Streams

Contenido

- Stream
  - Introducción
  - Creación
  - Sobre Tipo primitivos. Conversión
  - Estructuras desde stream
  - Métodos sobre stream
    - Transformaciones o Intermedios
    - Acciones o terminales
  - Metodos Intermedios o Transformaciones
    - Map, filter, peek, etc.
  - Tipo Optional
  - Métodos Terminales o Acciones
    - Foreach, findFirst,max, toArray, etc.
    - Collect. La clase Collectors
    - Reduce
- Spliterator

Motivaciones

- **La creciente necesidad de explotar con eficiencia los procesadores multicores:**
  - Código que de forma simple se ejecute en paralelo.

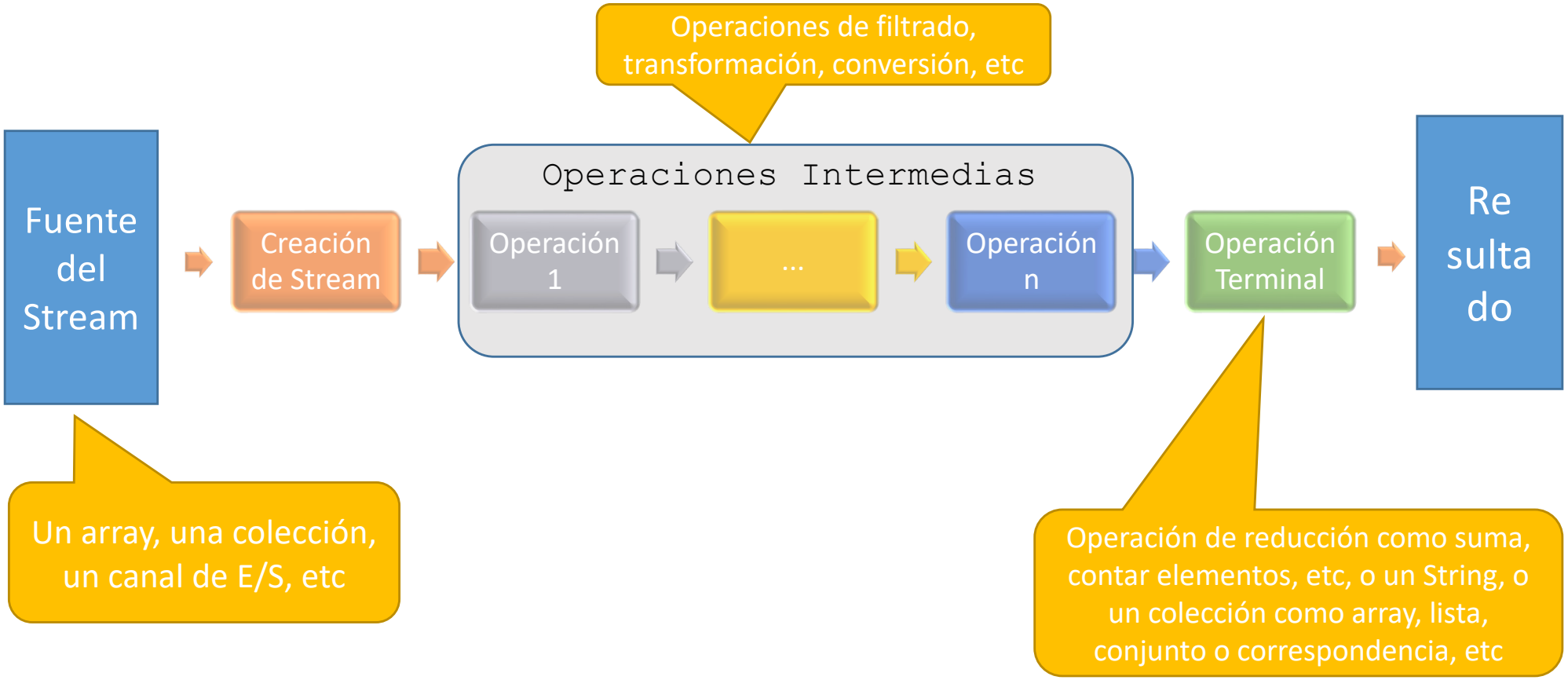
- **La creciente tendencia a manipular colecciones de datos con un estilo funcional:**
  - Permite composición, lazy, paralelismo y fusión.
  - Tomar la fuente de datos, manipularla de forma simple y extraer resultados.

- Se utiliza el concepto de Stream

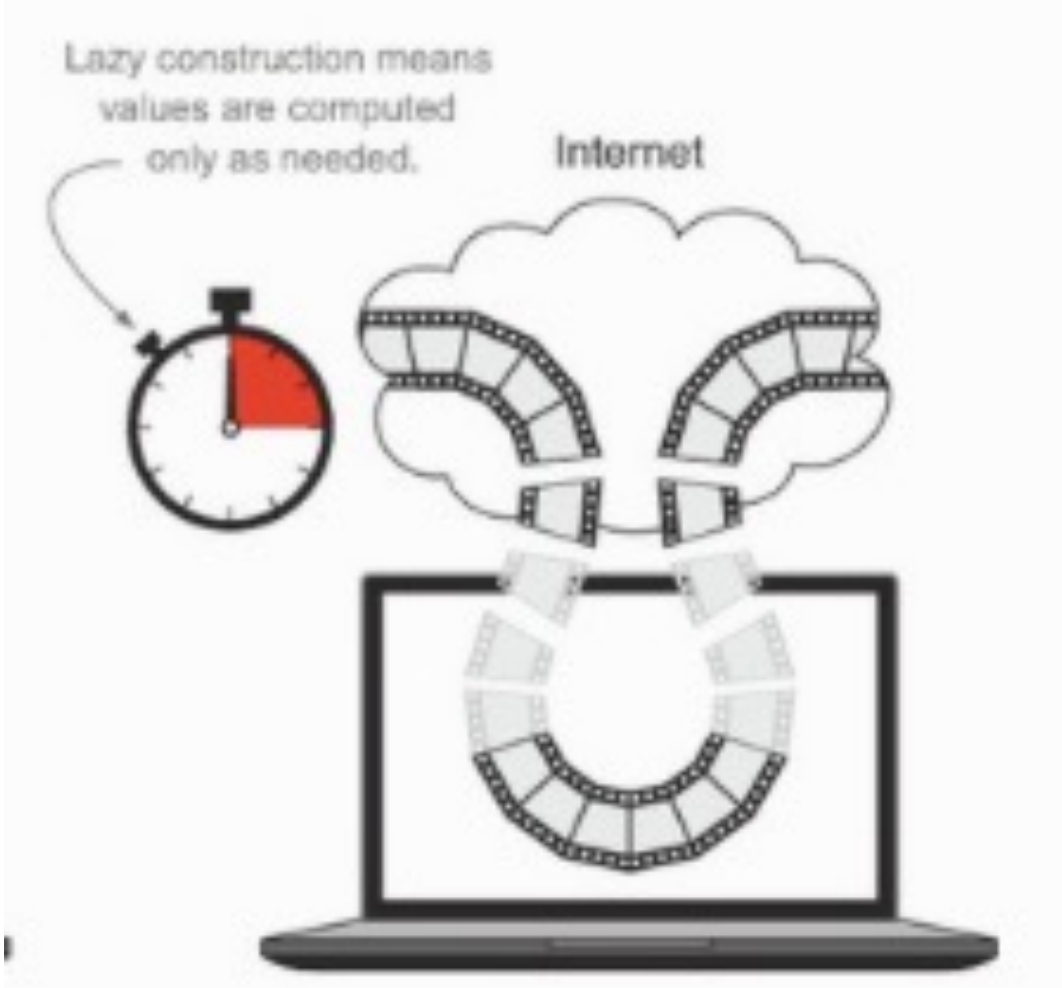
Stream

- Flujo de datos obtenido de una estructura o generado.
  - Permiten
    - Realizar diferentes tipos de operaciones con los datos del flujo.
    - Evaluación perezosa.
    - Paralelizar código automáticamente.
    - Generar secuencias infinitas (lazy).
  - Diseñados para trabajar con funciones de orden superior.
- Diferencias entre Colección y Stream.
  - Una colección es una estructura de datos residente en memoria y que debe ser poblada con todos sus datos antes de comenzar a trabajar con ella.
  - Un Stream es una estructura que computa datos bajo demanda. Un Stream no almacena datos. Simplemente los hace fluir.

Modelo de funcionamiento



Stream



- Imagen de : Raoul - Gabriel Urma , Mario Fusco , Alan Mycroft . "Java 8 in Action: Lambdas , streams, and functional-style programming " . iBooks .

Ejemplo inicial

Dada una lista de enteros, deseamos conocer la suma de los cuadrados de los elementos pares mayores que 10.

```
public int sumaCuadradoPares(List<Integer> lista) {
    int sum = 0;
    for (int i : lista)
            if (i % 2 == 0 && i > 10)
                        sum += i * i;
    return sum;
}
```

Problemas:

- El cliente debe maneja la iteración. Iteración externa.
- Mucho código para un problema tan simple.
- Se mezcla iteración, selección, cálculo y colección.
- El código es difícil de paralelizar.

Ejemplo inicial con stream

```
public int sumaCuadradoPares(List<Integer> lista) {
      return  lista.stream()
                  .filter(x -> x % 2 == 0)
                  .filter(x -> x > 10)
                  .mapToInt(x -> x * x)
                  .sum();
}
```

Ventajas:

- El cliente no maneja la iteración. Iteración interna.
- Código muy simple (estilo funcional composicional).
- Separación de iteración, selección, cálculo y colección.
- El código es fácil de paralelizar.

