trim())
               .filter((String x)->x.length()>0);
}
```

El API de java proporciona métodos adecuados para crear objetos de tipo *Stream<T>.* Son los métodos siguientes:

```
package java.util.stream;

public interface Stream<T>{
   static <T> StreamBuilder<T> builder();
   static <T> Stream<T> empty();
   static <T> Stream<T> of(T t);
   static <T> Stream<T> of(T... values);
   static <T> Stream<T> iterate(T seed, UnaryOperator<T> f);
   static <T> Stream<T> generate(Supplier<T> s);
   ...
}
public interface StreamBuilder<T> extends Consumer<T>{
   void accept(T t);
   default StreamBuilder<T> add(T t);
   Stream<T> build(
}
public class Streams{
   public static <T> Stream<T> concat(Stream<? extends T> a,
               Stream<? extends T> b) ;
   public static IntStream concat(IntStream a,IntStream b);
   public static LongStream concat(LongStream a, LongStream b);
   public static DoubleStream concat(DoubleStream a,
               DoubleStream b);
   public static <A,B,C> Stream<C> zip(Stream<? extends A> a,
            Stream<? extends B> b,
            BiFunction<? super A,? super B,? extends C> zipper) ;
}
```

- *static <T> StreamBuilder<T> builder():* Devuelve un objeto capaz de construir flujos de datos a partir del Stream dado.

- *static <T> Stream<T> empty():* Se construye un flujo de datos vacío.
- *static <T> Stream<T> of(T t):* Se construye un flujo de datos formado por el único elemento t.
- *static <T> Stream<T> of(T... values):* Se construye un flujo de datos formado por los elementos que se proporcionan como parámetros.
- *static <T> Stream<T> iterate(T s, UnaryOperator<T> f):* Se construye un flujo de datos infinito formado por los elementos s, f(s),f(f(s)),f(f(f(s))),…
- *static <T> Stream<T> generate(Supplier<T> s):* Se construye un flujo de datos infinito formado por los elementos que se van proporcionando desde la factoría s.

Los objetos de tipo *StreamBuilders* son objetos capaces de construir objetos de tipo *Stream<T>* a base de elementos individuales o a partir de otro *Stream<T>.* Estos objetos son una especialización del tipo *Consumer<T>* con métodos adicionales para añadir elementos y construir un flujo de datos a partir de los elementos añadidos.

Como hemos comentado arriba los flujos de datos pueden ser infinitos. Los métodos *iterate* y *generate* construyen flujos de datos infinitos. A partir de ellos podemos construir flujos de datos finitos partir de métodos que extraen sub-flujos o mediante la combinación de filtros y el método *getFirst().*

- *stream.limit(maxSize):* Flujo formado por los primeros *maxSize* elementos.
- *stream.substream(startInclusive,endExclusive):* Flujo formado por los elementos comprendidos entre las posiciones *startInclusive y endExclusive.*
- *stream.peek(consumer).filter(p).getFirst().get():* Con esta secuencia de métodos es posible encontrar el primer de un flujo infinito que no cumple el predicado *p.* Usando el método *peek* podemos gestionar todos los elementos del flujo infinito que cumplen p hasta el primero que no lo cumple.

La clase *Streams* agrupa a un conjunto de métodos diseñados para construir nuevos flujos de datos a partir de otros dados.

- *static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) :* Construye un nuevo flujo de datos concatenando los que se pasan como parámetros.
- *static <A,B,C> Stream<C> zip(Stream<? extends A> a,*
  Stream<? extends B> b, BiFunction<? super A,? super B,? extends C> zipper) :
  Construye un nuevo flujo de datos con elementos resultantes de operar un element del primer flujo con otro del segundo mediante la función zipper. La longitud del flujo resultante es el mínimo de la longitud de los flujos de entrada.

6. Ejemplos con Stream<T>

1. Encontrar el menor número primo mayor que 1000000.

