UNIDAD 9

LA API STREAM

BIBLIOGRAFÍA:
- Java 8 in Action, "Raoul-Gabriel Urma", Mannion Publications, 2015.
- Curso de Open webinnars, 2020.

9.1. INTRODUCCIÓN.

La gran mayoría de aplicaciones Java utilizan colecciones de datos de forma intensiva. Es una forma razonable de tener almacenados los datos que el programa debe procesar de alguna manera para realizar una tarea.

Sin embargo, procesar los datos de las colecciones, pese a ser algo necesario, tiene sus peros desde el punto de vista de un programador:

- Muchas veces necesitas hacer procesos similares a los que se realizan en las BD relacionales con SQL, y por tanto debes filtrar datos (WHERE...), agruparlos (GROUP BY...), ordenarlos (ORDER BY), etc. Sin embargo en SQL indicas lo que quieres hacer (es declarativo) y se hace, pero con colecciones (el trabajo es procedural) debes programar las operaciones cada vez que las hagas usando iteradores -> aumenta la complejidad -> aumenta el tiempo de implementación -> baja el rendimiento de programación. **¿Porqué no hacer algo similar a lo tenemos en SQL pero con las colecciones de datos?**
- Si necesitas eficiencia cuando trabajas con colecciones con muchos datos, la mejor opción es paralelizar los trabajos, pero desarrollar código paralelo con iteradores es complicado y depurarlo no es nada divertido. **¿Y si algo fuese capaz de paralelizar el procesamiento a los datos de forma automática y sencilla?**

Bueno, pues en Java 8, los desarrolladores de Java ofrecen el API Stream que podemos definir así: Es una actualización de Java que permite manipular las colecciones de datos de forma declarativa, paralelizable de forma transparente y donde las operaciones se pueden encadenar.

EJEMPLO 1: imagina que tienes una lista con los platos que es capaz de hacer una empresa de catering:

```
public class Plato {
    public static enum Tipo {CARNE, PESCADO, OTRO};

    private final String nombre;
    private final boolean vegano;
    private final int calorias;
    private final Tipo tipo;

    public Plato(String nombre, boolean vegano, int calorias, Plato.Tipo t){
        this.nombre= nombre;
        this.vegano= vegano;
        this.calorias= calorias;
        this.tipo= t;
    }

    public String getNombre() {  return nombre; }
    public boolean esVegano() { return vegano; }
    public int getCalorias() { return calorias; }
    public Tipo getTipo() { return tipo; }

    @Override
    public String toString() {  return nombre; }
}

List<Plato> menu = Arrays.asList(
        new Plato("cerdo", false, 800, Plato.Tipo.CARNE),
        new Plato("cordero", false, 700, Plato.Tipo.CARNE),
        new Plato("pollo", false, 400, Plato.Tipo.CARNE),
        new Plato("patatas fritas", true, 530, Plato.Tipo.CARNE),
        new Plato("arroz", true, 350, Plato.Tipo.OTRO),
        new Plato("fruta de temporada", true, 120, Plato.Tipo.OTRO),
        new Plato("pizza", true, 550, Plato.Tipo.OTRO),
        new Plato("trucha", false, 300, Plato.Tipo.PESCADO),
        new Plato("salmón", false, 450, Plato.Tipo.PESCADO)
);
```

Para que veas las ventajas de conocer la API Streams, vamos a hacer un par de ejemplos con estos datos. En primer lugar, una operación sencilla: supón que es necesario generar una lista con los nombres de los platos bajos en calorías (menos de 400 calorías) ordenados ascendentes (de menor a mayor) por las calorías.

Con colecciones	Con Streams

`List<Plato> ligeros = new ArrayList<>();`
 	
`// Filtrar datos usando un acumulador`
`for(Plato p: menu) {`
`  if( p.getCalorias() < 400 ) {`
`  ligeros.add( p );`
`  }`
`}`
`// Ordenar por Calorías`
`Collections.sort(ligeros,`
`  new Comparator<Plato>{`
`  public int compare(Plato p1, Plato p2){`
`  return Integer.compare(p1.getCalorias(),`
`  p2.getCalorias()`
`  );`
`  }`
`  }`
`  );`
`// Procesar lista para generar lista nombres`
`List<String> nombres= new ArrayList<>();`
`for(Plato p: ligeros) {`
`  nombres.add( p.getNombre() );`
`}`
	`import java.util.Comparator.Comparing;`
	`import java.util.stream.Collectors.toList;`
	
	`List<String> nombres =`
	`  menu.stream()`
	`  .filter(p -> p.getCalorias() < 400 )`
	`  .sorted(comparing(Plato::getCalorias))`
	`  .map(Plato::getNombre)`
	`  .collect( toList() );`
	`//Y si necesitas paralelismo, cambias`
	`//  menu.stream() por`
	`//  menu.parallelStream()`

Si observas el código de ambas formas de hacerlo, verás que con streams, procesas los datos encadenando operaciones de más alto nivel (filtrar, agrupar, ordenar, etc.) igual que ocurre con SQL por ejemplo. Pero además puedes encadenarlas como te interese. Esto significa que el trabajo de procesar los datos lo haces más rápido y más fácil, necesitas menos código (menos errores) y te puedes beneficiar de operaciones avanzadas implementadas por los equipos de desarrollo de Java, como paralelizar código.

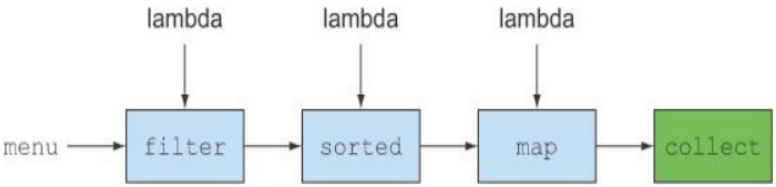


Figura 1: Encadenar operaciones de alto nivel.

9.3. STREAMS.

9.3.1. CONCEPTO Y COMPARACIÓN CON COLECCIONES.

Todas las clases que heredan de Collection soportan un método **stream()** definido en una interface disponible desde **java.util.stream.Stream**. No es la única forma de fabricar un stream. Pero, ¿Qué es un stream?

CONCEPTO DE STREAM

Un stream es una secuencia de elementos que vienen de un origen y que soporta operaciones de procesamiento. En detalle:

- **Secuencia de elementos**: como una colección, un stream afrece una interface para acceder a un conjunto de elementos, pero mientras las estructuras se centran en la complejidad de almacenarlos y accederlos de forma eficiente, los stream se preocupan de ofrecer operaciones de alto nivel como filtrado, ordenación, mapeo, etc.
- **Origen**: de donde provienen los elementos de un stream, normalmente arrays, colecciones de datos o recursos de entrada/salida.
- **Operaciones de procesamiento**: soportan operaciones similares a las bases de datos y a los lenguajes funcionales, operaciones como: filter, map, reduce, find, match, sort, etc. Se pueden ejecutar secuencialmente o en paralelo y tienen además 2 características interesantes:
  - **Pipelining:** el resultado de muchas operaciones es otro stream, lo que permite encadenar unas operaciones con otras.
  - **Iteración interna**: al contrario que las colecciones (iteran por los elementos usando un iterador externo y explícito) los stream iteran de forma interna.

EJEMPLO 2: obtener los nombres de los platos con más de 300 calorías de un menú (una lista de objetos Plato) y guardarlo en una lista de Strings.

```
import java.util.stream.Collectors.toList;

List<String> caloricos = menu.stream()
                              .filter( d -> d.getCalorias() > 300 )
                              .map( Plato::getNombre )
                              .limit(3)
                              .Collect( toList() );
System.out.println( caloricos );   // [cerdo, cordero, pollo]
```

DIFERENCIAS CON COLECCIONES

Al contrario que una colección, un stream no tiene porqué mantener todos sus elementos disponibles. Piensa en una película como en un stream o una colección de fotogramas.

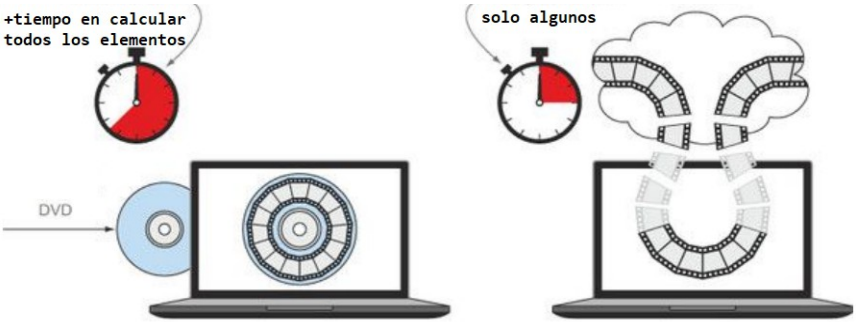


Figura 2: Colección vs stream.

Un DVD puede tener almacenada una película (toda la película normalmente). Es como una estructura de datos o colección. Cuando ves la misma película por streaming, solo unos pocos fotogramas del total están disponibles en el reproductor de tu equipo.

Otro ejemplo podría ser lo que ocurre cuando haces una búsqueda en

un buscador. No obtienes todos los recursos que coinciden (tardaría mucho). Solo recibes cuatro o cinco resultados y tienes la posibilidad de pedir los siguientes (un stream de resultados).

Desde un punto de vista filosófico, una colección es una secuencia de datos existente en el mismo tiempo pero dispersa en el espacio, mientras que un stream es la misma secuencia de datos en un mismo espacio, que está dispersa en el tiempo.

Otra diferencia es que para recorrer los elementos de una colección se usan iteradores externos a la colección, mientras que en el caso de los streams son internos.

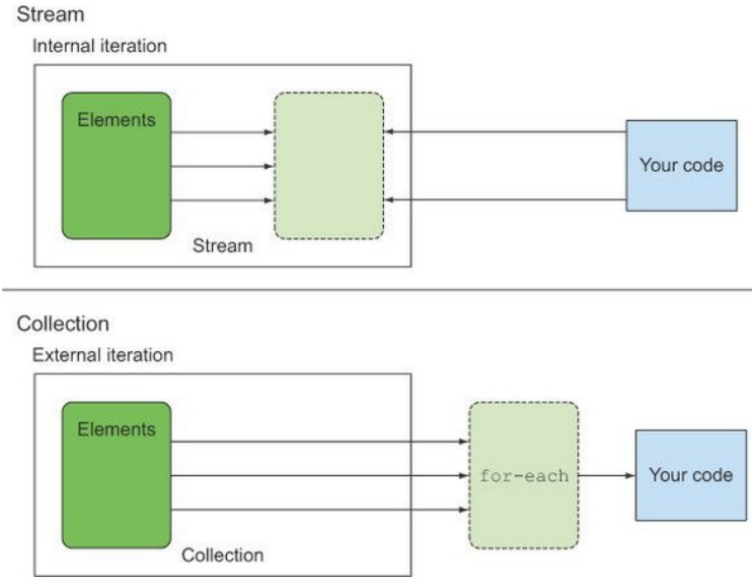


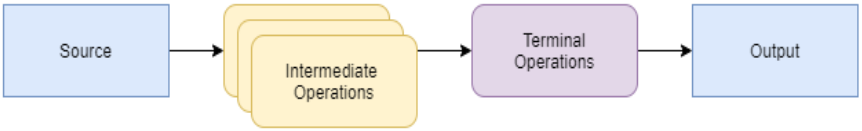
Figura 4: iteración interna (stream) / externa (colección).

Esto hace que a un stream no se le puedan añadir o eliminar elementos al contrario que a una colección. Además no todos los elementos del

stream tienen que existir cuando se empieza a trabajar con ellos.
La forma de trabajar con ellos puede esquematizarse así: tenemos el origen (de donde vienen los elementos del stream), y una serie de operaciones que podemos clasificar como:

- **Intermedias**: transforman un stream en otro stream.
  - **map()**: mapea, traduce o cambia los elementos de un stream.
  - **filter()**: filtra elementos del stream.
  - **distinct()**: elimina elementos duplicados de un stream.
  - **sorted()**: ordena los elementos de un stream.
  - **limit()**: devuelve otro stream limitado en tamaño a lo que indica el parámetro **maxSize**.
  - **skip()**: salta n elementos del stream.
- **Terminales**: acaban (consumen) el stream generarando una salida (a veces podría ser un stream almacenado en una colección o valores construidos a partir de varios elementos de un stream).
  - **collect()**: convierte un stream en una colección.
  - **toArray()**: convierte un stream en un array.
  - **count()**: cuenta los elementos del stream
  - **reduce()**: transforma elementos de un stream en un único valor.
  - **min()**: se queda con el mínimo valor del stream desde el punto de vista de un Comparator.
  - **max()**: devuelve el máximo elemento del stream (como min).
  - **AnyMatch()**: devuelve true si alguno de los elementos casa con el predicado indicado. Si es vacío devuelve false.
  - **allMatch()**: devuelve true si todos los elementos casan con el predicado que se pasa. Si el stream es vacío devuelve true.
  - **noneMatch()**: devuelve true si ninguno de los elementos casa con el predicado y true si está vacío.

- **findAny()**: devuelve un Optional con un elemento del stream.
- **findFirst()**: Devuelve el primer elemento del stream.



9.3.2. GENERAR STREAMS.

La manera más habitual es procesar los datos de una colección con streams usando el método **stream()** de las colecciones, pero también puedes generar rangos de valores numéricos con **range()** y **rangeClosed()** de **IntStream**, o desde los elementos de un array, un fichero o funciones genéricas que creen streams infinitos (que nuca se acaban).

STREAMS DESDE VALORES

Usando el método estático **Stream.of(lista de valores)**, puedes generar un stream con los valores que pasas como parámetro. También puedes generar un stream vacío usando el método **Stream.empty()**.

EJEMPLO 3: Imprimir los valores de varias cadenas

```
Stream<String> s = Stream.of("Java 8 ", "Lambdas ", " Y ", "Streams");
s.map(String::toUpperCase).forEach(System.out::println);
Stream<String> sVacio = Stream.empty();
```

STREAMS DESDE ARRAYS

A partir de un array a, puedes crear un stream con el método estático **Arrays.stream(a)**.

EJEMPLO 4: sumar un array de números.

```
int[] numeros = { 1, 2, 3, 4, 5 };
int suma= Arrays.stream( numeros ).sum();
```

STREAM DESDE FILES

El API de Java NIO (non-blocking I/O), usada pra operaciones con ficheros tiene muchos métodos que devuelven un stream como **Files.lines()**, que devuelve un stream de strings con las líneas de un fichero de texto.

EJEMPLO 5: contar las palabras distintas de un fichero.

```
long pu = 0;
try(Stream<String> lineas = Files.lines(Paths.get("datos.txt"),
                                        CharSet.defaultCharSet() ) {
    pu = lines.flatMap( linea -> Arrays.stream( linea.split("") ) )
              .distinct()
              .count()
}
catch(IOException e) { ... }
```

STREAM DESDE FUNCIONES

Stream tiene dos métodos estáticos para generar un stream desde una función: **Stream.iterate()** y **Stream.generate()**. Crean streams infinitos, que no tienen un número fijo de elementos, van creando elementos a medida que se necesitan al contrario que una colección de datos que los tiene creados previamente.

Normalmente interesa usar cierta cantidad limitada de ellos usando operaciones como **limit(n)**. El método **iterate()** nos sirve para ver un sencillo ejemplo.

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach( System.out::println );
```

- **iterate()**: acepta dos parámetros: un valor inicial y una operación lambda de tipo operador unario T -> f(T) que lo va transformando. En la siguiente iteración, el valor inicial será el

valor producido en la iteración actual. En el ejemplo anterior el valor inicial es 0, en la siguiente iteración cambia a 0 + 2 = 2, generando la serie: 0 -> 2 -> 4 -> 6 -> 8 -> ... -> 16 -> 18

- **generate()**: La operación generate() es similar y acepta un solo parámetro que es un **Supplier<T>**. Ejemplo:

```
// Genera 10 valores 1
Stream<Integer> si = Stream.generate( () -> 1).limit(10);
si.forEach( System.out::println );
```

EJERCICIO 1: Generar la serie de Fibonacci mediante iterate() de Streams.

Nota: la serie de Fibonacci es: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.... Los 2 primeros números de la serie son 0 y 1 y cada siguiente elemento es la suma de los dos anteriores. Podemos generar una primera pareja y luego pensar una lambda que genere la siguiente. La primera pareja será (0,1) y la siguiente (1,1) que es ((el 1 de la anterior), (la suma de los 2 anteriores 0+1)).

a) Debes generar (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21).... Para representar cada pareja usamos un array de 2 enteros que iremos creando en cada iteración. Debes pensar en la operación para generar el siguiente elemento: ???

```
Stream.iterate(new int[]{0, 1}, ???)
      .limit(20)
      .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

b) Si en vez de imprimir las parejas quieres imprimir la serie, mapea con map() cada tupla t a t[0].

El método **generate()** es similar a iterate(), pero produce un elemento cuando se le pide, no de forma iterativa, acepta una función de tipo

Supplier<T> para aportar nuevos valores.

EJEMPLO 6: Generar 5 números aleatorios.

```
Stream.generate( Math::random )
      .limit(5)
      .forEach( System.out::println );
```

EJERCICIO 2: Añade las cadenas "uno", "dos", "tres", "cuatro" y "dos" a una colección de tipo lista.

a) Crea un stream de la lista e imprime los elementos del stream usando el método `forEach()`.

b) Ordena los elementos con `sorted()` antes de imprimirlos.

c) Convierte (transforma=mapea) cada elemento a mayúscula antes de imprimirlo.

d) Filtra elementos que comiencen con la letra 'T'.

e) Elimina elementos repetidos.

9.3.3. OPERACIONES BÁSICAS.

Ya sabes que en un stream hay dos grandes categorías de operaciones:

- **Internas o intermedias**: generan otro stream como resultado. Son encadenables o pueden formar un pipeline de operaciones.
- **Finales o terminales**: cierran el stream y hacen que se ejecute. No devuelven como resultado un stream.

EJEMPLO 7: En el stream del ejemplo, indicar cuales son operaciones intermedias y cuales finales.

```
List<String> nombresTop3Calorias= menu.stream()
                                      .filter( d -> d.getCalorias() > 300)
                                      .map( Plato::getNombre )
                                      .limit(3)
                                      .Collect( toList() );
System.out.println( nompresTop3Calorias );   // [cerdo, cordero, pollo]
```

- Intermedias: filter, map y limit.
- Finales: collect

EJERCICIO 3. ¿Cuales son las operaciones terminales e intermedias en este stream?

```
long contador = menu.stream().
                        .filter( d -> f.getCalorias() > 300 )
                        .distinct()
                        .limit(3)
                        .count();
```

El trabajo con streams tiene una estructura de 3 pasos:

1. Hacer una consulta al origen de los elementos.
2. Encadenar varias operaciones intermedias.
3. Aplicar una operación final para que se ejecute el stream.

Tabla 2. Operaciones intermedias(todas devuelven Stream<T>).

Operación	Argumento	Descriptor de Función
filter	Predicate<T>	T -> boolean
map	Function<T,R>	T -> R
limit		
sorted	Comparator<T>	(T,T) -> int
distinct		

Tabla 3. Operaciones finales (ninguna devuelve stream).

Operación	Descriptor
forEach	Consume un elemento y aplica un lambda
count	Cuenta nº elementos
collect	Reduce el stream para crear colecciones como List, Map, etc.

