

Repaso de dos conceptos básicos sobre streams y su buen uso

1. Estilos de programación declarativa e imperativa:
Resolvamos un primer ejercicio con los dos estilos
2. Segundo ejercicio: “¿Qué pasa si mi stream no genera datos?”
Generación eficiente (*lazy evaluation*) de datos con streams
3. En ese caso, mejor no usar sólo streams y estilo declarativo
4. Si podemos usar estilo declarativo y streams, debemos hacerlo:
Tercer ejercicio y reparación del segundo ejercicio

Usaremos estos métodos para descomponer un problema sencillo:

Calcular el doble del primer número par mayor que 4 en una lista dada.

```
public class Auxiliar {  
  
    public static boolean esMayorQue4(int numero) {  
        return numero > 4;  
    }  
  
    public static boolean esPar(int numero) {  
        return numero % 2 == 0;  
    }  
  
    public static int duplica(int numero) {  
        return numero * 2;  
    }  
}
```

Solución **estilo imperativo** (secuencial): bucle hasta encontrar objetivo.

```
public class Main {  
    public static void main(String[] args) {  
        List<Integer> nums = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
        // Calcular el doble del primer número par mayor que 4  
  
        // estilo imperativo (secuencial)  
        // realiza 9 llamadas a métodos de Auxiliar  
        int resultado = 0;  
        for (int n : nums ) {  
            if (Auxiliar.esMayorQue4(n) && Auxiliar.esPar(n)) {  
                resultado = Auxiliar.duplica(n);  
                break;  
            }  
        }  
        System.out.println(resultado);  
    }  
}
```

Solución **estilo declarativo** (funcional): estilo adecuado con streams.

```
public class Main {  
    public static void main(String[] args) {  
        List<Integer> nums = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
        // Calcular el doble del primer número par mayor que 4  
  
        // estilo funcional (declarativo)  
        // aunque parezca complejo no lo es ...  
        // realiza las mismas llamadas a métodos de Auxiliar  
        System.out.println(  
            nums.stream()  
                .filter(Auxiliar::esMayorQue4)  
                .filter(Auxiliar::esPar)  
                .map(Auxiliar::duplica)  
                .findFirst()  
                .orElse(0) );  
    }  
}
```

Solución **estilo declarativo** (funcional): **lambdas** en vez de métodos.

```
public class Main {  
    public static void main(String[] args) {  
        List<Integer> nums = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
        // Calcular el doble del primer número par mayor que 4  
  
        // estilo funcional (declarativo) - con lambdas  
        // aunque parezca complejo no lo es ...  
        // realiza las mismas llamadas a métodos de Auxiliar  
        System.out.println(  
            nums.stream()  
                .filter( (n) -> n > 4 )  
                .filter( (n) -> n%2 == 0 )  
                .map( (n) -> n * 2 )  
                .findFirst()  
                .orElse(0) );  
    }  
}
```

Solución estilo declarativo (funcional): directo a **qué** hacer, no **cómo**.

El proceso de recorrido de la lista original queda oculto (en el stream)

El código se centra en qué operaciones hacer y no cómo procesar datos

Declara las operaciones que debemos realizar

```
System.out.println(  
    nums.stream()           // de todos los números dados  
        .filter( (n) -> n > 4 )    // me interesan solo los > 4  
        .filter( (n) -> n%2 == 0 ) // y de esos sólo los pares  
        .map( (n) -> n * 2)        // y además me interesa el doble  
        .findFirst()              // del primero.  
        .orElse(0) );  
}
```

Solución estilo declarativo (funcional): directo a **qué hacer no **cómo**.**

El proceso de recorrido de la lista original queda oculto (en el stream)

El código se centra en qué operaciones hacer y no cómo procesar datos

Declara las operaciones que debemos realizar

No obstante, en algunos (pocos) casos puede no ser lo apropiado.

Cambiemos ligeramente el problema:

Imprimir todos los números pares mayores que 4 en la lista dada

Imprimir todos los números pares mayores que 4 en la lista dada

```
public class Main {  
    public static void main(String[] args) {  
        List<Integer> nums = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
        // Imprimir todos los números pares mayores que 4  
  
        // estilo funcional (declarativo)  
        // aunque parezca complejo no lo es ...  
        // realiza las mismas llamadas a métodos de Auxiliar  
  
        nums.stream()  
            .filter( (n) -> n > 4 )  
            .filter( (n) -> n%2 == 0 )  
            .map( (n) -> { System.out.println(n); return n; } );  
    }  
}
```


Imprimir todos los números pares mayores que 4 en la lista dada

```
public class Main {  
    public static void main(String[] args)  
    {  
        List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
  
        // Imprimir todos los números pares mayores que 4  
  
        // estilo funcional (declarativo)  
        // aunque parezca complejo no lo es ...  
        // realiza las mismas llamadas a métodos de Auxiliar  
  
        nums.stream()  
            .filter( (n) -> n > 4 )  
            .filter( (n) -> n%2 == 0 )  
            .map( (n) -> { System.out.println(n); return n; } );  
    }  
}
```

¿Qué tiene de malo?

