Observables

Los Observables son colecciones Push 'perezosas' de múltiples valores. Rellenan el hueco que faltaba en la siguiente tabla:

	Un solo valor	Múltiples valores
Pull	Función	Iterador
Push	Promesa	Observable

Ejemplo: el siguiente Observable emite los valores `1`, `2`, `3` inmediatamente (de forma síncrona) al suscribirse a él, el valor `4` un segundo después de la suscripción y por último, se completa.

```
import { Observable } from "rxjs";

const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
```

Para invocar al Observable y poder ver estos valores, hay que *suscribirse* a él:

```
import { Observable } from "rxjs";

const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});

console.log("justo antes de la suscripción");
observable.subscribe({
```

```
  next(x) {
    console.log("obtenido el valor " + x);
  },
  error(err) {
    console.error("ha ocurrido un error: " + err);
  },
  complete() {
    console.log("listo");
  },
});
console.log("justo después de la suscripción");
```

Este código imprime lo siguiente por consola:

```
justo antes de la suscripción
obtenido el valor 1
obtenido el valor 2
obtenido el valor 3
just después de la suscripción
obtenido el valor 4
listo
```

Pull vs Push

Pull y *Push* son dos protocolos diferentes que describen cómo un *Producer* de datos se comunica con un *Consumer* de datos.

¿Qué es *Pull*? En los sistemas *Pull*, el *Consumer* es el que determina cuándo recibe los datos del *Producer*. El *Producer* no es consciente de cuándo se le proporcionan los datos al *Consumer*.

Todas las funciones de JavaScript son sistemas *Pull*. La función es el *Producer* de datos, y el código que llama a la función la consume *extrayendo/pulling* el valor de retorno de la llamada.

En ES2015, se introdujeron las funciones e iteradores *generator* (`function*`), otro tipo sistema *Pull*. El código que llama a `iterator.next()` es el *Consumer*, que "extrae" múltiples valores del iterador (el *Producer*).

	Productor	Consumidor
Pull	Pasivo: produce datos cuando se le piden	Activo: decide cuándo se piden los datos
Push	Activo: produce datos a su propio ritmo	Pasivo: reacciona a los datos recibidos

¿Qué es *Push*? En los sistemas *Push*, el *Producer* determina cuándo se le envían los datos al *Consumer*. El *Consumer* no es consciente de cuándo va a recibir los datos.

Las Promesas son los sistemas *Push* más comunes que hay hoy en día en JavaScript. Una Promesa (el *Producer*) hace entrega de un valor resuelto a *callbacks* registradas (los *Consumers*), pero, al contrario que las funciones, es la Promesa la que determina en qué momento se "empuja" el valor a las *callbacks*.

RxJS propone Observables, un nuevo sistema *Push* para JavaScript. Un Observable es un *Producer* de múltiples valores que "empuja" a los Observadores (*Consumers*).

> Una Función es una computación evaluada de forma perezosa que retorna un único valor de forma síncrona al ser invocada.
>
> Un Generador es una computación evaluada de forma perezosa que devuelve cero o (potencialmente) infinitos valores de forma síncrona en la iteración.
>
> Una Promesa es una computación que puede o no retornar finalmente un único valor.
>
> Un Observable es una computación evaluada de forma perezosa que puede retornar cero o (potencialmente) infinitos valores de forma síncrona o asíncrona desde el momento de la invocación o en adelante.

Observables como generalizaciones de funciones

Al contrario de la opinión popular, los Observables no son como los EventEmitters ni como las Promesas para múltiples valores. Los Observables pueden actuar como EventEmitters en algunos casos, como cuando son *multicasteados* mediante Subjects de RxJS, pero no es lo común.

