Preguntarse la diferencia entre un Stream vs Observable hoy en día es muy muy habitual. Poco a poco la programación funcional va entrando en nuestro código y el manejo de las capacidades de Java 8 se va convirtiendo en algo más y más habitual . Aun así hay conceptos que nos cuesta entender y uno de ellos es la diferencia que existe entre un Stream y un Observable. Vamos a echarlo un vistazo y aclarar estos temas. Para ello empezaremos con un ejemplo de un Stream de cadenas.

```
package com.arquitecturajava;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

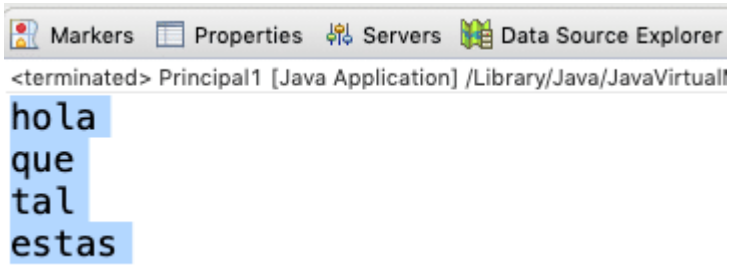
public class Principal1 {

  public static void main(String[] args) {
    List<String> lista= new ArrayList<>();
    lista.add("hola");
    lista.add("que");
    lista.add("tal");
    lista.add("estas");
    Stream<String> mistream=lista.stream();
    mistream.forEach((s)-> {
      System.out.println(s);
    });

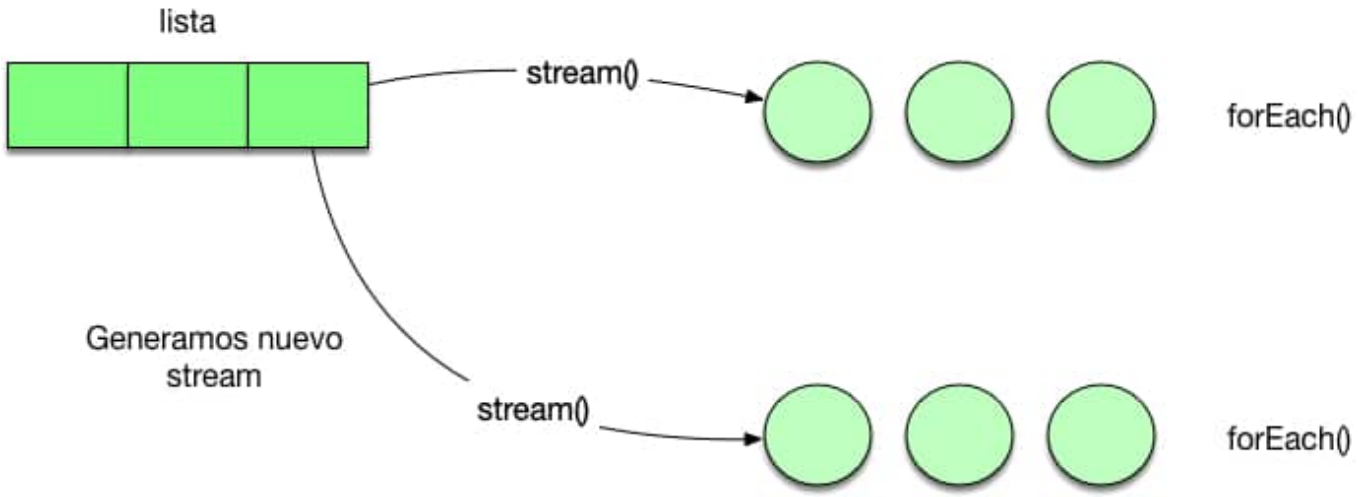
  }

}
```

Se trata de un código muy sencillo en el cual disponemos de un Array de cadenas y le recorremos con un stream imprimiendo los valores por la consola.



Esto es elemental , una de las características que tienen los streams es que se trata de un flujo de trabajo que solo se puede recorrer una vez.