Imprimir todos los números pares mayores que 4 en la lista dada

```
public class Main {  
    public static void main(String[] args)  
        List<Integer> nums = Arrays.asList(1  
  
    // Imprimir todos los números pares  
  
    // estilo funcional (declarativo)  
    // aunque parezca complejo no lo es ...  
    // realiza las mismas llamadas a métodos de Auxiliar  
  
        nums.stream()  
            .filter( (n) -> n > 4 )  
            .filter( (n) -> n%2 == 0 )  
            .map( (n) -> { System.out.println(n); return n; } );  
  
    }  
}
```

¿Qué tiene de malo?
Primero, que no funciona;
No imprime nada.

Imprimir todos los números pares mayores que 4 en la lista dada

```
public class Main {  
    public static void main(String[] args)  
        List<Integer> nums = Arrays.asList(1  
  
    // Imprimir todos los números pares  
  
    // estilo funcional (declarativo) MAL USADO: No Funciona  
    // aunque parezca complejo no lo es ...  
    // realiza las mismas llamadas a métodos de Auxiliar  
  
        nums.stream()  
            .filter( (n) -> n > 4 )  
            .filter( (n) -> n%2 == 0 )  
            .map( (n) -> { System.out.println(n); return n; } );  
  
    System.out.println("Fin");  
}  
}
```

¿Qué tiene de malo?
Primero, que no funciona;
No imprime nada.
Bueno, ahora imprime "Fin"
¿Por qué no imprime nada más?

Faltaba **la terminal**: la operación de reducción final que no sólo convierte los datos del stream final en **el resultado deseado**, sino que, **lo más importante**, hace que el stream inicial empiece a generar datos. Sin esa operación el stream es *lazy* (léase, eficiente) y no genera datos hasta que no se los pida *la operación terminal*. Ahora sí funciona el programa e imprime los números deseados, pero ...

```
// estilo (declarativo) MAL USADO: aunque funcione  
// aunque el código no lo es ...  
// realiza llamadas a métodos de Auxiliar
```

```
    nums.stream()  
        .filter( n -> n > 4 )  
        .filter( n -> n%2 == 0 )  
        .map( (n) -> { System.out.println(n); return n; } )  
        .collect( Collectors.toList() ); // terminal operation  
}  
}
```

... pero **el resultado deseado** no era la lista de números,
sino solamente imprimirlos;
es decir, nos valdría cualquier operación terminal
(p.ej.: ésta ignora todos los números del stream excepto el último
pero antes de ignorarlos ya los ha imprimido todos)

```
// estilo (declarativo) MAL USADO: aunque funcione  
// aunque el ejemplo no lo es ...  
// realiza llamadas a métodos de Auxiliar
```

```
nums.stream()  
    .filter( n -> n > 4 )  
    .filter( n -> n%2 == 0 )  
    .map( (n) -> { System.out.println(n); return n; } )  
    .reduce( (x,y) -> y ); // terminal operation  
  
}  
}
```

... y esta operación terminal también produce **el resultado deseado**,
y parece más simple (de escribir, que no de ejecutar):
simplemente cuenta cuántos números se imprimen ...
pero ese valor nos da igual.

```
// esto es declarativo) MAL USADO: aunque funcione  
// aunque el ejemplo no lo es ...  
// realmente son llamadas a métodos de Auxiliar
```

```
nums = ...  
    .filter( (n) -> n > 4 )  
    .filter( (n) -> n%2 == 0 )  
    .map( (n) -> { System.out.println(n); return n; }  
    .count() ); // terminal operation
```

```
}
```

```
}
```

Siempre que pongamos una operación terminal indiferente para el resultado deseado

con el mero objetivo de hacer trabajar al stream, deberíamos
reconsiderar nuestro diseño declarativo/funcional,
en ese caso quizá sería mejor el estilo imperativo con un bucle explícito.

```
// este (ejemplo declarativo) MAL USADO: aunque funcione  
// aunque el ejemplo no lo es ...  
// realmente son llamadas a métodos de Auxiliar
```

```
nums  
    .filter( (n) -> n > 4 )  
    .filter( (n) -> n%2 == 0 )  
    .map( (n) -> { System.out.println(n); return n; } )  
    .count() ); // terminal operation
```

```
}
```

```
}
```

Calcular el doble **del primer número** par mayor que 4 en una lista dada.

```
public class Main {
```

```
    public static
```

```
        List<Int>
```

```
    // Cal
```

```
    // estilo funcional (declarativo) BIEN USADO: ¡¡es el mejor!!
```

```
    // aunque parezca complejo no lo es ... es raro al principio
```

```
    // realiza las mismas llamadas a métodos de Auxiliar
```

```
    System.out.println(
```

```
        nums.stream()
```

```
            .filter(Auxiliar::esMayorQue4)
```

```
            .filter(Auxiliar::esPar)
```

```
            .map(Auxiliar::duplica)
```

```
            .findFirst()
```

```
            .orElse(0) );
```

```
    }
```

```
}
```

