Java Streams & Lamba Functions

Ejemplos básicos

Funciones Lambda

```
(int a, int b) -> a + b //Básico
(int a) -> {System.out.println(a); return a + b} //Varias sentencias y retorno
() -> new ArrayList() //Sin parámetros (¿Usado para Singleton?)
```

Interfaces funcionales

Interfaces con un único método a definir. Existen 4: `Predicate<T>, Supplier<T>, Consumer<T>, Function<T>`

- `Predicate<T>`
  - Valida criterios. Caso: ``(String s) -> s.length >10
  - Define el método `+test(T):boolean`
- `Supplier<T>`
  - Crea objetos. Caso: `() -> new ArrayList()`
  - Define el método `+get():T`
- `Consumer<T>`
  - Consume objetos. Caso: `(String s) -> s.trim`
  - Define el método `+accpet(T):void`
- `Function<T,R>`
  - Convierte un valor T a otro R. Caso: `(Integer s) -> String.valueOf(s)`
  - Define el método `+apply(T):R`

En realidad no se utilizan, pero son el proceso que se lleva a cabo de fondo en las funciones Lambda.

Streams

Tratamiento de pseudolistas de un tipo concreto. Siguen un modelo **filtro, mapeo, reducción** (selección, transformación de tipo, retorno de resultado).

```
List<Transaccion> transacciones = ...;
int sum = transacciones.stream() //Transforma la lista a Stream
    .filter(t -> t.getProveedor().getCiudad().equals("Valencia")) //Filtrado. Datatype: Stream
    .mapToInt(Transaccion::getPrecio) //Cast a Stream de enteros. Datatype: Stream<Int>
    .sum(); //Recibe los valores y opera
```

Inicialización de Streams

```
Stream planetas = Stream.of("Mercurio", "Venus", "Tierra"); //Directamente
Stream.empty(); //Vacío
//.iterate() es un iterador infinito que empieza en 1 e incrementa 2 cada iteración. Limit
devuelve el número de elementos que debe contener el Strean para detenerse. Si no existe
.limit(), el Stream no funcionará
Stream impares = Stream.iterate(1, x-> x + 2).limit(10);
List<String> algo = ...;
Stream<String> streamCanciones = canciones.stream();
```

Operaciones sobre colecciones de datos

- Básicas
  - `.filter(Predicate<T>)`, para filtrar según un criterio
  - `.distinct()`, para filtrar los elementos distintos. Se sirve del método `equals` del objeto en concreto
  - `.limit(n)`, que devuelve un Stream no mayor del tamaño `n` indicado
  - `.skip(n)`, que retorna un Stream sin los primeros `n` elemento
- Mapeo de datos
  - `map(Function<T,R>)` devuelve un Stream de tipo `R` tras aplicar una función (indicada por parámetro) a todos sus elementos.
  - `mapToDouble, mapToInt, mapToLong.`..
  - `flatMap(Function<T, Stream<R>>)` equivale a un map que transforma cada elemento del Stream en otro Stream para después concatenarlos
  - `flatMapToDouble, flatMapToInt, flatMapToLong.`...
- Terminales
  - `.count()`
  - `.max(Comparator<T>)` y `.min(Comparator<T>)`. Se especifica el comparador para poder hacer las comprobaciones pertinentes. **No depende de compareTo**
- Terminales con búsqueda
  - `.allMatch(Predicate<T>)`
  - `.anyMatch(Predicate<T>)`
  - `.noneMatch(Predicate<T>)`
  - `.findFirst()`, que devuelve el primer ítem. Puede ser `null` o un objeto
  - `.findAny()`, que devuelve cualquire ítem del Stream. Puede ser `null` o un objeto
  - `.reduce(BinaryOperator<T>)`, que realiza una operación en concreto para todos los elementos del Stream
- Recolectores (predecedidos por .collect())
  - `.counting()`, para contar elementos

    ```
    Long cuenta = numeros.stream().collect(counting());
    ```

  - `.joining(string)`, para unir cadenas de Strings

    ```
    String n = nombres.stream().collect(joining(", "));
    ```

- Agrupadores
  - `Map<R, List<T>> groupingBy(Function<T,R>)`, crea un Map que utiliza , uno de los atributos de , como clave. Agrupa en una lista los objetos que tengan el mismo valor en ese atributo concreto

    ```
    // Diccionario (o mapa) del estilo nombre_departamento y empleados que pertenecen a él
    Map<String, List<Empleado>> porDept =
    empleados.stream().collect(groupingBy(Empleado::getDepartamento()));
    // Diccionario (o mapa) del estilo nombre_departamento y # de empleados que pertenecen
    a él
    Map<String, Long> deptCant =
    empleados.stream().collect(groupingBy(Empleado::getDepartamento), counting());
    // TODO
    Map<String, Map<String, List<Empleado>>> = empleados.stream().collect(
        groupingBy(Empleado::getDepartamento, groupingBy(Empleado::getCiudad)));
    ```

  - Depuración
    - `.peek(Consumer<T>)`, lee los valores de un Stream y hace una operación concreta. Cualquier operación dentro de un peek **no afecta a los valores del Stream**.