> *Los Observables son como funciones con cero argumentos, pero los generalizan para permitir múltiples valores.*

Consideremos el siguiente caso:

```
function foo() {
  console.log("Hola");
  return 42;
}

const x = foo.call(); // igual que foo()
console.log(x);
const y = foo.call(); // igual que foo()
console.log(y);
```

Copy

Se obtiene la siguiente salida:

```
"Hola"
42
"Hola"
42
```

Se puede reescribir este comportamiento con Observables:

```
import { Observable } from "rxjs";

const foo = new Observable((subscriber) => {
  console.log("Hola");
  subscriber.next(42);
});

foo.subscribe((x) => {
  console.log(x);
});
foo.subscribe((y) => {
  console.log(y);
});
```

Copy

Y se obtiene la misma salida:

```
"Hola"
42
"Hola"
42
```

Esto ocurre porque tanto las funciones como los Observables son computaciones perezosas. Si no se llama a la función, el `console.log('Hola')` no ocurrirá. Lo mismo ocurre con los Observables, si no se 'llaman' (con subscribe), el `console.log('Hola')` no ocurrirá. Además, tanto las llamadas a funciones, como las suscripciones son operaciones aisladas: dos llamadas a funciones provocan dos efectos colaterales separados, y dos suscripciones a Observables provocan dos efectos colaterales separados. Al contrario que los EventEmitters, sin tener en cuenta la existencia de los suscriptores, comparten los efectos colaterales y son de ejecución *eager* o inmediata.

> *Suscribirse a un Observable es análogo a llamar a una función.*

Hay quien piensa que los Observables son asíncronos. Esto no es cierto. Si se rodea una llamada a una función con logs, así:

```
console.log("antes");
console.log(foo.call());
console.log("después");
```

Copy

Se podrá ver la siguiente salida:

```
"antes"
"Hola"
42
"después"
```

Aquí se puede ver el mismo comportamiento, con Observables:

```
console.log("antes");
foo.subscribe((x) => {
  console.log(x);
});
console.log("después");
```

La salida es:

```
"antes"
"Hola"
42
"después"
```

Lo que demuestra que la suscripción a `foo` es completamente síncrona, al igual que una función.

> *Los Observables pueden emitir valores de forma síncrona o asíncrona.*

¿Cuál es la diferencia entre un Observable y una función? Los Observables pueden "retornar" múltiples valores a lo largo del tiempo, algo que las funciones no pueden hacer. No se puede hacer esto:

```
function foo() {
  console.log("Hola");
  return 42;
  return 100; // Código muerto, nunca se llegará a ejecutar.
}
```

Las funciones solo pueden retornar un valor. Los Observables, sin embargo, pueden hacer esto:

```
import { Observable } from "rxjs";

const foo = new Observable((subscriber) => {
  console.log("Hola");
  subscriber.next(42);
  subscriber.next(100); // "retornar" un segundo valor
  subscriber.next(200); // "retornar" otro valor más
});

console.log("antes");
foo.subscribe((x) => {
  console.log(x);
});
console.log("después");
```

Con una salida síncrona:

```
"antes"
"Hola"
42
100
200
"después"
```

También se pueden retornar valores de forma asíncrona:

```
import { Observable } from "rxjs";

const foo = new Observable((subscriber) => {
  console.log("Hola");
  subscriber.next(42);
  subscriber.next(100);
  subscriber.next(200);
  setTimeout(() => {
    subscriber.next(300); // Ocurre asíncronamente
  }, 1000);
});

console.log("antes");
foo.subscribe((x) => {
  console.log(x);
});
console.log("después");
```

Con la siguiente salida:

```
"antes"
"Hola"
42
100
200
"después"
300;
```

Conclusión:

`func.call()` significa "dame un solo valor síncrono" `observable.subscribe()` significa "dame un número determinado de valores síncronos o asíncronos"

Anatomía de un Observable

Los Observables se crean mediante `new Observable` o un operador de creación, se suscribe a ellos mediante un *Observador*, durante la ejecución envían notificaciones `next`/ `error` / `complete`, y se puede

cancelar su ejecución. Estos cuatro aspectos están codificados en cada instancia del Observable, aunque algunos están relacionados con otros tipos, como el *Observador* y la *Suscripción*

Cometidos principales de un Observable:

- Crear Observables
- Suscribirse a un Observable
- Ejecutar el Observable
- Deshacerse de un Observable

Creando Observables

El constructor del `Observable` recibe un argumento: la función `suscribe`.