Volvamos al primer ejercicio

Calcular el doble **del primer número** par mayor que 4 en una lista dada.

```
public class Main {  
    public static void main(String[] args) {  
        List<Integer> nums = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
        // Calcular el doble del primer número par mayor que 4  
  
        // estilo funcional (declarativo) BIEN USADO: ¡¡es el mejor!!  
        // aunque parezca complejo no lo es  
        // realiza las mismas llamadas  
        System.out.println(  
            nums.stream()  
                .filter(Auxiliar::esMayorQue4)  
                .filter(Auxiliar::esPar)  
                .map(Auxiliar::duplica)  
                .findFirst()  
                .orElse(0) );  
    }  
}
```

En el primer ejercicio, sin duda la operación terminal `findFirst()` era parte del problema inicial; por eso el **estilo declarativo/funcional era el correcto.**

En Java 8 los streams no tienen un operación equivalente a
`findFirst()`

para el último elemento del stream.

Por eso, **en el segundo ejercicio**, usamos la lambda $(x,y) \rightarrow y$ como operación terminal indiferente para sólo el último número.

```
// estilo (declarativo) MAL USADO: aunque funcione  
// aunque el ejemplo no lo es ...  
// realiza llamadas a métodos de Auxiliar
```

```
    nums.stream()  
        .filter( (n) -> n > 4 )  
        .filter( (n) -> n%2 == 0 )  
        .map( (n) -> { System.out.println(n); return n; } )  
        .reduce( (x,y) -> y );           // terminal operation  
    }  
}
```

Quizá los diseñadores de Java 8 hayan querido avisarnos de que
findLast()

no tiene sentido con streams ni en el estilo declarativo.

La operación terminal indiferente o de tipo *findLast()* es la pista para no usar el estilo declarativo con streams en el segundo ejercicio.

```
// estilo (declarativo) MAL USADO: aunque funcione  
// aunque el ejemplo no lo es ...  
// realiza llamadas a métodos de Auxiliar
```

```
    nums.stream()  
        .filter( (n) -> n > 4 )  
        .filter( (n) -> n%2 == 0 )  
        .map( (n) -> { System.out.println(n); return n; } )  
        .reduce( (x,y) -> y );           // terminal operation  
    }  
}
```

En este segundo ejercicio, hemos visto *el caso* en que no conviene utilizar el estilo declarativo con streams

En todos los demás casos, es preferible ese estilo.

¿Por qué debemos usar el estilo declarativo con streams en todos los demás casos?

Lo veremos en el **tercer ejercicio**, muy similar a los anteriores pero con datos algo más complejos:

“Encontrar la acción (de bolsa) de precio más alto entre las que no llegan a los \$500”

Usaremos otros métodos auxiliares para descomponer el problema:

```
public class StockInfo {  
    private final String accion;  
    private final double precio;  
  
    public StockInfo(final String acc, final double p) {  
        accion = acc;  
        precio = p;  
    }  
  
    public String toString() {  
        return String.format("ticker: %s price: %g",  
                               accion, precio);  
    }  
    public Double getPrice() { return precio; }  
    public String getName() { return accion; }  
}
```

Usaremos otros métodos auxiliares para descomponer el problema:

```
import java.util.function.Predicate;

public class StockUtil {

    public static StockInfo getStockInfo(final String acc) {
        return new StockInfo(acc, YahooFinance.getPrice(acc));
    }

    public static Predicate<StockInfo> isPriceLessThan(final double p) {
        return stockInfo -> stockInfo.getPrice() < p;
    }

    public static StockInfo pickHigh(final StockInfo stockInfo1,
                                     final StockInfo stockInfo2) {
        return stockInfo1.getPrice() > stockInfo2.getPrice() ?
            stockInfo1 : stockInfo2;
    }
}
```

Usaremos otros métodos auxiliares para descomponer el problema:

```
public class Tickers {
    public static final List<String> symbols = Arrays.asList(
        "MMM",      "ABT",      "ABBV",      "ACN",      "ACE",
        "ACT",      "ADBE",      "ADT",      "AES",      "AET",
        /* y otras 490 más */ );
}

public class YahooFinance {
    public static double getPrice(final String acc) {
        URL url =
            new URL("http://ichart.finance.yahoo.com/table.csv?s="+acc);
        BufferedReader reader = new BufferedReader(new
            InputStreamReader(url.openStream()));
        String d = reader.lines().skip(1).limit(1).findFirst().get();
        String[] items = d.split(",");
        double precio = Double.parseDouble(items[items.length-1]);
        return precio;
    }
}
```

Todo lo anterior eran sólo métodos auxiliares para el **tercer ejercicio**:

“Encontrar la acción (de bolsa) de precio más alto entre las que no llegan a los \$500”

Veremos tres (o cuatro) estilos de solución:

- Imperativa (descompuesta en pasos sucesivos)
- Imperativa compacta (todo en un mismo bucle)
- Declarativa con stream (sin bucle)
- Y otra más ...