Otro ejemplo con stream

Sumar las longitudes de las cadenas que comienzan con una letra mayor que la dada.

```
List<String> ls = List.of("Juana", "Maria", "Antonio",
                          "Pedro", "Ana", "Paco", "Luis");

public int sumaLongitudesMayores(List<String> lista, char c) {
    int sum = 0;
    for (String s: lista)
        if (s.toUpperCase().charAt(0) > c)
            sum += s.length();
    return sum;
}
```

sumaLongitudMayores(ls, ‘J’) → 18

Otro ejemplo con stream

Sumar las longitudes de las cadenas que comienzan con una letra mayor que la dada.

```
List<String> ls = List.of("Juana", "Maria", "Antonio",
                          "Pedro", "Ana", "Paco", "Luis");

public int sumaLongitudesMayores(List<String> lista, char c) {
      return lista.stream()
                  .map(String::toUpperCase)
                  .filter(s -> s.charAt(0) > c)
                  .mapToInt(String::length)
                  .sum();
}
```

```
sumaLongitudMayores(ls, 'J')     →    18
```

Creación de Stream

- Desde una colección se crean con el método stream() incluido por defecto en la interfaz Collection:

```
public default Stream<T> stream()
```

- También hay métodos de clase por defecto en la interfaz Stream:

```
public static <T> Stream<T> of(T obj)
public static <T> Stream<T> of(T ... values)
public static <T> Stream<T>
              iterate(T seed, UnaryOperator<T> func)
public static <T> Stream<T>
              generate(Supplier<T> sup)
public static Stream<T> iterate(T seed, Predicate<? super T> hasNext,
                                                  UnaryOperator<T> next)
public static Stream<T>  ofNullable(T t)
```

- Y en la clase Arrays:

```
public static <T> Stream<T> stream(T []  array)
```

Ejemplo de creación de Stream

- Ejemplo:
  - Crear un stream con los números pares positivos hasta el 100

```
Stream.iterate(2, x -> x <= 100, x -> x + 2);
```

  - Los stream son perezosos:

```
Stream.iterate(2, x -> x + 2)
      .takeWhile(x -> x <= 100);
```

Concurrencia versus paralelismo

- La concurrencia permite que varios hilos progresen simultáneamente en el mismo núcleo de la CPU. Los hilos necesitan coordinarse e "interrumpirse" para realizar su trabajo.
  - Piénsalo como un malabarista que utiliza una sola mano (un solo núcleo de CPU) con múltiples bolas (hilos). Sólo pueden sostener una bola en todo momento (haciendo el trabajo), pero la bola cambia con el tiempo (interrumpiendo y cambiando a otro hilo). Incluso con sólo dos bolas, tienen que hacer malabares con la carga de trabajo.

- El paralelismo consiste en ejecutar múltiples tareas literalmente al mismo tiempo, como en varios núcleos de la CPU.
  - El malabarista utiliza ahora ambas manos (más de un núcleo de CPU) para sostener dos bolas a la vez (haciendo el trabajo simultáneamente). Si sólo hay dos bolas en total, pueden sostener ambas al mismo tiempo. Si hay más, habrá también concurrencia.

Fuente: A Functional Approach to Java (Second Early Release) (Ben Weidig)

Streams paralelos

- Es posible paralelizar un stream simplemente enviándole el mensaje parallel() de la interfaz Stream.

```
public Stream<T> parallel()
```

- Para volver a convertirlo en secuencial.

```
public Stream<T> sequential()
```

- O crearlo directamente paralelo desde una colección con el método por defecto disponible en la interfaz Collection

```
default public Stream<T> parallelStream()
```

Streams de tipos básicos

- Se pueden crear Stream para tipos básicos.
- Las interfaces disponibles son:
  - IntStream, LongStream, DoubleStream
  - Estas interfaces **definen los mismos métodos que Stream aunque especializados para estos tipos**, y además añaden otros.

- Y en la clase Arrays:

```
public static IntStream stream(int [] array)
public static LongStream stream(long [] array)
public static DoubleStream stream(double [] array)
```

Métodos extras para tipos básicos

Mostramos para IntStream (similar para los otros)

- Métodos factoría para crear IntStream

```
public static IntStream range(int startInc, int endExc)
public static IntStream rangeClosed(int startInc, int endInc)
```

- Métodos de instancia

```
public Stream<Integer> boxed()
public int sum()
public IntSummaryStatistics summaryStatistics()
     (max, min, count, sum, average)
```

Métodos en la clase Random que generan IntStream

Podemos generar XxxStream con valores aleatorios.

Utilizamos métodos de instancia de la clase Random:

- Métodos de instancia de la clase Random para crear IntStream

```
public IntStream ints(int origin, int bounds)
public IntStream ints(long numElem, int origin, int bounds)

// Genera 100 valores de entre 0,1,2,3,4 y 5
IntStream is = (new Random()).ints(100, 0, 6);
```

- Igual se pueden crear
  - LongStream con el método longs
  - DoubleStream con el método doubles

Creando IntStream de un String (java9)

- Se ha incorporado un nuevo método a la clase String

```
IntStream chars()
```

```
public static IntStream stringAIntStreamEnMinus(String s) {
    return s.toLowerCase().chars();
}
```

Stream para tipos basicos versus referencias

- No es lo mismo IntStream que Stream<Integer>

```
int [] array = {1,2,3,4,5}
Arrays.stream(array)                    // IntStream
Stream.of(array)                        // Stream<int[]>
```

```
Integer[] array = {1,2,3,4,5}
Arrays.stream(array)                    // Stream<Integer>
Stream.of(array)                        // Stream<Integer>
```

Conversión de XXXStream a Stream<XXX>

- Es posible convertir un XXXStream en Stream<XXX> con el método boxed()

```
int [] array = {1,2,3,4,5};
IntStream is = Arrays.stream(array);          // IntStream
Stream<Integer> isi = is.boxed();             // Stream<Integer>
```

- Posteriormente veremos cómo hacer la conversión contraria.

Creando una estructura a partir de un Stream

- Se pueden generar diferentes estructuras.
  - Sea st una variable que referencia un stream
- Arrays

```
st.toArray(Tipo[]::new)
```

- Listas

```
st.collect(Collectors.toList())
```

- Conjuntos

```
st.collect(Collectors.toSet())
```

- Correspondencias

```
st.collect(Collectors.groupingBy(...))
```

- String

```
st.collect(Collectors.joining(delim)).toString()
```

Métodos sobre Stream

- Métodos fundamentales:
  - Métodos Transformaciones o Intermedios. Producen otro stream.
    - `map, flatMap`
    - `filter,`
    - `skip, limit,`
    - `sorted,`
    - `peek,`
    - `distint,`
    - `takeWhile, dropWhile,`
    - `parallel, sequential.`
  - Métodos Acciones o Terminales. Producen un dato (o void)
    - `forEach,`
    - `findFirst, findAny,`
    - `min, max,`
    - `noneMatch, allMatch anyMatch,`
    - `count,`
    - `iterator,`
    - `reduce,`
    - `toArray,`
    - `collect.`

Uso de Stream

Normalmente, el proceso que se sigue es el siguiente:

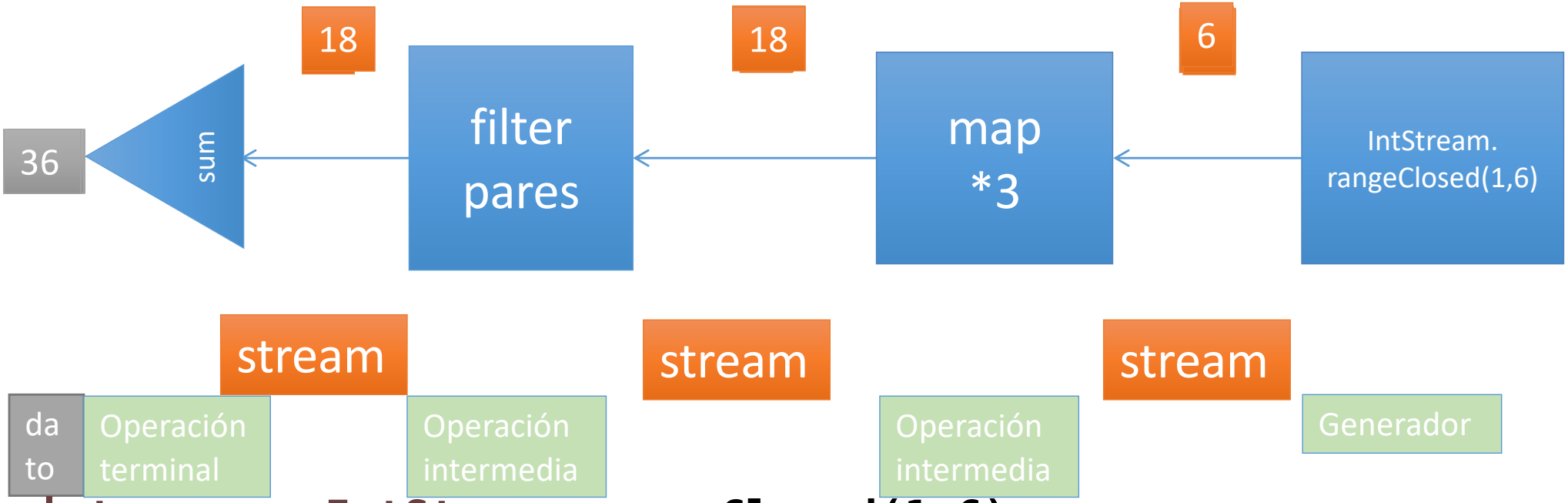
- Se crea un stream.
- Se manipula sus datos con diferentes métodos intermedios encadenando llamadas.
- Finalmente se utiliza un método terminal para extraer la información.

No se ejecuta nada mientras no haya un método terminal que lo demande.

Características de un Stream

- Un stream puede tener activa unas características que ayudan a la hora de realizar operaciones con él.
- Esas características pueden cambiar a lo largo de la vida de un stream.
  - SIZED se conoce su tamaño.
  - DISTINCT Los elementos son todos distintos (con equals o con == para los stream de tipos básicos).
  - SORTED Los elementos están ordenados según su orden natural.
  - ORDERED Los elementos tienen un orden que debe mantenerse.
- Por ejemplo:
  - Un Stream sobre un HashSet activa la característica DISTINCT.
  - La operación filter(…) mantiene las características DISTINCT y SORTED pero anula SIZED.
  - La operación sorted() mantiene DISTINCT y SIZED y activa la característica SORTED.