9.3.2.1. FILTRADO.

La operación intermedia **filter()** acepta un predicado y devuelve otro stream con los elementos que cumplen el predicado.

Filtrar con un predicado.

Quedarnos con los platos que sean veganos:

```
List<Plato> vegetarianos= menu.stream()
                              .filter(Plato::esVegano)
                              .collect( toList() )
```

Eliminar objetos repetidos.

Un objeto se considera repetido si coinciden en **hashCode()** y **equals()**. El método **distinct()** se encarga de eliminar los repetidos del stream. Ejemplo: quedarse con los números pares sin repetidos:

```
List<Integer> numeros = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numeros.stream()
       .filter(i -> i % 2 == 0)
       .distinct()
       .forEach( System.out::println );
```

Truncar el stream

Para quedarnos solamente con n elementos del stream, podemos usar el método limit(n). Ejemplo: los 3 platos más calóricos:

```
List<Plato> platos = menu.stream()
                         .filter(d -> d.getCalories() > 300)
                         .limit(3)
                         .collect( toList() );
```

Saltar elementos

También podemos no coger por el principio un número de elementos, saltarlos con el método **skip(n)**.

```
List<Plato> platos = menu.stream()
                         .filter(d -> d.getCalories() > 300)
                         .skip(2)
                         .collect( toList() );
```

EJERCICIO 4: Muestra mediante un stream los primeros 2 platos de carne.

9.3.2.2. MAPEO.

En SQL puedes indicar seleccionar no todas las columnas de las filas de una tabla, sino solamente las que te interesen en cierto momento o incluso hacer aparecer nueva información que se genera a partir de la información existente (columnas calculadas). Esta misma operación, seleccionar solo ciertas características de un objeto o hacer aparecer nuevos datos a partir de los que hay, en el API stream, se realizan con **map()** y **flatmap()**.

El método **map()** admite una función de parámetro que se aplica a cada elemento del stream de datos. Mapear es "crear una nueva versión de" cada objeto.

EJEMPLO 8: Obtener los nombres de los platos.

```
// El resultado es un Stream<String> porque getNombre() devuelve String
List<String> nombrePlatos= menu.stream()
                               .map( Plato::getNombre )
                               .collect( toList() );
```

EJEMPLO 9: Obtener la longitud de las cadenas de un Stream<String>.

```
List<String> palabras= Arrays.asList("Java8", "Lambdas", "Ole", "Ole");
List<Integer> longitudes= palabras.stream()
                                  .map( String::length )
                                  .collect( toList() );
```

EJEMPLO 10: Encadenar operaciones map. Primero sacamos el nombre de los platos y luego la longitud de los nombres.

```
List<String> nombrePlatos= menu.stream()
                               .map( Plato::getNombre )
                               .map( String::length )
                               .collect( toList() );
```

Imagina que ahora queramos obtener las letras no repetidas que aparecen en las palabras de un stream de strings. Podrías pensar en separar las letras de cada palabra en un array de esta forma:

```
palabras.stream()
        .map( w -> w.split("") )
        .distinct()
        .collect( toList() );
```

Pero así mapeas cada palabra a un **String[]** (array de String) y por tanto generas un stream de tipo **Stream<String[]>**. Pero lo que quieres es un **Stream<String>** donde cada string sea una letra.

Necesitas un un stream de chars no un stream de arrays. Hay un método llamado **Arrays.stream()** que acepta un array y produce un stream:

```
String[] arrayPalabras= {"Adiós", "Mundo"};
Stream<String> streamPalbras= Arrays.stream(arrayPalabras);
```

Si lo usas en el pipeline anterior:

```
palabras.stream()
        .map( w -> w.split("") )
        .map( Arrays::stream )
        .distinct()
        .collect( toList() );
```

Tampoco acaba de funcionar porque generas un stream de streams, es decir obtienes **Stream<Stream<String>>**.

La solución es usar **flatMap()**, que combina el resultado no con el stream de salida, sino con el contenido de cada stream que recibe. Lo que hace es eliminar los diferentes streams que recibe combinándolos en un solo stream de salida.

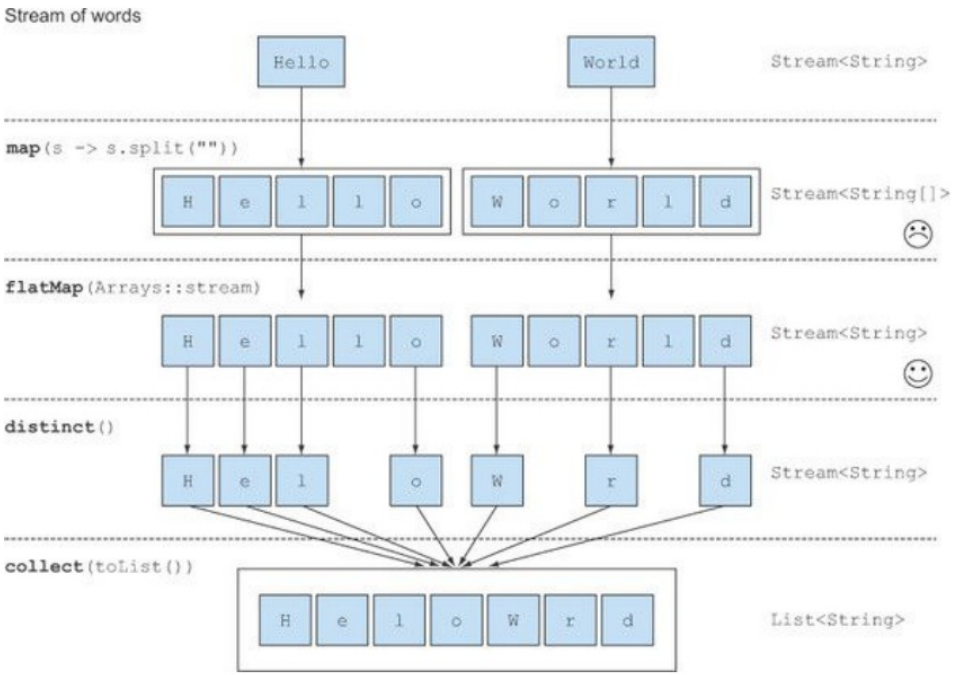


Figura 5: Ejemplo de Aplicar flatMap().

EJEMPLO 11: usando flatMap() para unir varios stream en un elemento de otro.

```
palabras.stream()
        .map( w -> w.split("") )
        .flatMap( Arrays::stream )
        .distinct()
        .collect( toList() );
```

EJERCICIO 5. Resuelve estos problemas:

a) Dada una lista de números, por ejemplo [1, 2, 3, 4], devuelve el cuadrado de cada uno. Deberías obtener [1, 4, 9, 16, 25].

b) Dadas dos listas de números, ¿Cómo devolver una lista de todas las posibles parejas de sus elementos? Por ejemplo, dadas las listas [1, 2, 3] y [3, 4] deberías obtener [(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]. Por sencillez, representa cada pareja como un array de 2 enteros.

c) Devolver parejas de todos los números, pero solo las parejas cuya suma sea divisible por 3. Por ejemplo:,(2, 4) y (3, 3) serían válidas

9.3.2.3. FINDING Y MATCHING.

Para comprobar si hay elementos de un conjunto que cumplen ciertas propiedades, el API streams ofrece los métodos allMatch, anyMatch, noneMatch, findFirst y findAny.

AL MENOS UN ELEMENTO

El método **anyMatch()** sirve para averiguar si al menos uno de los elementos cumple un predicado. Es una operación terminal que devuelve un booleano.

EJEMPLO 28: Imprimir si el menú tiene algún plato vegetariano.

```
if( menu.stream().anyMatch( Plato::isVegeno) ) {
    System.out.println("Menú incluye algún plato vegetariano!!");
}
```

TODOS LOS ELEMENTOS

El método **allMatch()** es similar pero devuelve true si todos los elementos cumplen el predicado.

EJEMPLO 29: Indicar si todos los platos del menú son saludables (menos de 1000 calorías).

```
boolean saludable= menu.stream().allMatch( p -> p.getCalorias() < 1000 );
```

NINGÚN ELEMENTO

El opuesto de allMatch es **noneMatch()**. Devuelve true si ningún elemento del stream cumple la condición.

EJEMPLO 30: Indicar si todos los platos del menú son saludables (no hay ninguno con 1000 o más calorías).

```
boolean saludable= menu.stream().noneMatch( p -> p.getCalorias() >= 1000 );
```

Estas 3 operaciones (anyMatch, allMatch y noneMatch) usan evaluación en cortocircuito, es decir, a veces no necesitan procesar todo el stream para saber el resultado que deben devolver. Por ejemplo en anyMatch(), en el momento en que encuentre el primer elemento que cumple el predicado, devuelve true.

BUSCAR UN ELEMENTO Y LA CLASE OPTIONAL

El método **findAny()** devuelve un elemento arbitrario del stream actual. Se utiliza en combinación con otras operaciones.

EJEMPLO 31: encontrar cualquier plato que sea vegetariano.

```
Optional<Plato> p= menu.stream()
                       .filter( Plato::esVegeno)
                       .findAny();
```

¿Qué es Optional en el código anterior?
La clase **Optional<T>** se encuentra en **java.util.Optional** y es una clase contenedora que representa la existencia o ausencia de un valor. En el ejemplo anterior, si el método **findAny()** no encuentra ningún elemento, para evitar devolver **null** (que puede ser una fuente de errores), siempre se devuelve un **Optional** en el que puedes comprobar sin errores si tiene valor o no mediante estos métodos:

- **isPresent()** devuelve true si contiene un valor.
- **ifPresent(Consumer<T> block)** ejecuta el bloque de código si hay un valor.
- **T get()** devuelve el valor si está presente o en otro caso lanza la excepción NoSuchElement.
- **T orElse(T other)** devuelve un valor si está presente o en otro caso devuelve el valor por defecto **other**.

En el ejemplo anterior, necesitas explícitamente comprobar si hay un plato en el objeto Optional devuelto para acceder a su nombre:

EJEMPLO 32: encontrar cualquier plato que sea vegetariano e imprimir su nombre.

```
Optional<Plato> p = menu.stream()
                        .filter( Plato::esVegeno)
                        .findAny()
                        .ifPresent( p -> System.out.println( p.getNombre() ) );
```

BUSCAR EL PRIMER ELEMENTO

Cuando en un stream los elementos vienen en cierto orden, tiene sentido estar interesados en algunos elementos que ocupen posiciones determinadas. Por ejemplo, encontrar el primer elemento lo realiza el método **findFirst()**. Es parecido a **findAny()**.

EJEMPLO 33: dada una lista de números, encuentra el cuadrdado que sea divisible por 3.

```
List<Integer> li= Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> resultado= li.stream()
                              .map( x -> x * x )
                              .filter( x -> x % 3 == 0 )
                              .findFirst();   // 9
```

Nota: si vas a usar paralelismo, es mejor usar **findAny()**.

9.3.2.4. REDUCING.

Ya hemos visto que algunos métodos finales, reducen todo el stream a un valor boolean como anyMatch(), etc. Ahora incrementamos la complejidad de las consultas, realizando operaciones de reducción o de resumen (resumir la información) y en los siguientes apartados aprenderemos a realizar operaciones más complejas aún con collect(), etc.

RESUMIR ELEMENTOS

Si queremos calcular la suma de los números que tiene una lista podemos usar este código por ejemplo:

```
int sum = 0;
for (int x : lista) { sum += x; }
```

Cada elemento es combinado repetidamente mediante el operador suma con el resultado de procesar los elementos anteriores. De esta forma reduces una lista de valores a un solo valor. Además se utiliza:

- Un valor inicial de la variable que resume la lista
- La operación mediante la cual se combinan los elementos.

El método **reduce()** permite transformar los elementos de una lista en un único valor, utilizando un valor inicial del resultado, y aplicando una operación de forma repetida a los elementos. La versión del código anterior:

```
int sum = lista.stream().reduce(0, (a, b) -> a + b);
```

El método tiene 2 argumentos:

- El primero es un valor inicial.
- El segundo es una operación binaria **BinaryOperator<T>** que combina dos elementos para producir un nuevo valor, en el ejemplo (a, b) -> a + b.

Cuando llega el primer elemento del stream, el primer parámetro de la expresión lambda (a) es el valor inicial y (b) es el elemento. Se combinan y la siguiente vez (a) es el valor actual del proceso y (b) el siguiente elemento, se combinan, y así repetidamente.

Puedes hacer este código más conciso usando referencia a métodos, porque la clase Integer tiene un método static que suma dos números:

```
int sum= lista.stream().reduce(0, Integer::sum);
```

SIN VALOR INICIAL

Hay una variante de **reduce()** que no utiliza valor inicial, pero en ese caso devuelve un objeto **Optional** por si acaso el pipeline de operaciones tiene un filtrado por ejemplo y no tiene elementos que combinar:

```
Optional<Integer> sum = lista.stream().reduce( (a, b) -> (a + b) );
```

MÁXIMOS Y MÍNIMOS

Es similar:

```
Optional<Integer> max = lista.stream().reduce( Integer::max );
Optional<Integer> min = lista.stream().reduce( Integer::min );
```

También puedes pasar un lambda pero la referencia es más legible (en vez de Integer::min puedes usar (x,y) -> x < y ? x:y).

EJERCICIO 6: ¿Cómo podrías contar el número de platos en el menú usando operaciones map y reduce en vez del método count() de un stream?

> **NOTA**: Usar reduce es mejor que hacer el trabajo de forma iterativa cuando usas paralelismo.
>
> ```
> int sum = lista..parallelStream().reduce(0, Integer::sum);
> ```

Sin embargo hay un precio a pagar, las lambdas que reducen no pueden cambiar de estado y las operaciones deben ser asociativas para poder aplicarlas en cualquier orden.

NOTA: *hay operaciones que no tienen estado interno (como map y filter) que cogen un elemento y generan otro o no sin fijarse en lo anterior. Pero operaciones como reduce(), sum(), max(), etc. necesitan tener un estado interno para tener en cuenta o recordar lo ya procesado. En estos casos el estado interno es pequeño, un int o un double. Pero operaciones como ordenaciones o eliminar elementos son muy sensibles a la cantidad de elementos con los que tienen que trabajar.*

9.3.2.5. PRIMITIVAS ESPECIALIZADAS.

Cuando los datos son numéricos hay un problema de ineficiencia en **reduce()** porque hay que hacer muchas operaciones de boxing y unboxing entre int e Integer al calcular por ejemplo la suma de las calorías de un menú:

```
int sumCal= menu.stream()
                .map(Plato::getCalorias)
                .reduce(0, Integer::sum);
```

¿Podríamos evitarlo llamando al método sum() de esta forma?

```
int sumCal= menu.stream()
                .map(Plato::getCalorias)
                .sum();
```

No es posible porque map() genera un Stream<T> y aunque T sean enteros, la interface Stream no define un método sum(). Sin embargo, para elementos numéricos se definen formas de convertir streams genéricos a streams numéricos especializados. Los métodos **mapToInt**, **mapToDouble** y **mapToLong** funcionan como el método map() pero

devuelven un stream especializado en vez del **Stream<T>** genérico.

EJEMPLO 34: Calcular la suma de calorías de un menú de forma eficiente:

```
int sumCal= menu.stream()
                .mapToInt(Plato::getCalorias)
                .sum();
```

Las conversiones son en las dos direcciones, de streams especializados a genéricos y viceversa.

```
IntStream is= menu.stream().mapToInt(Plato::getCalorias);
Stream<Integer> s = is.boxed();
```

OPTIONAL ESPECIALIZADO

Igual que está el Optional<T> también hay Optional especializados y OptionalInt podemos usarlo para tener valores por defecto:

```
OptionalInt oi= menu.stream()
                    .mapToInt(Plato::getCalorias)
                    .max();
int maxCal= oi.orElse(1);
```

RANGOS DE VALORES

También podemos generar rangos de valores enteros usando IntStream, por ejemplo números del 1 al 100. Para ello, cuenta con 2 métodos **range()** y **rangeClosed()**. Ambos tienen de parámetros el primer y el último valor pero range() es exclusivo y rangeClosed() es inclusivo:

```
IntStream pares= IntStream.rangeClosed(1, 100)
                          .filter( n -> n % 2 == 0 );
System.out.println( pares.count() );  // 50
```

EJEMPLO 35: Buscar tripletas de números enteros (a,b,c) que cumplan el teorema de pitágoras. Por sencillez, la tripleta la almacenamos en un array de int: int[]{a,b,c} representa (a, b, c).

Necesitas comprobar que la raíz cuadrada de a * a + b * b sea un entero, que no tenga decimales. Esto puedes expresarlo mediante un filtro:

```
filter(b -> Math.sqrt(a * a + b * b) % 1 == 0)
```

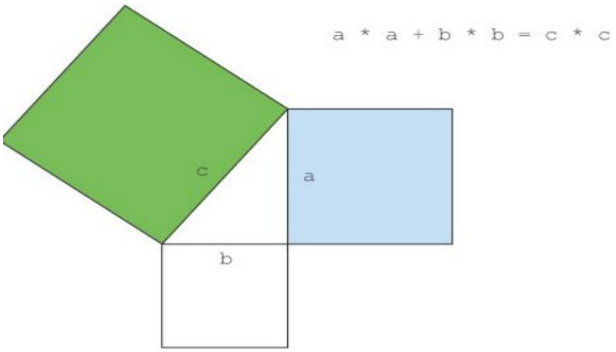


Figura 6: teorema de Pitágoras.