Versión imperativa: descompuesta paso a paso

```
public class ImperativeStyle {  
    public static void findStock(List<String> acciones) {  
        // Obtener y almacenar informacion de las acciones  
        List<StockInfo> stocks = new ArrayList<>();  
        for (String acc: acciones) stocks.add(StockUtil.getStockInfo(acc));  
        // Quedarse sólo con las acciones de precio menor que $500  
        List<StockInfo> stocksMenorQue500 = new ArrayList<>();  
        for (StockInfo stockInfo : stocks)  
            if (StockUtil.isPriceLessThan(500).test(stockInfo))  
                stocksMenorQue500.add(stockInfo);  
        // Encontrar la acción de mayor precio entre las menores que $500  
        StockInfo precioMax = new StockInfo("", 0.0);  
        for (StockInfo stockInfo : stocksLessThan500) {  
            precioMax = StockUtil.pickHigh(precioMax, stockInfo);  
        }  
        System.out.println(precioMax); // Informar de la acción y su precio  
    }  
}
```

En main ejecutamos: `ImperativeStyle.findStock(Tickers.symbols);`

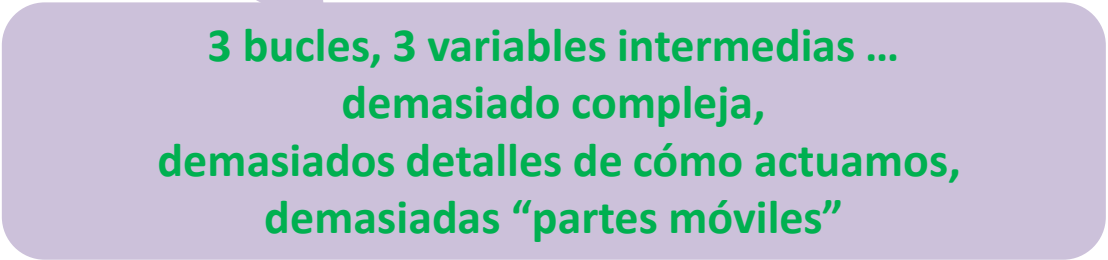
Versión imperativa: descompuesta paso a paso

```
public class ImperativeStyle {  
    public static void findStock(List<String> acciones) {  
  
        List<StockInfo> stocks = new ArrayList<>();  
        for (String acc: acciones) stocks.add(StockUtil.getStockInfo(acc));  
  
        List<StockInfo> stocksMenorQue500 = new ArrayList<>();  
        for (StockInfo stockInfo : stocks)  
            if (StockUtil.isPriceLessThan(500).test(stockInfo))  
                stocksMenorQue500.add(stockInfo);  
  
        StockInfo precioMax = new StockInfo("", 0.0);  
        for (StockInfo stockInfo : stocksLessThan500) {  
            precioMax = StockUtil.pickHigh(precioMax, stockInfo);  
        }  
        System.out.println(precioMax);  
    }  
}
```

En main ejecutamos: `ImperativeStyle.findStock(Tickers.symbols);`

Versión imperativa: descompuesta paso a paso

```
public class ImperativeStyle {  
    public static void findStock(List<String> acciones) {  
  
        List<StockInfo> stocks = new ArrayList<>();  
        for (String acc: acciones) stocks.add(StockUtil.getStockInfo(acc));  
  
        List<StockInfo> stocksMenorQue500 = new ArrayList<>();  
        for (StockInfo stockInfo : stocks)  
            if (StockUtil.isPriceLessThan(500).test(stockInfo))  
                stocksMenorQue500.add(stockInfo);  
  
        StockInfo precioMax = new StockInfo("", 0.0);  
        for (StockInfo stockInfo : stocksLessThan500) {  
            precioMax = StockUtil.pushHigh(precioMax, stockInfo);  
        }  
        System.out.println(precioMax);  
    }  
}
```



3 bucles, 3 variables intermedias ...
demasiado compleja,
demasiados detalles de cómo actuamos,
demasiadas “partes móviles”

