PROGRAMACIÓN FUNCIONAL

- Diferente de la programación imperativa
- Lenguaje matemático formal
- En este paradigma, la salida de una función depende exclusivamente de sus parámetros de entrada.
- Más expresivo (con menos código) y elegante.
- Presente en otros lenguajes de programación desde hace tiempo.

Una función depende únicamente de los parámetros de entrada, para generar la salida. Uso de código encadenado, es más elegante programar así. En payton se utiliza mucho, java lo incluye a partir de java 8.

EXPRESIÓN LAMBDA

- Básicamente es un método abstracto, una función sin nombre.
- Su estructura es

```
() -> expresión

(parámetros) -> expresión

(parámetros) -> { sentencias; }
```

Detrás de expresión, no va ;

Mira estos ejemplos:

El primero es para crear un arrayList

El segundo, recibe dos parámetros de tipo entero, y devuelve la suma de ambos.

El tercero, recibe un parámetro a, lo imprime por consola y retorna true.

EJEMPLOS DE EXPRESIONES LAMBDA

```
() -> new ArrayList<>()

(int a, int b) -> a+b

(a) -> {
     System.out.println(a);
     return true;
 }
```

INTERFAZ FUNCIONAL

- Se trata de un interfaz con un solo método (que no sea static ni default) (sin contar equals, ...).
- En lugar de implementar una clase, o una clase anónima, podemos utilizar una expresión lambda.

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Una clase anónima no se crea en un fichero aparte, sino en un punto de nuestro código, eso no lo hemos visto.

Una interfaz funcional, es por ejemplo, la interfaz Comparator. Que comprara dos objetos de un mismo tipo. Solo tiene como método abstracto Compare, por tanto, es funcional. Para usar esa interfaz funcional, implementando la clase, como ha hecho Rubén en clase, a través de una clase anónima (así lo hemos hecho otros, igual que en mis ejemplos) o mediante una expresión Lambda.

Ejemplo: Vamos a ordenar una lista de números, de las 3 formas posibles:

La lista:

```
List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

//1ª Forma: una clase que implementa comparator
lista.sort(new MayorAMenor());
```

Crear una clase, MayorAMenor, que implemente la Interfaz Comparator:

```
4
5 public class MayorAMenor implements Comparator<Integer> {
6
7
8     /*
9      * ESTE MÉTODO SOBREESCRIBE EL "ORDEN NATURAL"
10     * DE LOS NÚMEROS DE FORMA QUE:
11     *
12     * - Si el primer número es menor que el segundo, devuelve un valor positivo.
13     * - Si son iguales devuelve un cero.
14     * - Si el primer número es mayor que el segundo, devuelve un valor negativo.
15     */
16
17    @Override
18    public int compare(Integer o1, Integer o2) {
19        return -(o1.compareTo(o2));
20    }
```

Ordenamos con una clase anónima, la creamos en un punto de nuestro código y la declaramos e instanciamos a la vez, esto nos ahorra el uso de un nuevo fichero, hace lo mismo que el ejemplo anterior:

```
//Eclipse permite convertir esto en una expresión
lista.sort(new Comparator<Integer>() {

    @Override
    public int compare(Integer o1, Integer o2) {
        return -(o1.compareTo(o2));
    }

});
```

Otra manera, con expresión Lambda, (pulsa crtl+1) y te convierte la clase en una expresión Lambda, se puede usar esta expresión porque la interfaz Comparator es funcional, sino no sería posible porque no sabría cuál de los métodos abstractos usar, pero como solo tiene uno.

```
//3ª Forma: una expresion lambda
//Los tipos de datos ¡no son obligatorios! Se infieren del contexto
lista.sort((Integer n1, Integer n2) -> -(n1.compareTo(n2)));
```

Incluso podría poner esto:

```
lista.sort((n1, n2) -> -(n1.compareTo(n2)));
```

Y no especificar el tipo de datos, porque como sabe que tipo de datos tiene la lista, n1 y n2 ya supone que son enteros, por el contexto, puedo quitarles el tipo. Esto hace que el código quede más conciso todavía.

COLLECIONES y forEach

- Nuevo bucle para recorrer colecciones
- Recibe como parámetro un objeto que instancie una interfaz funcional, `Consumer<T>`.
- Perfecto para usar expresiones lambda.

ACCESO A MÉTODOS

- Se puede usar el operador ::
- Abrevia aun más la expresión

String::valueOf	x-> String.valueOf(x)
Object::toString	x -> x.toString()
x::toString	() -> x.toString()
ArrayList::new	() -> new ArrayList
System.out::println	x -> System.out.println(x)

```
//1ª Forma clásica de imprimir los elementos de una lista
for(Integer i: lista)
    System.out.println(i);

//2ª Uso de forEach
lista.forEach(e -> System.out.println(e));

//3ª Uso de referencias a métodos con ::
lista.forEach(System.out::println);

//4º Uso de un bloque de sentencias que suma un valor fijo a los
//elementos de la lista
lista.forEach((e) -> {
    e = e+1;
    System.out.println(e);
});
```

API STREAM

- Nos permite trabajar con una colección con si se tratara de un flujo de información.
- Permite realizar fácilmente operaciones de filtrado, transformación, ordenación , agrupación y presentación de información.

Todas las colecciones pueden usar el método STREAM().

La interfaz stream tiene métodos que permiten, hacer filtrados, ordenación… por ejemplo, tenemos una lista (igual que el ejemplo de antes) de enteros, puedo usar el método filter() para visualizar solo los números mayores o iguales a 5 de mi lista, observa:

```
//2º Imprimir solo los mayores o iguales que 5
lista
    .stream()
    .filter((x) -> x >= 5)
    .forEach(System.out::println);
```

Otro ejemplo, los ordena también:

```
//3º Imprimir solo los mayores o iguales que 5, ordenados inversamente
lista
    .stream()
    .filter((x) -> x >= 5)
    .sorted((n1, n2) -> -(n1.compareTo(n2)))
    .forEach(System.out::println);
```

Mira este ejemplo, agrupo todos los valores en un solo valor, haciendo antes un mapeo, creo un stream(), mapeo de Integer a int, filtro y sumo. Y todo eso se almacena en resultado. Es una forma de mostrar el código muy diferente a lo que estamos acostumbrado, pero está bien que os vaya sonando, sobre todo por si veis código así, sepáis de donde viene.

```
//4º Sumar todos los elementos mayores o igual que 5
int resultado = lista
                    .stream()
                    .mapToInt(v -> v.intValue())
                    .filter((x) -> x >= 5)
                    .sum();
System.out.println(resultado);
```