Métodos sobre Stream




```
int suma = IntStream.rangeClosed(1,6)
                    .map(x -> x * 3)
                    .filter(x -> x % 2 == 0)
                    .sum();
```

El proceso se realiza dato a dato bajo demanda de sum

```
1 -> 3 ->          3 -> 9 ->          5 -> 15 ->
2 -> 6 -> 6        4 -> 12 -> 12      6 -> 18 -> 18
```

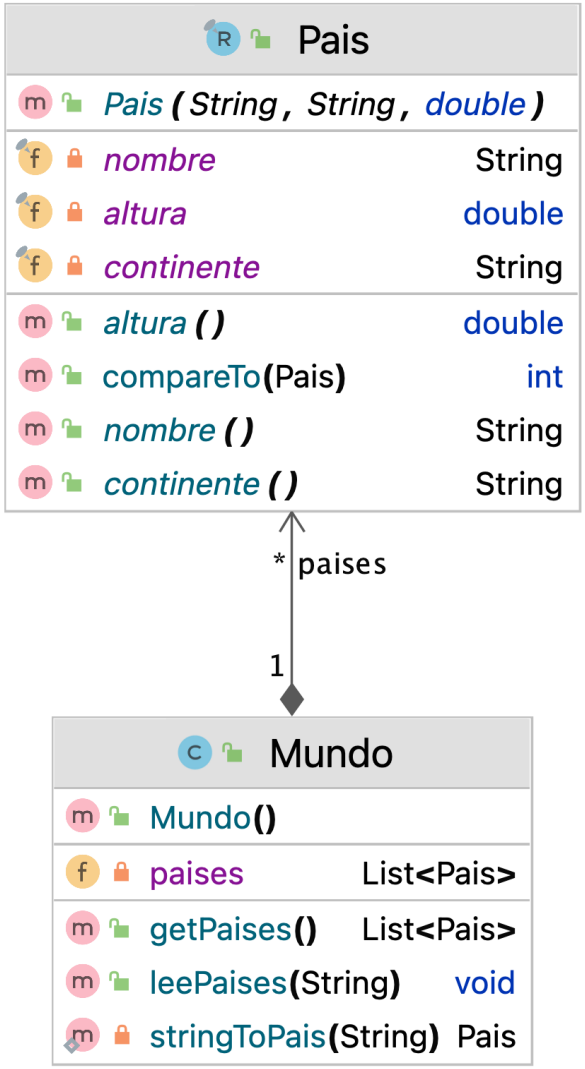
Métodos stateless y statefull

- Stateless: métodos que procesan dato a dato y no necesitan mantener información almacenada.
- Statefull: métodos que deben almacenar alguna información entre el tratamiento de un dato y otro.

```
int suma = IntStream.rangeClosed(1,6)
                    .map(x -> x * 3)
                    .filter(x -> x % 2 == 0)
                    .sum();
```

`map` y `filter` son stateless.
`sum` es statefull.

Ejemplo de apoyo



Métodos **intermedios** sobre Stream

map

```
Stream<R>  map(Function<? super T,? extends R> mapper)
```

```
public interface Function<T,R> {
        R apply(T t);
}
```

Crea un nuevo stream con los resultados de aplicar a cada elemento del stream una función.

Ejemplo: Obtener una lista con el nombre de todos los países

Métodos **intermedios** sobre Stream

map

Ejemplo

```
Stream<R>  map(Function<? super T,? extends R> mapper)
```

Lista con el nombre de todos los países:

```
public List<String> getNombrePaises() {
    return paises.stream().map(Pais::nombre)
            .collect(Collectors.toList());
}
```

Con **mapToInt** se obtiene un **IntStream**

Con **mapToLong** se obtiene un **LongStream**

Con **mapToDouble** se obtiene un **DoubleStream**

Desde **XXXStream** existe `Stream<T> mapToObj(XXXFunction<T> mapper)`

Métodos **intermedios** sobre Stream

filter

```
Stream<T>  filter(Predicate<? super T> predicate)
```

```
public interface Predicate<T> {
        boolean test(T t);
}
```

Crea un nuevo stream con los elementos del stream que verifican el predicado.

Ejemplo: Obtener la lista de los países de un continente dado

Métodos **intermedios** sobre Stream

filter

Ejemplo

```
Stream<T>  filter(Predicate<? super T> predicate)
```

Lista de los países de un continente dado:

```
public List<Pais> getPaisesDelContinente(String continente) {
    return paises.stream()
                 .filter(p -> p.continente().equals(continente))
                 .collect(Collectors.toList());
}
```

Métodos **intermedios** sobre Stream

flatMap

```
Stream<R>  flatMap(Function<? super T,
                           ? extends Stream<? extends R> mapper)
```

```
public interface Function<T,R> {
        R apply(T t);
}
```

Crea un nuevo stream con los resultados de aplicar a cada elemento del stream una función mapper que genera otro stream.

Lista de países de varios continentes dados

Métodos **intermedios** sobre Stream

flatMap

Ejemplo

```
Stream<R>  flatMap(Function<? super T,
                           ? extends Stream<? extends R> mapper)
```

Lista de países de varios continentes dados

```
List<String> continentes = List.of("Europe", "Asia", "South America");
public List<Pais> paisesDe(List<String> continentes) {
   return continentes.stream()
         .flatMap(cont -> paises.stream().
               .filter(p-> p.continente().equals(cont)))
         .collect(Collectors.toList());
}
```

Con `flatMapToInt` se obtiene un `IntStream`

Con `flatMapToLong` se obtiene un `LongStream`

Con `flatMapToDouble` se obtiene un `DoubleStream`

Métodos **intermedios** sobre Stream

peek

```
Stream<T> peek(Consumer<? super T> action)
```

```
public interface Consumer<T> {
        void accept(T t);
}
```

Con cada elemento del Stream realiza una acción y devuelve el mismo stream

Ejemplo: mostrar los elementos de un stream en la consola conforme se van procesando y seguir con él.

Métodos **intermedios** sobre Stream

peek

Ejemplo

```
Stream<T> peek(Consumer<? super T> action)
```

Muestra los elementos de un stream conforme se van procesando y seguir con él.

```
public List<Pais> getYMuestraPaisesDelContinente(String continente) {
    return paises.stream()
            .filter(p -> p.continente().equals(continente))
                .peek(System.out::println)
                .collect(Collectors.toList());
}
```

Métodos **intermedios** sobre Stream

distinct

`Stream<T>  distinct()`

Devuelve un stream con los datos de otro stream pero sin repetir.

Se utiliza

equals para stream de objetos

== para stream de tipos básicos

```
public List<String> getContinentes() {
    return paises.stream()
            .map(Pais::continente)
            .distinct()
            .collect(Collectors.toList());
}
```

Métodos **intermedios** sobre Stream

`limit`

```
Stream<T>   limit(long n)
```

Devuelve un stream con los primeros n datos de otro stream.

Igualmente existe

```
Stream<T> skip(long n)
```

que elimina los n primeros datos del stream

Métodos **intermedios** sobre Stream

sorted

```
Stream<T>   sorted()
Stream<T>   sorted(Comparator<? super T> comp)
```

Devuelve un stream con los elementos del stream ordenados por orden natural o el comparador dado.

```
public List<String> getListContinentes() {
    return paises.stream()
            .map(Pais::continente)
            .distinct()
            .sorted()
            .collect(Collectors.toList());
}
```

Métodos **intermedios** sobre Stream

takeWhile

```
Stream<T>   takeWhile(Predicate<? super T> predicate)
```

Toma elementos del stream mientras cumplan un predicado.

- En cuanto un elemento no cumpla el predicado se termina el stream.

Lista de países del continente con altura menor a una dada

```
List<Pais> extraeMenoresDeContinente(String continente, double maxAltura) {
        return paises.stream()
                        .filter(pais->pais.continente().equals(continente))
                        .sorted(Comparator.comparingDouble(Pais::altura))
                        .takeWhile(pais->pais.altura() < maxAltura)
                        .collect(Collectors.toList();
}
```

Métodos **intermedios** sobre Stream

dropWhile

```
Stream<T>   dropWhile(Predicate<? super T> predicate)
```

Elimina elementos del stream mientras cumplan un predicado.

- En cuanto un elemento no cumpla el predicado se devuelve el stream restante.

Lista de países del continente con altura mayor o igual a una dada

```
List<Pais> extraeMayoresDeContinente(String continente, double maxAltura) {
        return paises.stream()
                        .filter(pais->pais.continente().equals(continente))
                        .sorted(Comparator.comparingDouble(Pais::altura))
                        .dropWhile(pais->pais.altura() < maxAltura)
                        .collect(Collectors.toList();
}
```

Métodos **terminales** sobre Stream

forEach

```
void forEach(Consumer<? super T> action)
```

```
public interface Consumer<T> {
        void accept(T t);
}
```

Con cada elemento del Stream realiza una acción.

Ejemplo: Mostrar los elementos de un stream por pantalla.

Métodos **terminales** sobre Stream

forEach

Ejemplo

```
void forEach(Consumer<? super T> action)
```

Muestra los nombre de los países por la consola

```
public void muestraNombrePaises() {
    paises.stream()
            .map(Pais::nombre)
            .forEach(System.out::println);
}
```

Métodos **terminales** sobre Stream

allMatch

```
public interface Predicate> {
        booblean test(T t);
}
```

```
boolean allMatch(Predicate<? super T> pred)
```

Comprueba si todos los elementos verifican un predicado.

Ejemplo: todos los países tienen altura superior a una dada?

```
public boolean todosMayores(double altura) {
    return paises.stream()
                 .allMatch(p -> p.altura() > altura);
}
```

```
boolean noneMatch(Predicate<? super T> pred)
boolean anyMatch(Predicate<? super T> pred)
```

Métodos **terminales** sobre Stream

`findFirst`

```
Optional<T>      findFirst()
```

Devuelve un opcional con el primer elemento del stream o el opcional empty si el stream está vacío.

Ejemplo: País con altura mas baja de un continente dado de entre los mas altos de una altura dada.

Métodos **terminales** sobre Stream

`findFirst`

Ejemplo

```
Optional<T>      findFirst()
```

Pais con altura mas baja de un continente dado de entre los mas altos de una altura dada.

```
Optional<Pais> masBajoDeContinente(String continente, double minAltura) {
        return paises.stream()
                .filter(pais->pais.continente().equals(continente))
                .sorted(Comparator.comparingDouble(Pais::altura))
                .findFirst();
}
```

Métodos **terminales** sobre Stream

max

```
Optional<T>  max(Comparator<? super T> comparator)
```

Devuelve un opcional con el máximo elemento del stream según el comparador proporcionado.

Ejemplo: obtener el país con mayor altura.

Métodos **terminales** sobre Stream

max

Ejemplo

```
Optional<T>  max(Comparator<? super T> comparator)
```

Obtener el país con mayor altura.

```
public Pais paisMayorAltura() {
    return paises.stream()
            .max(Comparator.comparingDouble(Pais::altura))
            .get();
}
```

```
Optional<T> min(Comparator<? super T> comparator)
```

Métodos **terminales** sobre Stream

count

```
Long    count()
```

Devuelve la longitud del stream.

Ejemplo: cuántos países hay que verifiquen un predicado dado?

Métodos **terminales** sobre Stream

count

Ejemplo

```
Long    count()
```

Cuántos países hay que verifiquen un predicado dado:

```
public long numPaisesQueVerifican(Predicate<Pais> pred) {
    return paises.stream()
            .filter(pred)
            .count();
}
```

Métodos **terminales** sobre Stream

- Tres operaciones terminales fundamentales:

`toArray`

Genera un array con los datos de stream

`reduce`

Genera un dato inmutable a partir de los datos del stream

`collect`

Genera un dato mutable a partir de los datos del stream

Métodos **terminales** sobre Stream

toArray

```
Object []  toArray()
T []       toArray(IntFunction<T []>)
```

Devuelve un array del tipo proporcionado con los elementos del stream.

