# Algoritmos paralelos IV Streams

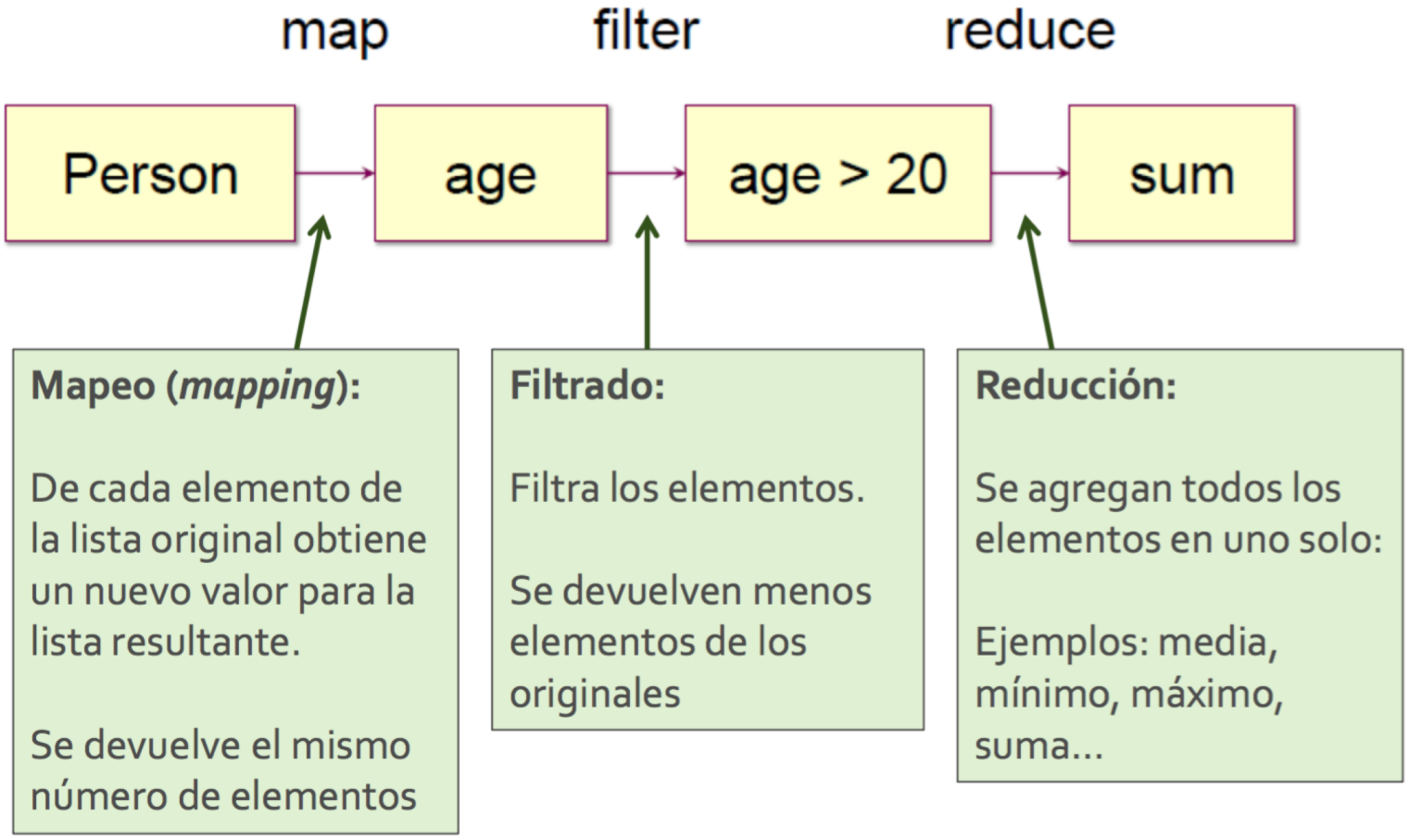
PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 5

Universidad
Rey Juan Carlos

# Java 8 Streams

## Algoritmos Paralelos

- La programación imperativa es muy **verbosa** porque se indica qué y cómo se consigue.
- La programación declarativa es más **compacta**, basta con indicar qué se quiere.
- Los ***streams*** de Java nos permiten aplicar operaciones de mapeo, filtrado y reducción a colecciones de datos.


map
filter
reduce
Person
age
age > 20
sum
Mapeo (*mapping*):
De cada elemento de la lista original obtiene un nuevo valor para la lista resultante.
Se devuelve el mismo número de elementos
Filtrado:
Filtra los elementos.
Se devuelven menos elementos de los originales
Reducción:
Se agregan todos los elementos en uno solo:
Ejemplos: media, mínimo, máximo, suma...

