PROGRAMACIÓN FUNCIONAL Y STREAMS

Los dos conceptos claves que se incorporaron a Java para poder utilizar técnicas de programación funcional fueron:

- las expresiones lambda
- el concepto de Stream

INTRODUCCIÓN AL CONCEPTO DE STREAM

¿Qué es un Stream?. Se entiende mejor al verlo funcionar que al definirlo. La típica definición es muy genérica, pero hay que empezar por algo:

Un Stream es una secuencia de elementos que provienen de un origen y esta secuencia soporta operaciones de agregados sobre sus elementos.

Una colección también es "una secuencia de elementos", así que puede parecer que no es más que un tipo de colección pero el concepto de Stream es realmente diferente ya que permite una nueva forma de trabajar ya que incorpora:

- un estilo de programación declarativa usando programación funcional
- novedades en términos de eficiencia permitiendo aprovechar automáticamente las características multiproceso de los modernos procesadores. Los Streams pueden ser secuenciales o paralelos. Las acciones en un stream secuencial ocurren de forma secuencial en un único thread, mientras que en un stream paralelo pueden ocurrir todas a la vez en múltiples threads.

API Stream

Hay todo un enjambre de interfaces y clases para soportar el concepto de Stream, A todos ellos les llamamos API Stream que provee utilidades para permitir programar "al estilo de programación funcional" sobre flujos de valores. El API Stream está contenido principalmente en el paquete java.util.stream. ¡consulta API!

Y dentro del API Stream el punto central es el Interface Stream.

COMO CREAR UN Stream

Hay muchas formas, vamos a ver algunas formas y más adelante surgirán otras nuevas. Ahora hay que fijarse que todo Stream tiene un origen "source", es decir una fuente de datos. La fuente puede ser de naturaleza diversa, desde un array, a datos provenientes de disco o de la tarjeta de red.

crear un Stream con Stream.of()

Ya que un Stream es un interface, no podemos hacer directamente un new, hay muchas formas de crear un Stream, un ejemplo sencillo es *con el método static of() del interface Stream*

Recuerda que desde java 8 los interfaces pueden tener ciertos métodos con código, concretamente son los métodos default y los static. Stream.of() es un método static de Stream y se usa para crear objetos Stream. Se le puede pasar por parámetro un conjunto de valores separados por comas o un array

En estos ejemplos, para demostrar el funcionamiento del Stream utilizamos el método foreach() que veremos formalmente más adelante como "una operación de terminación"

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

class App {

   public static void main(String[] args) {
```

```
        //of(T... values) => lista de valores separadas por comas o array
        Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
        stream1.forEach(p -> System.out.print(p));
        System.out.println("");

        Stream<Integer> stream2 = Stream.of(new Integer[]{1, 2, 3, 4, 5, 6, 7, 8, 9});
        stream2.forEach(p -> System.out.print(p));
        System.out.println("");

        Stream<String> stream3 = Stream.of(new String[]{"hola", "adios", "chao"});
        stream3.forEach(p -> System.out.print(p));
        System.out.println("");

        Stream<String> stream4 = Stream.of("hola;adios;chao".split(";"));
        stream4.forEach(p -> System.out.print(p));
        System.out.println("");

    }
}
```

crear un Stream con el método stream()

Una forma interesante para crear un Stream es con el método stream() que desde java 8 tienen los objetos Collection

Si consultamos *Collection* en el API DE JAVA i8!. Vemos ahí el método Stream

`default Stream<E>` **`stream()`**

Returns a sequential `Stream` with this collection as its source.

Observa que indica que devuelve un stream secuencial. También hay un método para parallelStream() que puede devolver un Stream paralelo que usa internamente varios Threads, pero de esto no vemos nada en este boletín.

O sea, que el método *stream()*, aplicado sobre una objeto que implementa Collection devuelve un objeto *Stream* que me da "otra visión" de los datos y de cómo trabajar con ellos, esta visión complementa y a veces incluso sustituye a la tradicional ya que va ligada a la potencia de la programación funcional cuando se aplica al procesamiento de datos.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class App{
    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1,2,3,4,5,6);
        //Crear Stream con método stream()
        Stream<Integer> stream = list.stream();
        stream.forEach(p -> System.out.println(p));
    }
}
```