Ejemplo: crear un array de países de un continente dado

Métodos **terminales** sobre Stream

toArray

Ejemplo

```
T[]   toArray(IntFunction<T []>)
```

Array de países de un continente dado:

```
public Pais[] arrayDePaises(String continente) {
    return paises.stream()
            .filter(p -> p.continente().equals(continente))
            .toArray(Pais[]::new);
}
```

Métodos **terminales** sobre Stream

reduce

```
Optional<T>      reduce(BinaryOperator<T> accumulator)
```

Devuelve un opcional tras reducir con el accumulator los elementos del stream

$$(\cdots((e_1 \odot e_2) \odot e_3) \odot \cdots \odot e_n)$$

Ejemplo: Obtener la suma de las alturas de todos los países.

Métodos **terminales** sobre Stream

reduce

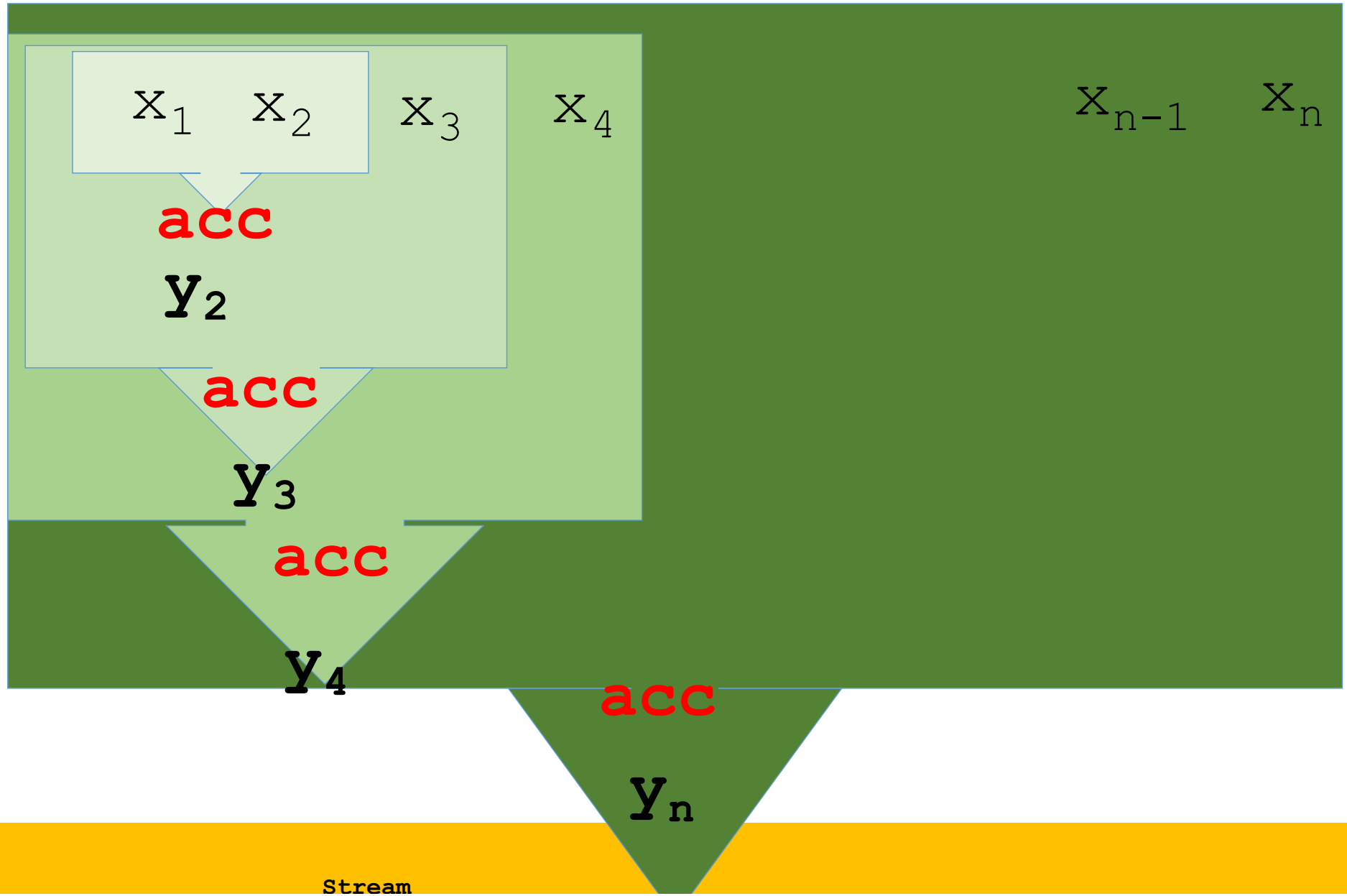
Ejemplo

```
Optional<T>      reduce(BinaryOperator<T> accumulator)
```

Obtener la suma de las alturas de todos los países:

```
public double sumaAlturas() {
    return paises.stream()
            .map(Pais::altura)
            .reduce(Double::sum)
            .get();
}
```

reduce(BinaryOperator<T> acc)



Métodos **terminales** sobre Stream

reduce

```
T     reduce(T identity, BinaryOperator<T> accumulator)
```

Opera a partir de indentity con todos los datos del stream con el operador accumulator

$$(\cdots((identity \odot e_1) \odot e_2) \odot e_3) \odot \cdots \odot e_n)$$

Ejemplo: calcula la suma de todas las alturas

Métodos **terminales** sobre Stream

reduce

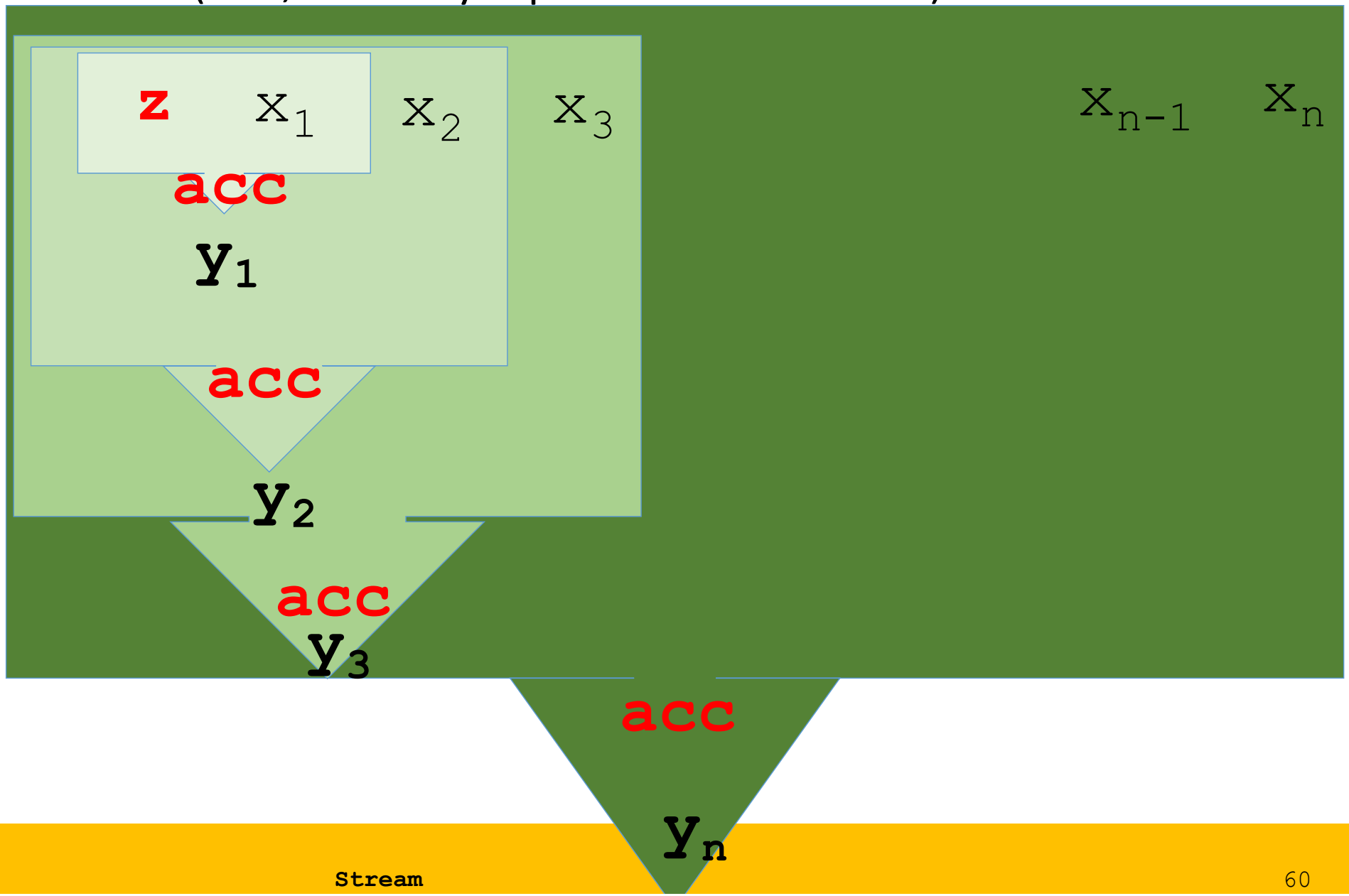
Ejemplo

```
T     reduce(T identity, BinaryOperator<T> accumulator)
```

Suma de todas las alturas:

```
public double suma2Alturas() {
    return paises.stream()
            .map(Pais::altura)
            .reduce(0d, Double::sum);
}
```

reduce(T z, BinaryOperator<T> acc)



Métodos **terminales** sobre Stream

reduce

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

- Opera a partir de identity con todos los datos del stream con el operador accumulator combinando los resultados con combiner.
- Puede ejecutarse en paralelo.