```
Comparator<BigInteger> natural =
         Comparator.<BigInteger>naturalOrder();
BigInteger r = Stream.iterate(new BigInteger("3"),
                   (BigInteger x)->x.nextProbablePrime())
   .filter(x -> natural.compare(x, new BigInteger("1000000"))>0)
   .findFirst()
   .get();
System.out.println(r);
```

2. Encontrar la suma de los múltiplos de 7 comprendidos entre 1000 y 2000

```
Long r = LongStream.range(1001L,2000L)
      .filter(x->x%7==0)
      .sum();
System.out.println(r);
```

3. Un método que guarde en un fichero de texto n primeros números primos. Su signatura será la siguiente:

```
static void crearFicheroNumerosPrimos(String file, Long n){
   Stream<String> f = Stream.iterate(
          BigInteger.ONE,(BigInteger x)-> x.nextProbablePrime())
      .limit(n)
      .map((BigInteger x)->x.toString());
   Streams2.streamToFile(f,file);
}
```

4. Un método que guarde en un fichero de texto el cuadrado de los n primeros números primos que acaben en 7.

```
static void crearFicheroCuadradosPrimos(String file, Long n){
   Stream<String> f = Stream.iterate(
         BigInteger.ONE, (BigInteger x)-> x.nextProbablePrime())
      .limit(n)
      .filter((BigInteger x)->x.mod(
            new BigInteger("10")).equals(new BigInteger("7")))
      .<BigInteger>map((BigInteger x)->x.multiply(x))
      .<String>map((BigInteger x)->x.toString())
       ;
   Streams2.streamToFile(f,file);
}
```

5. Un método que devuelva un *Stream<Long>* a partir de un fichero de texto que contiene en cada línea un número entero. La signatura del método será:

```
static Stream<Long>  obtenerStreamEnteros(String file){
```

```
      return Streams2.streamFromFile(file)
             .<Long>map(Numbers::stringToLong);
}
```

Por comodidad se ha diseñado una clase *Numbers* con métodos del tipo *stringToLong. stringToInteger, etc.* Estos métodos conviertes la excepción *NumberFormatException* en *IllegalArgumentException.*

6. De los puntos leídos de un fichero obtener aquellos cuyas coordenadas sean de la forma (X,X)

```
Streams2.streamFromFile(file)
      .<Punto2D>map(Punto2D::create)
      .filter(x->x.getX().equals(x.getY()))
      .forEach(x->System.out.println(x));
```

7. Un método que dado un fichero de texto con una fecha escrita en cada línea, genere otro fichero con las fechas ordenadas y que estén entre dos fechas dadas. La signatura del método será:

```
Streams2.streamFromFile(file)
      .<Calendar>map(x-> stringToCalendar(x))
      .distinct()
      .sorted()
      .filter(x-> Predicates.estaEnAbierto(x,
                 stringToCalendar("2/1/2009"),
                 stringToCalendar("2/1/2011"))
      .forEach(x->System.out.println(calendarToString(x)));
```

Por comodidad se ha diseñado una clase Predicate*s* con métodos del tipo *estaEnAbierto, estaEnCerrado, mayorQue, etc.*

8. Obtener las diferentes palabras de un fichero ordenadas por orden alfabético. Suponemos los siguientes delimitadores ,;.().

```
Streams2.streamFromFile(file)
 .<String>flatMap(x->Streams2.streamFromString(x,"[ ,;.()]"))
 .distinct()
 .sorted()
 .forEach(x->System.out.println(x))
 ;
```

7. Una primera factoría de objetos de tipo Collector<T>

Recordamos que un objeto de tipo Collector (recolector) es de la forma :

```
public interface Collector<T,A,R>{
      BiConsumer<A,T> accumulator();
      BinaryOperator<A> combiner();
      Function<A,R> transformer();
      Supplier<A> supplier();
      ...
}
```

Dónde:

- T - es el tipo de los elementos del flujo de entrada
- A - es el tipo de acumulador mutable
- R – es el tipo del resultado de la operación de reducción.

Objetos de este tipo son usados como parámetros en el método *collect* del tipo *Stream<T>.* Este método tiene el prototipo::

```
<R,A> R collect(Collector<? super T,A,R> collector);
```

De una forma general podemos decir que el método *collect* toma un flujo de datos y acumula un valor de tipo R usando como acumulador un objeto de tipo A. Cada objeto de tipo Collector indica una acumulación concreta.

El API de java proporciona la clase *Collectors* como una factoría de objetos de tipo *Collector<T,A,R>.* Todos los métodos de esta clase son *static* por lo que no lo haremos explícito.

Agruparemos los métodos en varios grupos según su funcionalidad. En primer lugar vemos un conjunto de de recolectores (Collector) adecuados para acumular un flujo de datos en varios agregados de datos Collection, List, Set, Map y algunos de sus subtipos.

En el caso de acumular en un Map se ofrecen tres métodos. En los tres métodos se proporcionan dos funciones *keyMapper* (f1) y *valueMapper* (f2). Los pares clave-valor del Map resultante se forman al aplicar ambas funciones a cada elemento *e* del flujo de entrada. Los pares tienen, por lo tanto la forma $(f1(e), f2(e))$.