Generar las tuplas de dos elementos, pasarles el filtro para quedarte solamente con las correctas y luego usar una operación map para conseguir el tercer elemento y formar la tripleta de números:

```
stream.filter(b -> Math.sqrt(a * a + b * b) % 1 == 0)
      .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Generando los valores b en el intervalo [1, 100]:

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .boxed()
         .map(b -> new int[]{a, b, (int) Math.sqrt(a*a + b*b)} );
```

Se llama a boxed() después de que filter() genere un Stream<Integer> a partir del IntStream devuelto por rangeClosed. Esto es porque map() devuelve un array de int para cada elemento del stream. Y el map de un IntStream expera que se devuelva otro int por cada elemento, pero no

es lo que quieres! Así que puedes usar **mapToObj()** de IntStream:

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .mapToObj(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Para generar los valores a, hacemos lo mismo que con b y todo junto:

```
Stream<int[]> triplesPitagoricos=  IntStream.rangeClosed(1, 100)
                                           .boxed()
 .flatMap(a -> IntStream.rangeClosed(a, 100)
                        .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
          .mapToObj(b -> new int[]{a, b, (int)Math.sqrt(a * a + b * b)})
        );
```

Una mejora para hacer una sola raíz cuadrada:

```
Stream<double[]> triplesPitagoricos=  IntStream.rangeClosed(1, 100)
                                                .boxed()
 .flatMap(a -> IntStream.rangeClosed(a, 100)
        .mapToObj(b -> new double[]{a, b, Math.sqrt(a * a + b * b)} )
        .filter( t -> t[2] % 1 == 0 )
        );
 triplesPitagoricos.limit(5)
                   .forEach(t ->
                     System.out.println(t[0] + ", " + t[1] + ", " + t[2])
                    );
```

9.4. COLECTORES PARA PROCESOS COMPLEJOS.

Vamos a usar estos datos que son una lista de comerciales y operaciones de venta (Transaccion) que han realizado. Las clases de definen así:

```
public class Comercial{
    private final String nombre;
    private final String ciudad;

    public Comercial(String n, String c){this.nombre= n; this.ciudad= c;}
    public String getNombre(){ return this.nombre; }
    public String getCiudad(){ return this.ciudad; }
    @Override
```

```
        public String toString(){
            return "Comercial:" + this.nombre + " en " + this.ciudad;
        }
    }

    public class Transaccion{
        private final Comercial comercial;
        private final int year;
        private final int valor;

        public Transaccion(Comercial c, int y, int v){
            this.comercial= c;
            this.year= y;
            this.valor= v;
        }
        public Comercial getComercial(){ return this.comercial; }
        public int getYear(){ return this.year; }
        public int getValor(){ return this.valor; }
        @Override
        public String toString(){
            return "{" + this.comercial + ", " + "year: "+ this.year + ", " +
                   "valor:" + this.valor +"}";
        }
     }
```

Los datos:

```
Comercial mario = new Comercial("Mario","Londres");
Comercial raul = new Comercial("Raul","Milan");
Comercial alicia = new Comercial("Alicia","Seattle");
Comercial boro = new Comercial("Boro","Londres");
List<Transaccion> tran= Arrays.asList(
                            new Transaccion(boro,    2011,  300),
                            new Transaccion(raul,    2012, 1000),
                            new Transaccion(raul,    2011,  400),
                            new Transaccion(mario,   2012,  710),
                            new Transaccion(mario,   2012,  700),
                            new Transaccion(alicia,  2012,  950)
                        );
```

Las operaciones que puedes usar con colectores a parte de transformar los datos a una lista con **toList()** que hemos usado en algunos ejemplos:

- Agrupar datos por moneda y obtener datos de cada grupo. Por ejemplo: agrupar la lista de transacciones por moneda y obtener la suma de valores de transacciones para cada una (devolviendo

un Map<Currency, Integer>)
- Particionar una lista de transacciones en 2 grupos: rentable y norentable (devolviendo un Map<Boolean, List<Transaccion>>)
- Crear agrupaciones multinivel como agrupar por ciudades y luego por rentables y no rentables (devolviendo un Map<String, Map<Boolean, List<Transaccion>>>)

Está claro, se puede hacer sin streams, usando las colecciones donde están los datos. Por ejemplo:

```
Map<Currency, List<Transaccion>> traAgruMon =  new HashMap<>();
for(Transaccion t: trans) {
    Currency c= t.getMoneda();
    List<Transaccion> lt= traAgruMon.get(c);
    if(lt == null) {
        lt= new ArrayList<>();
        traAgruMon.put(c, lt);
    }
    lt.add( t );
}
```

Pero mucho más complicado que con streams:

```
Map<Currency, List<Transaccion>> traAgruMon =
    trans.stream().collect( groupingBy(Transaccion::getCurrency) );
```

COLECTORES COMO REDUCTORES AVANZADOS

Los colectores se usan mucho porque ofrecen una forma muy sencilla y flexible de definir criterios para recoger los datos y producir una colección de ellos. De hecho, llamar a un colector dispara una operación de reducción (parametrizada por un Collector) sobre los elementos del stream. Esta reducción es interna, y se muestra en la figura 7.

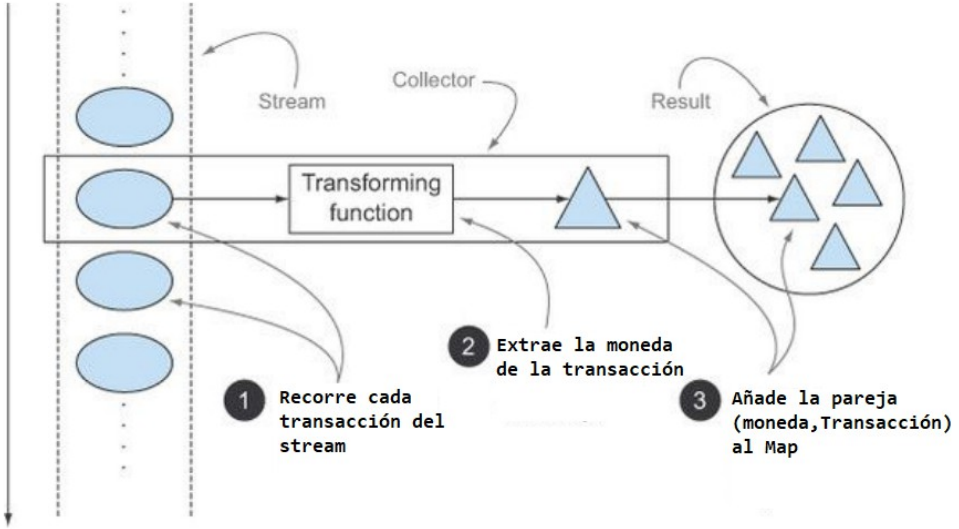


Figura 7. Agrupar transacciones por moneda aplica una reducción.

Normalmente el Collector aplica la función de transformación identidad al elemento (que no tiene ningún efecto) como en toList(), y añade el elemento a una estructura de datos o colección. Pero en el ejemplo, lo que hace es calcular la moneda de la ciudad y luego usar la moneda como clave de la pareja (moneda, transacción) antes de guardarla a un Map. La interfaz Collector ya tiene predefinidos algunos métodos para estas transformaciones y también podemos crear los nuestros propios:

```
List<Transaccion> listaTran= tran.stream().collect(Collectors.toList());
```

Los colectores predefinidos ofrecen 3 funcionalidades:

- Reducir y resumir streams de elementos a un único valor.
- Agrupar elementos.
- Particionar elementos.

REDUCIR Y RESUMIR

Se usan cuando quieres transformar varios elementos de un stream en un único valor. Puede ser tan sencillo como calcular el acumulado de una serie de números o más complejo.

EJEMPLO 36: Contar: contar el número de platos de un menú usando el método **counting()** de los colectores y con el count() de stream, que hacen lo mismo en este caso.

```
long nPlatos= menu.stream().collect( Collectors.counting() );
long nPlatos1= menu.stream().count();
```

Si importas los métodos estáticos podrás escribir directamente **counting()** de esta forma:

```
import java.util.stream.Collectors.*;
  :
long nPlatos= menu.stream().collect( counting() );
```

Los métodos **Collectors.maxBy** y **Collectors.minBy**, aceptan de argumento un Comparator para comparar los elementos del Stream.

EJEMPLO 37: encontrar máximos y mínimos: encontrar el plato con más calorías del menú.

```
Import java.util.Collectors.*;
  :
Comparator<Plato> compCalor= Comparator.comparingInt(Plato::getCalorias);
Optional<Plato> maxCalor= menu.stream()
                              .collect( maxBy( compCalor) );
```

Para acumular tienes el método Collectors.summingInt que acepta una función que mapea un objeto a un int que se usa para acumular su valor a un valor inicial de 0.

EJEMPLO 38: sumar: calcular el nº total de calorías del menú:

```
int totalCalor= menu.stream()
```

```
                .collect( summingInt( Plato::getCalorias ) );
```

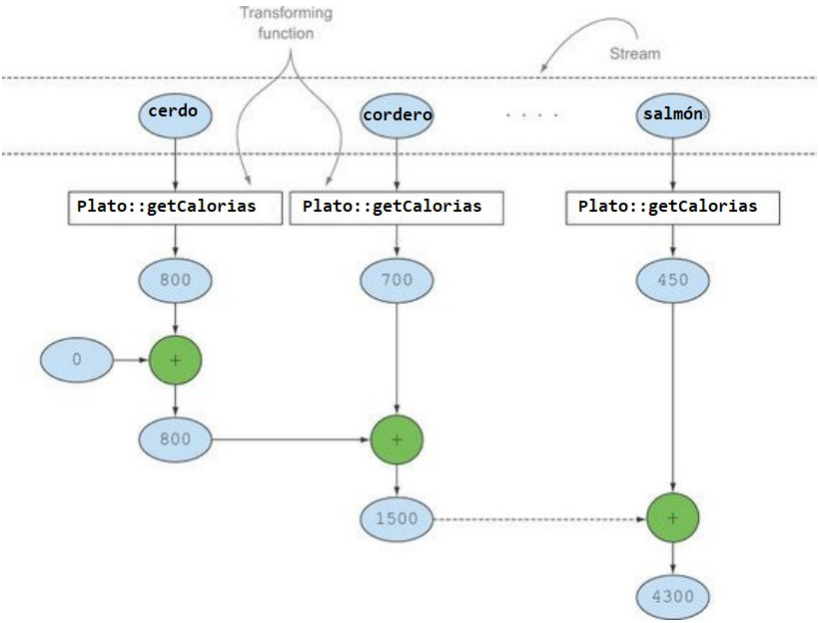


Figura 8. Proceso de acumulación con el colector summingInt

Los métodos **Collectors.summingLong** y **Collectors.summingDouble** son equivalentes para tipos long y double.

También hay disponibles **Collectors.averagingInt**, **averagingLong** y **averagingDouble** para calcular la media de estos valores:

```
double avgCalor=    menu.stream()
                        .collect( averagingInt( Plato::getCalorias) );
```

A veces podrías necesitar calcular varios de estos valores resumen (contarlos, máximo, mínimo, la suma, la media) y para que puedas hacerlos en una sola operación, puedes utilizar el método **summarizingInt**, summarizingLong, y summarizingDouble que devuelven objetos **IntSummaryStatistics**, **LongSummaryStatistics** y

DoubleSummaryStatistics.

EJEMPLO 39: obtener datos estadísticos de las calorías del menú:

```
IntSummaryStatistics mEsta= menu.stream()
                                .collect(
                                        summarizingInt(Plato::getCalorias)
                                        );
IntSummaryStatistics{count=9,   sum=4300,   min=120,   average=477.777778,
max=800}
```

También podemos unir elementos String usando el método **joining** que usa toString() de cada objeto o los elementos si son Strings y los une en un único string. También tiene un método sobrecargado que admite un separador:

EJEMPLO 40: concatenar cadenas: obtener una cadena con todos los nombres de los platos.

```
String menuCorto= menu.stream().map( Plato::getNombre)
                      .collect( joining() );
// Como Plato implementa toString() el paso de mapear lo podemos ahorrar
menuCorto= menu.stream().collect( joining() );
// Ambas formas obtiene la cadena "cerdocorderopollo...truchasalmón"
menuCorto= menu.stream()..map( Plato::getNombre).collect( joining(", ") );
```

MÉTODO REDUCING

Todos los colectores anteriores son casos particulares de un colector genérico llamado **Collectors.reducing** que acepta 3 argumentos:

- Valor de inicio: el primer valor usado en el proceso de reducción. Si quisieras hacer un acumulado sería 0.
- Función de transformación: transforma los objetos del stream en el dato que quieras resumir.
- Operador binario: mezcla el valor anterior con el valor devuelto por la función de transformación.

EJEMPLO 41: calcular la suma de calorías.

```
int totCalor= menu.stream()
              .collect( reducing( 0, Plato::getCalorias, (i,j) -> i+j) );
```

Tiene una versión sobrecargada con un solo parámetro que es el operador binario. El valor inicial se asume que es el primer elemento del stream y la función de transformación es la función identidad. Como no usa realmente un valor inicial como resultado, podría acabar no calculando nada si no recibe elementos. Por eso devuelve un objeto Optional<T>.

EJEMPLO 42: encontrar el plato con el máximo de calorías.

```
Optional<Plato> maxCalor=
     menu.stream()
         .collect(
            reducing(
                   (d1,d2) -> d1.getCalorias() > d2.getCalorias() ? d1:d2
                    )
                 );
```

COLLECT VS. REDUCE

Muchas veces puedes obtener los mismos resultados usando la operación reducir y la operación collect. Por ejemplo podemos cambiar collect(toList()) por este código que usa reduce() para obtener lo mismo:

```
Stream<Integer> s= Arrays.asList(1, 2, 3, 4, 5, 6).stream();
List<Integer> li= s.reduce(
                    new ArrayList<Integer>(),
                   (List<Integer> l, Integer e) -> { l.add(e); return l; },
       (List<Integer> l1, List<Integer> l2) -> { l1.addAll(l2); return l1; }
                     );
```

Pero esta solución tiene 2 problemas: uno semántico y otro práctico. El problema semántico es que el método reduce() intenta combinar dos elementos para obtener otro, es una reducción immutable. Mientras que collect() está pensado para cambiar un contenedor, así que el código anterior está mutando la lista que usa de contenedor. Este

problema filosófico se vuelve práctico si usas paralelismo, porque podría corromperse la lista al usarla al mismo tiempo varios threads. Si usas paralelismo, utiliza collect para almacenar resultados en colecciones.

Puedes hacer las operaciones de diferentes formas, usando reducing en el colector:

```
int totalCal= menu.stream()
                  .collect(
                     reducing(0,                    // Valor inicial
                              Plato::getCalorias,   // Fun. transformación
                              Integer::sum)         // Fun. agregación
                           );
```

O mapeando y reduciendo con reduce():

```
int totalCal= menu.stream().
                  .map(Plato::getCalorias).reduce(Integer::sum).get();
```

O mapeando el stream a un IntStream e invocando sum():

```
int totalCal = menu.stream().mapToInt(Plato::getCalorias).sum();
```

De todas las posibilidades, ¿Cául es mejor? En este caso la úmtima es más corta y eficiente. Pero en general tendrás que decantarte por la facilidad de entenderla.

EJERCICIO 7: Joining strings con reducing. ¿Cuál de estas 3 sentencias que utiliza el colector reducing es el equivalente del colector joining de strings?

```
// Original
String s= menu.stream().map(Plato::getNombre).collect( joining() );
```

1.
```
String s= menu.stream()
               .map(Plato::getNombre)
               .collect( reducing( (s1, s2) -> s1 + s2 ) )
               .get();
```

2.
```
String s= menu.stream()
                .collect(
                     reducing( (d1,d2) -> d1.getNombre()+d2.getNombre() )
                        )
                .get();
```

3.
```
String s= menu.stream()
                .collect(
                    reducing( "", Plato::getNombre, (s1, s2) -> s1 + s2 )
                        );
```

9.4.1. AGRUPAR.

Una operación común disponible en las bases de datos relaciones es agrupar la información por una o varias propiedades. Hacerlo con código imperativo es una tarea complicada y propensa a errores. Hacerlo con el API stream es más sencillo porque trabajas a más alto nivel, de forma declarativa. Imagina que queremos agrupar los platos del menú por su tipo. Para agrupar podemos usar el colector **Collectors.groupingBy**:

```
Map<Plato.Tipo, List<Plato>> plaGrup= menu.stream()
                                        .collect(groupingBy(Plato::getTipo));
```

Que genera el siguiente Map con el tipo de clave y una lista de Plato como elemento:

```
{PESCADO=[trucha, salmón], OTRO=[patatas fritas, arroz, fruta del tiempo,
pizza], CARNE=[cerdo, cordero, pollo]}
```

El método groupingBy() acepta una función que le devuelve la propiedad por la que realizar la agrupación. A esta función podemos llamarla **función de clasificación**.

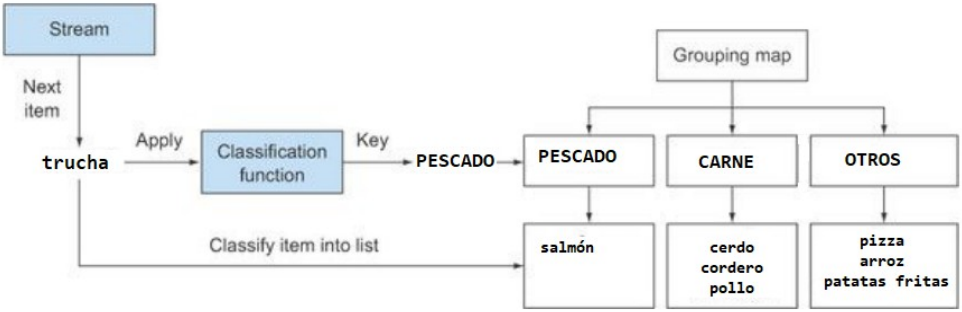


Figura 9. Clasificación de un elemento del stream al agrupar.

Pero no siempre es posible usar una referencia a método, por ejemplo si quieres clasificar como dieta los platos por debajo de 400 calorías, como normales los que estén entre 400 y 700 y como pesados los que estén por encima de 700, el creador de la clase Plato no ha programado esta funcionalidad, así que o la modificas o debes utilizar una función lambda como función de clasificación:

```
public enum NivelCal { DIETA, NORMAL, PESADO }
Map<NivelCal, List<Plato>> pGrup=
    menu.stream()
      .collect( groupingBy(
                p -> { if (p.getCalorias() <= 400)
                           return NivelCal.DIETA;
                       else if(p.getCalorias()<=700)
                           return NivelCal.NORMAL;
                       else return NivelCal.PESADO;
                     } )
             );
```

AGRUPAMIENTO MULTINIVEL

Hay una versión sobrecargada de groupingBy() que tiene de parámetros 2 funciones de clasificación. Esto permite realizar agrupamientos a dos niveles.

```
Map<Plato.tipo, Map<NivelCal, List<Plato>>> pGrupTipoNiv=
```

```
menu.stream()
        .collect(
             groupingBy(
                  Plato::getTipo,
                  groupingBy(
                       p -> { if (p.getCalorias() <= 400)
                                  return NivelCal.DIETA;
                              else if(p.getCalorias()<=700)
                                  return NivelCal.NORMAL;
                              else return NivelCal.PESADO;
                             }
                            )
                   )
                );
```

El resultado es un Map que tiene de clave el tipo de plato y de elemento otro Map de clave el nivel calórico y de elemento una lista de platos:

```
{ CARNE={DIETA=[pollo], NORMAL=[cordero], PESADO=[cerdo]},
  PESCADO={DIETA=[trucha], NORMAL=[salmón]},
  OTROS={DIETA=[arroz, fruta del tiempo], NORMAL=[patatas fritas, pizza]}
}
```

RECOGER DATOS DE SUBGRUPOS

Cuando haces subgrupos, el segundo parámetro colector no tiene porqué ser otro **groupingBy()**, podemos contar cuantos elementos hay en el grupo, o hacer una suma, o buscar el máximo del grupo, etc.

EJEMPLO 41: contar cuantos platos de cada tipo hay:

```
Map<Plato.Tipo, Long> tp= menu.stream()
                              .collect( groupingBy(Plato::getTipo, counting()));
```

El Map resultado:

```
{CARNE=3, PESCADO=2, OTRAS=4}
```

Observa también que el groupingBy(f) con un solo parámetro es en realidad una abreviatura de groupingBy(f, toList()).

EJEMPLO 42: plato con más calorías de cada tipo:

```
Map<Plato.Tipo, Optional<Plato>> maxPorTipo=
    menu.stream()
        .collect(
             groupingBy(
              Plato::getTipo,
              maxBy( comparingInt(Plato::getCalorias) )
             )
         );
```

El Map resultado:

```
{PESCADO=Optional[salmón], OTRAS=Optional[pizza], CARNE=Optional[cerdo]}
```

Si quieres adaptar el resultado devuelto, por ejemplo quedarte con el plato y coger algún dato suyo, puedes usar **Collectors.collectingAndThen()**.

```
Map<Plato.Tipo, Plato> maxPorTipo=
    menu.stream()
        .collect(
             groupingBy(
                Plato::getTipo,
                collectingAndThen(
                    maxBy( comparingInt(Plato::getCalorias) ),
                    Optional::get
                )
             )
         );
```

Este método acepta dos parámetros, el primero es un colector y el segundo es una función de transformación. Devuelve otro colector. La siguiente figura resumen como funciona.