Ejemplo: calcula la suma de todas las alturas

Métodos **terminales** sobre Stream

reduce

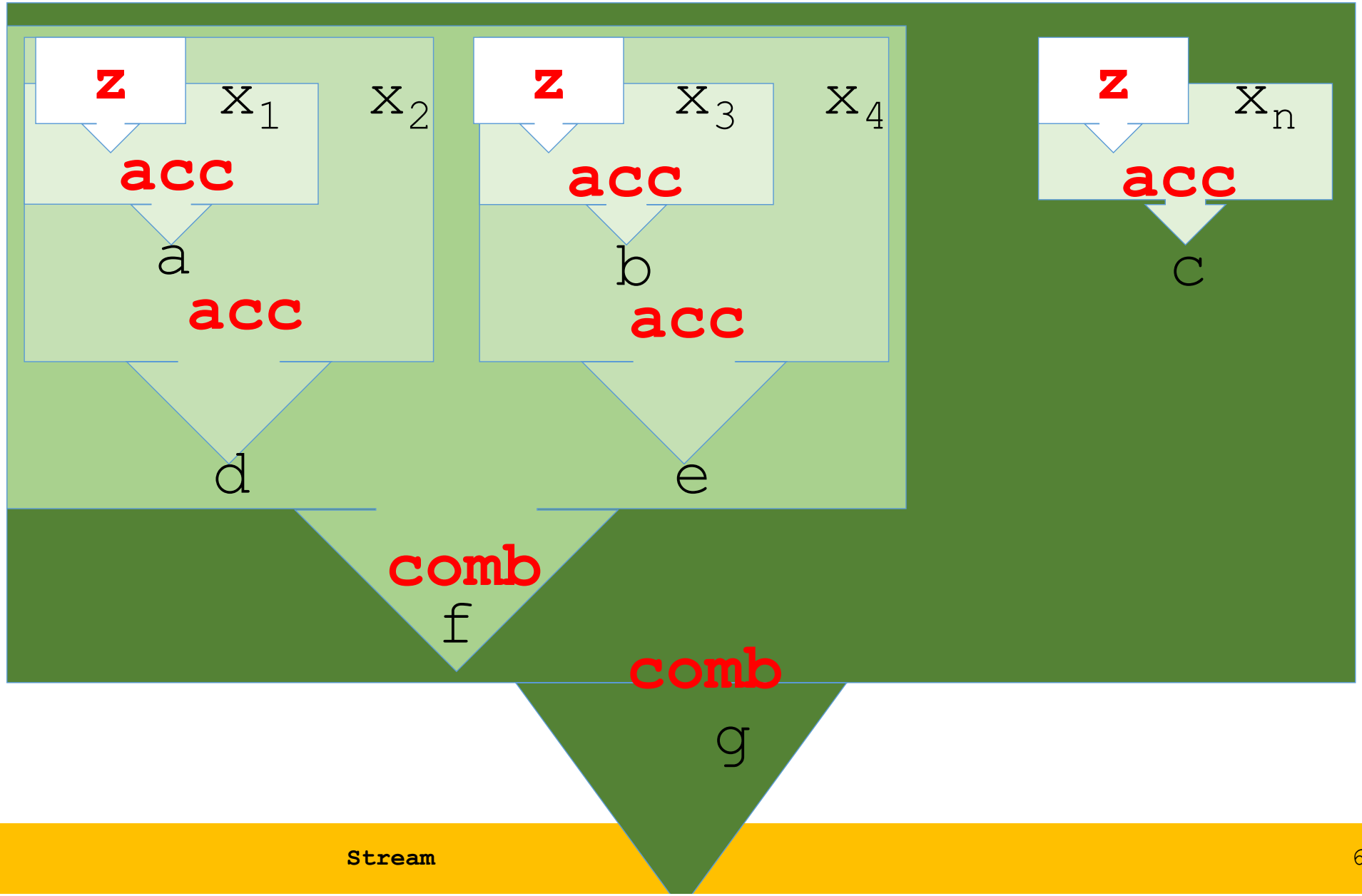
Ejemplo

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

Suma de todas las alturas:

```
public double suma3Alturas() {
    return paises.stream()
            .map(Pais::altura)
            .reduce(0d, Double::sum, Double::sum);
}
```

reduce(R z, BiFuction<? super T,R,R> acc, BinaryOperator<R,R> comb)



Métodos **terminales** sobre Stream

collect

```
R collect( Supplier<R> suplier,
           BiConsumer<R,? super T> accumulator,
           BiConsumer<R,R> combiner)
```

Devuelve una estructura a partir de:

- **Una función que crea una estructura de datos.**
- **Una función que acumula a la estructura un nuevo elemento.**
- **Una función que combina dos estructuras.**

Ejemplo: devolver un conjunto ordenado con los países de un continente dado.

Métodos **terminales** sobre Stream

collect

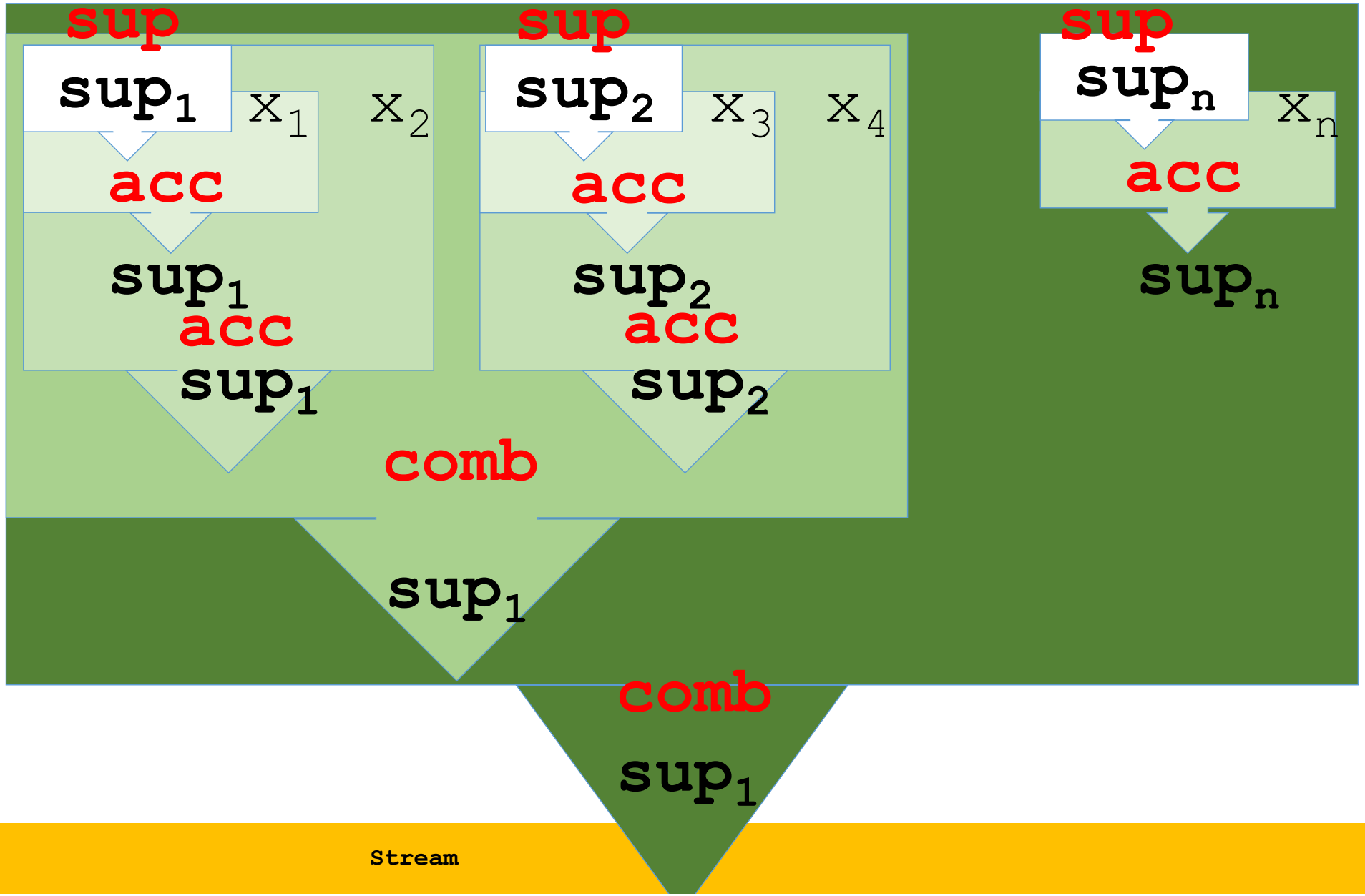
Ejemplo

```
R collect( Supplier<R> suplier,
           BiConsumer<R,? super T> accumulator,
           BiConsumer<R,R> combiner)
```

Conjunto ordenado de países de un continente dado:

```
public Set<Pais> paisesOrdDelContinente(String continente) {
    return paises.stream()
            .filter(p -> p.continente().equalsIgnoreCase(continente))
            .collect(TreeSet<Pais>::new
                    ,Set::add
                    ,Set::addAll);
}
```

collect(Supplier<R> sup, BiConsumer<R,? super T> acc, BiConsumer<R,R> comb)



Métodos **terminales** sobre Stream

collect

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

Devuelve una estructura a partir de un Collector<T,A,R>.

```
publi
```

supplier: Crea un nuevo contenedor

accumulator: Acumula un nuevo dato al contenedor

combiner: Combina dos contenedores

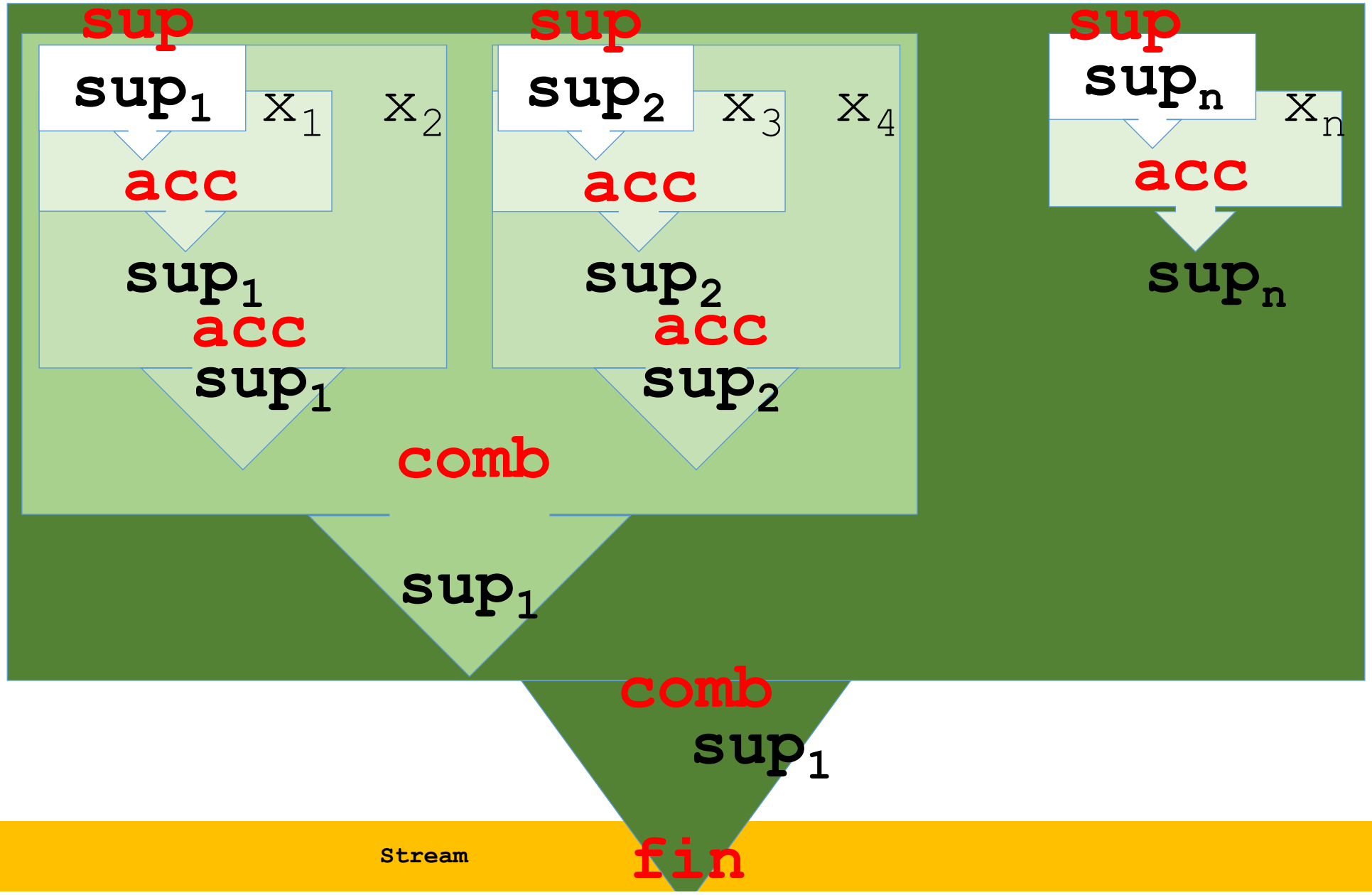
finisher: Realiza una transformación final con el contenedor

characteristics: CONCURRENT, IDENTITY_FINISH, UNORDERED

```
}
```

- La clase Collectors simplifica la creación de colectores.