En main ejecutamos: `ImperativeStyle.findStock(Tickers.symbols);`

Versión imperativa compacta: todos los pasos en un solo bucle

```
public class ImperativeCompactStyle {  
    public static void findStock(List<String> acciones) {  
        String accionMax = "";  
        Double precioMax = 0.0;  
        for (String acc: acciones) { // para cada accion  
            // obtener su precio  
            Double precio = StockUtil.getStockInfo(acc).getPrice();  
            if (precio < 500) { // si el precio es menor que $500  
                if (precio > precioMax) { // y si es mayor q mayor hasta ahora  
                    precioMax = price; // actualizar precio máximo  
                    accionMax = acc; // y recordar la acción de ese precio  
                }  
            }  
        }  
        // Informar de la acción seleccionada y su precio  
        System.out.println(String.format("ticker: %s price: %g",  
                                         candidateTicker, candidatePrice));  
    }  
}
```

En main: `ImperativeCompactStyle.findStock(Tickers.symbols);`


Versión imperativa compacta: todos los pasos en un solo bucle

```
public class ImperativeCompactStyle {  
    public static void findStock(List<String> acciones) {  
        String accionMax = "";  
        Double precioMax = 0.0;  
        for (String acc: acciones) {  
  
            Double precio = StockUtil.getStockInfo(acc).getPrice();  
            if (precio < 500) {  
                if (precio > precioMax) {  
                    precioMax = price;  
                    accionMax = acc;  
                }  
            }  
        }  
  
        System.out.println(String.format("ticker: %s price: %g",  
                                         candidateTicker, candidatePrice));  
    }  
}
```

En main: `ImperativeCompactStyle.findStock(Tickers.symbols);`

Versión imperativa compacta: todos los pasos en un solo bucle

```
public class ImperativeCompactStyle {  
    public static void findStock(List<String> acciones) {  
        String accionMax = "";  
        Double precioMax = 0.0;  
        for (String acc: acciones) { // para cada accion  
            // obtener su precio  
            Double precio = StockUtil.getStockInfo(acc).getPrice();  
            if (precio < 500) { // si el precio es menor que $500  
                if (precio > precioMax) { // y si es mayor q mayor hasta ahora  
                    precioMax = price; // actualizar precio máximo  
                    accionMax = acc; // y recordar la acción de ese precio  
                }  
            }  
        }  
        // Informar de la acción seleccionada y su precio  
        System.out.pr  
    }  
}
```



A pesar de minimizar variables intermedias y tener sólo 1 bucle, seguimos centrados en los detalles de cómo actuamos: inicializaciones, cambios en variables, ...

En main: `ImperativeCompactStyle.findStock(Tickers.symbols);`

Ambos estilos de solución son muy similares:

- Imperativa (descompuesta en pasos sucesivos)
- Imperativa compacta (todo en un mismo bucle)

La imperativa compacta puede parecer más clara aquí, pero se debe a la simplicidad del problema

Un problema complejo se debe descomponer, a pesar de los defectos que tiene el estilo imperativo.

Entonces, veamos otra forma de descomposición en el siguiente estilo de solución:

- Declarativa con stream (sin bucle)

Versión declarativa con stream: se centra en *qué* hacemos, no *cómo*

```
import java.util.Comparator;
import java.util.stream.Stream;

public class DeclarativeStyle {
    public static void findStock(Stream<String> acciones) {
        System.out.println(
            acciones
                .map(StockUtil::getStockInfo)
                .filter(StockUtil.isPriceLessThan(500))
                .max(Comparator.comparingDouble(StockInfo::getPrice)).get()
        );
    }
}
```

En el main ejecutamos:

```
DeclarativeStyle.findStock(Tickers.symbols.stream());
```


Versión declarativa con stream: **sin bucle explícito fuera del stream**

```
import java.util.Comparator;
import java.util.stream.Stream;

public class DeclarativeStyle {
    public static void findStock(Stream<String> acciones) {
        System.out.println(
            acciones
                .map(StockUtil::getStockInfo)
                .filter(StockUtil.isPriceLessThan(500))
                .max(Comparator.comparingDouble(StockInfo::getPrice)).get()
        );
    }
}
```

En el main ejecutamos:

```
DeclarativeStyle.findStock(Tickers.symbols.stream());
```

Es otra forma de descomposición con el estilo de solución

- Declarativa con stream (sin bucle)

No hay bucles explícitos (ni siquiera “fors avanzados”, sin índices)

No hay “partes móviles” (variables intermedias que cambian)

No hay detalles de cómo se ejecutan los pasos

Se lee (casi) como la descripción del problema (declarativo):

“Encontrar la acción (de bolsa) de precio más alto entre las que no llegan a los \$500”

```
acciones.map( StockUtil::getStockInfo )  
    .filter( StockUtil. isPriceLessThan(500) )  
    .max( Comparator.comparingDouble(StockInfo::getPrice) )  
    .get()
```

La solución no es compleja; no confundir “rara” o nueva con compleja.

Alguna pega tendrá ... ¿Quizá sea más lenta de ejecutarse? Al contrario.

¿Hay diferencias significativas en velocidad de ejecución?