Crear un Stream con Stream.generate()

Crea un Stream infinito, el siguiente ejemplo crea infinitas fechas. Tendrás que parar la ejecución del programa para salir.

```
import java.util.Calendar;
import java.util.stream.Stream;
public class App {
    public static void main(String... args) {
        Stream<Calendar> stream = Stream.generate(() -> Calendar.getInstance());
        stream.forEach(System.out::println);
```

```
    }
}
```

Para hacer finito el Stream a la salida de generate() se le aplica el método limit(). Por ejemplo limitamos a 5 fechas.

```
import java.util.Calendar;
import java.util.stream.Stream;
public class App {
    public static void main(String... args) {
        Stream<Calendar> stream = Stream.generate(() -> Calendar.getInstance()).limit(5);
        stream.forEach(System.out::println);

    }
}
```

otro ejemplo para generar 5 números aleatorios

```
 import java.util.stream.Stream;
public class App {
    public static void main(String... args) {
        Stream<String> stream = Stream.generate(()
                -> Double.toString(Math.random() * 1000)).limit(5);

        stream.forEach(System.out::println);

    }
}
```

otras formas de crear un Stream

Hay muchas, más o menos útiles según el contexto. Irán saliendo.

UN EJEMPLO DE PROGRAMACIÓN FUNCIONAL CON STREAM JAVA.

programación funcional en java=> usar api Stream.

No hay mejor forma de entender que es la programación funcional que con un ejemplo que abarca todos los conceptos clave aunque sea informalmente.

La explicación que sigue es una simplificación y resumen de
http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Un objeto que implementa el interface Stream, permite mezclar un agregado de datos con una serie de operaciones que se ejecuten sobre ellos. Estas operaciones se escriben con una sintaxis nueva en el lenguaje. Las operaciones también tienen características novedosas de computación(posibilidad de paralelismo). Se utiliza el término "agregado" para no usar el nombre "colección" ya que conceptualmente un stream se diferencia de una colección. Dicho esto, a menudo una colección se utiliza como "fuente de datos" para la generación del stream por lo que en los ejemplos de stream a menudo intervienen colecciones.

Nota del profesor: lo de arriba es prácticamente una traducción del artículo. Lo de sintaxis nueva se refiere a las lambda, por lo demás todos son interfaces, métodos, etc. con la sintaxis tradicional

La idea genérica del trabajo con Streams es la siguiente: **definir "flujos de trabajo"(un conjunto de operaciones) sobre agregados (agrupaciones) de datos.**

Ejemplo de procesamiento de los datos de una colección sin usar Stream

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Transaction{
    public static final int GROCERY=20;
    private int Id;
    private int value;
    private int type;

    public Transaction(int Id, int value, int type) {
        this.Id = Id;
        this.value = value;
        this.type = type;
    }

    public int getId() {
        return Id;
    }

    public void setId(int Id) {
        this.Id = Id;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

}
class App{
    public static void main(String[] args){
            Transaction[] arrayTransacciones={new Transaction(1,100,33),new Transaction(3,80,20),new Transaction(6,120,20),new Transaction(7,40,35),new
Transaction(10,50,20)};
        List<Transaction> transactions=Arrays.asList(arrayTransacciones);

        List<Transaction> groceryTransactions = new ArrayList<>();
        for(Transaction t: transactions){
            if(t.getType() == Transaction.GROCERY){
                groceryTransactions.add(t);
            }
        }
        Collections.sort(groceryTransactions, new Comparator<Transaction>(){
            public int compare(Transaction t1, Transaction t2){
                return t1.getValue()>=t2.getValue() ? -1:1;
                //return t2.getValue()-t1.getValue();
            }
        });
        List<Integer> transactionsIds = new ArrayList<>();
        for(Transaction t: groceryTransactions){
            transactionsIds.add(t.getId());
        }
        //comprobamos
        for(Integer id:transactionsIds)
            System.out.print(id+" ");
    }
}
```

El ejemplo anterior hace esencialmente:

1. crea una lista con objetos Transaction
2. Los elementos de esa lista los filtra quedándose sólo con los que son de tipo GROCERY (valor 20 en Type)
3. El resultado del filtro lo guarda en una lista llamada groceryTrasactions y ordena esa lista por Value
4. de la anterior lista ordenada generamos una lista que simplemente contiene los ids
5. listamos los productos de dichos ids

Se pudo hacer un poco más compacta la solución pero está así a propósito para fijarse en la forma de programar imperativa: creamos un algoritmo que consta de 5 pasos y vamos dando estos pasos de forma ordenada

Fíjate cómo haríamos esto en SQL si existiera una tabla Transactions