collect(Collector<? super T, A, R> col) *sup, acc, comb, fin*



Características de un Stream

collect

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

Características:

```
enum  Collector.Characteristics
    {CONCURRECT, IDENTITY_FINISH, UNORDERED}
```

- ***CONCURRECT***: Puede ejecutarse en paralelo. Se usa combiner.
- ***IDENTITY_FINISH***: La función final es la identidad y puede obviarse.
- ***UNORDERED***: Las operaciones pueden no respetar el orden de los elementos.

Métodos **terminales** sobre Stream. Creando un colector

`collect`

```
public class MiColector implements Collector<String, Set<String>, TreeSet<String>> {
    public Supplier<Set<String>> supplier() {
        return HashSet::new;
    }
    public BiConsumer<Set<String>, String> accumulator() {
       return Set::add;
    }
    public BinaryOperator<Set<String>> combiner() {
       return (s1,s2) ->{s1.addAll(s2); return s1;};
    }
    public Function<Set<String>, TreeSet<String>> finisher() {
        return TreeSet::new;
    }
    public Set<Collector.Characteristics> characteristics() {
        return new HashSet<Characteristics>(
                Arrays.asList(Characteristics.CONCURRENT,
                    Characteristics.UNORDERED));
    }
}
```

Métodos **terminales** sobre Stream. Usando el colector `collect`

Ejemplo

Conjunto de países usando un colector

```
public Set<String> paisesConColector() {
    return paises.stream()
            .map(Pais::nombre)
            .collect(new MiColector());
}
```

La clase Collectors

collect

La clase `Collectors` proporciona muchos colectores especializados.

Contiene métodos estáticos como:

```
<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)
```

Varias versiones

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

Veremos colectores especializados

```
counting()                                  mapping()
summingInt(), summingLong(), summingDouble()
summarizingInt(), summarizingLong(), summarizingDouble()
partitioningBy()                            reducing()
groupingBy()                                joining()
maxBy(Comparator<? super T>)                minBy(Comparator<? super T>)
```

Colectores elementales para colecciones

`collect`

`toList`
`toSet`
`toMap`

```
TipoIndicadoEnColector      collect(Collector)
```

Devuelve una estructura según indique el colector.

```
<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)
```

Ejemplo: devolver una lista con los intérpretes de un género:

Colectores elementales para colecciones

collect

```
TipoIndicadoEnColector      collect(Collector)
```

toList
toSet
toMap

Ejemplo

Si queremos una lista `collect(Collectors.toList())`

Si queremos un conjunto `collect(Collectors.toSet())`

Si queremos un map

```
collect(Collectors.toMap(
        Function<? super T,? extends K> keyMapper,
        Function<? super T,? extends U> valueMapper))
```

CUIDADO: Las claves no pueden estar repetidas

Colectores que definen el tipo de colección

`collect` `toCollection`

```
TipoIndicadoEnColector      collect(Collector)
```

El método

```
static <T,C extends Collection<T>> Collector<T,?,C>

Collectors.toCollection(TipoColeccion::new)
```

permite recolectar en cualquier colección. Se necesita un Supplier que cree la colección.

Ejemplo: devolver el conjunto ordenado de los continentes

Colectores que definen el tipo de colección

`collect` `toCollection`

`TipoIndicadoEnColector collect(Collector)`

Ejemplo

```
static <T,C extends Collection<T>> Collector<T,?,C>
            Collectors.toCollection(TipoColeccion::new)
```

Conjunto ordenado de los continentes:

```
public Set<String> continentesOrd() {
    return paises.stream()
            .map(Pais::continente)
            .collect(
                Collectors.toCollection(TreeSet<String>::new));
}
```

Colector que transforma y colecciona

`collect`

`mapping`

```
TipoIndicadoEnElColector collect(Collector)
```

El método

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

permite extraer un dato con mapper y coleccionarlo según el colector donwstream.

Ejemplo: devolver el conjunto de alturas de un continente dado.

Colector que transforma y colecciona

`collect`

`mapping`

Ejemplo

```
TipoIndicadoEnElColector collect(Collector)
```

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

Conjunto de alturas de un continente dado:

```
public Set<Double> alturasDelContinente(String continente) {
    return paises.stream()
            .filter(p -> p.continente().equals(continente))
            .collect(Collectors.mapping(Pais::altura
                                        ,Collectors.toSet()));
}
```

Colector que cuenta

`collect`

`counting`

```
TipoIndicadoEnColector      collect(Collector)
```

El método

```
static <T> Collector<T,?,Long>  Collectors.counting()
```

permite contar los datos contenidos en un stream.

Ejemplo: cuántos países hay con altura menor a una dada

Colector que cuenta

`collect`

`counting`

Ejemplo

```
TipoIndicadoEnColector      collect(Collector)
```

```
   static <T> Collector<T,?,Long>  Collectors.counting()
```

Cuántos países hay con una altura menor a una dada:

```
public long numPaisesMenoresQue(double maxAlt) {
    return paises.stream()
            .filter(p -> p.altura() < maxAlt)
            .collect(Collectors.counting());
}
```

Colector que transforma y suma

`collect` `summingInt`

```
TipoIndicadoEnColector      collect(Collector)
```

El método

```
static <T> Collector<T,?,Integer>
   Collectors.summingInt(ToIntFunction<? super T> mapper)
```

permite sumar (Integer) los datos devueltos por mapper.

Ejemplo: devuelve el número de países con altura menor a una dada

Colector que transforma y suma

`collect`

`summingInt`

Ejemplo

```
TipoIndicadoEnColector collect(Collector)
```

```
static <T> Collector<T,?,Integer>
      Collectors.summingInt(ToIntFunction<? super T> mapper)
```

Número de paises con altura menor a una dada:

```
public int numIntPaisesMenoresQue(double d) {
    return paises.stream()
            .filter(p -> p.altura() < d)
            .collect(Collectors.summingInt(pais -> 1));
}
```

Existen tambien **summingLong** y **summingDouble**

Colector que particiona creando una correspondencia

`collect` `partitioningBy`

```
TipoIndicadoEnColector      collect(Collector)
```

El método

```
static <T> Collector<T,?,Map<Boolean,List<T>>>
   Collectors.partitioningBy(Predicate<? super T> predicate)
```

permite crear una correspondencia con claves true y false y valores una lista de los elementos que hacen el predicado true y false.

Ejemplo: crea una correspondencia separando los países que verifican o no un predicado dado.

Colector que particiona creando una correspondencia

`collect` `partitioningBy`

Ejemplo

```
TipoIndicadoEnColector      collect(Collector)
```

```
static <T> Collector<T,?,Map<Boolean,List<T>>>
     Collectors.partitioningBy(Predicate<? super T> predicate)
```

Crea una correspondencia separando los países que verifican o no un predicado dado:

```
public Map<Boolean, List<Pais>> separaPor(Predicate<Pais> pred) {
    return paises.stream()
            .collect(Collectors.partitioningBy(pred));
}
```

Colector que agrupa en una correspondencia

`collect` `groupingBy`

```
TipoIndicadoEnColector      collect(Collector)
```

El método

```
static <T,K> Collector<T,?,Map<K,List<T>>>
Collectors.groupingBy(Function<? super T,? extends K> classifier)
```

permite crear una correspondencia con los claves devueltas por la función classifier.

Ejemplo: Crear una correspondencia con la lista de países por continente

Colector que agrupa en una correspondencia

`collect` `groupingBy`

`TipoIndicadoEnColector collect(Collector)`

Ejemplo

```
static <T,K> Collector<T,?,Map<K,List<T>>>
  Collectors.groupingBy(Function<? super T,? extends K> classifier)
```

Correspondencia con la lista de países por continente:

```
public Map<String, List<Pais>> paisesPorContinente() {
    return paises.stream()
            .collect(Collectors.groupingBy(Pais::continente));
}
```

Colector que agrupa en una correspondencia

`collect` `groupingBy/2`

```
TipoIndicadoEnColector  collect(Collector)
```

El método

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
      Collectors.groupingBy(Function<? super T,? extends K> classifier,
                            Collector<? super T,A,D> downstream)
```

permite crear una correspondencia con las claves devueltas por la función classifier , y con los valores coleccionados por el colector downstream.

Ejemplo: Crear una correspondencia con el número de paises por continente.

Ejemplo: Crear una correspondencia con el conjunto de alturas por continente.

Colector que agrupa en una correspondencia

`collect` `groupingBy/2` Ejemplo

```
TipoIndicadoEnColector      collect(Collector)
```

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
      Collectors.groupingBy(Function<? super T,? extends K> classifier,
                            Collector<? super T,A,D> downstream)
```

Correspondencia con el número de países por continente:

```
public Map<String, Long> numPaisesPorContinente() {
    return paises.stream()
            .collect(Collectors.groupingBy(Pais::continente
                                           ,Collectors.counting()));
}
```

Colector que agrupa en una correspondencia

`collect` `groupingBy/2` Ejemplo

```
TipoIndicadoEnColector  collect(Collector)
```

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
      Collectors.groupingBy(Function<? super T,? extends K> classifier,
                            Collector<? super T,A,D> downstream)
```

Correspondencia con el conjunto de alturas por continente:

```
public Map<String, Set<Double>> alturasPorContinente() {
    return paises.stream()
            .collect(Collectors
                  .groupingBy(Pais::continente
                              ,Collectors.mapping(Pais::altura
                                                  ,Collectors.toSet())));
}
```

Colector que agrupa en una correspondencia

`collect` `groupingBy/2`

Ejemplo

```
TipoIndicadoEnColector  collect(Collector)
```

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
      Collectors.groupingBy(Function<? super T,? extends K> classifier,
                            Collector<? super T,A,D> downstream)
```

Correspondencia con el número de países por continente ahora devolviendo un Map<String,Integer>:

```
public Map<String, Integer> numIntPaisesPorContinente() {
    return paises.stream()
            .collect(Collectors.groupingBy(
                            Pais::continente
                            ,Collectors.summingInt(p -> 1)));
}
```

Colector que agrupa en una correspondencia

`collect`

`groupingBy/2`

Ejemplo

```
TipoIndicadoEnColector collect(Collector )
```

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
      Collectors.groupingBy(Function<? super T,? extends K> classifier,
                            Collector<? super T,A,D> downstream)
```

Correspondencia con el conjunto ordenado de continentes por altura:

```
public Map<Double, Set<String>> continentesPorAltura() {
    return paises
            .stream()
            .collect(Collectors.groupingBy(
                   Pais::altura
                  ,Collectors.mapping(Pais::continente,
                                      Collectors.toCollection(TreeSet::new))));
}
```

Colector que agrupa en una correspondencia

`collect` `groupingBy/3`

```
TipoIndicadoEnElColector collect(Collector)
```

El método

```
static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M>
Collectors.groupingBy( Function<? super T,? extends K> classifier,
                       Supplier<M> mapFactory,
                       Collector<? super T,A,D> downstream)
```

permite crear una correspondencia del tipo dado por mapFactory con las claves devueltas por la función y los valores obtenidos por el colector downstream.

Ejemplo: Correspondencia ordenada con conjunto de continentes ordenado por alturas

Colector que agrupa en una correspondencia

`collect`

`groupingBy/3`

```
TipoIndicadoEnElColector collect(Collector)
```

Ejemplo

```
static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M>
Collectors.groupingBy( Function<? super T,? extends K> classifier,
                       Supplier<M> mapFactory,
                       Collector<? super T,A,D> downstream)
```

Correspondencia ordenada con conjunto de continentes ordenado por alturas

```
public Map<Double, Set<String>> continentesPorAlturaOrd() {
    return paises.stream()
            .collect(Collectors.groupingBy(
                    Pais::altura,
                    TreeMap::new,
                    Collectors.mapping(Pais::continente
                                       ,Collectors.toCollection(TreeSet::new))));
}
```

Colector que genera un CharSequence

`collect` `joining`

```
TipoIndicadoEnElColector collect(Collector)
```

Los métodos

```
static Collector<CharSequence, ?, String>
                                        Collectors.joining()
static Collector<CharSequence, ?, String>
                        Collectors.joining(CharSequence delimiter)
static Collector<CharSequence, ?, String>
                        Collectors.joining(CharSequence del,
                                           CharSequence pre,
                                           CharSequence por)
```

crean un string concatenando los string del stream, el primero concatena todos los string, el segundo usa el delimitador intermedio y el tercero además pone un prefijo y sufijo.

Ejemplo: Crea un string con todos los continentes en la forma

$< int_1 - int_2 - int_3 - ... - int_n >$

Colector que genera un CharSequence

`collect`

`joining`

Ejemplo

```
TipoIndicadoEnElColector collect(Collector)
```

```
static Collector<CharSequence, ?, String>
                    Collectors.joining()
static Collector<CharSequence, ?, String>
                    Collectors.joining(CharSequence delimiter)
static Collector<CharSequence, ?, String>
                    Collectors.joining(     CharSequence del,
                                            CharSequence pre,
                                            CharSequence por)
```

Crea un string con todos los continentes en la forma

$< int_1 - int_2 - int_3 - ... - int_n >$

```
public String stringDeContinentes() {
    return paises.stream()
            .map(Pais::continente)
            .distinct()
            .collect(Collectors.joining("<", ",", ">"));
}
```

Creación de Stream sobre ficheros

- En la clase java.nio.file.Files tenemos los métodos de clase:

```
Stream<String> lines(Path)  // Stream de lineas
Stream<String> lines(Path, CharSet)  // Stream de lineas
Stream<Path>   list(Path)   // Stream de paths de ficheros
```

- En la clase java.io.BufferedReader tenemos el método:

```
Stream<String> lines(Path)  // Stream de lineas
```

- En estos casos, el stream debe cerrarse con close.
  - Mejor utilizarlo como un recurso Closeable dentro de un try

Creación de Stream sobre ficheros

- Ejemplo:
  - Crear un stream con las líneas contenidas en la sección **meta** de un fichero html.

```
Files.lines(htmlFile)
     .dropWhile(line -> !line.contains("<meta>")
     .skip(1)
     .takeWhile(line -> !line.contains("</meta>")
```

Ejemplos mas complejos

¿Qué devuelve el siguiente método?

```
public Map<String, Long>  nppl () {
      return paises.stream()
                  .map(Pais::continente)
                  .collect(
                        Collectors.groupingBy(
                              Function.identity(),
                              Collectors.counting()));
}
```

Ejemplos mas complejos

¿Qué devuelve el siguiente método?

```
public  Map<String, Map<Double, Long>> ' nppccua() {
    return paises.stream()
            .collect(Collectors.groupingBy(
                  Pais::continente,
                  Collectors.groupingBy(
                  pais -> ((int)(pais.altura()*10)/10.0,
                  Collectors.counting())));
}
```

Ejemplos mas complejos

¿Qué devuelve el siguiente método?

```
public  Map<Double, Map<String, List<Pais>>>  lppayc() {
    return paises.stream()
            .collect(Collectors.groupingBy(
                    pais -> ((int)(pais.altura()*10))/10.0,
                    Collectors.groupingBy(
                            Pais::continente)));
}
```

Ejemplos mas complejos

¿Qué devuelve el siguiente método?

```
public Map<Long, Set<Integer>>  nlpp() {
      Map<Integer, Long> map =
            paises.stream().collect(
            Collectors.groupingBy(pais->pais.nombre().length),
                        Collectors.counting()));
      return map.entrySet().stream()
            .collect(Collectors.groupingBy(
                  Map.Entry::getValue,
                  TreeMap::new,
                  Collectors.mapping(
                              Map.Entry::getKey,
                              Collectors.toSet())));
}
```

Rendimiento: Filtra y Seno

```
public static List<Double> filtraySenoIterador(List<Integer> li) {
	List<Double> res = new ArrayList<>();
	for (int i : li) {
		if (i % 2 == 0) {
			res.add(Math.sin(i));
		}
	}
	return res;
}

public static List<Double> filtraySenoSecuencial(List<Integer> li) {
	return li.stream()
		.filter(x -> x % 2 == 0).map(Math::sin).collect(Collectors.toList());
}

public static List<Double> filtraySenoParalelo(List<Integer> li) {
	return li.parallelStream()
		.filter(x -> x % 2 == 0).map(Math::sin).collect(Collectors.toList());
}
```

Rendimiento: Correspondencia

```
public static Map<Integer, Set<Integer>> correspondenciaIterador(List<Integer> li) {
        Map<Integer, Set<Integer>> map = new HashMap<>();
        for (int i : li) {
                int clave = i % 10;
                Set<Integer> set = map.computeIfAbsent(clave, HashSet::new);
                set.add(i);
        }
        return map;
}

public static Map<Integer, Set<Integer>>  correspondenciaSecuencial(List<Integer> li) {
        return li.stream()
                .collect(
                 Collectors.groupingBy(
                        x -> x % 10,
                        Collectors.toSet()));
}

public static Map<Integer, Set<Integer>> correspondenciaParalelo(List<Integer> li) {
        return li.parallelStream()
                        .collect(
                         Collectors.groupingBy(
                                x -> x % 10,
                                Collectors.toSet()));
}
```

Rendimiento: Máximo

```
public static int maximoIterador(List<Integer> li) {
      int m = Integer.MIN_VALUE;
      for (int dd: li)
            if (dd > m )
                  m = dd;
      return m;
}

public static int maximoSecuencial(List<Integer> li) {
      return li.stream().max(Integer::compare).get();
}


public static int maximoParalelo(List<Integer> li) {
      return  li.parallelStream().max(Integer::compare).get();
}
```

Rendimiento

MicroBechMark.
Se utiliza una lista de 4.000.000 enteros
Tiempo en milisegundos
Mac OSX Sequoia 15.0.1
16GB GB 2133 MHz LPDDR3
2,7 GHz Intel Core i7 (núcleos 4, hilos 16)
JDK-23

	filtraYSeno	Correspondencia	Máximo
Iterador	147,751 ± 33,662	805,200 ± 60,148	4,732 ± 0,212
Stream Secuencial	134,843 ± 31,635	730,351 ± 57,461	15,503 ± 0,220
Stream Paralelo	87,328 ± 33,427	888,045 ± 160,251	4,420 ± 0,393

Atención a los efectos laterales

```
public class Acumulador {
   public long total;

   public void agrega(long n) {
       total += n;
   }
}
```

Atención a los efectos laterales

Secuencial 500000500000
Paralelo 182332175326

```
public class StreamExamples {
	public static long acumulaSecuencial(long n) {
		Acumulador ac = new Acumulador();
		LongStream.rangeClosed(1,n).forEach(ac::agrega);
		return ac.total;
	}

	public static long acumulaParalelo(long n) {
		Acumulador ac = new Acumulador();
		LongStream.rangeClosed(1,n).parallel().forEach(ac::agrega);
		return ac.total;
	}
	public static void main(String[] args) {
		System.out.println("Secuencial " +
                                        acumulaSecuencial(1_000_000));
		System.out.println("Paralelo " + acumulaParalelo(1_000_000));
	}
}
```

Atención a los efectos laterales

```
public class Acumulador {
   public long total ;

   public synchronized void agrega(long n) {
       total += n;
   }
}
```

Atención a los efectos laterales

Secuencial 500000500000
Paralelo 500000500000

```
public class StreamExamples2 {
	public static long acumulaSecuencial(long n) {
		AtomicLong ac = new AtomicLong(0);
		LongStream.rangeClosed(1,n).forEach(ac::addAndGet);
		return ac.get();
	}

	public static long acumulaParalelo(long n) {
		AtomicLong ac = new AtomicLong(0);
		LongStream.rangeClosed(1,n).parallel().forEach(ac::addAndGet);
		return ac.get();
	}
	public static void main(String[] args) {
		System.out.println("Secuencial " +
                                        acumulaSecuencial(1_000_000));
		System.out.println("Paralelo " + acumulaParalelo(1_000_000));
	}
}
```

La interfaz Spliterator

- Una nueva definición de iterador que incorpora un nuevo método para iterar en sustitución de next y hashNext.

