Streams paralelos

Fundamentos

- Streams que utilizan la multitarea para realizar las operaciones
- Dividen la tarea a realizar entre varios procesos que se ejecutan de manera concurrente
- Los Stream paralelos siguen siendo objetos Stream, que utilizan los mismos métodos estudiados, pero operando de forma más eficiente
- Deben evitarse en tareas de ordenación

Creación de un Stream paralelo

- A partir de una colección. Utilizamos el método parallelStream() en lugar de stream():

```
List<Integer> nums=List.of(20,100,80,25);
Stream<Integer> pst=nums.parallelStream();
```

- A partir de un Stream secuencial. Si ya disponemos de un Stream estándar o secuencial, podemos obtener u Stream paralelo a través del método parallel():

```
Stream<Integer> st=Stream.of(20,100,80,25);
Stream<Integer> pst=st.parallel();
```

Utilización de Stream paralelo

- **Una vez creado, se utiliza de la misma manera que los streams secuenciales, ya que se trata del mismo tipo de objeto:**

```
List<Integer> nums=List.of(20,100,80,25);
Stream<Integer> pst=nums.parallelStream();
//muestra total de números pares
System.out.println(pst.filter(n->n%2==0).count());
```

- **Precaución con:**

```
List<Integer> nums=List.of(20,100,80,25);
Stream<Integer> pst=nums.parallelStream();
//intenta mostrar los números ordenados
pst.sorted().forEach(s->System.out.println(s));
```

Cada proceso ordena un bloque de números, el resultado final **no está ordenado**

Convertir a secuencial

- Se puede crear un Stream paralelo para realizar operaciones costosas y volver a obtener después uno secuencial para tareas que con un paralelo no pueden hacerse.

- Se emplea el método sequential() de Stream:

```
List<Integer> nums=List.of(20,100,80,25,39,1,7,100,25,4,2,20);
Stream<Integer> pst=nums.parallelStream();
//muestra ordenados los números pares no repetidos
pst
    .distinct()
    .filter(n->n%2==0)
    .sequental()
    .forEach(n->System.out.println(n));
```