Acabamos de ver como usar Java equals y hashcode

Spring Reactor es el framework de programación reactiva de Spring y poco a poco tendremos que aprender a utilizarlo . Los conceptos fundamentales que el Framework soporta son las clases Flux y Mono de las cuales ya hablé en algún artículo anterior. La programación reactiva esta fuertemente basada en la programación funcional . Vamos a ver un ejemplo sencillo.

```
package com.arquitecturajava.reactor;

import reactor.core.publisher.Flux;

public class Principal2 {

	public static void main(String[] args) {

		Flux<Integer> flux1 = Flux.just(2, 3, 4, 5, 6, 1, 7);

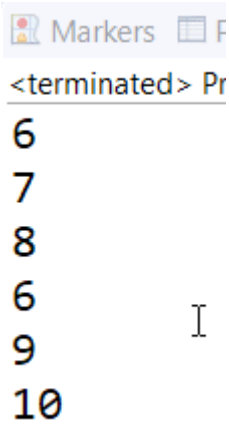
		flux1.filter(e -> e >= 5).map(e -> e +
1).subscribe(System.out::println);
		Flux<Integer> flux2 = Flux.just(3, 3, 3, 4, 5, 8, 9);

		flux2.filter(e -> e >= 5).map(e -> e +
1).subscribe(System.out::println);
	}

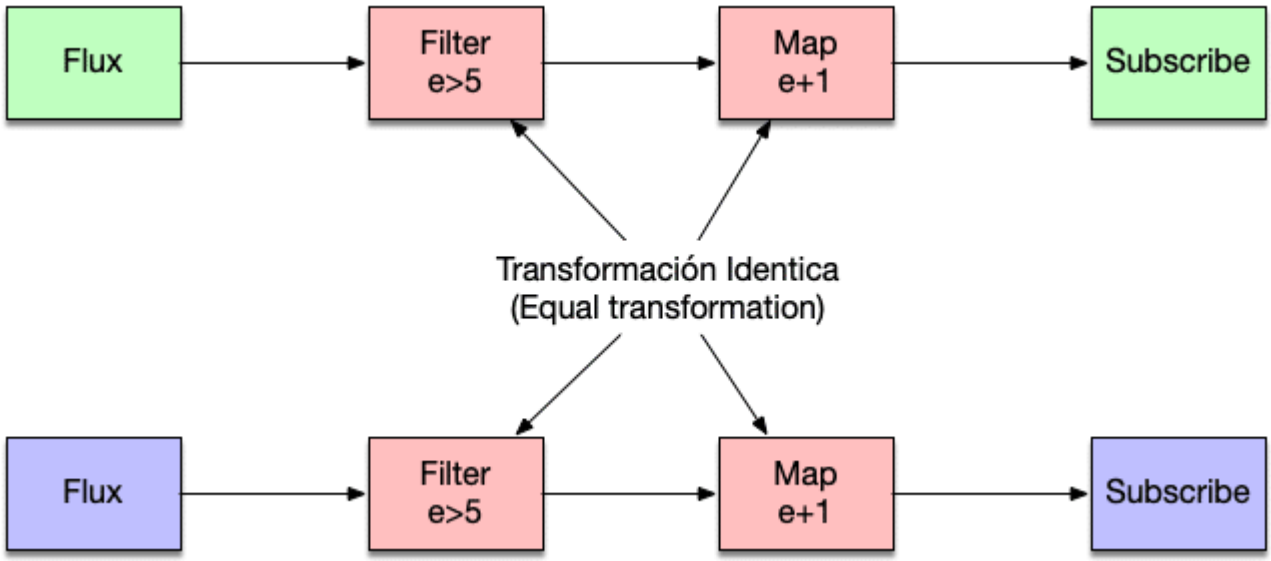
}
```

En este caso nos encontramos ante un sencillo Flux que contiene una lista de números . Recorremos el Flux y nos quedamos con los valores que superen el 5 una vez hecho eso realizamos una transformación y le sumamos al numero 1 . Es como si nos quedaramos con las personas que han aprobado un examen y por aprobar sumáramos un punto a la nota final

. Si ejecutamos el código el resultado sera:



¿Cómo de correcto es el código? . La realidad es que el código es bastante correcto y nos sirve para abordar las transformaciones que queremos. Ahora bien si nos fijamos nos daremos cuenta que ambas transformaciones son idénticas y estamos de alguna manera repitiendo código. ¿Cómo podemos solventar este problema?



Este es un problema habitual de la programación funcional . Ya que muchos de nosotros entre los cuales me incluyo . Solemos usar la programación funcional como un conjunto de operaciones con lambdas y métodos de referencia que nos facilitan transformaciones complejas . Pero no abordamos temas como su reutilización. En este caso vamos a modificar

nuestro código para poder reutilizar este concepto a nivel de Spring Reactor y poder compartir la transformación.

```
package com.arquitecturajava.reactor;

import java.util.function.Function;

import reactor.core.publisher.Flux;

public class Principal4 {

	public static void main(String[] args) {

		Function<Flux<Integer>,Flux<Integer>> transformacion=
flux->flux.filter(e -> e >= 5).map(e -> e + 1);
		Flux<Integer> flux1 = Flux.just(2, 3, 4, 5, 6, 1, 7);

		//flux1.filter(e -> e >= 5).map(e -> e +
1).subscribe(System.out::println);
flux1.transform(transformacion).subscribe(System.out::println);

		Flux<Integer> flux2 = Flux.just(3, 3, 3, 4, 5, 8, 9);

		//flux2.filter(e -> e >= 5).map(e -> e +
1).subscribe(System.out::println);
flux2.transform(transformacion).subscribe(System.out::println);

	}

}
```