```
SELECT id
```

```
FROM Transactions
WHERE type=20;
ORDER BY value;
```

SQL nos ofrece una forma de programar "declarativa" ya que decimos/declaramos lo que queremos, no necesitamos indicar los pasos ordenados para indicar cómo obtenerlo

Cada estilo de programación será mejor o peor que el otro según el contexto, pero por lo que vemos en el ejemplo anterior cuando trabajamos procesando datos (filtrando, ordenando, …), la programación declarativa parece muy conveniente.

El ejemplo anterior con Stream

El api Stream nos va a permitir trabajar de forma declarativa al procesar datos. Observa el siguiente ejemplo cuya sintaxis no tienes por el momento que entender al 100%, pero sí intuir la idea de funcionamiento

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collector;
import java.util.stream.Collectors;

class Transaction{
    public static final int GROCERY=20;
    private int Id;
    private int value;
    private int type;

    public Transaction(int Id, int value, int type) {
        this.Id = Id;
        this.value = value;
        this.type = type;
    }

    public int getId() {
        return Id;
    }

    public void setId(int Id) {
        this.Id = Id;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

}
class App{
    public static void main(String[] args){
            Transaction[] arrayTransacciones={new Transaction(1,100,33),new Transaction(3,80,20),new Transaction(6,120,20),new Transaction(7,40,35),new
Transaction(10,50,20)};
        List<Transaction> transactions=Arrays.asList(arrayTransacciones);

        List<Integer> transactionsIds =
                transactions.stream()
                        .filter(t -> t.getType() == Transaction.GROCERY)
                        .sorted(Comparator.comparing(Transaction::getValue).reversed())
                        .map(Transaction::getId)
                        .collect(Collectors.toList());
        //comprobamos
        for(Integer id:transactionsIds)
            System.out.print(id+" ");
    }
}
```

Por lo tanto esencialmente hacemos dos cosas:

- generar un Stream de la lista con el método stream()
- el objeto Stream copia los datos de la lista y se lanzan a una especie de cadena de manipulación en la que se van aplicando operaciones similarmente a como en una cadena de montaje de coches va manipulando la materia prima inicial. Observa que los datos originales de la lista no se tocan

- la primera operación la acomete el método filter() que devuelve un Stream filtrado.
- El Stream filtrado se pasa a sorted() que utiliza un comparador para ordenar y devuelve un Stream ordenado.
- el map devuelve también un Stream pero sólo con el campo que indicamos
- por último el método collect genera una lista para almacenar el resultado

Si queremos ver todo con expresiones lambda, sin referencias a métodos

```
List<Integer> transactionsIds =
        transactions.stream()
            .filter(t -> t.getType() == Transaction.GROCERY)
            .sorted( (t1,t2)->t2.getValue()-t1.getValue())
            .map( t->t.getId())
            .collect(Collectors.toList());
```

Cada operación la escribimos en línea a parte por legibilidad pero podríamos poner todo en una línea o varias líneas, por ejemplo

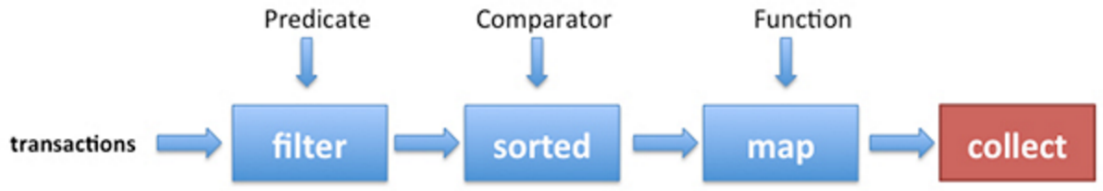
```
transactions.stream().filter(t -> t.getType() == Transaction.GROCERY).sorted( (t1,t2)->t2.getValue()-t1.getValue()) .map( t->t.getId()).collect(Collectors.toList());
```

Observa que en definitiva todo lo anterior es una instrucción java de asignación en la que su parte derecha es una cadena de operaciones donde la salida de una es la entrada de otra.