Parece que no ... a la vista de estos tiempos de ejecución en segundos:

	Procesadores		
Estilos	Core 2 Duo E8500	Core2 Quad Q9300	i5-2430M
Imperativo	599,5	604,7	601,7
Imperativo Compacto	602,4	613,1	598,8
Declarativo/Stream	598,3	599,2	594,5

¿Hay diferencias significativas en velocidad de ejecución?

Parece que no ...

o quizá sí: **esos nuevos tiempos de ejecución son 50% y 25% menores**

	Procesadores		
Estilos	Core 2 Duo E8500	Core2 Quad Q9300	i5-2430M
Imperativo	599,5	604,7	601,7
Imperativo Compacto	602,4	613,1	598,8
Declarativo/Stream	598,3	599,2	594,5
¿?	303,0	152,9	151,7

¿Hay diferencias significativas en velocidad de ejecución?

El estilo más rápido es el declarativo con `parallelStream()` ...

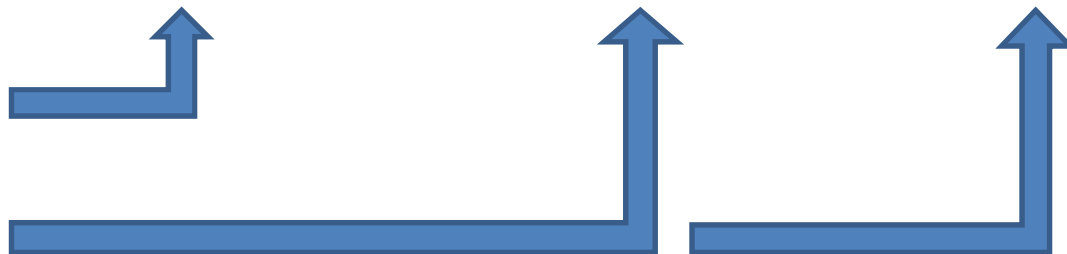
Pero **¿cómo se programa eso?**

	Procesadores		
Estilos	Core 2 Duo E8500	Core2 Quad Q9300	i5-2430M
Imperativo	599,5	604,7	601,7
Imperativo Compacto	602,4	613,1	598,8
Declarativo/Stream	598,3	599,2	594,5
Declarativo/Parallel Stream	303,0	152,9	151,7

Procesador con 2 núcleos:



Procesador con 4 núcleos:



¿Cómo se paraleliza la versión declarativa/stream?

```
import java.util.Comparator;
import java.util.stream.Stream;

public class DeclarativeStyle {
    public static void findStock(Stream<String> acciones) {
        System.out.println(
            acciones
                .map(StockUtil::getStockInfo)
                .filter(StockUtil.isPriceLessThan(500))
                .max(Comparator.comparingDouble(StockInfo::getPrice)).get()
        );
    }
}
```

En el main ejecutamos:

```
DeclarativeStyle.findStock(Tickers.symbols.stream());
```

¿Cómo se paraleliza la versión declarativa/stream?

Sin cambiar nuestro código en absoluto ...

Procesábamos un stream y lo seguimos haciendo igual, pero ...

```
import java.util.Comparator;  
import java.util.stream.Stream;
```

```
public class DeclarativeStyle {  
    public static void findStock(Stream<String> acciones) {  
        System.out.println(  
            acciones  
                .map(StockUtil::getStockInfo)  
                .filter(StockUtil.isPriceLessThan(500))  
                .max(Comparator.comparingDouble(StockInfo::getPrice)).get()  
        );  
    }  
}
```

En el main:

```
DeclarativeStyle.findStock(Tickers.symbols.stream());
```

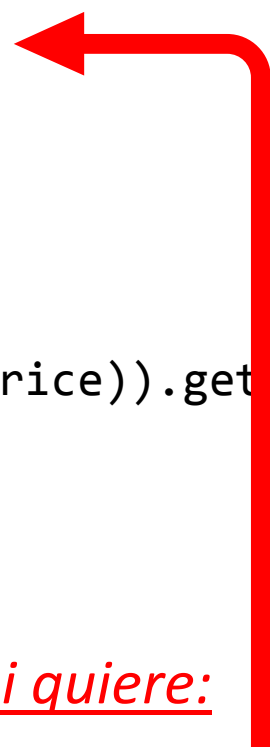
¿Cómo se paraleliza la versión declarativa/stream?

Sin cambiar nuestro código en absoluto ...

Procesábamos un stream y lo seguimos haciendo igual, pero ...

```
import java.util.Comparator;
import java.util.stream.Stream;
```

```
public class DeclarativeStyle {
    public static void findStock(Stream<String> acciones) {
        System.out.println(
            acciones
                .map(StockUtil::getStockInfo)
                .filter(StockUtil.isPriceLessThan(500))
                .max(Comparator.comparingDouble(StockInfo::getPrice)).get()
        );
    }
}
```



En el main, el cliente nos puede pasar un parallelStream, si quiere:

```
DeclarativeStyle.findStock(Tickers.symbols.parallelStream() );
```


Antes de terminar, ¿podemos conciliar ambos estilos?