```
public class Collectors {

<T,C extends Collection<T>> Collector<T,?,C> toCollection(
```

```
        Supplier<C> collectionFactory);
<T> Collector<T,?,List<T>> toList();
<T> Collector<T,?,Set<T>> toSet();
<T,K,U> Collector<T,?,Map<K,U>> toMap(
        Function<? super T,? extends K> keyMapper,
        Function<? super T,? extends U> valueMapper);
<T,K,U> Collector<T,?,Map<K,U>> toMap(
        Function<? super T,? extends K> keyMapper,
        Function<? super T,? extends U> valueMapper,
        BinaryOperator<U> mergeFunction);
<T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(
        Function<? super T,? extends K> keyMapper,
        Function<? super T,? extends U> valueMapper,
        BinaryOperator<U> mergeFunction,
        Supplier<M> mapSupplier);
```

La primera posibilidad de acumular un flujo de datos de tipo *T* en un *Map<K,U>* es con el método que toma dos funciones de tipo funcional $\boldsymbol{T \rightarrow K}$ y $\boldsymbol{T \rightarrow U}$**.** En este caso no se admiten claves duplicadas. Si las hubiera se produciría una excepción. El Map resultante admite el método *forEach(Biconsumer<K,U> c).*

```
stream.collect(Collectors.<Person,String,Integer>
   toMap(Person::getDni, Person::getSalary))
   .forEach((x,y)->System.out.println(x+","+y)
  );
```

Si hubiera claves duplicadas podemos proporcionar un operador binario para acumular todos los valores asociados a la misma clave.

```
stream.collect(Collectors.<Person,Integer,Integer>
   toMap(Person::getPosition, Person::getSalary, Integer::sum))
   .forEach((x,y)->System.out.println(x+","+y)
  );
```

Además podemos escoger un subtipo de Map. En este caso un Sorted Map con las claves ordenas en orden inverso.

```
stream.collect(Collectors.<Person,Integer,Integer,
        SortedMap<Integer,Integer>>
   toMap(
     Person::getPosition,
     Person::getSalary,
     Integer::sum,
     ()->new TreeMap<Integer,Integer>(
         Comparator.<Integer>naturalOrder().reversed())
    ))
   .forEach((x,y)->System.out.println(x+","+y)
```

```
  );
```

El siguiente grupo de métodos es adecuado para obtener resúmenes estadísticos del flujo de entrada: número de elementos, media, máximo, mínimo, etc.

```
public class Collectors {

<T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(
       ToDoubleFunction<? super T> mapper);
<T> Collector<T,?,LongSummaryStatistics> summarizingLong(
       ToLongFunction<? super T> mapper);
<T> Collector<T,?,IntSummaryStatistics> summarizingInt(
       ToIntFunction<? super T> mapper);
<T> Collector<T,?,Double> summingDouble(
       ToDoubleFunction<? super T> mapper);
<T> Collector<T,?,Long> summingLong(
       ToLongFunction<? super T> mapper);
<T> Collector<T,?,Integer> summingInt(
       ToIntFunction<? super T> mapper);
<T> Collector<T,?,Double> averagingDouble(
       ToDoubleFunction<? super T> mapper);
<T> Collector<T,?,Double > averagingLong(
       ToLongFunction<? super T> mapper);
<T> Collector<T,?,Double > averagingInt(
       ToIntFunction<? super T> mapper);
<T> Collector<T,?,Long> counting();


<T> Collector<T,?,Optional<T>> maxBy(
       Comparator<? super T> comparator);
<T> Collector<T,?,Optional<T>> minBy(
       Comparator<? super T> comparator);
     ...
}
```

El siguiente grupo de métodos lleva a cabo la operación de reducción equivalente a la reducción inmutable ya comentada anteriormente.