Además, suelen utilizarse como segundo parámetro del groupingBy funciones que reducen todos los elementos del grupo (sumas, medias, etc.)

```
Map<Plato.Tipo, Integer> sumCalPorTipo=
    menu.stream()
        .collect(
             groupingBy(Plato::getTipo,
                        summingInt(Plato::getCalorias)
                       )
              );
```

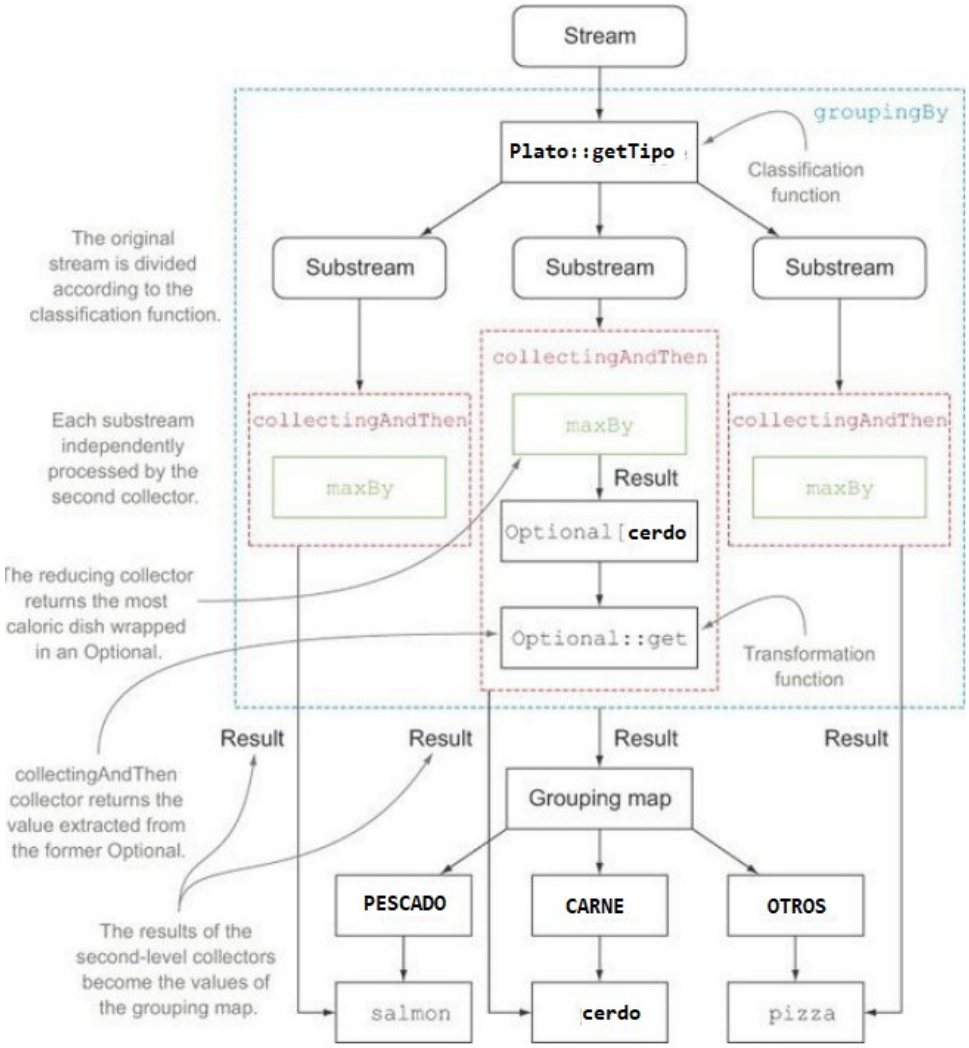


Figura 10: funcionamiento de collectingAndThen().

También se puede necesitar mapear con **mapping()**. Acepta dos parámetros, el primero una función de transformación que modifica un

elemento generando un stram y un colector que vuelve a acumular los elementos.

EJEMPLO 43: queremos saber el nivel de calorías de cada tipo de Plato:

```
Map<Plato.Tipo, Set<NivelCal>> ncPorTipo=
    menu.stream()
        .collect(
            groupingBy(Plato::getTipo,
                       mapping(
                                p -> { if (p.getCalorias() <= 400)
                                           return NivelCal.DIETA;
                                       else if (p.getCalorias() <= 700)
                                           return NivelCal.NORMAL;
                                       else return CaloricLevel.PESADO; },
                                toSet()
                       ) // mapping
            ) // grouping
        );  // collect
```

Dando como resultado:

```
{OTROS=[DIETA, NORMAL], CARNE=[DIETA, NORMAL, PESADO], PESCADO=[DIETA, NORMAL]}
```

Si te interesa localizar algún tipo concreto de nivel calórico puedes usar un hashSet utilizando **toCollection()**:

```
Map<Plato.Tipo, Set<NivelCal>> ncPorTipo=
   menu.stream()
       .collect(
          groupingBy(Plato::getTipo, mapping( if (p.getCalorias() <= 400)
                                           return NivelCal.DIETA;
                                       else if (p.getCalorias() <= 700)
                                           return NivelCal.NORMAL;
                                       else return CaloricLevel.PESADO; },
                             toCollection(HashSet::new) )));
```

9.4.3. PARTICIONAMIENTO.

Es un caso de especial de agrupamiento: tiene una función de particionamiento (como la función de clasificación) que devuelve un booleano, lo que implica que el Map resultado tendrá de clave un

booleano y solo dos posibles claves. Se realiza con el colector **partitioningBy()**.

EJEMPLO 44: invitas a un amigo vegetariano a cenar, así que quieres dividir el menú en dos trozos, los platos vegetarianos y los que no.

```
Map<Boolean, List<Plato>> parti=
    menu.stream()
        .collect(
              partitioningBy( plato::esVegano )
         );
```

Para recuperar los platos que sean vegetariaos del Map:

```
List<Plato> pv= parti.get(true);
```

Podrías haber conseguido lo mismo con:

```
List<Plato> pv= menu.stream().filter(Plato::isVegeno).collect( toList() );
```

VENTAJAS DEL PARTICIONAMIENTO

Particionar tiene la ventaja de que haces las dos divisiones a la vez en la misma sentencia. Además, como pasaba con groupingBy(), tiene una versión sobrecargada con dos parámetros para tener más de un nivel.

EJEMPLO 45: invitas a un amigo vegetariano a cenar, así que quieres dividir el menú en dos trozos, los platos vegetarianos y los que no.

```
Map<boolean, Map<Plato.tipo, List<Plato>>> parti=
    menu.stream()
        .collect(
              partitioningBy( plato::esVegano,
                              groupingBy( Plato::getTipo )    )
         );
```

Que produce un mapa a 2 niveles:

```
{ false={PESCADO=[trucha,salmón], CARNE=[cerdo, cordero, pollo]},
  true={OTROS=[patatas fritas, arroz, fruta del tiempo, pizza]}
}
```

Otro ejemplo, encuentra el plato más calórico tanto vegetariano como no vegetariano:

```
Map<Boolean, Plato> maxCal=
    menu.stream()
        .collect(
             partitioningBy(
                   Plato::esVegano,
                   collectingAndThen(
                     maxBy(comparingInt(Plato::getCalorias) ),
                     Optional::get )
             )
         );
```

Produce el resultado: `{false=cerdo, true=pizza}`

EJERCICIO 8: Usando partitioningBy puedes combinar varios colectores, como usar un segundo partitioningBy. ¿Cuál será el resultado de los siguientes particionamientos multinivel?

1. ```
   menu.stream()
          .collect(
               partitioningBy(
                  Plato::esVegano,
                  partitioningBy( p -> p.getCalorias() > 500 )
               )
           );
   ```

2. ```
   menu.stream()
          .collect( partitioningBy(Plato::esVegano,
                                   partitioningBy(Plato::getTipo)
                                  ) );
   ```

3. ```
   menu.stream().collect(partitioningBy(Plato::esVegano, counting()));
   ```

EJEMPLO 46: Procesamiento avanzado: Particionar números en primos y no primos. Necesitamos un método que acepte un número entero n y particione los n primeros números en primos y no primos. Hacemos un predicado que compruebe si un nº es primo:

```
public boolean esPrimo(int candidato) {
```
```

```
        return InsStream.range(2, candidato)
                        .noneMatch(i -> candidato % i == 0);
    }
```

Una sencilla optimización para probar no hasta el candidato-1 sino hasta la raíz cuadrdada del candidato:

```
    public boolean esPrimo(int candidato) {
        int raiz= (int) Math.sqrt((double) candidato);
        return IntStream.rangeClosed(2, raiz)
                        .noneMatch(i -> candidato % i == 0);
    }
```

La parte difícil está hecha, ahora queda particionar los n primeros números:

```
    public Map<Boolean, List<Integer>> particionaPrimos(int n) {
        return IntStream.rangeClosed(2, n)
                        .boxed()
                        .collect(
                           partitioningBy( c -> esPrimo(c) )
                         );
    }
```

Tabla 6.1. Métodos estáticos prefabricados de la clase Collectors

Método Factoría	Tipo Devuelto	Usado para..	Ejemplo de uso:
toList	List<T>	Deja elementos en lista	List<P> p= ms.collect(toList());
toSet	Set<T>	Deja elementos en Set sin duplicados	Set<P> p= ms.collect(toSet());
toCollection	Collection<T>	Deja elementos en Collection creada por el supplier	Collection<P> p= ms.collect(toCollection(), ArrayList::new);
counting	Long	Cuenta elementos	long np= ms.collect(counting());
summingInt	Integer	Suma valores	int t= ms.collect(summingInt(Plato::getCalorias));
averagingInt	Double	Calcula el valor medio	double avg= ms.collect(averagingInt(Plato::getCalorias));

Método Factoría	Tipo Devuelto	Usado para..	Ejemplo de uso:
summarizingInt	IntSummaryStatistics	Recoge estadísticas como max, min, total y media	IntSummaryStatistics s= ms.collect(summarizingInt(Plato::getCalorias));`
joining	String	Concatena strings	String s= ms.map(Plato::getNombre).collect(joining(", "));`
maxBy	Optional<T>	Máximo valor de la propiedad indicada por un Comparator encapsulado en un Optional	Optional<P> m= ms.collect(maxBy(comparingInt(P:: getCalorias)));`
minBy	Optional<T>	Igual para mínimo	Optional<P> m= ms.collect(minBy(comparingInt(P:: getCalorias)));`
reducing	Tipo producido por la función de reducción	Reduce el stream a un único valor desde un valor inicial de forma iterativa combinando valores con un BinaryOperator.	int t= ms.collect(reducing(0, Plato::getCalorias, Integer::sum));`
collectingAndThen	The type returned by the transforming function	Rompe otro colector y aplica una función de transformación a sus resultados	int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));`
groupingBy	Map<K, List<T>>	Agrupa elementos de un stream usando como valor una de sus propiedades como claves en el mapa resultado	Map<Dish.Type, List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));`
partitioningBy	Map<Boolean, List<T>>	Particiona elementos aplicando un predicado a cada uno de ellos.	Map<Boolean, List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));`

9.4.3 USO AVANZADO DE COLECTORES.

La interface **Collector** ofrece un conjunto de métodos para implementar reducciones (transformar streams en otras cosas). Por ejemplo las operaciones toList() o groupingBy() implementan esta interface. Pero tu también puedes crear tus propias operaciones de reducción.

La definición de la interface Collector:

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

Donde:

- T es el genérico que representa el tipo de los elementos del stream.
- A es el tipo del acumulador, el objeto que se usa para generar resultados parciales durante el proceso.
- R es el tipo del objeto (normalmente, aunque no siempre, una colección) resultado de la operación.

Por ejemplo podrías implementar una clase ToListCollector<T> que transforma los elementos de un Stream<T> en un List<T> y donde el tipo intermedio coincide con el final, indicando la siguiente definición:

```
public class ToListCollector<T> implements Collector<T,List<T>,List<T>> {
  ...
}
```

Ahora examinamos los 5 métodos que declara la interface, donde los 4 primeros devuelven una función que será invocada por el método collect y la última, characteristics, da un conjunto de características

que es una lista para que la usen los anteriores de cara a aplicar optimizaciones (por ejemplo, paralelización) que tienen permitido usar.

MÉTODO SUPPLIER

Debe devolver un Supplier de un resultado vacío—devuelve una función que crea una instancia de un acumulador vacío usado durante el proceso de reducción.

EJEMPLO 47. definir un supplier para ToListCollector. Debe devolver una lista si reducimos un strem<T> vacío.

```
public Supplier<List<T>> supplier() {
    return () -> new ArrayList<T>();
}
// O también:
public Supplier<List<T>> supplier() { return ArrayList::new; }
```

MÉTODO ACCUMULATOR

Devuelve la función que realiza la operación de reducción. Cuando se atraviese el n-ésimo elemento del stream, se aplica esta función con 2 parámetros, el acumulador que contiene el resultado de reducir los primeros n-1 elementos anteriores del stream y el propio n-ésimo elemento. La función devuelve void porque el resultado es la modificación del acumulador, al que la función cambia su estado de forma interna.

EJEMPLO 48: implementar el método accumulator para ToListCollector:

```
public BiConsumer<List<T>, T> accumulator() {
    return (lista, elemento) -> lista.add(elemento);
}
// Con referencia a métodos
public BiConsumer<List<T>, T> accumulator() { return List::add; }
```

MÉTODO FINISHER

Devuelve una función que al ser aplicada acaba el proceso de acumulación después de haber atravesado todos los elementos del stream.

EJEMPLO 49: implementar el finisher de ToListCollector. Como el resultado final coincide con el obtenido al procesar el último elemento, no hace falta aplicar ninguna transformación adicional, por eso se utiliza la función identidad.

```
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```

Estos 3 primeros 3 métodos son suficientes para ejecutar reducciones secuenciales de streams. La figura 11 visualiza de manera lógica los procesamientos que podríamos realizar. La implementación real podría ser más compleja (realizar operaciones previas en pipeline antes de aplicar collect, realizar la reducción en paralelo, etc.).

MÉTODO COMBINER

Define como los acumuladores que se generan durante el proceso de reducción de las diferentes subpartes del stream se combinan cuando se procesan en paralelo.

EJEMPLO 50: Implementar combiner para ToListCollector. En este caso es simple, hay que añadir la lista de la segunda subparte al final de la lista generada por la primera subparte:

```
public BinaryOperator<List<T>> combiner() {
  return (list1, list2) -> { list1.addAll(list2); return list1; }
}
```

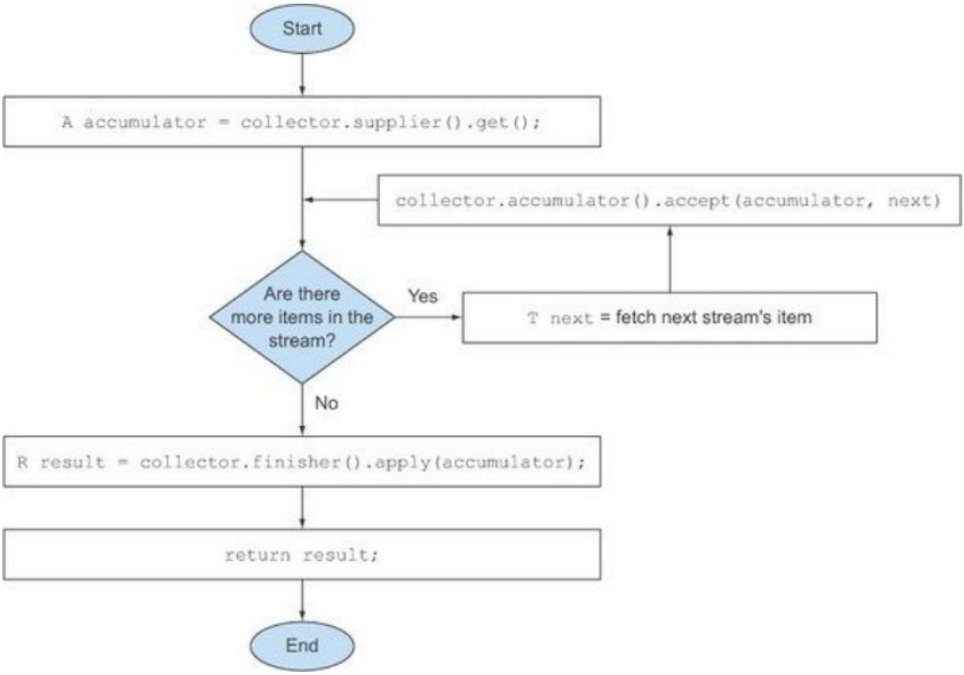


Figura 11. Pasos lógicos de un proceso de reducción (simplificado).

Este cuarto método permite reducir el stream en paralelo. Usa el framework fork/join y la abstracción Spliterator. La figura 12 representa la forma de trabajar. El *stream* original se divide recursivamente en substreams hasta alcanzar el nivel de división adecuado (los cálculos paralelos podrían ser más lentos que los secuenciales si el tamaño es demasiado pequeño). Cada substream se procesa en paralelo usando reducción secuencial (los 3 métodos anteriores) y finalmente, los resultados parciales se combinan.

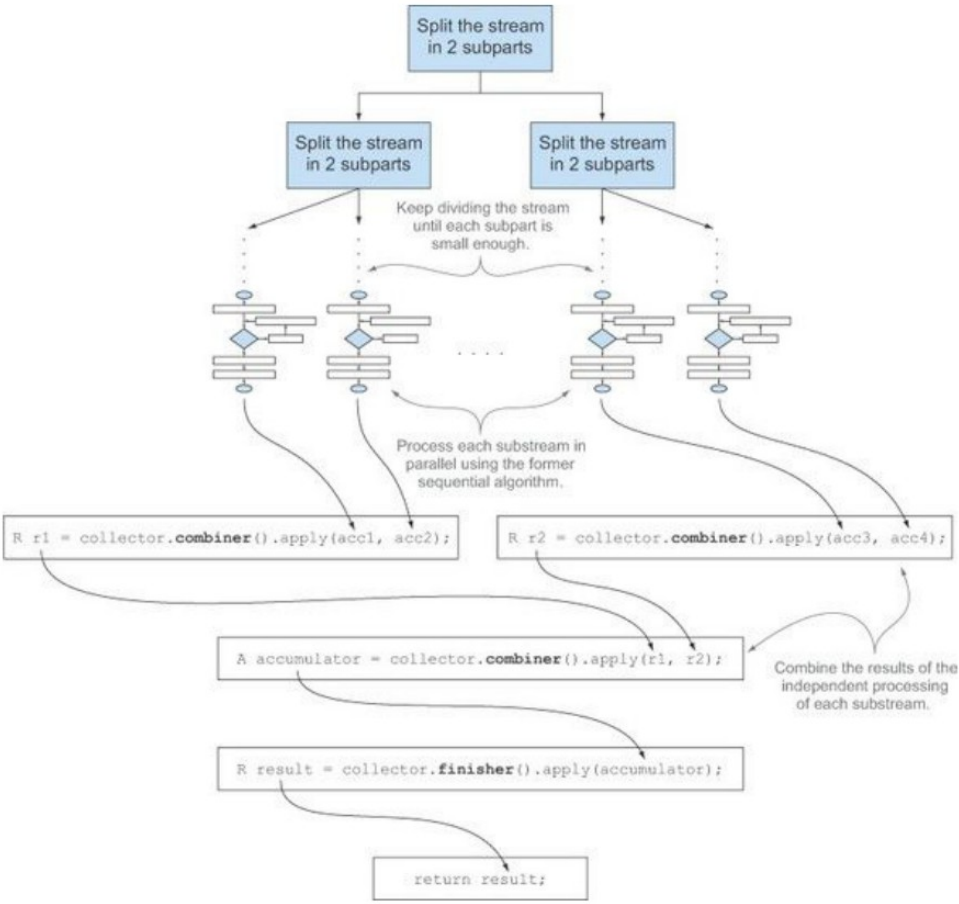


Figura 12. Paralelizar el proceso usando el método combiner.

MÉTODO CHARACTERISTICS

Devuelve un conjunto immutable de Characteristics que define el comportamiento del colector. Contiene 3 elementos:

- UNORDERED: el resultado no es afectado por el orden que los elementos son atravesados o acumulados.
- CONCURRENT: la función accumulator puede llamarse

concurrentemente (varios hilos paralelos). Si no tiene también la característica UNORDERED, puede aplicar reducción en paralelo solamente si se aplica a datos que vengan desordenados desde el origen.

- IDENTITY_FINISH: indica que la función devuelta por el método finisher es la identidad y por tanto su uso puede evitarse. También significa que es seguro no comprobar el casting desde A a R.

EJEMPLO 51: implementar el método para ToListCollector y poner todo junto. Tendrá IDENTITY_FINISH, pero no UNORDERED porque si lo aplicas a un stream donde los elementos vengan ordenados y quieres mantener ese orden, se perdería. Sería CONCURRENT, pero solamente se utilizará la concurrencia cuando los elementos del stream de origen vengan desordenados. Así que la clase quedaría así definida:

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import static java.util.stream.Collector.Characteristics.*;


public class ToListCollector<T>
       implements Collector<T, List>T>, List<T> {

    @Override
    public Supplier<List<T>> supplier() { return ArrayList::new; }

    @Override
    public BiConsumer<List<T>, T> accumulator(){return List::add;}

    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.identity();
    }

    @Override
```

```
        public BinaryOperator<List<T>> combiner() {
          return (list1, list2) -> { list1.addAll(list2);
                                     return list1; }
        }

        @Override
        public Set<Characteristics> Characteristics() {
            return Collections.unmodificableSet(
                   EnumSet.of( IDENTITY_FINISH, CONCURRENT)
                                               );
        }
    }
```