Nota profesor : decimos "operaciones" en lugar de "funciones" porque filter(), sorted() en realidad son métodos…en java conseguimos una aproximación a la programación funcional de forma que encadenando los métodos de Stream simulamos el efecto de que la salida de una función es la entrada de otra en la programación funcional pura. Además a estos métodos podemos pasarle expresiones lambda simulando otra característica de la programación funcional en la que a una función se le puede pasar de parámetro a otra función.

Un Stream como flujo de trabajo

Decíamos que la idea genérica de un Stream es la siguiente: Los streams permiten definir "flujos de trabajo" formados por un conjunto de operaciones sobre agrupaciones de datos.
Respecto a nuestro ejemplo, esto se resume con el siguiente gráfico
datos->objetos Transaction
operaciones -> filter, sorted,map,collect



En el gráfico se aprecia además que los parámetros de las operaciones(los métodos) son instancias de Interfaces funcionales, por lo que podemos escribirlos como expresiones lambda.

Estilo de programación declarativa.

Observa que estamos haciendo un estilo de programación declarativa (la programación funcional es un caso concreto de programación declarativa), de forma que decimos lo que queremos, algo parecido a SQL. Podemos establecer esta relación no muy rigurosa:

select -> map
from ->stream()
where ->filter()
order by->sorted()
Además observa que en SQL hay una iteración automática sobre cada miembro resultante del from similar a la iteración automática que hace el Stream
fila de select=> cada elemento del stream

poder "simular SQL" desde un lenguaje de propósito general como Java o Python es un gran logro. Piensa que SQL es sencillo porque es muy específico, sólo vale para trabajar con datos y en un formato muy concreto(tablas)

MÁS CONCEPTOS SOBRE STREAMS vs COLECCIONES

(es una traducción/resumen de Getting Started with Streams del link)