Es decir una vez recorrido no puedo volver a invocar el método forEach sino que lo que tengo que hacer es volver a solicitar que me generen otro Stream.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class Principal1 {
```

```
  public static void main(String[] args) {

    List<String> lista = new ArrayList<>();

    lista.add("hola");
    lista.add("que");
    lista.add("tal");
    lista.add("estas");

    Stream<String> mistream = lista.stream();

    mistream.forEach((s) -> {

      System.out.println(s);
    });

    Stream<String> mistream2 = lista.stream();

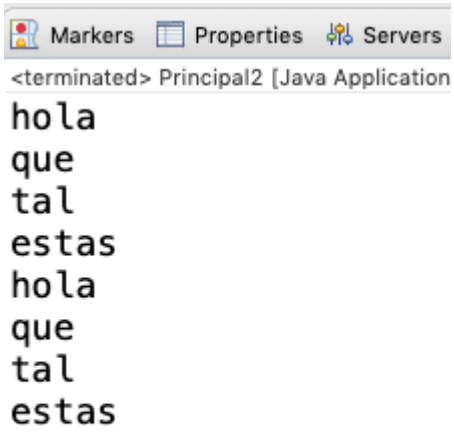
    mistream2.forEach((s) -> {

      System.out.println(s);
    });

  }

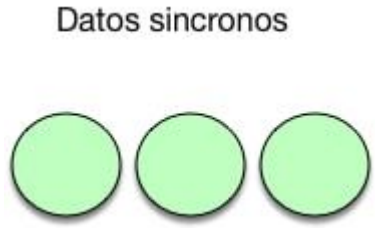
}
```

El resultado se ve en la consola:



Streams y sus datos

Otra de las características que definen a los Streams es que son totalmente síncronos es decir contienen los datos desde el momento inicial y los procesan de forma totalmente síncrona uno tras otro.



Streams vs Observable

Para poder hacer uso de Observables necesitaremos de entrada incluir una dependencia de Maven.

```
<dependency>
	<groupId>io.reactivex.rxjava2</groupId>
	<artifactId>rxjava</artifactId>
```

```
      <version>2.2.10</version>
    </dependency>
```

Hecho esto es momento de ver el código pero utilizando un observable.

```
package com.arquitecturajava;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

import io.reactivex.Observable;

public class Principal2 {

  public static void main(String[] args) {

    List<String> lista = new ArrayList<>();

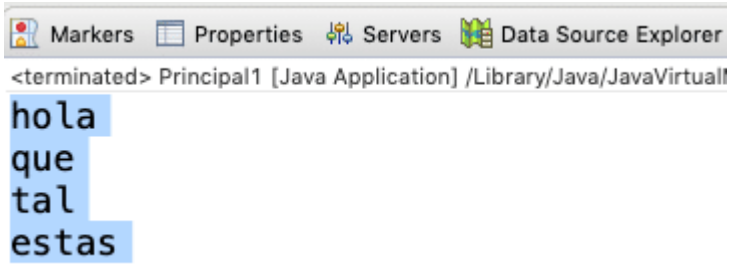
    lista.add("hola");
    lista.add("que");
    lista.add("tal");
    lista.add("estas");

    Observable<String> flujo=Observable.fromIterable(lista);