```
public class Collectors {

<T> Collector<T,?,Optional<T>> reducing(
       BinaryOperator<T> op);
<T> Collector<T,?,T> reducing(
       T identity,
       BinaryOperator<T> op);
<T,U> Collector<T,?,U> reducing(
       U identity,
```

```
        Function<? super T,? extends U> mapper,
        BinaryOperator<U> op);
      ...
}
```

La operación de agrupación (*groupBy*) es adecuada para agrupar los elementos de un flujo de datos según el valor devuelto por una función aplicada a cada uno de ellos. Se admite agrupamientos de varios niveles para lo que hay que proporcionar un *Collector* como parámetro adicional. El resultado de estos métodos es un `Map<K,List<T>>>` o más en general, si el agrupamiento es de dos niveles `Map<K, Map<U,List<T>>>>>`. Para varios nieles más de agrupamiento es similar.

Los métodos de partición (*partitioning*) son un caso particular de los de agrupamiento cuando a cada elemento del flujo de entrada le asociamos un valor lógico mediante un predicado.

```
public class Collectors {

<T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
        Function<? super T,? extends K> classifier);
<T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(
        Function<? super T,? extends K> classifier,
        Collector<? super T,A,D> downstream);
<T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(
        Function<? super T,? extends K> classifier,
        Supplier<M> mapFactory,
        Collector<? super T,A,D> downstream);

<T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
         Predicate<? super T> predicate);
<T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
         Predicate<? super T> predicate,
         Collector<? super T,A,D> downstream)
      ...

}
```

Como ejemplo veamos cómo podemos agrupar un flujo de personas por su posición y posteriormente por su salario dentro de cada posición.

```
Stream<Person> stream= ...
stream.collect(Collectors.groupingBy(Person::getPosition,
     Collectors.<Person,Integer>groupingBy(Person::getSalary)))
  .forEach((x,y)->System.out.println(x+","+y.toString()));
```

Y podemos esperar un resultado del tipo:

```
2,{31=[José]}
3,{19=[Rafael], 42=[Mariano]}
1,{20=[Antonio], 10=[Pepe], 15=[Juan]}
```

El método *mapping* es un adaptador de objetos de tipo *Collector<U,A,R>,* es decir que aceptan flujos de tipo U*,* en otro de tipo *Collector<T,A,R>,* es decir que acepta flujos de tipo T, proporcionando una función de tipo *Function<T,U>*.

```
<T,U,A,R> Collector<T,?,R> mapping(
        Function<? super T,? extends U> mapper,
        Collector<? super U,A,R> downstream);
```

Los métodos *joining* son adecuados para acumular flujos de String con delimitadores y posibles sufijos y prefijos.

```
public class Collectors {

Collector<CharSequence,?,String> joining();
Collector<CharSequence,?,String> joining(
       CharSequence delimiter);
Collector<CharSequence,?,String> joining(
        CharSequence delimiter,
        CharSequence prefix,
        CharSequence suffix);
      ...
}
```

Ejemplos

1. Acumula los elementos en una lista

```
List<String> list = people.stream() .map(Person::getName).collect(Collectors.toList());
```

2. Acumula los elementos en un SortedSet

```
SortedSet<String> list =people.stream()
      .map(Person::getName).collect(Collectors.toCollection(TreeSet::new));
```

3. Convierte los elementos del flujo de datos en String y los concatena separados por comas

```
String joined = things.stream()
        .map(Object::toString)
        .collect(Collectors.joining(", "));
```

4. Encuentra el empleado mejor pagado

```
Employee highestPaid = employees.stream()
        .collect(Collectors.maxBy(Comparator.comparing(Employee::getSalary)));
```

5. Agrupa empleados por departamento

```
Map<Department, List<Employee>> byDept  = employees.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));
```

6. Encuentra el empleado mejor pagado por departamento

```
Map<Department, Employee> highestPaidByDept = employees.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment,
                Collectors.maxBy(Comparator.comparing(Employee::getSalary))));
```

7. Agrupa los estudiantes en aprobados y suspensos

```
Map<Boolean, List<Student>> passingFailing =
  students.stream()
        .collect(Collectors.partitioningBy(s -> s.getGrade() >=LIMIT));
```

8. Problemas propuestos

1. En una clase de utilidad `EjerciciosIterables` escriba los siguientes métodos usando las clases *Iterables, Iterables2 o Query* junto con *Ordering*, *Ordering2, Functions* y *Predicates.*

a) Un método que guarde en un fichero de texto los números primos hasta un número `n` dado. Su signatura será la siguiente:

```
public static void crearFicheroNumerosPrimos(String nomFich, Long n)
```

b) Un método que guarde en un fichero de texto el cuadrado de los números primos hasta un número `n` dado. Su signatura será la siguiente:

```
public static void crearFicheroCuadradosNumerosPrimos(String nomFich, Long num)
```

c) Un método que devuelva un `Iterable<Integer>` a partir de un fichero de texto que contiene en cada línea un número entero. La signatura del método será:

```
public static Iterable<Integer> obtenerIterableEnteros(String nomFich)
```

d) Un método devuelva un `Iterable<Long>` que vaya recorriendo los números primos menores que un número `n` y que terminen en un dígito dado. La signatura del método será:

```
public static Iterable<Long> iterablePrimoTerminadoEn (Integer digito, Long num)
```

e) Un método que devuelva un `Iterable<Punto>` con los puntos del plano cuya coordenada sea `(X,X)`, siendo `X` un número primo menor que un número dado. La signatura del método será:

```
public static Iterable<Punto> iterablePuntoPrimo(Long num)
```

f) Un método que dado un fichero de texto con una fecha escrita en cada línea, genere otro fichero con las fechas ordenadas y que estén entre dos fechas dadas. La signatura del método será:

```
public static void crearFicheroFechasEntreOrd(String nomFich,Fecha fIni, Fecha fFin)
```

Suponga definido un método que dado el nombre del fichero original, devuelve el nombre de un fichero de salida, que será exactamente igual que el original, pero con el prefijo `"Ord_"`. Así, por ejemplo, el fichero de salida para un fichero llamado `"Fechas.txt"` será `"Ord_Fechas.txt"`, mientras que si el fichero de entrada es `"./res/Fechas.txt"`, el de salida será `"./res/Ord_Fechas.txt"`. La signatura de este método es:

```
public static String getNombreFicheroOrdenado (String nomFich)
```

g) Un método que dado un fichero de texto con una fecha escrita en cada línea, devuelva un `Iterable<Integer>` que permita iterar sobre los años de las fechas incluidas en el fichero. La signatura del método será:

```
public static Iterable<Integer> iterableDiasFecha(String nomFich)
```

h) Un método que a partir de dos cadenas de texto, devuelva un `Iterable<String>` que permita recorrer las palabras de las dos cadenas ordenadas. La signatura del método será:

```
public static Iterable<String> iterablePalabrasOrdenadas(String c1, String c2)
```

2. Sean los tipos `Cancion` y `Album` especificados mediante las interfaces siguientes:

```
public interface Cancion extends
Comparable<Cancion>,
Copiable<Cancion> {

   String getNombre();

   String getInterprete();

   Hora getDuracion();

   Integer getAño();

   String getGenero();

   Integer
getNumeroDeReproducciones();

   void setNumeroDeReproducciones
(Integer n);

   Integer getCalificacion();

   void setCalificacion (Integer
c);

   Boolean getReproducir();

   void setReproducir (Boolean r);

}
```

```
public interface Album {

  String getNombre();

  Integer getAño();

  Integer
getNumeroDeCanciones();

  Boolean
getEsRecopilacion();

  List<Cancion>
getCanciones();

  void
setCanciones(List<Cancion>
v);

  Hora duracionAlbum();

  Double
calificacionAlbum();

}
```

El criterio de ordenación natural de `Cancion` es por nombre de canción y a igualdad de nombre de canción, por intérprete. En una clase de utilidad `Canciones`, escriba los siguientes métodos:

a) Un método `getGenero` que dado un `Iterable<Cancion>` y un género musical, obtenga otro `Iterable<Cancion>` compuesto solamente por las canciones de ese género. La signatura del método será:

```
public static Iterable<Cancion> getGeneros (
        Iterable<Cancion> itCancion, String genero)
```

b) Un método `getGenerosOrdenadosCalificacion` que a partir de un `Iterable<Cancion>` obtenga un `Iterable` con todas las canciones de la biblioteca de dos géneros determinados, ordenados por su calificación. La signatura del método será:

```
public static Iterable<Cancion> getGenerosCalificacion(
        Iterable<Cancion> itCancion, String genero1,
   String genero2)
```

c) Un método `getCancionesRecopilatorios` que a partir de un `Iterable<Album>` obtenga un `Iterable` con todas las canciones de todos los álbumes que sean recopilatorios. La signatura del método será:

```
public static Iterable<Cancion>
getCancionesRecopilatorios(
            Iterable<Album> itAlbum)
```