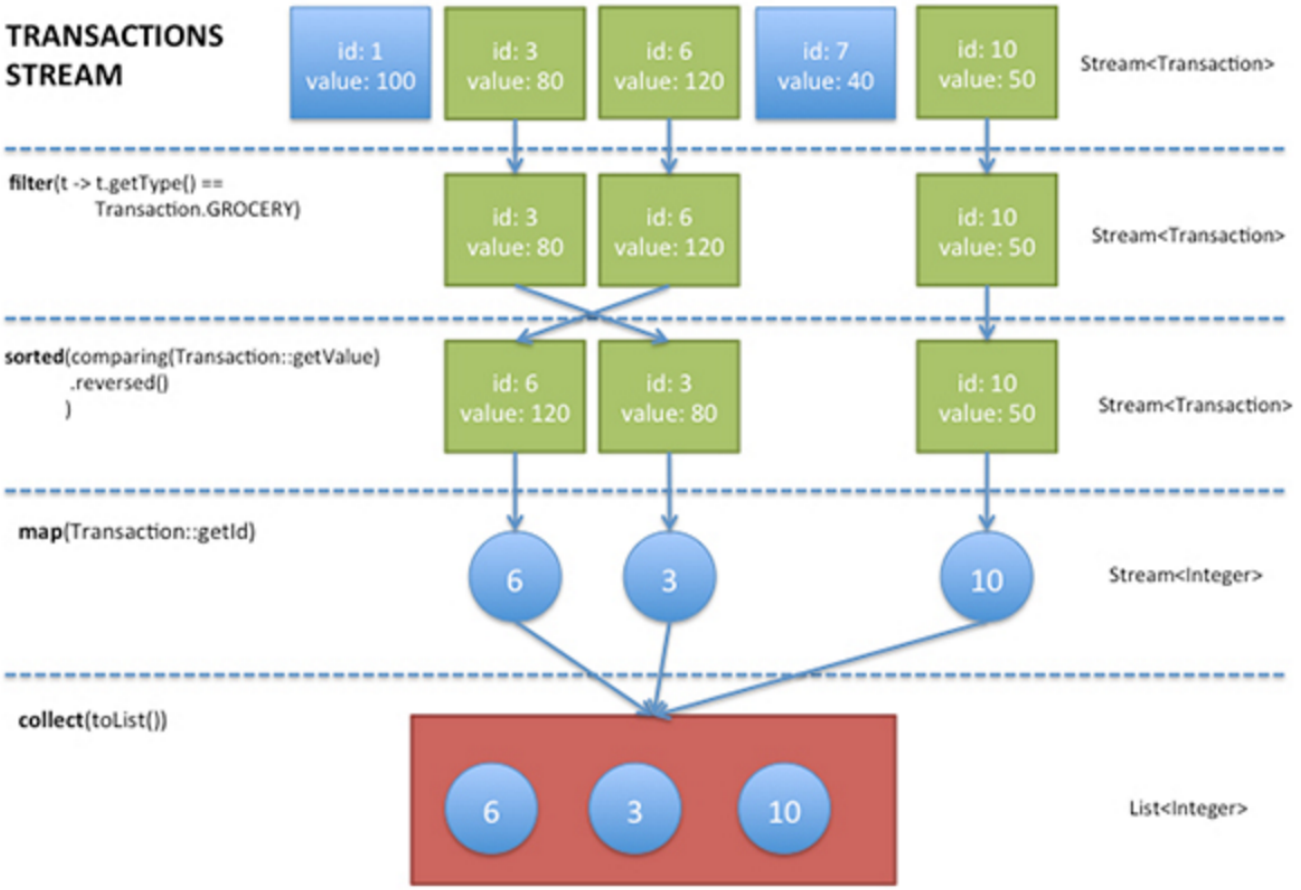
Ahora explicamos un poco más la definición de Stream: una **secuencia de elementos** de un **origen** que soporta un **agregado de operaciones.**

- Secuencia de elementos: se trabaja con una secuencia de elementos todos del mismo un tipo. El stream va suministrando esta secuencia y se va procesando al vuelo, es decir, la secuencia no se almacena como una colección. Imagina que haces un for sobre una colección y cada iteración del for manda el dato al stream.
- Origen: Los datos que un Stream tienen un origen(una fuente), por ejemplo: colecciones, arrays, recursos de I/O. resources.
- Agregado de operaciones: los streams soportan operaciones de estilo declarativo como filter, map, reduce, find, match, sorted, etc.

Dos conceptos importantes de los Streams que los hacen diferentes a la forma de trabajar directamente con colecciones:

- **Pipelining:** ya que muchos métodos de Stream devuelven otro Stream, se puede concebir el agregado de operaciones como una cadena de operaciones que forma larga tubería. Esto tiene muchas ventajas que aquí no discutimos(cortocircuitos, procesamiento paralelo,etc.)
- **iteración interna.** Cuando trabajo con colecciones sin Streams hay bucles que recorren las colecciones, aquí la iteración es automática **(esto nos libera del método imperativo de programación a la hora de pensar)**

En el siguiente gráfico se aprecia cómo trabaja con más detalle el Stream del ejemplo



Ejemplo que demuestra que los datos del origen entran en el Stream pero no se modifican

```
import java.util.Arrays;
import java.util.List;

class App{
    public static void main(String[] args){
        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);

        System.out.print("Los datos originales de myList se mantienen: ");
        for(String s:myList)
            System.out.print(s+" ");

    }
}
```

los datos del arraylist entran de uno en uno, como una secuencia en el stream y se van procesando. Cada dato de arraylist "se copia" y viaja por el agregado de operaciones para ser manipulado. Algo parecido a cuando hacemos un SELECT SQL al que le entra una fila de la tabla y la procesa, pero no modifica la tabla.