El método Collector de la API de Java .toList() es como esta clase pero con algunas optimizaciones como el uso de la clase singleton Collections.emptyList(). Podrías utilizarla por tanto de forma segura para coleccionar los platos de un menú en vez de la del API toList() con la salvedad de que debes crear una instancia, mientras que la de Java es una clase factoría:

```
List<Plato> m = menuStream.collect(new ToListCollector<Plato>());
```

En vez de:

```
List<Plato> menu = menuStream.collect(toList());
```

Ahora, después de examinar el comportamiento de un Collector personalizado, vamos a implementar uno con el objetivo de mejorar el rendimiento de realizar cierto trabajo. Anteriormente, en el apartado de las particiones, habíamos creado un colector que dividía los n primeros números naturales en dos grupos: primos y no primos.

```
public Map<Boolean, List<Integer>> particionPrimos(int n) {
    return IntStream
           .rangeClosed(2, n)
           .boxed()
           .collect(
               partitioningBy(candidato -> esPrimo(candidato)
             );
```

```
}
```

Se puede mejorar el rendimiento actuando sobre el método esPrimo() reduciendo el número de divisores que tiene que comprobar hasta la raíz cuadrada de n.

```
public boolean esPrimo(int candidato) {
    int raiz = (int) Math.sqrt((double) candidato);
    return IntStream
            .rangeClosed(2, raiz)
            .noneMatch( i -> candidato % i == 0);
}
```

¿Mejorable? Sí, implementando un Collector personalizado.
Un número no primo, tendrá divisores primos. Bueno, dividamos solamente por los primos anteriores a la raíz del candidato a ser primo. Si no tiene, es primo. Pero para hacer esto, necesitamos tener acceso a los resultados anteriores, y eso solamente lo podremos tener si nos creamos uno personalizado. Vamos paso a paso. Primero modificamos esPrimo:

```
public static boolean esPrimo( List<Integer> primos,
                               int candidato ) {
    return primos
            .stream()
            .noneMatch( i -> candidato % i == 0 );
    }
```

Mantenemos limitar los divisores a comprobar y hay que parar de procesar si el candidato es divisible por un primo. Para hacerlo podríamos usar filter(p -> p <= raiz) pero procesaremos todo el stream. Y si la lista de primos o el candidato son muy grandes bajará el rendimiento. Para evitarlo definimos otro método que nos devuelva una lista ordenada y nos devuelva solamente los valores que sean inferiores:

```
public static <A> List<A> soloEstos(List<A> lista, Predicate<A> p)
```

```
{
    int i = 0;
    for( A item : lista ) {
        if( !p.test(item) ) return lista.sublist(0,i);
        i++;
    }
    return lista;
}
```

Y lo usamos en el método esPrimo():

```
public static boolean esPrimo(List<Integer> primos, int candidato)
{
    int raiz = (int) Math.sqrt( (double)candidato);
    return soloEstos(primos, i -> i <= raiz)
           .stream()
           .noneMatch( p -> candidato % p == 0 );
}
```

Ahora mejoramos el colector y lo primero es definir la firma de la clase teniendo en cuenta la definición de Collector (public interface Collector<T, A, R>) donde T, A y R son el tipo de elementos, el tipo de objetos que acumulan y el tipo del resultado final. El tipo de los elementos será Integer y el de los acumuladores y resultado serán Map<Boolean, List<Integer>> por tanto quedará definida así:

```
public class ColectorPrimos
             implements Collector<Integer,
                                  Map<Boolean,List<Integer>>,
                                  Map<Boolean, List<Integer>>> {
```

El Supplier solo crea el Map que sirve como acumulador y define dos listas vacías, una para la clave true (los primos) y otra para la clave false (los valores no primos):

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>() {
                        {  put(true, new ArrayList<Integer>());
                           put(false, new ArrayList<Integer>());
                        }
```

```
                        };
    }
```

De esta forma, en cada iteración podremos acceder a los valores previamente calculados. El acumulador quedará así:

```
    public BiConsumer<Map<Boolean, List<Integer>>, Integer>
    accumulator() {
        return (Map<Boolean, List<Integer>> acu, Integer candidato) ->
               {
                   acu.get( esPrimo( acu.get(true), candidato) )
                      .add(candidato);
               };
    }
```

El método combiner() solo tiene que combinar dos acumuladores parciales, es decir, dos mapas, añadiendo a la lista de primos y no primos los valores calculados de uno y otro.

```
    public BinaryOperator<Map<Boolean, List<Integer>>> combiner(){
        return (Map<Boolean, List<Integer>> m1,
                Map<Boolean, List<Integer>> m2) ->
                { m1.get(true).addAll( m2.get(true) );
                  m1.get(false).addAll( m2.get(false) );
                  return m1;
                };
    }
```

Si lo piensas, en realidad este colector no puede utilizarse en paralelo porque el algoritmo es completamente secuencial. Esto quiere decir que no debe llamarse nunca al método combiner() , así que sería mejor dejar su implementación vacía o incluso mejor lanzar una excepción de tipo UnsupportedOperation. Lo hemos implementado por hacer el ejemplo completo.
Como el acumulador coincide con el resultado el último método será la función identidad:

```
    public Function<Map<Boolean, List<Integer>>,
                    Map<Boolean, List<Integer>>> finisher() {
```

```
        return Function.identity();
    }
```

Y como características fijaremos no CONCURRENT y no UNORDERED pero sí IDENTITY_FINISH:

```
    public Set<Characteristics> characteristics() {
      return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
    }
```

Ahora puedes usarlo en lugar del método prefabricado partitioningBy() y obtendrás los mismos resultaados:

```
    public Map<Boolean, List<Integer>> particionPrimos2(int n) {
        return IntStream
                .rangeClosed(2, n)
                .boxed()
                .collect( new ColectorPrimos() );
    }
```

Para que puedas comparar el rendimiento obtenido con ambos te propongo que pruebes este código y sustituyas lo subrayado por cada colector.

```
    public class RendimientoColectores {
        public static void main(String[] args) {
            long test = Long.MAX_VALUE;
            for( int i = 0; i < 10; i++ ) {
                long inicio = System.nanoTime();
                particionPrimos(1_000_000);
                long dura = (System.nanoTime() - inicio) / 1_000_000;
                if(dura < test ) test = dura;  // El mínimo
            }
            System.out.printf("Ejecución más rápida en %d ms\n",test);
        }
    }
```

9.4. PROGRAMACIÓN PARALELA Y ASÍNCRONA.

9.4.1 INTRODUCCIÓN AL PARALELISMO.

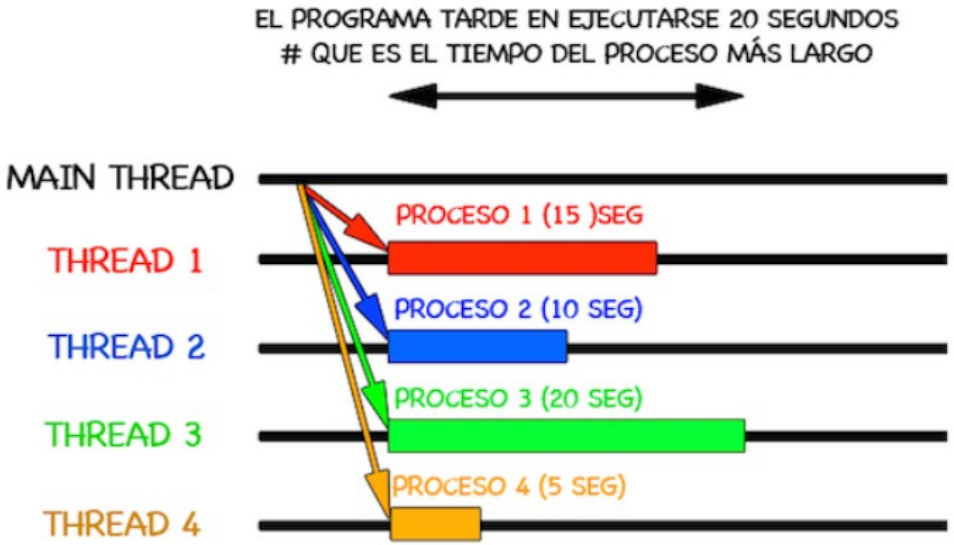
El subproceso de tipo hilo (thread) o proceso ligero es la unidad de procesamiento más pequeña que el sistema operativo puede manejar para asignarle CPU. La diferencia con un proceso normal es que comparte la memoria global de su proceso principal y es más rápido y menos costoso de crear.

En esencia la multitarea nos permite ejecutar varios procesos a la vez, es decir, de forma concurrente y por tanto eso nos permite hacer programas que se ejecuten en menor tiempo y sean más eficientes.

Evidentemente no podemos ejecutar infinitos procesos de forma concurrente ya que el hardware tiene sus limitaciones, pero raro es el dispositivo que no tenga más de un núcleo (en un procesador con dos núcleos se podrían ejecutar dos procesos a la vez). Para que veáis la diferencia en un par de imágenes, supongamos que tenemos un programa secuencial en el que se han de ejecutar 4 procesos, uno detrás de otro y estos tardan unos segundos:

En un hardware donde se ejecuten en paralelo (como la siguiente figura), siempre que cada uno sea independiente del resto. Sin embargo, esto no siempre sucede, por tanto necesitamos formas de sincronizar los threads, que puedan esperarse, comunicarse, compartir recursos de forma segura, notificarse o avisarse de cosas, etc. (no entraremos en ese jardín aunque ya sabes que es algo que tienes

pendiente de mirar).



CLASE THREAD E INTERFACE RUNNABLE

En Java para utilizar la multitarea podemos usar la clase **Thread** (creamos una clase que la extienda y sobreescriba run()) o bien una clase que implemente la interface **Runnable** (Thread lo hace) que nos obliga a implementar run().

EJEMPLO: Usar ejecución no paralela.

En este ejemplo vamos a simular el proceso de cobro de un supermercado donde unos clientes van con un carro lleno de productos y una cajera les cobra, pasándolos uno a uno por el escaner de la caja registradora. En este caso la cajera debe de procesar la compra cliente a cliente, es decir, que primero le cobra al cliente 1, luego al cliente 2 y así sucesivamente. Para ello vamos a definir una clase

"Cajera" y una clase "Cliente" el cual tendrá un "array de enteros" que representa los productos que ha comprado y el tiempo que la cajera tarda en pasar el producto por el escaner. Si tenemos un array con [1,3,5] significa que el cliente ha comprado 3 productos y que la cajera tarda en procesar el producto 1 '1 segundo', el producto 2 '3 segundos' y el producto 3 en '5 segundos', con lo cual tarda en cobrar al cliente toda su compra '9 segundos'. Explicado este ejemplo vamos a ver como hemos definido estas clases:

```
// Clase "Cajera.java":
public class Cajera {
    private String nombre;

    public void procesarCompra( Cliente c, long t) {
        System.out.println( "Cajera " + this.nombre +
                " COMIENZA A PROCESAR COMPRA DE CLIENTE " +
                c.getNombre() + " EN EL TIEMPO: " +
                (System.currentTimeMillis() - t) / 1000 + "seg");
        for(int i = 0; i < c.getCarroCompra().length; i++) {
            this.esperar(c.getCarroCompra()[i]);
            System.out.println( "Procesado producto " + (i + 1) +
                     " ->Tiempo: " +
                    (System.currentTimeMillis() - t)/1000 +
                    "seg");
       }
       System. out.println( "La cajera " + this.nombre +
                    " TERMINA DE PROCESAR " + c.getNombre() +
                    " EN EL TIEMPO: " +
                    (System.currentTimeMillis() - t) / 1000 +
                    "seg");
    }

    private void esperar(int segundos) {
        try {
            Thread.sleep(segundos * 1000);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
} // fin clase Cajera
```

```
// Clase "Cliente.java":
public class Cliente {
    private String nombre;
    private int[] carroCompra;
    // Constructor, getter y setter...
}
```

Si ejecutásemos este programa con 2 Clientes y un solo proceso, se procesaría primero la compra del Cliente 1 y después la del Cliente 2, con lo cual se tarda el tiempo del Cliente 1 + Cliente 2. A continuación vamos a ver como programamos el método Main para lanzar el programa. Ten en cuenta que aunque hayamos puesto dos objetos de la clase Cajera (cajera1 y cajera2) no significa que tengamos dos cajeras trabajando en paralelo, lo que estamos diciendo es que dentro del mismo hilo se ejecuten primero los métodos de la cajera1 y después los métodos de la cajera2, por tanto a nivel de procesamiento es como si tuviésemos una sola cajera:

```
public class Ejemplo1 {

  public static void main(String[] args) {
    Cliente c1 = new Cliente("Cliente 1", new int[]{2, 2, 1, 5} );
    Cliente c2 = new Cliente("Cliente 2", new int[]{ 1, 3, 5, 1});
    Cajera ca1 = new Cajera( "Cajera 1");
    Cajera ca2 = new Cajera( "Cajera 2");
    // Tiempo inicial de referencia
    long t0 = System.currentTimeMillis();
    ca1.procesarCompra(c1, t0);
    ca2.procesarCompra(c2, t0);
  }
}
```

EJEMPLO 2: Definir el programa usando paralelismo.

Modificamos la clase "Cajera.java" para que herede de la clase Thread.

```
public class CajeraThread extends Thread {
```

```
        private String nombre;
        private Cliente c;
        private long t0;   // Tiempo de creación
        // Constructor, getter & setter...
        @Override
        public void run() {
            System.out.println("Cajera " + this.nombre +
                       " COMIENZA A PROCESAR CLIENTE " +
                       this.c.getNombre() + " EN TIEMPO: " +
                       (System.currentTimeMillis() - this.t0) / 1000 +
                       "seg");
            for(int i = 0; i < this.c.getCarroCompra().length; i++) {
                this.esperar( c.getCarroCompra()[i] );
                System.out.println( "Procesado producto " + (i + 1) +
                " de cliente " + this.c.getNombre() + " ->Tiempo: " +
                (System.currentTimeMillis() - this.t0) / 1000 +
                "seg");
            }
            System.out.println( "La cajera " + this.nombre +
                       " TERMINA DE PROCESAR " + this.c.getNombre() +
                       " EN TIEMPO: " +
                       (System.currentTimeMillis() - this.t0) / +
                       "seg");
        }

        private void esperar(int segundos) {
            try {
                Thread.sleep(segundos * 1000);
            } catch (InterruptedException ex) {
               Thread.currentThread().interrupt();
            }
        }
    } // clase
```

Observa que la clase "CajeraThread" hereda de la clase Thread "extends Thread" y sobreescribe el método "run()". Este método es imprescindible sobreescribirlo (ya que es un método que esta en la clase Runnable y la clase Thread la implementa) porque en él se va a codificar la funcionalidad que se ha de ejecutar en un hilo diferente, es decir, que lo que se programe en el método "run()" se va a ejecutar

de forma paralela al hilo principal (secuencial en su hilo). En esta clase "CajeraThread" se pueden sobreescribir más métodos para que hagan acciones sobre el hilo o thread como por ejemplo, parar el thread, ponerlo en reposo, etc. A continuación programamos otra clase desde donde ejecutar de nuevo la misma simulación:

```
public class Ejemplo2 {

  public static void main(String[] args) {
    Cliente c1 = new Cliente("Cliente 1", new int[]{2, 2, 1, 5} );
    Cliente c2 = new Cliente("Cliente 2", new int[]{ 1, 3, 5, 1});
    long tiempo = System.CurrentTimeMillis();
    Cajera ca1 = new CajeraThread( "Cajera 1", c1, tiempo);
    Cajera ca2 = new CajeraThread( "Cajera 2", c2, tiempo);
    ca1.start();
    ca2.start();
  }
}
```

EJEMPLO 3: Igual pero implementando Runnable

```
public class Ejemplo3 implements Runnable{
    private Cliente c;
    private Cajera ca;
    private long t;

    public Ejemplo3(Cliente c, Cajera ca, long t) {
        this.ca = ca;
        this.c = c;
        this.t = t;
    }

    public static void main(String[] args) {
        Cliente c1 = new Cliente("Cliente 1", new int[]{2,2,1,5});
        Cliente c2 = new Cliente("Cliente 2", new int[]{1,3,5,1});
        Cajera ca1 = new Cajera( "Cajera 1");
        Cajera ca2 = new Cajera( "Cajera 2");
        // Tiempo inicial de referencia
        long t0 = System.currentTimeMillis();
        Runnable hilo1 = new Ejemplo3(c1, ca1, t0);
```

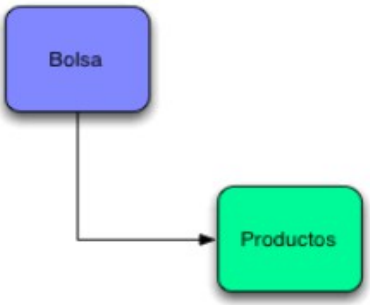
```
            Runnable hilo2 = new Ejemplo3(c2, ca2, t0);
            new Thread(hilo1).start();
            new Thread(hilo2).start();
        }

        @Override
        public void run() {
            this.ca.procesarCompra( this.c, this.t);
        }
    }
```

Sin embargo, ya extiendas Thread o implementes Runnable, el método run() no es capaz de devolver ningún dato/objeto como resultado de su trabajo, como mucho, los distintos Threads pueden comunicarse a través de variables (pues todos acceden a las variables globales del programa, pues pertenecen al mismo proceso) o deben utilizar mecanismos interprocesos.

SINCRONIZAR THREADS CON WAIT Y NOTIFY

Todos conocemos la clase Object y los métodos principales como equals() y hashCode(). Sin embargo a mucha gente le cuesta entender para que sirven **wait()** y **notify()**, dos métodos que relacionados con la programación concurrente. Para probarlos vamos a crear dos clases sencillas (Bolsa y Producto) donde una Bolsa contiene varios Productos.