```
boolean tryAdvance(Consumer<? super T> accion)
```

Si hay elemento, lo consume y ejecuta la acción devolviendo true. En otro caso devuelve false.

- Otro método importante, divide el spliterador en dos: en el receptor y el devuelto. Si no es posible dividir devuelve null;

```
Spliterator<T> trySplit()
```

- Y un tercero por defecto que ejecuta un acción con todos los elementos restantes en el spliterator:

```
default void
    forEachRemaining(Consumer<? super T> accion)
```

La interfaz Collection

- Define un nuevo método por defecto.

```
Spliterator<T> spliterator()
```

- Por lo que todas las colecciones disponen de un spliterator.

Interfaces anidados en Spliterator

- Spliterator para tipos básicos:

```
static interface   Spliterator.OfDouble

static interface   Spliterator.OfInt

static interface   Spliterator.OfLong
```

Caracteríticas de un Spliterator

- Un Spliterator puede incorporar las siguientes características (static int):

CONCURRENT. Los elementos pueden ser tratados concurrentemente.

DISTINCT. Todos los elementos son distintos (según equals)

IMMUTABLE La fuente del spliterator no se modificará.

NONNULL. No hay elementos nulos.

ORDERED. Los elementos tienen definido un orden de secuencia.

SIZED Se conoce el número de elementos del spliterator.

SORTED Los elementos están ordenados.

SUBSIZED Se conocen los tamaños de los Spliterator que resultan de utilizar trySplit().

Creación de un Stream a partir de un Spliterator

- La clase StreamSupport (java.util.stream) permite generar un stream a partir de un Spliterator o de un Supplier de Spliterator:

```
static <T> Stream<T>
    stream(Spliterator<T> spliterator, boolean parallel)

static <T> Stream<T>
      stream(Supplier<? extends Spliterator<T>> supplier,
                                int characteristics, boolean parallel)
```

```
static <T> Stream<T>   intStream(SlitIterator.OfInt d, boolean parallel)
static <T> Stream<T>   longStream(SlitIterator.OfLong d, boolean parallel)
static <T> Stream<T>   doubleStream(SlitIterator.OfDouble d, boolean parallel)
```

Y también con Supplier

Spliterator y Stream (4 hilos)

```
Stream<String> ss =
      Arrays.asList("hola", "a", "todos", "como", "están",
                    "nosotros", "muy", "bien").parallelStream();

int total = ss.reduce(0, (a,b) -> a + b.length(), Integer::sum)
```

hola a todos como están nosotros muy bien

hola a todos como | están nosotros muy bien

hola a | todos como | están nosotros | muy bien

hola | a | todos | como | están | nosotros | muy | bien

(0, hola) **(0, a)** **(0, todos)** **(0, como)** **(0,estan)** **(0, nosotros)** **(0, muy)** **(0, bien)**

4 **1** **5** **4** 5 8 **3** **4**

5 **9** 13 **7**

14 **20**

34

MiArrayList

```
public class MiArrayList<T> {
	protected T[] elements;
	protected int size;
	public MiArrayList() {
		elements = (T[]) new Object[10];
		size = 0;
	}
	private void ensureCapacity() {
		if (size >= elements.length) {
			elements = Arrays.copyOf(elements, 2*elements.length);
		}
	}
	public void append(T x) {
		ensureCapacity();
		elements[size] = x;
		size++;
	}
	public Spliterator<T> spliterator() {
		return new SpliteratorArrayList(0, size);
	}

	public Stream<T> stream() {
			return StreamSupport.stream(spliterator(), false);
	}
	...
```

Spliterator para MiArrayList

```
...
private class SpliteratorArrayList implements Spliterator<T> {
        int ini; // inclusive
        int fin; // exclusive
        private static final int MIN_SIZE_TO_SPLIT = 4;
        public SpliteratorArrayList(int i, int f) {
                ini = i;
                fin = f;
        }
        public long estimateSize() {
                return fin - ini;
        }
        public int characteristics() {
                return CONCURRENT | NONNULL | ORDERED | SIZED | SUBSIZED;
        }
        public boolean tryAdvance(Consumer<? super T> action) {
                boolean res = ini < fin;
                if (res) {
                        action.accept(elements[ini]);
                        ini++;
                }
                return res;
        }
        ...
```

Spliterator para MiArrayList

```
        ...
        public Spliterator<T> trySplit() {
            Spliterator<T> split = null;
            if (fin - ini  >= MIN_SIZE_TO_SPLIT ) {
                int num = (fin - ini) / 2;
                split =
                   new SpliteratorArrayList(ini, ini + num);
                ini = ini + num;
            }
            return split;
        }
    }
}
```

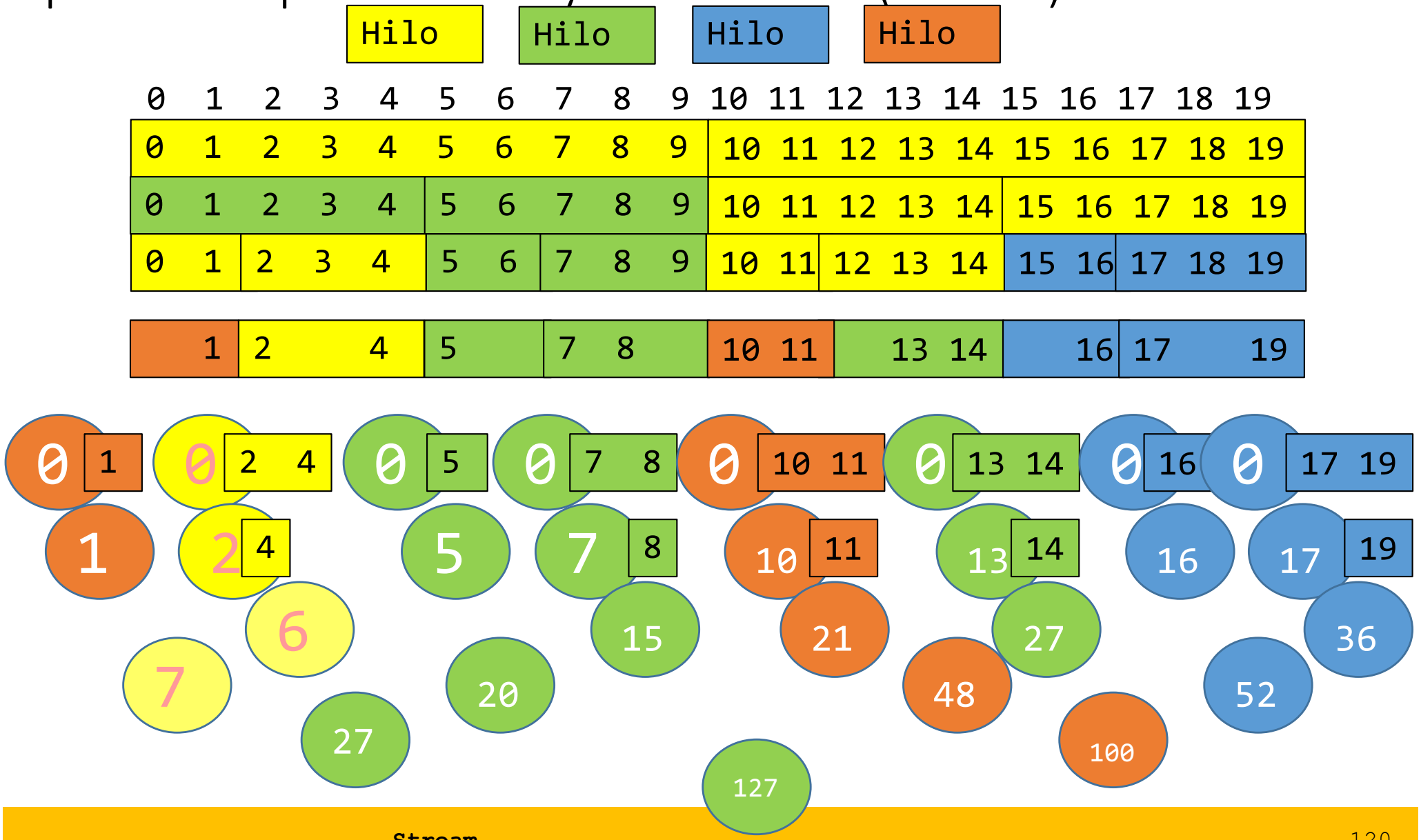
Demo de Spliterator para MiArrayList

```
public class MiArrayListSplitDemo {

    public static void main(String [] args) {
        MiArrayList<Integer> al = new MiArrayList<>();
        IntStream.range(1,20).forEach(al::append);

        int sum =
          al.parallelStream()
            .filter(x -> x % 3 != 0))
            .reduce(0, Integer::sum, Integer::sum);
    }
}
```

Spliterator para MiArrayList. Stream (4 hilos)



Operación no incluida en Java

- Hay una operación interesante que Java no incorpora:

```
public class Util {

    public static Stream<C>
          zipWith(Stream<? extends A> a,
                  Stream<? extends B> b,
                  BiFunction<? super A, ? super B,
                             ? extends C> zipper) {

          ...
     }

}
```

Operaciones no incluidas en Java: zipWith

```
public static <A, B, C> Stream<C> zipWith(Stream<? extends A> a, Stream<? extends B> b,
                    BiFunction<? super A, ? super B, ? extends C> zipper) {
          Objects.requireNonNull(zipper);
          Spliterator<? extends A> aSpliterator = Objects.requireNonNull(a).spliterator();
          Spliterator<? extends B> bSpliterator = Objects.requireNonNull(b).spliterator();

          // Zipping looses DISTINCT and SORTED characteristics
          int both = aSpliterator.characteristics() & bSpliterator.characteristics()
                              & ~(Spliterator.DISTINCT | Spliterator.SORTED);
          int characteristics = both;

          long zipSize = ((characteristics & Spliterator.SIZED) != 0)
                    ? Math.min(aSpliterator.getExactSizeIfKnown(), bSpliterator.getExactSizeIfKnown()) : -1;

          Iterator<A> aIterator = Spliterators.iterator(aSpliterator);
          Iterator<B> bIterator = Spliterators.iterator(bSpliterator);
          Iterator<C> cIterator = new Iterator<C>() {
                    @Override
                    public boolean hasNext() {
                              return aIterator.hasNext() && bIterator.hasNext();
                    }

                    @Override
                    public C next() {
                              return zipper.apply(aIterator.next(), bIterator.next());
                    }
          };

          Spliterator<C> split = Spliterators.spliterator(cIterator, zipSize, characteristics);
          return StreamSupport.stream(split, a.isParallel() || b.isParallel());

}
```