En este caso estamos declarando un objeto de tipo Function . Recordemos que este

interface Funcional es el encargado de definir una operación que recibe un parámetro de entrada y devuelve un resultado como salida. A través de este interface definimos la operación de transformación para luego aplicarla a los dos Flux que tenemos.

```
package com.arquitecturajava.reactor;

import java.util.function.Function;

import reactor.core.publisher.Flux;

public class Principal4 {

	public static void main(String[] args) {

		Function<Flux<Integer>,Flux<Integer>> transformacion=
flux->flux.filter(e -> e >= 5).map(e -> e + 1);
		Flux<Integer> flux1 = Flux.just(2, 3, 4, 5, 6, 1, 7);

		//flux1.filter(e -> e >= 5).map(e -> e +
1).subscribe(System.out::println);
flux1.transform(transformacion).subscribe(System.out::println);

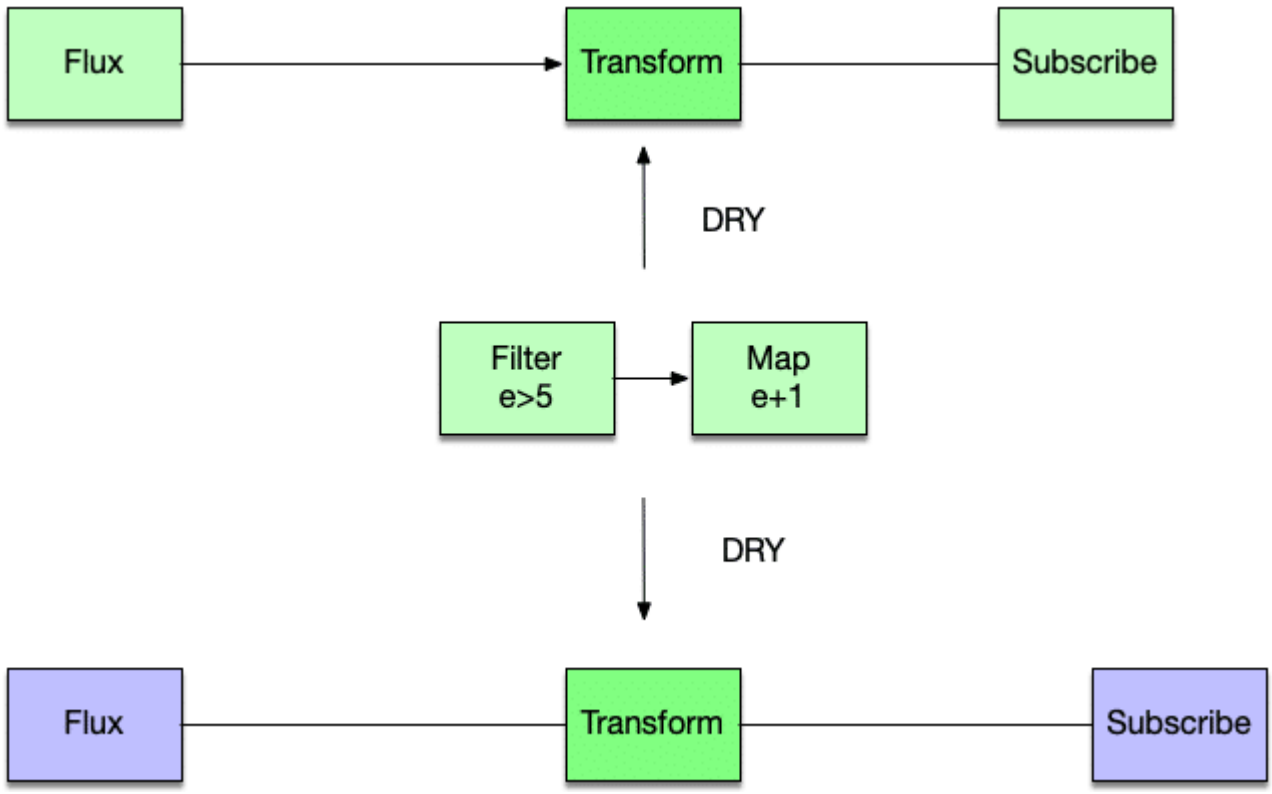
		Flux<Integer> flux2 = Flux.just(3, 3, 3, 4, 5, 8, 9);

		//flux2.filter(e -> e >= 5).map(e -> e +
1).subscribe(System.out::println);
flux2.transform(transformacion).subscribe(System.out::println);

	}

}
```

De esta manera podremos reutilizar la transformación en varios lugares. Usando el método transform que el objeto Flux soporta.



De esta forma conseguiremos que una misma transformación se reutilice en varios sitios

- ¿Que es Spring WebFlux?
- ¿Que es Spring Boot?
- Spring Boot JPA
- Reactor Framework