```
import java.util.ArrayList;

public class Bolsa {
    private ArrayList<Producto> lp = new ArrayList<>();
```

```
    public void addProducto(Producto p) {
        if( !estaLlena() ) lp.add(p);
    }

    public ArrayList<Producto> getListaProductos() { return lp; }
    public int getSize() { return lp.size(); }
    public boolean estaLlena() { return lp.size() >= 5; }
}

// Fichero Producto.java
public class Producto {
    private String nombre;

    public String getNombre() { return nombre; }

    public void setNombre(String nombre) { this.nombre = nombre; }
}
```

Lo único que tiene de peculiar la bolsa que hemos creado es que no permite que se le añadan más de 5 elementos. Ahora vamos a construir dos Threads un primer Thread que va llenando poco a poco la bolsa (thread main) y otro Thread que cuando la bolsa este llena la envía. Vamos a ver el Thread que envía la bolsa:

```
public class HiloEnvio extends Thread {
    private Bolsa bolsa;

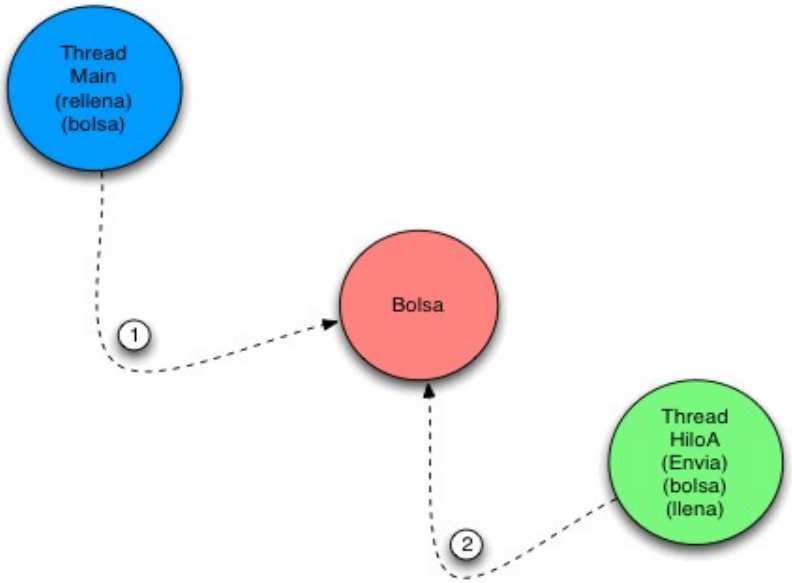
    public HiloEnvio(Bolsa bolsa) {
        super();
        this.bolsa = bolsa;
    }

    @Override
    public void run() {
        if( bolsa.estaLlena() != true ) {
            try {
                synchronized(bolsa) {  bolsa.wait(); }
            }
            catch(InterruptedException e) {
                e.printStackTrace();
            }
```

```
                System.out.println("Enviando bolsa con " +
                                   bolsa.getSize()+ " elementos");
            }
        }

        public Bolsa getBolsa() { return bolsa; }
        public void setBolsa(Bolsa bolsa) { this.bolsa = bolsa; }
    }
```

Este Hilo se encarga de sacarnos por pantalla el mensaje "Enviando bolsa con 5 elementos". Para ello recibe la Bolsa como parámetro y chequea que esta llena. Usa un bloque de código sincronizado (syncronized). Este bloque de código coordina el trabajo entre nuestros dos Threads y permite que un Thread llene la Bolsa y cuando este llena el otro Thread la envíe.



Lamentablemente ambos Threads deben de estar sincronizados ya que hasta que la bolsa no este llena no la podemos enviar. Para sincronizar

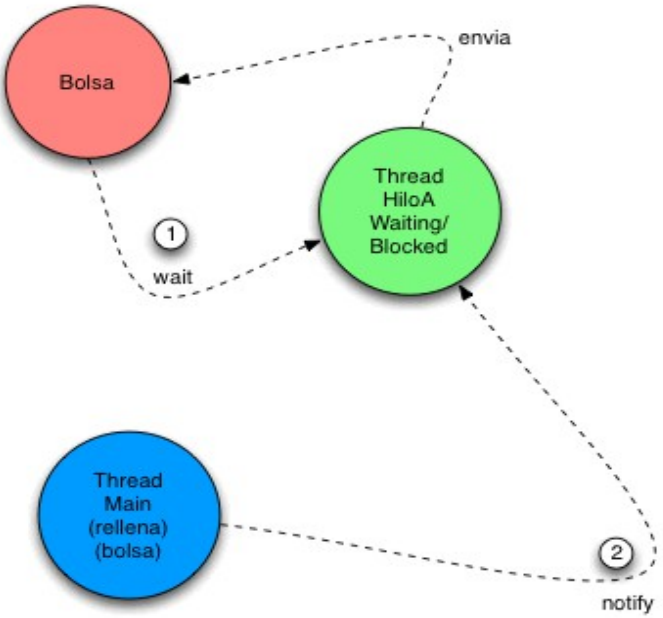
el trabajo de los Threads usamos la pareja wait-notify. De tal forma que nuestro primer HiloEnvio se pondrá a esperar (wait) hasta que la Bolsa este llena antes de enviarla. Para rellenar la bolsa usamos el Thread del programa principal que cada segundo añadirá un nuevo elemento a la lista.

```
public class Principal {

    public static void main(String[] args) {
        Bolsa bolsa = new Bolsa();
        HiloEnvio hilo= new HiloEnvio(bolsa);
        hilo.start();
        for(int i = 0; i <= 10; i++) {
            Producto p = new Producto();
            try {
                synchronized(bolsa) {
                    Thread.sleep(1000);
                    if( bolsa.estaLlena() ) {
                        bolsa.notify();
                    }
                }
            }
            catch(InterruptedException e) {
                 e.printStackTrace();
            }
            bolsa.addProducto(p);
            System.out.println(bolsa.getSize());
        }
    } // fin main
} // fin clase Principal
```

El uso de wait() y notify() es muy habitual en programación concurrente y son parte de la clase Object ya que lo que bloqueamos y sincronizamos es el acceso a cada uno de los objetos que comparten los threads.

Wait() bloquea al thread hasta que otro lo desbloquee. Si hay varios bloqueados en el mismo objeto, se libera uno de ellos y el resto sigue bloqueado.



SERVICIO JAVA EXECUTOR

Java Executor Service pertenece al API de Java7 y es una de las clases que nos permite gestionar la programación concurrente de una forma más sencilla y óptima. Vamos a ver un ejemplo, para ello nos vamos a construir una clase **Tarea** que realice un pequeño bucle por pantalla.

```
public class Tarea implements Runnable {
    private String nombre;

    public Tarea(String nombre) {
        super();
        this.nombre = nombre;
    }
```

```
    @Override
    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println(nombre);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Como podemos ver es una clase normal que implementa el interface Runnable y que tiene un pequeño bucle que ejecutamos cada segundo. Vamos a crear un programa con 3 tareas y las ejecutamos cada una en su hilo.

```
public class Principal {

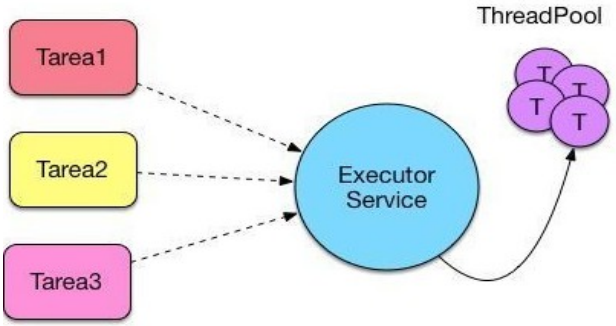
    public static void main(String[] args) {
        Thread t1 = new Thread( new Tarea("tarea1") );
        t1.start();
        Thread t2 = new Thread( new Tarea("tarea2") );
        t2.start();
        Thread t3 = new Thread( new Tarea("tarea3"));
        t3.start();
    }
}
```

La funcionalidad es un poco repetitiva. ¿Podemos realizarla de otra forma? Vamos a construir un Stream con tres cadenas y convertir estas cadenas a objetos Runnables. Hecho esto invocaremos al método **execute()** de un Executor Service que se encarga de automatizar la ejecución de cualquier objeto Runnable.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.stream.Stream;
```

```
public class PrincipalExecutor {
    public static void main(String[] args) {
       Stream<String> flujo =
                          Stream.of("tarea1", "tarea2", "tarea3");
       ExecutorService servicio = Executors.newCachedThreadPool();
       flujo.map(t->new Tarea(t)).forEach( servicio::execute );
    }
}
```

El resultado será identico pero tendrá varias ventajas. En primer lugar la reducción de código y la flexibilidad que tenemos si utilizáramos un stream infinito. En segundo lugar, al no ser nosotros los que inicializamos los Threads, dejamos al API de Java que se encargue de hacerlo de la forma correcta. Por ejemplo en este caso si tuviéramos muchas tareas que ejetutar, el API cachearía un numero determinado de Threads y usaría solo estos.



INTERFACE CALLABLE

Una clase que implementa la interfaz Callable debe implementar su método call() que es similar al run() de Runnable, pero al contrario que este, si devuelve resultado.

```
import java.util.concurrent.Callable;

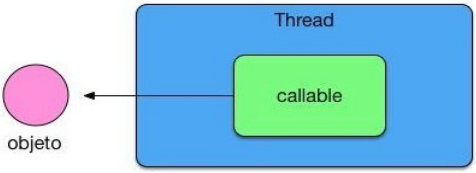
public class MiCallable implements Callable<Integer> {
```

```
    @Override
    public Integer call() throws Exception {
        int total = 0;
        for(int i = 0; i < 5; i++) {
            total += i;
            try {
                Thread.sleep(300);
            }
            catch(InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println( Thread.currentThread().getName() );
        return total;
    }

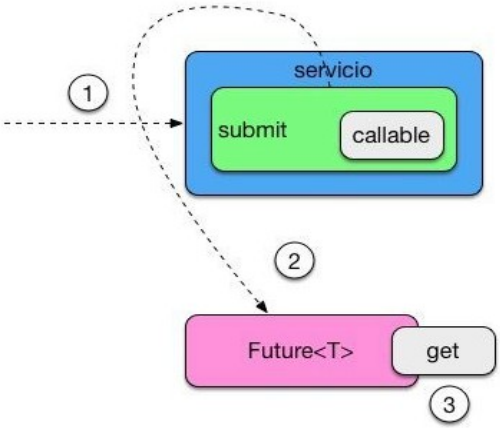
}

// Fichero PrincipalCallable
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class PrincipalCallable {
   public static void main(String[] args) {
      try {
        ExecutorService servici = Executors.newFixedThreadPool(1);
        Future<Integer> resul = servici.submit(new MiCallable());
        if( resul.isDone() ) {
            System.out.println( resul.get() );
        }
      }
      catch( InterruptedException e ) {
          e.printStackTrace();
      }
      catch( ExecutionException e ) {
          e.printStackTrace();
      }
   }
}
```



Cuando invoquemos el servicio recibiremos de forma automática una variable de tipo **Future**, la cual recibirá en un futuro un valor que podremos imprimir usando el método **get()**.




```
public class CallableDemo {

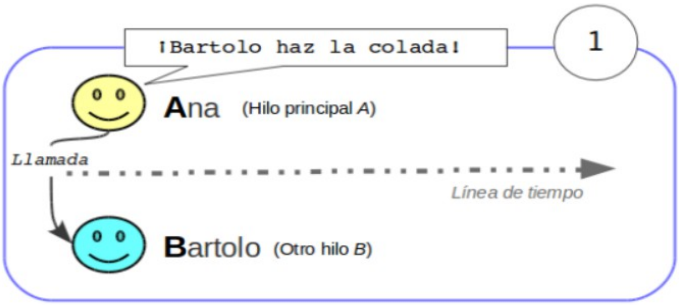
    public static void main(String[] args) {
    // Este método requiere la asistencia del grupo de subprocesos
    // para implementar subprocesos múltiples
    // El subprocesamiento múltiple implementado tiene un retorno.
        MyCallable callable = new MyCallable();
        ExecutorService eS = Executors.newFixedThreadPool(1);
        Future<String> f = eS.submit(callable);
        try {
           // bloqueo hasta que el hilo se ejecute y devuelva val
           String result = f.get();
           System.out.println(result);
```

```
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    static class MyCallable implements Callable<String> {
        @Override
        public String call() throws Exception {
            System.out.println("Hilo en ejecución ->" +
                               Thread.currentThread().GetName() );
            return "Ejecución completada";
        }
    }
```

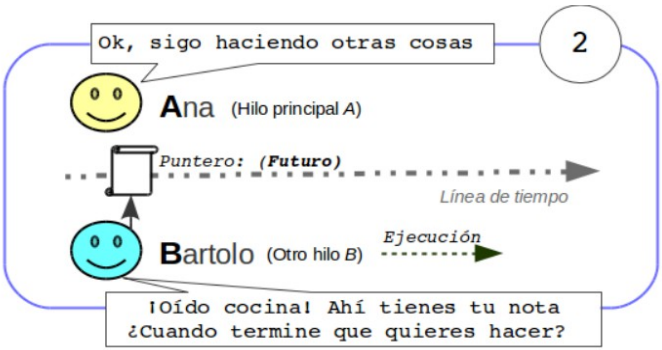
PATRÓN FUTURO-PROMESA Y PROGRAMACIÓN ASÍNCRONA

Imaginemos dos procesos: uno será el principal que llamará a un método que devuelve un futuro de un valor. El otro será el que calcule realmente el valor de manera asíncrona ejecutado en un segundo plano con respecto al principal. Voy a poner un ejemplo para hacerlo más didáctico. Ana (el hilo A, el principal) le dice a Bartolo (el hilo B) que haga la colada (llamada al método).

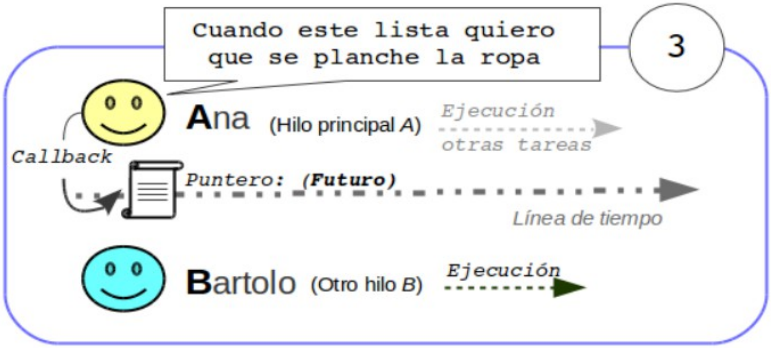


El proceso principal obtiene un futuro como resultado de llamar a un método que se lanzará en segundo plano gracias al hilo B. En el ejemplo,

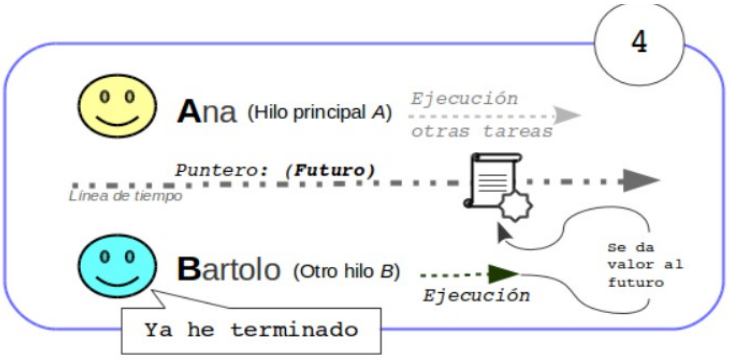
Bartolo manda una nota (representa el futuro) a Ana que le permitirá apuntar lo que se quiere que se haga con la colada cuando haya terminado Bartolo. El hilo A no queda a la espera del resultado de la acción ejecutada por el hilo B y queda liberado para realizar otras operaciones. En el ejemplo Ana no espera a que Bartolo termine y así puede seguir haciendo otras tareas.



Este futuro es en realidad un artefacto que permite asignar, desde el proceso principal una función callback. En el siguiente paso, Ana decidirá que es lo que quiere hacer cuando termine la colada, es decir, asignará la función callback que se ejecutará cuando se obtenga el resultado del segundo proceso ejecutado por Bartolo.



Cuando, gracias al segundo proceso, se haya calculado finalmente el valor, se ejecutará la función de regreso a la que se le pasará como argumento el valor final obtenido. En el ejemplo tenemos la función "planchar la ropa" y el resultado del proceso: "la ropa lavada".



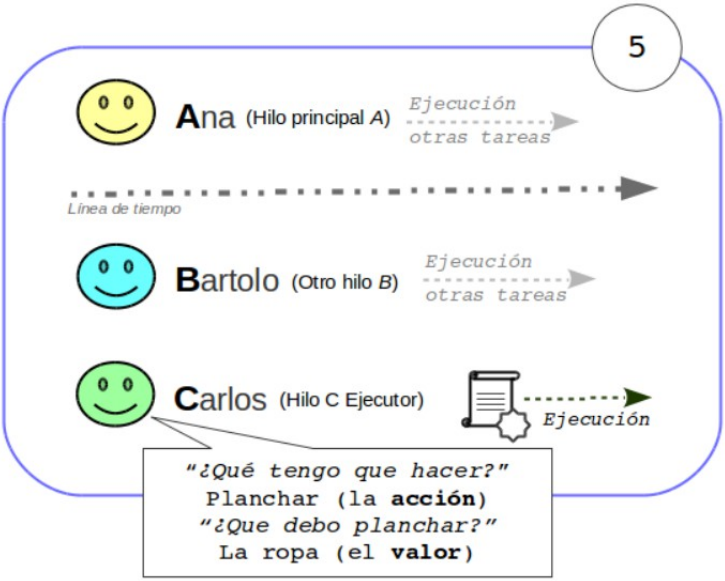
Contextos de ejecución

La ejecución de las funciones callback con el valor final como argumento se hará dentro de lo que se llama un contexto de ejecución. Este no es más que un pool de hilos que permite independizar la ejecución de la función con el valor final tanto del hilo principal como del segundo proceso, es decir, se ejecutarán en un tercer hilo "reutilizable". En el ejemplo que estamos siguiendo es Carlos (el hilo C) el que plancha la ropa.

En la implementación del patrón que he propuesto a cada una de las funciones callback y de orden superior definidas en el interfaz futuro se le puede pasar un contexto de ejecución.

A grandes rasgos, esta es la filosofía: el futuro permite configurar desde la ejecución principal que hacer cuando finalmente exista el valor. Se hace sin tener que bloquear hilos para permitir la asincronía. Teniendo esto en mente y sin perder de vista lo simple de este

concepto, base de todo, vamos a ir añadiendo nuevos conceptos e ideas que nos permitirán ir completando la visión general del patrón.



Dos caminos

Añadiendo una nueva capa a las funciones callback. El resultado de la ejecución de cualquier método puede ser, o bien su valor, o bien que su ejecución haya acabado con un error, luego el resultado final que contiene un futuro puede ser:

- un valor correcto.
- o también una excepción.

¿Por qué no contemplar esta circunstancia con dos caminos? Un camino para cuando el valor sea correcto con su correspondiente función callback (el método **onSuccess**) y otro que nos permita definir otra función de regreso cuando se produzca un error (el método **onFailure**). Signatura de un **onSuccess()** y **onFailure()** de **AlternativeFuture<T>**:

```
void onSuccesful( final Consumer<T> function, final Executor executor );
```

El argumento function es una función cuyo argumento es de tipo T (es el tipo del valor final del futuro) y resultado void.