Volvamos a ver el ejercicio 2:

```
public static void main(String[] args) {  
    List<Integer> nums = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
  
    // Imprimir todos los números pares mayores que 4  
  
    // estilo funcional (declarativo)  
    // aunque parezca complejo no lo es ...  
    // realiza las mismas llamadas a métodos de Auxiliar  
  
    nums.stream()  
        .filter( (n) -> n > 4 )  
        .filter( (n) -> n%2 == 0 )  
        .map( (n) -> {System.out.println(n); return n;} );  
  
}
```

Antes de terminar, ¿podemos conciliar ambos estilos?

Volvamos a ver el ejercicio

```
public static void main(String[] args) {
    List<Integer> nums = ...

    // Imprimir todos los números ...

    // estilo funcional (declarativo) ...
    // aunque parezca complejo ...
    // realiza las mismas tareas que los métodos de Auxiliar

    nums.stream()
        .filter( (n) -> n > 4 )
        .filter( (n) -> n%2 == 0 )
        .map( (n) -> {System.out.println(n); return n;} );
}
```

¿Estaba mal diseñada toda la solución al Ejercicio 2,
o sólo parte de ella?

Antes de terminar, ¿podemos conciliar ambos estilos?

Volvamos a ver el ejercicio

```
public static void main(String[] args) {
    List<Integer> nums = new ArrayList<>();

    // Imprimir todos los números de 1 a 100
    // estilo imperativo (declarativo)
    // aunque parezca complejo, es más fácil de entender ...
    // realiza las mismas 100 iteraciones a métodos de Auxiliar

    nums.stream()
        .filter( (n) -> n > 4 )
        .filter( (n) -> n%2 == 0 )
        .map( (n) -> {System.out.println(n); return n;} );
}
```

¿Estaba mal diseñada toda la solución al Ejercicio 2,
o sólo parte de ella?

**Ese map produce un stream que
no era necesario para nuestro objetivo:**
imprimir todo los números.

Antes de terminar, ¿podemos conciliar ambos estilos?

Volvamos a ver el ejercicio

```
public static void main( String[] args ) {  
    List<Integer> nums = new ArrayList<>( 10 );  
    for( int i = 0; i < 10; i++ ) {  
        nums.add( new Integer( i ) );  
    }  
  
    // Imprimir todos los números de la lista  
    // usando el estilo tradicional  
    for( Integer n : nums ) {  
        System.out.println( n );  
    }  
  
    // estilo funcional (declarativo)  
    // aunque parezca complejo, en realidad es más sencillo ...  
    // realiza las mismas 11 llamadas a los métodos de Auxiliar  
    // que el código anterior  
    nums.stream()  
        .filter( (n) -> n > 4 )  
        .filter( (n) -> n%2 == 0 )  
        // .map( (n) -> {System.out.println(n); return n;} );  
        .forEach( (n) -> System.out.println(n) );  
}
```

Con **forEach()** se puede volver a aplicar un **bucle externo al stream**, que ejecuta su parámetro para cada elemento del stream pero sin retornar ningún resultado.

Antes de terminar, ¿podemos conciliar ambos estilos?

Volvamos a ver el ejercicio

```
public static void main(
```

```
List<Integer> nums =
```

```
// Imprimir todos los
```

```
// estilo funcional (declarativo)
```

```
// aunque parezca complejo ...
```

```
// realiza las mismas llamadas a los métodos de Auxiliar
```

```
    nums.stream()
```

```
        .filter( (n) -> n > 4 )
```

```
        .filter( (n) -> n%2 == 0 )
```

```
        // .map( (n) -> {System.out.println(n); return n;} );
```

```
        .forEach( (n) -> System.out.println(n) );
```

```
}
```

forEach() es la **operación terminal** que antes “inventábamos” de forma indiferente al resultado con el único fin de hacer que el stream generase datos, pero ahora esta reducción substituye al map() incorrecto, y ya **no es una operac. terminal indiferente al resultado**

Un último detalle sobre **forEach()** para conciliación de estilos

En el **ejercicio 2**, generamos nosotros mismos el stream

Lo hacemos a partir de una colección `nums.stream()`

Por tanto, sabemos que será un stream secuencial (no paralelo)

Y por eso es seguro utilizar **forEach()** con ese stream ...

```
// estilo funcional (declarativo)
// aunque parezca complejo no lo es ...
// realiza las mismas llamadas a métodos de Auxiliar
```