# Java 8 Streams



- Un **stream** es una secuencia de elementos que solo se procesan una vez.
- Pueden ser dinámicos o estáticos.
- Permite desarrollar operaciones muy **eficientes** sin estructuras de datos intermedias.

- Un *stream* representa una **secuencia de elementos** y puede ejecutar una serie de operaciones sobre ellos

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

- Las operaciones sobre *streams* pueden ser intermedias o finales.
  - **Intermedia**: devuelven un *stream* para poder procesarlos en cadena.
  - **Finales**: devuelven o bien *void* o un resultado que no es un *stream*.

http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

- El tipo de operaciones con streams que se hacen en cadena se denominan ***operation pipeline***.

# Java 8 Streams



- Operaciones intermedias:
  - **Filter:** elimina ciertos elementos.
  - **Map:** por cada elementos obtiene un nuevo valor.
  - **Sorted:** ordena los elementos.
  - **Peek:** aplica una operación a cada elemento.
  - **Distinct:** filtra dejando solo los distintos.
  - **Limit:** limita el número de elementos.
  - **Skip:** ignora los primeros elementos.
  - **Range:** devuelve un rango de elementos

- Operaciones finales:
  - **Count:** cuenta los elementos.
  - **Min, max**: obtiene el máximo y el mínimo.
  - **anyMatch, allMatch, noneMatch**: indica si se cumple o no el criterio.
  - **findFirst, findAny**: devuelve el elemento que cumpla el criterio.
  - **mapToInt**: convierte a IntStream.
  - **toArray**: devuelve el contenido como array.
  - **forEach, forEachOrdered**: ejecuta por cada elemento.
  - **Reduce**: mecanismo genérico de reducción.
  - **Collect**: mecanismo genérico para recolectar los elementos

- La mayoría de operaciones con *streams* aceptan una expresión lambda como parámetro.
  - **Expresión lambda**: interfaz funcional que especifica el comportamiento de la operación.
- Hemos visto cómo crear un *Thread* con expresiones lambda:

```
Thread th = new Thread(() -> System.out.println("THREAD_CODE"));
```

- Estas operaciones deben funcionar sin intromisión y sin estado.
  - **Sin intromisión**: no modifica los datos del *stream*. En el ejemplo anterior, no añadimos ni eliminamos datos del mismo.
  - **Sin estado**: la operación es determinista, no depende de ninguna variable externa que pueda cambiar durante la ejecución.

- Los *streams* se pueden crear a partir de diferentes tipos de **colecciones**.
  - Las listas y conjuntos pueden crear streams secuenciales y paralelos.
- El método *stream()* aplicado sobre una lista devuelve un stream **secuencial**.
- No es necesario crear una colección para trabajar con *streams*:

```
Stream.of("a1", "a2", "a3");
```

- Java 8 permite crear *streams* especiales que trabajan con los **tipos primitivos** *int*, *long* y *double*.
  - *IntStream*, *LongStream*, *DoubleStream*
  - IntStream.*range(1, 4);*
- Podemos **transformar** un *stream* genérico en uno específico mediante la operación *mapToInt()*, y revertir la operación con *mapToObj()*:

```
Stream.of(1.0, 2.0, 3.0)
      .mapToInt(Double::intValue)
      .mapToObj(i -> "a" + i)
      .forEach(System.out::println);
```

- Las operaciones intermedias **no se ejecutarán** si no existe ninguna operación final.
  - Evaluación perezosa
- Operación intermedia no ejecutada:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

- Las operaciones intermedias **no se ejecutarán** si no existe ninguna operación final.
  - Evaluación perezosa
- Operación intermedia ejecutada:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

- La salida del código anterior es la siguiente:

```
filter: d2
forEach: d2
filter: a2
forEach: a2
filter: b1
forEach: b1
filter: b3
forEach: b3
filter: c
forEach: c
```

- La operación en cadena se sigue de manera **vertical**.

- Este comportamiento reduce el número de operaciones que se realizan en cada elemento:

```
Stream.of("d2", "a2", "b1", "b3", "c")
        .map(s -> {
            System.out.println("map: " + s);
            return s.toUpperCase();
        })
        .anyMatch(s -> {
            System.out.println("anyMatch: " + s);
            return s.startsWith("A");
        });
```

- En este código, map solo se ejecuta dos veces, en lugar de mapear todo el stream.

- ¿Podemos reducir el número de operaciones que realiza este código?