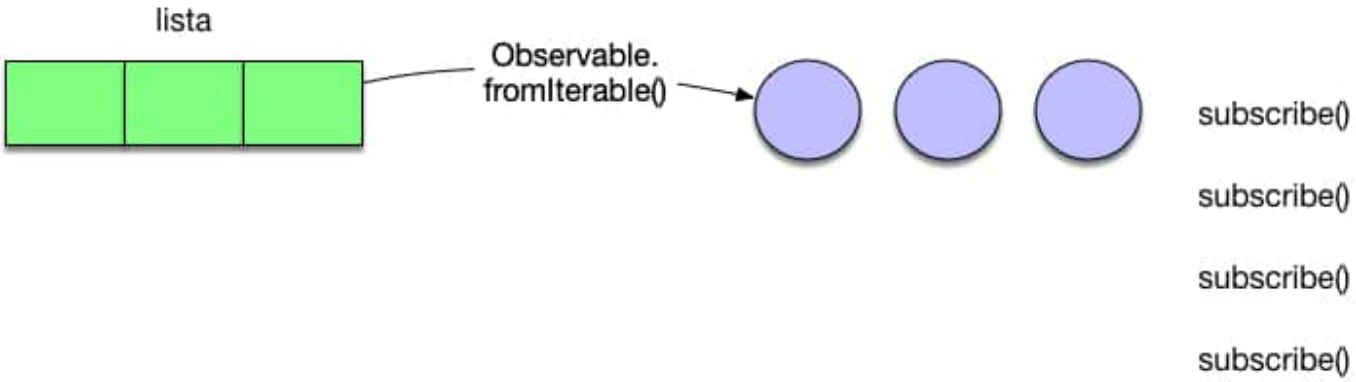
    flujo.subscribe((s)-> {
      System.out.println(s);
    });
  }

}
```

El resultado por la consola es idéntico:



Cabe preguntarse en qué se diferencia un Stream vs Observable. La realidad es que nosotros nos podemos subscribir varias veces a un observable algo que no sucede con un stream.



Veamos un ejemplo:

```
package com.arquitecturajava;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

import io.reactivex.Observable;
```

```
public class Principal2 {

  public static void main(String[] args) {

    List<String> lista = new ArrayList<>();

    lista.add("hola");
    lista.add("que");
    lista.add("tal");
    lista.add("estas");

    Observable<String> flujo = Observable.fromIterable(lista);

    flujo.subscribe((s) -> {

      System.out.println(s);
    });

    flujo.subscribe((s) -> {

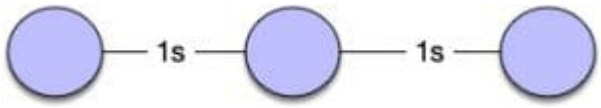
      System.out.println(s);
    });
  }

}
```

El resultado será idéntico al anterior caso en el cual creábamos dos streams

```
Markers  Properties  Servers
<terminated> Principal2 [Java Application
hola
que
tal
estas
hola
que
tal
estas
```

Esa es la primera diferencia importante , la segunda diferencia es que los observable han sido diseñados para gestionar situaciones de programación asíncrona compleja de forma relativamente sencilla. Es decir yo puedo convertir la secuencia que tenemos de textos en una secuencia asíncrona en la cual cada elemento se genera pasado un segundo.



Veamoslo:

```
package com.arquitecturajava;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;

import io.reactivex.Observable;

public class Principal2 {
```

```
  public static void main(String[] args) {

    List<String> lista = new ArrayList<>();

    lista.add("hola");
    lista.add("que");
    lista.add("tal");
    lista.add("estas");

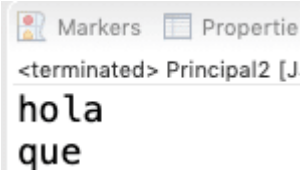
    Observable<String> flujo = Observable.fromIterable(lista);
    Observable<String> flujoLento=flujo
        .concatMap((s)->Observable.just(s).delay(1,TimeUnit.SECONDS));
    flujoLento.subscribe((mensaje) -> {

      System.out.println(mensaje);
    });

    try {
      Thread.sleep(6000);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

}
```

En este caso hemos usado el operador concatMap de los observables que nos permite esperar a que ambos observables estén disponibles el de la lista y el del intervalo para emitir un nuevo item. De esta forma convertiremos el flujo de trabajo en un flujo asíncrono e iremos viendo como se imprimen los elementos uno a uno cada segundo.



Estas son las dos diferencias mas importantes de Stream vs Observable. El primero solo se ejecuta una vez y es sincrono. El segundo se puede ejecutar varias veces y tiene fuertes capacidades asíncronas.

1. Hot vs Cold Observable con Rx.js
2. Usando Rx Observables en JavaScript
3. Flux vs Mono ,Spring y la programación reactiva
4. Oracle Java Stream