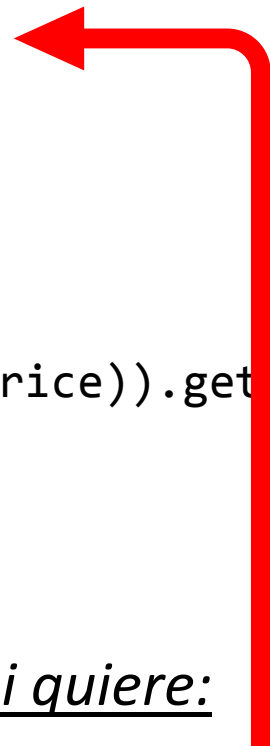
```
nums.stream()
    .filter( (n) -> n > 4 )
    .filter( (n) -> n%2 == 0 )
    .forEach( (n) -> System.out.println(n) );
```

```
}
```

Un último detalle sobre **forEach()** para conciliación de estilos

En el **ejercicio 3**, vimos la ventaja de recibir un stream como parámetro
Nuestro código funcionaba igual si nos pasaban un `parallelStream`

```
public class DeclarativeStyle {  
    public static void findStock(Stream<String> acciones) {  
  
        .map(StockUtil::getStockInfo)  
        .filter(StockUtil.isPriceLessThan(500))  
        .max(Comparator.comparingDouble(StockInfo::getPrice)).get()  
    };  
}  
}
```



En el main, el cliente nos puede pasar un `parallelStream`, si quiere:

```
DeclarativeStyle.findStock(Tickers.symbols.parallelStream());
```

Un último detalle sobre **forEach()** para conciliación de estilos

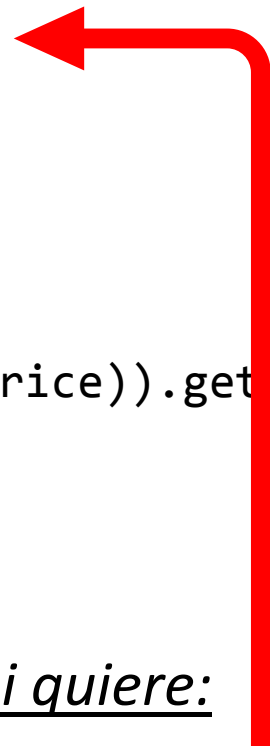
En el **ejercicio 3**, vimos la ventaja de recibir un stream como parámetro

Nuestro código funcionaba igual si nos pasaban un `parallelStream`

Pero, esto no habría sido cierto si hubiésemos usado **forEach()**

forEach() es no determinista, no garantiza el orden de ejecución

```
public class DeclarativeStyle {  
    public static void findStock(Stream<String> acciones) {  
  
        .map(StockUtil::getStockInfo)  
        .filter(StockUtil.isPriceLessThan(500))  
        .max(Comparator.comparingDouble(StockInfo::getPrice)).get()  
    };  
}  
}
```



En el main, el cliente nos puede pasar un `parallelStream`, si quiere:

```
DeclarativeStyle.findStock(Tickers.symbols.parallelStream());
```


Un último detalle sobre `forEach()` para conciliación de estilos

En el **ejercicio 3**, vimos la ventaja de recibir un stream como parámetro

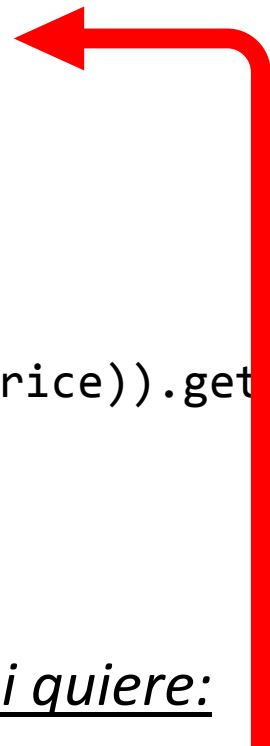
Nuestro código funcionaba igual si nos pasaban un `parallelStream`

Pero, esto no habría sido cierto si hubiésemos usado `forEach()`

`forEach()` es no determinista, no garantiza el orden de ejecución

`forEachOrdered()` respeta el “*encounter order*” del stream (ver API)

```
public class DeclarativeStyle {  
    public static void findStock(Stream<String> acciones) {  
  
        .map(StockUtil::getStockInfo)  
        .filter(StockUtil.isPriceLessThan(500))  
        .max(Comparator.comparingDouble(StockInfo::getPrice)).get()  
    };  
}  
}
```



En el main, el cliente nos puede pasar un `parallelStream`, si quiere:

```
DeclarativeStyle.findStock(Tickers.symbols.parallelStream());
```

Conclusión

Dos razones de enorme peso para preferir el estilo declarativo con streams:

- 1. Refleja más directamente *qué* hacer (en vez de *cómo* hacerlo)**
- 2. La paralelización del código es trivial (decisión del SW cliente)**

Pero, tampoco debemos usar ese estilo siempre y sin pensar: vimos un caso en que no era lo adecuado y vimos cómo corregirlo.

Agradecimientos: Ejercicios y código de *Venkat Subramaniam*, autor del libro “Functional Programming in Java: Harnessing the Power Of *Java 8*” (ver bibliografía)