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

- Sí, solo cambiando el **orden** de las operaciones

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

- ¿Qué pasa si incluimos una operación con estado?
  - Esa operación se ejecuta de manera horizontal

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

- De nuevo, el orden es importante.
- Si ejecutamos el filtro **antes** de la ordenación, reducimos el número de operaciones ejecutadas.
- En el ejemplo anterior *sorted* no llega a ejecutarse, ya que no hay más de un elemento filtrado.

- Los streams de Java se **cierran** en cuanto ejecutamos una operación terminal.

```
Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

    stream.anyMatch(s -> true);    // ok
    stream.noneMatch(s -> true);   // exception
```

- La última instrucción lanza una excepción por ejecutarse sobre un *stream* cerrado.

- Solución: crear un nuevo stream por cada operación final que queramos ejecutar

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

    streamSupplier.get().anyMatch(s -> true);   // ok
    streamSupplier.get().noneMatch(s -> true);  // ok
```

# Operaciones avanzadas

- Supongamos las siguientes definiciones

```
public class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return name;
    }
}
```

```
List<Person> persons =
Arrays.asList(
        new Person("Max", 18),
        new Person("Peter", 23),
        new Person("Pamela", 23),
        new Person("David", 12));
```

- **Collect**
  - Operación terminal que transforma los elementos de un *stream* en un tipo diferente de datos.
  - Acepta un objeto de la clase Collector

```
List<Person> filtered = persons
    .stream()
    .filter(p -> p.name.startsWith("P"))
    .collect(Collectors.toList());
```

- **Collect**
  - Se puede utilizar con colectores más complejos

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) ->
        System.out.format("age %s: %s\n", age, p));
```

- **Collect**
  - Se puede utilizar con colectores más complejos

```
Double averageAge = persons
     .stream()
     .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);

IntSummaryStatistics ageSummary =
     persons
          .stream()
          .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
```

- **Collect**
  - Se puede utilizar con colectores más complejos

```
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(Collectors.joining
            (" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
```

- **Reduce**
  - Combina todos los elementos del stream en un único resultado.
  - Primera variante:

```
persons
      .stream()
      .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
      .ifPresent(System.out::println);
```

- **Reduce**
  - Combina todos los elementos del stream en un único resultado.
  - Segunda variante:

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });

System.out.format("name=%s; age=%s", result.name, result.age);
```

- **Reduce**
  - Combina todos los elementos del stream en un único resultado.
  - Tercera variante:

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age,
               (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum);
```

- Los streams se pueden ejecutar en **paralelo** para incrementar el rendimiento sobre un conjunto de datos **grande**.
  - Utilizan el framework de **Fork / Join** para ello.
  - Para **modificar** el número de threads modificamos el siguiente parámetro de la máquina virtual:
    - -Djava.util.concurrent.ForkJoinPool.common.parallelism=5
  - Se crean mediante *parallelStream()* o utilizando *parallel()* sobre un stream secuencial.

- ¿Qué pasa si ordenamos en paralelo un array?
- Insertamos la operación ordenar justo antes de la operación terminal *forEach*

```
.sorted((s1, s2) -> {
    System.out.format("sort: %s <> %s [%s]\n",
        s1, s2, Thread.currentThread().getName());
    return s1.compareTo(s2);
})
```

- Si analizamos la salida, parece que se ejecuta secuencial.
  - La ordenación paralela de Java 8 solo paraleliza si supera un umbral de granularidad.

- En resumen, los streams paralelos pueden mejorar a los secuenciales cuando consideramos un **tamaño de datos grande**.
- Algunas operaciones como *reduce* o *collect* necesitan operaciones adicionales de combinación que no son necesarias en los secuenciales.
- Están implementados con Fork / Join pool, con todas las ventajas y desventajas que ello conlleva.

- Cuidado al usar streams paralelos
  - Paralelizar tiene un coste de gestión asociado
    - División del trabajo, gestión de hilos, consolidación de resultados, etc.
  - Para pocos datos o ciertos algoritmos, no merece la pena paralelizar, porque es más lento.
  - Siempre debemos comparar con la versión secuencial, para unos datos de entrada habituales.

# Ejercicio



- Crear un fichero con *n* líneas y *m* números (float) separados por un espacio.
  - *n* y *m* deben ser números razonablemente grandes.
- Utilizar los streams de Java para procesar dicho fichero.
  - Se debe calcular el promedio de cada línea.
  - Finalmente se devuelve el máximo de todos los promedios.

# Algoritmos paralelos IV Streams

PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 5

Universidad
Rey Juan Carlos