El uso de Java Stream forEach poco a poco se va haciendo hueco entre los desarrolladores que utilizan programación funcional. El problema fundamentalmente que existe con el método forEach es que estamos muy acostumbrados a usar un bucle clásico forEach y nos funciona bastante bien .Así que da un poco de pereza cambiarlo . ¿ Aporta alguna ventaja el uso de un forEach a nivel de Streams o es lo mismo de siempre?. Vamos a verlo.

Java forEach clásico

Supongamos que disponemos de la clase Persona.

```
package com.arquitecturajava;

public class Persona {

private String nombre;
private String apellidos;
private int edad;
public String getNombre() {
return nombre;
}
public void setNombre(String nombre) {
this.nombre = nombre;
}
public int getEdad() {
return edad;
}
public void setEdad(int edad) {
this.edad = edad;
}
public String getApellidos() {
return apellidos;
```

```
}
public void setApellidos(String apellidos) {
this.apellidos = apellidos;
}
public Persona(String nombre, String apellidos, int edad) {
super();
this.nombre = nombre;
this.apellidos = apellidos;
this.edad = edad;
}

}
```

Podemos construir una lista de Personas en código:

```
Persona p1 = new Persona("juan", "sanchez", 20)
Persona p2 = new Persona("ana", "gomez", 12);
Persona p3 = new Persona("pedro", "gutierrez", 40);
List<Persona> lista=Arrays.asList(p1,p2,p3);
```

Ahora es el momento de decidir como recorrer la lista. En principio lo más natural es recorrerla con un bucle forEach de Java:

```
for (Persona p:lista) {
// formato clasico
System.out.println(p.getNombre());
System.out.println(p.getApellidos());
System.out.println(p.getEdad());

}
```

El resultado se mostrará por la consola y estamos bastante contentos con el:

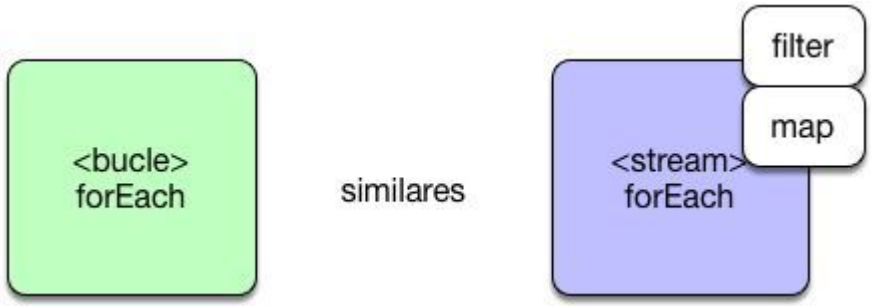
Java Stream forEach , una alternativa

Eso sí podemos hacer lo mismo y utilizar el método forEach de los streams:

```
lista.stream().forEach((p)-> {
System.out.println(p.getNombre());
System.out.println(p.getApellidos());
System.out.println(p.getEdad());
});
```

El resultado no variará:

Es cierto que al convertirlo en Stream podemos tener acceso a operaciones como map o filter que le dan una mayor flexibilidad . Ahora bien esto no es lo importante ya que a nosotros únicamente nos interesa la estructura forEach.



¿No aporta o si aporta?

En estos momentos el uso de Streams no nos aporta demasiado. Sin embargo vamos a ver otro ejemplo complementario. Supongamos que tenemos que recorrer un diccionario con Personas (hashmap) con los siguientes elementos:

```
Map < String, Persona > mapa = new HashMap < String, Persona > ();

mapa.put(p1.getNombre(), p1);
mapa.put(p2.getNombre(), p2);
mapa.put(p3.getNombre(), p3);
```

No es muy difícil recorrerlo con un bucle forEach:

```
for (String nombre: mapa.keySet()) {

Persona pSeleccionada=mapa.get(nombre);

System.out.println(pSeleccionada.getNombre());
System.out.println(pSeleccionada.getApellidos());
System.out.println(pSeleccionada.getEdad());
}
```

Bueno no es muy difícil , pero no queda muy claro y los datos salen desordenados .

```
ana
gomez
12
pedro
gutierrez
40
juan
sanchez
20
```

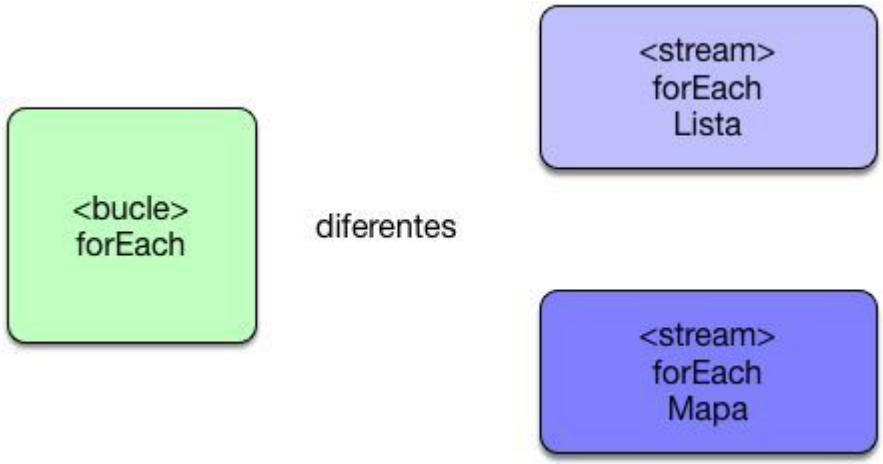
Lo que estamos haciendo es acceder a la colección de keys o claves del diccionario .Una vez tenemos las claves accedemos a cada elemento del diccionario usando el método get y pasando la clave, de ahí que salgan desordenados. ¿Cómo sería esto con el uso de streams?. Vamos a verlo:

```
mapa.forEach((k,v)->{

System.out.println(v.getNombre());
System.out.println(v.getApellidos());
System.out.println(v.getEdad());

});
```

El código es mucho más natural ya que usa una sobrecarga del método forEach y nos permite acceder directamente a los valores de un diccionario.



Muchas veces nos olvidamos de todas la novedades que trae Java 8 a nivel de programación

funcional para simplificarnos el trabajo. El uso de Java Stream forEach es uno de ellos.

Otros artículos relacionados

1. Java 8 Stream y workflows
2. Java 8 Lambda Expressions (I)
3. Java 8 Lambda y forEach (II)
4. Oracle streams