```
function: T => void
```

Esta será la función que se ejecutaría cuando de resultado final del futuro ha ido bien.

```
void onFailure( final Consumer<Throwable> function, final Executor e );
```

En este caso el argumento function es una función cuyo argumento es un Throwable (la excepción que se reportaría si se produciría un error) y resultado void.

```
function: Throwable => void
```

Está será la función que se ejecutará cuando el futuro se haya solventado con un error.

¿Cuando se ejecuta la función callback?

Para dar respuesta a esta pregunta nos debemos fijar en que existan las dos cosas necesarias para la ejecución de la función: que exista la función y que exista el valor. Hay entonces dos posibilidades:

- Puede que ya haya asignada una o varias funciones callback al futuro, entonces la ejecución de las mismas será cuando se obtenga el valor.
- O bien puede ser que se obtenga primero el valor y el disparador de que la función se ejecute será la propia asignación de la función callback.

Es importante tener en mente que un futuro ("barra" promesa, más adelante), sólo se le puede asignar el valor una vez, además otra característica es que una función callback sólo se ejecutará una vez. Sin embargo se puede asignar al futuro, a lo largo del tiempo, todas las

funciones callback que se deseen.

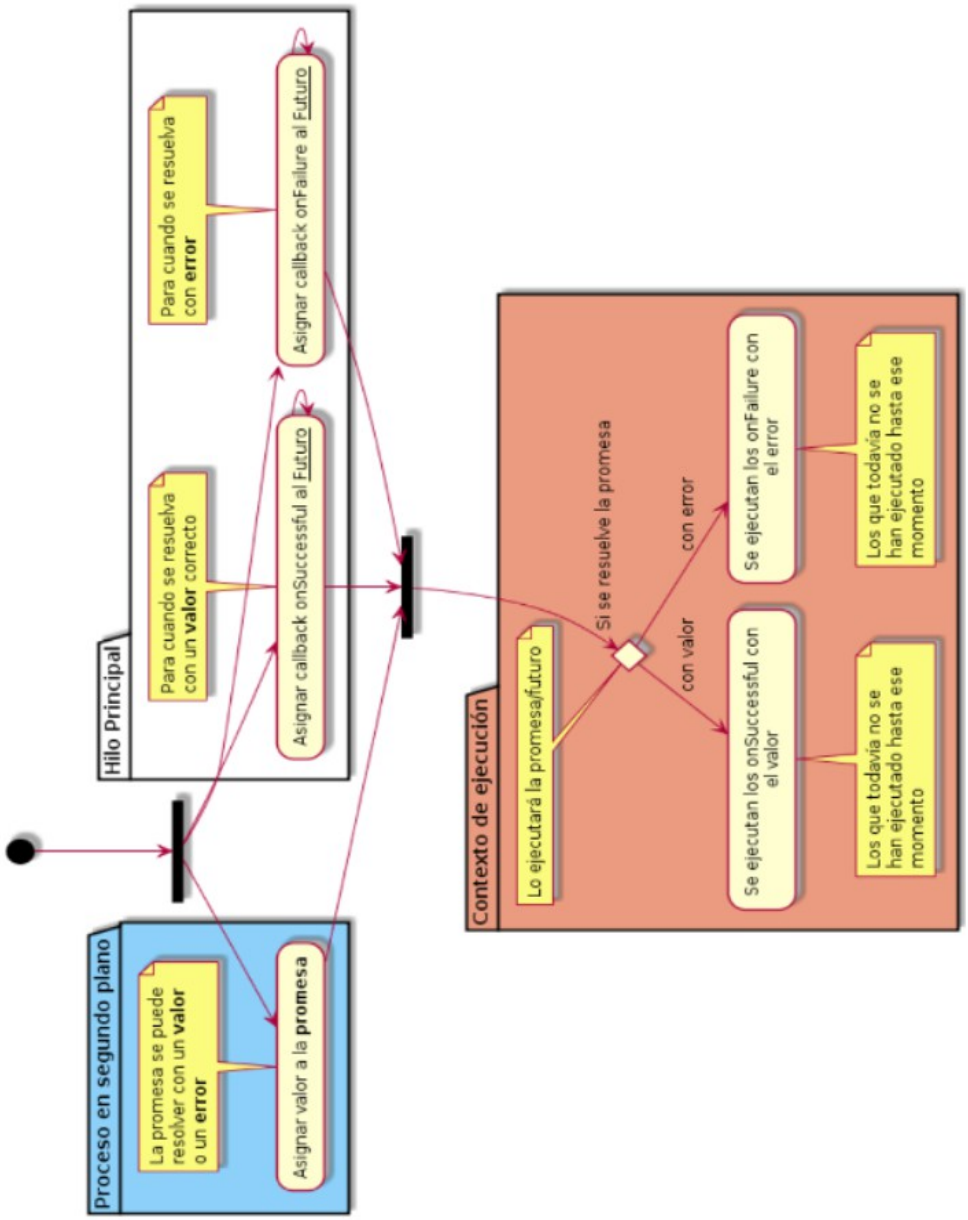
Promesas

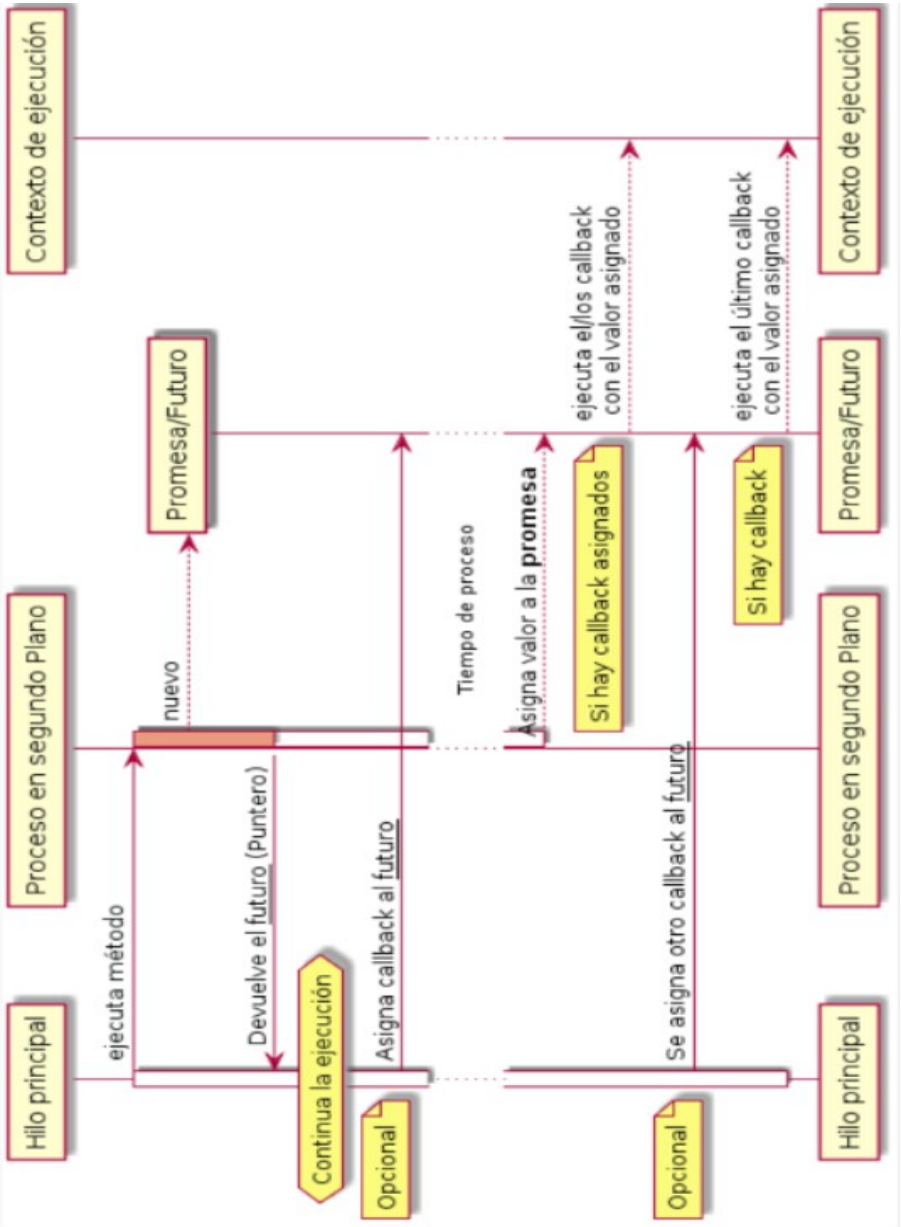
Si has llegado hasta aquí probablemente ya te habrá asaltado la duda: ¿Cómo se da valor al futuro? ¿Cómo se le asigna un valor? La respuesta: a través de la promesa. Podemos decir que la promesa y el futuro son las dos caras de la misma moneda y que una promesa está relacionada con un futuro. La idea básica que debemos tener en mente es la siguiente:

- el futuro permite asignar la función callback para tomar decisiones una vez se haya obtenido el valor.
- la promesa permite dar valor a ese futuro.

Si volvemos al ejemplo anterior, será más sencillo de entender. Si recordamos el proceso principal llamaba a un método que permite en un segundo proceso calcular un valor de manera asíncrona en un tiempo indeterminado (puede ser antes o después). Mientras que el proceso principal tiene como herramienta al interfaz futuro que le permite indicar que es lo que se debe de hacer una vez se haya calculado el valor, el interfaz de la promesa será el instrumento que tiene el segundo proceso encargado de calcular el valor para asignar el valor a ese futuro. En síntesis Ana usaría el futuro y Bartolo la promesa.

Las dos siguientes figuras muestran un esquema del funcionamiento de este patrón.


Proceso en segundo plano
La promesa se puede resolver con un valor o un error
Asignar valor a la promesa
Hilo Principal
Para cuando se resuelva con un valor correcto
Para cuando se resuelva con error
Asignar callback onSuccessful al Futuro
Asignar callback onFailure al Futuro
Contexto de ejecución
Lo ejecutará la promesa/futuro
Si se resuelve la promesa
con valor
con error
Se ejecutan los onSuccessful con el valor
Se ejecutan los onFailure con el error
Los que todavía no se han ejecutado hasta ese momento
Los que todavía no se han ejecutado hasta ese momento


Contexto de ejecución
Proceso en segundo Plano
Promesa/Futuro
Hilo principal
ejecuta método
nuevo
Devuelve el futuro (Puntero)
Continua la ejecución
Asigna callback al futuro
Opcional
Tiempo de proceso
Asigna valor a la promesa
Si hay callback asignados
ejecuta el/los callback con el valor asignado
Se asigna otro callback al futuro
Opcional
Si hay callback
ejecuta el último callback con el valor asignado
Contexto de ejecución
Proceso en segundo Plano
Promesa/Futuro
Hilo principal

Implementación de Promesa y Futuro

Sólo existe una implementación que implementa las dos interfaces: la promesa y el futuro. La clase ***AlternativePromiseImp*** tiene un atributo que permite almacenar el valor del futuro. Este se asignará gracias a los métodos que expone el interfaz promesa AlternativePromise.
Existen dos atributos más en AlternativePromiseImp que son del tipo de cola FIFO (el primero que entra será el primero que sale). Permiten almacenar las funciones callback gracias a los métodos expuestos por el interfaz AlternativeFuture.
Una de las colas guardará las funciones que se van a ejecutar cuando mediante la promesa relacionada se asigne un valor correcto al futuro. Las funciones se van añadiendo a esta cola utilizando el método 'onSuccesful' del interfaz AlternativeFuture.

La otra cola FIFO almacenará las funciones callback que se ejecutarán cuando la promesa relacionada se resuelva con un error. De la misma manera, se van añadiendo estas las funciones gracias al método 'onFailure' del interfaz AlternativeFuture.

Bien cuando se asigna un valor a la promesa o bien cuando se añade una función callback al futuro se comprueba que existan los requisitos necesarios (la función y el valor del futuro). Si es así se ejecutará las funciones callback incluidas en la cola de callbacks con el valor final. Según se vayan ejecutando las funciones se eliminarán de la cola.

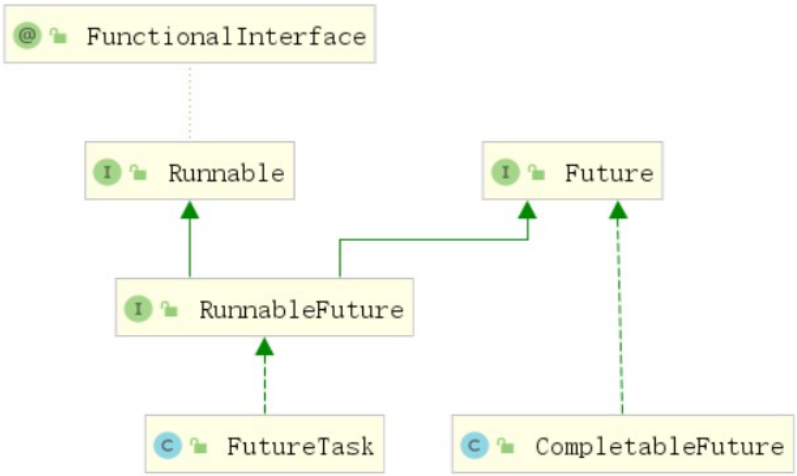
El Futuro como mónada.

Se puede dar una vuelta de tuerca más y seguir añadiendo capas a la cebolla. El patrón Futuro/promesa permite adoptar un nuevo patrón funcional: la monada. Os dejo pospuesto investigar sobre el asunto.

9.4.2 FUTUROS Y PROMESAS EN JAVA.

La programación asíncrona nos da la capacidad de "diferir" la ejecución de una función a la espera de que se complete una operación, normalmente de I/O (red, disco duro, …), y así evitar bloquear la ejecución hasta que se haya completado la tarea en cuestión. Esto es posible gracias a que las funciones son ciudadanos de primer nivel (first-class citizens) y pueden ser pasadas como argumentos de otras funciones tal como hacemos con las variables.

La librería **java.util.concurrent** aporta la interface **Future<T>** para indicar un valor de tipo T que estará disponible en el futuro. Sin embargo las posibilidades son muy limitadas hasta la oparición de futuros completables, que facilitan la implementación de operaciones asíncronas.



INTRODUCCIÓN A COMPLETABLEFUTURE

Future es una clase agregada por Java 5 para describir el resultado de

un cálculo asincrónico. puedes usar el método **isDone()** para verificar si el cálculo está completo o usar **get()** para que bloquee el hilo de llamada hasta que se complete el cálculo y devuelva el resultado, también puedes usar el método **cancel()** que detiene la ejecución de la tarea.
A pesar de que Future y sus métodos proporcionan la capacidad de ejecutar tareas de forma asincrónica, los resultados de esas tareas solo se pueden obtener bloqueando o sondeando. El método de bloqueo es obviamente contrario a nuestra intención original de programación asincrónica. El método de sondeo consume recursos innecesarios de la CPU y los resultados del cálculo no pueden obtenerse a tiempo. Lo ideal es usar el patrón de diseño del observador para notificar al oyente cuando se completan los resultados del cálculo.

En Java 8, se ha agregado una nueva clase con aproximadamente 50 métodos: CompletableFuture, que proporciona una extensión muy poderosa de Future, que puede ayudarnos a simplificar la complejidad de la programación asincrónica y proporciona capacidades de programación funcional, que pueden pasarse a través de devoluciones de llamada.

La clase **CompletableFuture** implementa la interfaz **Future**, por lo que aún puedes usarla como antes para obtener resultados bloqueando o sondeando, pero no se recomienda este método.

CompletableFuture y FutureTask pertenecen a la clase de implementación de la interfaz Future, y ambos pueden obtener el resultado de ejecución del hilo.

CompletableFuture proporciona 4 métodos estáticos para crear una operación asincrónica.

- `static CompletableFuture<Void> runAsync(Runnable runnable)`
- `public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)`
- `public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)`
- `public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)`

El método sin especificar Executor usa **ForkJoinPool.commonPool()** como su grupo de subprocesos para ejecutar código asincrónico. Si se especifica un grupo de subprocesos, el grupo de subprocesos especificado se utiliza para la operación. Todos los métodos son similares.

- El método runAsync no admite valores de retorno.
- supplyAsync puede admitir el valor de retorno.

Método de devolución de llamada cuando se completa el cálculo
Cuando se completa el resultado del cálculo de CompletableFuture o se lanza una excepción, se puede ejecutar una acción específica. Los principales métodos son los siguientes:

- `public CompletableFuture<T> whenComplete(BiConsumer<? super T,? super Throwable> action);`
- `public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T,? super Throwable> action);`
- `public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor);`
- `public CompletableFuture<T> exceptionally(Function<Throwable,? extends T> fn);`

whenComplete() puede manejar resultados de cálculo normales y anormales.

BiConsumer <? Super T ,? super Throwable> puede definir la acción de procesamiento. La diferencia entre whenComplete() y whenCompleteAsync() es que en la primera el hilo que ejecuta la tarea

actual continúa ejecutándose cuando finaliza y en la segunda se envía la tarea al grupo de subprocesos para su ejecución. El método no termina con Async, lo que significa que Action usa el mismo hilo para ejecutar, y Async puede usar otros hilos para ejecutar (si usa el mismo grupo de hilos, también puede ser seleccionado para ejecución por el mismo hilo)

Ejemplo:

```
public class CompletableFutureDemo {

    public static void main(String[] args)
           throws ExecutionException, InterruptedException {
      CompletableFuture f = CompletableFuture
        .supplyAsync(
            new Supplier<Object>() {
               @Override
               public Object get() {
                System.out.println(Thread.currentThread().getName() +
                                   "\t completableFuture");
                int i = 10 / 0;
                return 1024;
            }
        })
        .whenComplete(new BiConsumer<Object, Throwable>() {
            @Override
            public void accept(Object o, Throwable throwable) {
                System.out.println("-------o= " + o.toString());
                System.out.println("-------throwable= " + throwable);
            }
        })
        .exceptionally(new Function<Throwable, Object>() {
            @Override
            public Object apply(Throwable throwable) {
                System.out.println("throwable= " + throwable);
                return 6666;
            }
        });
        System.out.println( f.get() );
    }
}
```

Métodos de manejo

Cuando se completa la tarea hay que pocesar los resultados y **handle()** y **handleAsync()** se ejecuta después de completar la tarea, y también puede manejar tareas anormales.

- `public <U> CompletionStage<U> handle(BiFunction<? super T, Throwable, ? extends U> fn);`
- `public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn);`
- `public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn,Executor executor);`

Método de serialización de hilos

thenApply(): Cuando un subproceso depende de otro subproceso porque debe obtener el resultado devuelto por la tarea anterior y devuelver el valor de retorno a la tarea actual.

ThenAccept(): consume los resultados de procesamiento. Recibe el resultado de procesamiento de la tarea y consume el procesamiento, no hay resultado de retorno.

thenRun(): siempre y cuando se complete la tarea anterior, se ejecutará el ejecutable thenRun, pero después de procesar la tarea, se ejecutará la operación posterior de thenRun.

Con Async se ejecuta de forma asincrónica, se refiere a no ejecutarse en el hilo actual.

- `public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)`
- `public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn)`
- `public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)`
- `public CompletionStage<Void> thenAccept(Consumer<? super T> action);`
- `public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);`
- `public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action,Executor executor);`
- `public CompletionStage<Void> thenRun(Runnable action);`

- `public CompletionStage<Void> thenRunAsync(Runnable action);`
- `public CompletionStage<Void> thenRunAsync(Runnable action,Executor executor);`

Function<? super T,? extends U>

- T: el tipo de resultado devuelto por la tarea anterior
- U: el tipo de valor de retorno de la tarea actual

Ejemplo:

```
public static void main(String[] args)
       throws ExecutionException, InterruptedException {
    CompletableFuture<Integer> f = CompletableFuture
      .supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            System.out.println(Thread.currentThread().getName() +
                 "\t completableFuture");
            return 1024;
        }
    })
      .thenApply(new Function<Integer, Integer>() {
        @Override
        public Integer apply(Integer o) {
          System.out.println("then, resultado devuelto antes: " + o);
          return  o * 2;
        }
    })
      .whenComplete(new BiConsumer<Integer, Throwable>() {
        @Override
        public void accept(Integer o, Throwable throwable) {
            System.out.println("-------o=" + o);
            System.out.println("-------throwable=" + throwable);
        }
    })
      .exceptionally(new Function<Throwable, Integer>() {
        @Override
        public Integer apply(Throwable throwable) {
            System.out.println("throwable=" + throwable);
            return 6666;
        }
    })
    .handle(new BiFunction<Integer, Throwable, Integer>() {
        @Override
        public Integer apply(Integer integer, Throwable throwable) {
```

```
                System.out.println("handle o=" + integer);
                System.out.println("handle throwable=" + throwable);
                return 8888;
            }
        });
        System.out.println(future.get());
    }
```

Combinación de dos tareas, ambas deben completarse

Ambas tareas deben completarse para activar la nueva tarea.

thenCombine(): combina dos futuros, obtiene el resultado de retorno de los dos futuros y devuelve el valor de retorno de la tarea actual.

thenAcceptBoth(): combina dos futuros, obtiene el resultado de dos tareas futuras y luego procesa la tarea, no hay valor de retorno.

runAfterBoth(): combina dos futuros, no necesita obtener el resultado del futuro, solo necesita dos futuros para procesar la tarea, luego procesa la tarea.

```
public <U,V> CompletableFuture<V> thenCombine(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn);

public <U,V> CompletableFuture<V> thenCombineAsync(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn);

public <U,V> CompletableFuture<V> thenCombineAsync(
    CompletionStage<? extends U> other,
    BiFunction<? super T,? super U,? extends V> fn, Executor executor);

public <U> CompletableFuture<Void> thenAcceptBoth(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action);

public <U> CompletableFuture<Void> thenAcceptBothAsync(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action);

public <U> CompletableFuture<Void> thenAcceptBothAsync(
    CompletionStage<? extends U> other,
    BiConsumer<? super T, ? super U> action, Executor executor);
```

```
public CompletableFuture<Void> runAfterBoth(CompletionStage<?> other,
                                            Runnable action);

public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                 Runnable action);

public CompletableFuture<Void> runAfterBothAsync(CompletionStage<?> other,
                                                 Runnable action,
                                                 Executor executor);
```

Ejemplo:

```
public static void main(String[] args) {
    CompletableFuture.supplyAsync(() -> {
        return "hello";
    }).thenApplyAsync(t -> {
        return t + " world!";
    })
 .thenCombineAsync(
    CompletableFuture.completedFuture(" CompletableFuture"), (t, u) -> {
        return t + u;
    }).whenComplete((t, u) -> {
        System.out.println(t);
    });
}
```

Combinar dos tareas, al menos una se ejecuta

Cuando se completa cualquiera de las dos tareas futuras, la tarea se ejecuta.

- **applyToEither()**: una de las dos tareas se completa, obtiene su valor de retorno, procesa la tarea y tiene un nuevo valor de retorno.
- **acceptEither()**: se completa una de las dos tareas, obtiene su valor de retorno, procesa la tarea, no hay un nuevo valor de retorno.
- **runAfterEither()**: se completa una de las dos tareas, no es necesario obtener el resultado del futuro, la tarea se procesa y no hay valor de retorno.

```
public <U> CompletableFuture<U> applyToEither(
    CompletionStage<? extends T> other, Function<? super T, U> fn);
```

```
public <U> CompletableFuture<U> applyToEitherAsync(
    CompletionStage<? extends T> other, Function<? super T, U> fn);

public <U> CompletableFuture<U> applyToEitherAsync(
    CompletionStage<? extends T> other, Function<? super T, U> fn,
    Executor executor);

public CompletableFuture<Void> acceptEither(
    CompletionStage<? extends T> other, Consumer<? super T> action);

public CompletableFuture<Void> acceptEitherAsync(
    CompletionStage<? extends T> other, Consumer<? super T> action);

public CompletableFuture<Void> acceptEitherAsync(
    CompletionStage<? extends T> other, Consumer<? super T> action,
    Executor executor);

public CompletableFuture<Void> runAfterEither(CompletionStage<?> other,
                                              Runnable action);

public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
                                                   Runnable action);

public CompletableFuture<Void> runAfterEitherAsync(CompletionStage<?> other,
                                                   Runnable action,
                                                   Executor executor);
```

Combinación de tareas múltiples

- `public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs);`
- `public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs);`

allOf(): espera que se completen todas las tareas
anyOf(): siempre que se complete una tarea

REPASO EXPLICADO

Considera un método:

```
public void Future<String> leePagina(URL url)
```

Lee una página web en un hilo separado. Cuando lo llamas:

```
Future<String> contenido = leePagina(url);
```

El método se jecutará y lo almacenas como un Future<String>. Ahora imagina que quieres extraer todas las URL's de la página para construir un rastreador web. Tenemos una clase llamada Parser con un método público:

```
public static List<URL> getLinks(String pagina)
```

¿Cómo aplicar este método al objeto futuro? La única manera es llamar al método get() del futuro para obtener su valor cuando esté disponible:

```
String pagina = contenido.get();
List<URL> links = Parser.getLinks(pagina);
```

Pero la llamada a get() es bloqueante y nuestro thread quedará suspendido hasta que el otro complete su trabajo.

Vamos a cambiar el método leepagina() para que devuelva un CompletableFuture<String>. Como el nuevo valor tiene un método thenApply() al que podemos pasarle la función que debe realizar con el resultado, nuestro thread no tiene que quedar bloqueado:

```
CompletableFuture<String> contenidos = leePagina(url);
CompletableFuture<List<String>> links =
                          contenidos.thenApply( Parser::getLinks );
```

Devuelve otro futuro. Cuando el primer futuro se haya completado, se ejecutará el método getLinks() en otro thread y devolverá el resultado final. Esta facilidad para componer resultados es la clave de la clase CompletableFuture porque soluciona el mayor problema de la programación asíncrona. La solución tradicional para evitar bloqueos era usar eventos. Los programadores registraban manejadores de eventos para la siguiente tarea, que también era asíncrona, de forma que se pensaba en términos de "primer paso", "segundo paso" y la

lógica del proceso se dispersaba en estos pasos. Pero manejar errores de esta forma es un desafío. Imagina que en el paso 2, necesitamos volver al paso 1, implementar control de flujo en este esquema también es horrible.

Pipeline de Composición de Futuros

En los apartados anteriores de streams hemos visto como podíamos realizar procesos y que sus resultados fuesen la entrada de otros o acabasen consumidos en un proceso final. De la misma manera podemos crear un pipeline de futuros. Comenzamos generando un CompletableFuture, normalmente con el método estático **supplyAsync()**. Este método necesita un **Supplier<T>**, que es una función sin parámetros que devuelve un tipo T. La función se llamará en un Thread separado. Aplicado a nuestro ejemplo:

```
CompletableFuture<String> contenidos =
        CompletableFuture.supplyAsync( () -> bloqueanteLeePagina(url) );
```

Y también tenemos el método **runAsync()** que acepta un Runnable, y devuelve un CompletableFuture<Void>. Este se usa cuando solo quieres ejecutar una acción detrás de otra, sin pasar datos entre ellas.

> ***Nota**: recuerda que todos los métodos acabados en Async tienen dos variantes. Una ejecuta la acción en el ForkJoinPool común. La otra tiene un parámetro de tipo java.util.concurrent.Executor, y usa ese executor para ejecutar la acción.*

Completable Futures

A continuación puedes llamar a los métodos **thenApply()** o **thenApplyAsync()** para ejecutar otra acción en el mismo thread o en otro. En ambos métodos aportas una función que devuelve un CompletableFuture<U>, donde U es el tipo devuelto por la función.

Aplicado a nuestro ejemplo:

```
CompletableFuture<List<String>> links =
    CompletableFuture
    .supplyAsync( () -> bloqueanteLeePagina(url) )
    .thenApply( Parser::getLinks );
```

Puedes decidir añadir pasos adicionales de procesamiento. O bien simplemente imprimir los resultados como en este ejemplo:

```
CompletableFuture<Void> links =
    CompletableFuture
    .supplyAsync( () -> bloqueanteLeePagina(url) )
    .thenApply(Parser::getLinks)
    .thenAccept( System.out::println );
```

El método **thenAccept()** acepta un Consumer, una función que devuelve void, es decir, usa los datos de las etapas anteriores. Normalmente nunca necesitas llamar a get() para obtener un futuro.

> **Nota**: *no necesitas iniciar explícitamente el procesamiento. El método supplyAsync() lo hace.*

Composición de Operaciones Asíncronas

Ya hemos comentado un gran número de métodos que trabajan con futuros completables. Y para cada uno hay una versión Async. Ya hemos comentado thenApply(). Sus llamadas `CompletableFuture<U> future.thenApply(f);` y `CompletableFuture<U> future.thenApplyAsync(f);` devuelven un futuro que aplica f al resultado cuando esté disponible y la segunda llamada la aplica en otro Thread.

El método **thenCompose()** en vez de tomar una función `T -> U`, toma una función `T -> CompletableFuture<U>`. Que parece un poco abstracto pero es bastante natural. Piensa en el ejemplo, en vez de indicar un método `public String bloqueanteLeepagina(URL url)` es más elegante un método que devuelva un futuro:

```
public CompletableFuture<String> LeePagina(URL url)
```

Tabla 6-1 Añadir acciones a un CompletableFuture<T> Object

Method	Parameter	Description
thenApply	T -> U	Apply a function to the result.
thenCompose	T -> CompletableFuture<U>	Invoke the function on the result and execute the returned future.
handle	(T, Throwable) -> U	Process the result or error.
thenAccept	T -> void	Like thenApply, but with void result.
whenComplete	(T, Throwable) -> void	Like handle, but with void result.
thenRun	Runnable	Execute the Runnable with void result.

Ahora imagina que otro método obtiene la URL del usuario, por ejemplo cuando este pulse un botón:

```
public CompletableFuture<URL> getURLInput(String prompt)
```

Aquí tenemos dos funciones `T -> CompletableFuture<U>` y `U -> CompletableFuture<V>`. Se van a componer en una función `T -> CompletableFuture<V>` llamando a la segunda función cuando la primera se haya completado. Eso es exactamente lo que hace la composición, ejecutar algo tras ejecutar otra cosa que le proporciona el dato con lo que tiene que trabajar.

El tercer método de la tabla se ocupa de algo que hemos ignorado en esta última explicación y son los errores. Cuando una excepción se lanza dentro de un CompletableFuture, se captura y se envuelve en un ExecutionException cuando el método get() se usa. Pero quizás no se lalme nunca. Así que para manejarla, se usa el método handle(). La función indicada se llamará con el resultado (o null si no hay) y la excepción (o null si no hay).

Y ahora recordamos los métodos que combinan dos futuros en paralelo (no uno detrás de otro). Los 3 primeros ejecutan un `CompletableFuture<T>`

y `CompletableFuture<U>` y los siguientes 3 ejecutan dos CompletableFuture<T>. Tan pronto como no de ellos acaba, los del otro se ignoran.

Por último, los métodos estáticos **allOf()** y **anyOf()** aceptan un número variable de futuros completables y devuelven un `CompletableFuture<Void>`.

Method	Parameters	Description
thenCombine	CompletableFuture<U>, (T, U) -> V	Execute both and combine the results with the given function.
thenAcceptBoth	CompletableFuture<U>, (T, U) -> void	Like thenCombine, but with void result.
runAfterBoth	CompletableFuture<?>, Runnable	Execute the runnable after both complete.
applyToEither	CompletableFuture<T>, T -> V	When a result is available from one or the other, pass it to the given function.
acceptEither	CompletableFuture<T>, T -> void	Like applyToEither, but with void result.
runAfterEither	CompletableFuture<?>, Runnable	Execute the runnable after one or the other completes.
static allOf	CompletableFuture<?>...	Complete with void result after all given futures complete.
static anyOf	CompletableFuture<?>...	Complete with void result after any of the given futures completes.

> ***Nota**: técnicamente hablando, los métodos de esta sección aceptan parámetros de tipo CompletionStage, no CompletableFuture. Que es una interface con al menos 40 métodos actualmente solo implementada por la clase CompletableFuture.*

9.5. EJERCICIOS.

Dado el siguiente conjunto de clases e instancias, realiza los siguientes ejercicios:

```
public class Vendedor{
    private final String nombre;
    private final String ciudad;

    public Vendedor(String n, String c){
        this.nombre = n;
        this.ciudad = c;
    }

    public String getNombre(){ return this.nombre; }

    public String getCiudad(){ return this.ciudad; }

    public String toString(){
        return "Vendedor: " + this.nombre + " en " + this.ciudad;
    }
}

public class Venta {
    private final Vendedor ven;
    private final int year;
    private final int valor;

    public Venta(Vendedor ven, int year, int valor) {
        this.ven = ven;
        this.year = year;
        this.valor = valor;
    }

    public Vendedor getVendedor(){ return this.ven; }

    public int getYear(){ return this.year;  }

    public int getValor(){ return this.valor; }

    public String toString(){
        return "{" + this.ven + ", " +
               "year: " + this.year + ", " +
```

```
                "valor:" + this.valor + "}";
    }
}

public class Empleado {
    private String nombre;
    private int edad;

    public Empleado(String n, int e) { nombre = n; edad = e; }
    public getNombre() { return nombre; }
    public getEdad() { return edad; }
}

Vendedor r = new Vendedor("Raul",    "Castellon");
Vendedor m = new Vendedor("Mario",   "Valencia");
Vendedor a = new Vendedor("Alfonso", "Castellon");
Vendedor b = new Vendedor("Bruno",   "Castellon");
List<Vendedor> v = Arrrays.asList(r, m, a, b );
List<Venta> lv = Arrays.asList( new Venta(b, 2021, 300),
                                new Venta(r, 2020, 1000),
                                new Venta(r, 2021, 400),
                                new Venta(m, 2020, 710),
                                new Venta(m, 2021, 700),
                                new Venta(a, 2021, 950)
                              );
Empleado e1 = new Empleado("Jose", 21);
Empleado e2 = new Empleado("Marta", 19);
Empleado e3 = new Empleado("Maria", 31);
Empleado e4 = new Empleado("Maria", 18);
Empleado e5 = new Empleado("Jose", 26);
List<Empleado> le = new Arrays.asList(e1, e2, e3, e4, e5);
```

EJERCICIO 0. En un concurso de la TV se sortean 3 premios a los espectadores que envíen un SMS. Hay que escoger de manera aleatoria a los 3 ganadores de una lista de objetos Espectador que tenemos a continuación. Soluciona los siguientes apartados:

```
class Espectador {
        private String nombre;
        private String telefono = null;
        private Espectador(String n, String t) {
                nombre = n;
                telefono = t;
        }
        // getters y setters, toString(), equals(), compareTo()...
}

ArrayList<Espectador> lista = new ArrayList<>();
lista.add(new Espectador("Mar", "9912345678"));
lista.add(new Espectador("Santi", "9912342222"));
lista.add(new Espectador("Alvarado", "9912345770"));
lista.add(new Espectador("Violeta", "9912345678"));
lista.add(new Espectador("Jaime", "9912345888"));
lista.add(new Espectador("Paco", "9912345111"));
lista.add(new Espectador("Ignacio", "11"));
```

1. Obtener números de letéfono.
2. Obtener números de teléfono correctos (eliminar los que no tengan longitud de 10).
3. Eliminar de la lista anterior los repetidos.
4. Desordenar los números del paso anterior.
5. Quedate con los 3 primeros del paso anterior.
6. Mostrarlos por consola.

EJERCICIO 1. Cuando defines pipelines puedes depurar su funcionamiento añadiendo sentencias que muestren como se aplican. Modifica este trozo de código para imprimir por consola el elemento con el que trabaja la operación map y la operación filter.

```
Stream<String> ns = Stream.of("Maria", "Jose", "Vicente", "Ana");
Stream<String> pJ = ns.map( (s) -> { return s.toUpperCase(); } )
                      .filter( (s)-> { returns.startsWith("J"); } );
```

EJERCICIO 2. Encontrar ventas del año 2020 y ordenarlas por su importe (orden creciente).

EJERCICIO 3. ¿Cuales son las ciudades (sin repetidos) donde trabajan los vendedores?

EJERCICIO 4. Encuentra los vendedores que han realizado ventas en Castellon, ordenados por su nombre.

EJERCICIO 5. Devuelve un string con el nombre de todos los vendedores ordenados alfabéticamente.

EJERCICIO 6. ¿Hay algún comercial en Milan?

EJERCICIO 7. Imprime los valores de las ventas de los vendedores que viven en Madrid.

EJERCICIO 8. ¿Cuál es la venta con el valor más alto?

EJERCICIO 9. Encuentra la venta con el valor más bajo.

EJERCICIO 10. Dada una lista de empleados, encuentra aquellos cuya edad sea mayor de 30 e imprime sus nombres.

EJERCICIO 11. ¿Cuántos empleados mayores de 25 años hay?

EJERCICIO 12. Muestra datos del empleado cuyo nombre es "Jose"

EJERCICIO 13. Muestra la edad del empleado que sea mayor que el resto.

EJERCICIO 14. Muestra los empleados ordenados por su edad.