El siguiente ejemplo crea un Observable que emite el string 'hola' cada segundo a un *suscriber*.

```
import { Observable } from "rxjs";

const observable = new Observable(function subscribe(subscriber) {
  const id = setInterval(() => {
    subscriber.next("hola");
  }, 1000);
});
```

Copy

> *Los Observables se pueden crear con `new Observable`. Más comúnmente, los Observables se crean mediante funciones de creación, como `of`, `from`, `interval` etc.*

En el ejemplo anterior, la función `suscribe` es la pieza más importante para describir al Observable. Vamos a ver qué sinifica suscribirse.

Suscripción a un Observable

Se puede realizar una suscripción a un Observable `observable` de la siguiente manera:

```
observable.subscribe((x) => console.log(x));
```

Copy

No es una coincidencia que `observable.subscribe` y `subscribe` en `new Observable (function subscribe(subscriber) {...})` tengan el mismo nombre. En la biblioteca, son distintos, pero por razones prácticas se pueden considerar conceptualmente iguales.

Esto muestra que las llamadas `subscribe` no se comparten entre los diversos Observadores del mismo Observable. Al hacer una llamada a `observable.subscribe` desde un Observador, la función `subscribe in new Observable(function subscribe(subscriber) {...})` se ejecuta para dicho suscriptor. Cada llamada a `observable.subscribe` provoca un *setup* independiente para dicho suscriber.

> *Suscribirse a un Observable es como llamar a una función, proporcionando callbacks donde se recibirán los datos.*

Esto es drásticamente diferente a las APIs de manejo de eventos como `addEventListener` / `removeEventListener`. Con `observable.subscribe`, el Observador no se registra como un listener en el Observable. El Observable ni siquiera mantiene una lista de Observadores agregados.

La llamada `subscribe` es simplemente una forma de que comience la "ejecución Observable" y de enviar valores o eventos a un Observador de dicha ejecución.

Ejecutando Observables

El código dentro de `new Observable(function subscribe(subscriber) {...})` representa una "ejecución Observable", una computación perezosa que solo se da para el Observador que se suscribe. La ejecución produce múltiples valores a lo largo del tiempo, bien síncrona o asíncronamente.

Hay tres tipos de valores que una Ejecución Observable puede emitir:

> notificación "Next": envía un valor como puede ser un Number, un String, un Object, etc.
>
> notificación "Error": envía un Error JavaScript o una excepción.
>
> notificación "Complete": no envía ningún valor.

Las notificaciones "Next" son las más importantes, y las más comunes: representan los datos que se envían al suscriptor. Las notificaciones "Error" y "Complete" solo pueden ocurrir una vez durante la Ejecución Observable, y solo puede darse una de ellas.

La mejor forma de expresar estas restricciones es mediante la Gramática Observable o Contract, escrita mediante una expresión regular:

```
next*(error|complete)?
```
Copy

> *En una Ejecución Observable, se pueden emitir desde cero a infinitas notificaciones. En el caso de que se diera una notificación "Error" o "Complete", ninguna otra notificación podrá emitirse a partir de ese momento.*

En el siguiente ejemplo podemos ver una Ejecución Observable que emite tres notificaciones "Next", y se completa:

```
import { Observable } from "rxjs";
```
Copy

```
const observable = new Observable(function subscribe(subscriber) {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();
});
```

Los Observables se adhieren estrictamente al Contrato Observable, por lo que el siguiente código no puede llegar a emitir la notificación "Next" 4:

```
import { Observable } from "rxjs";

const observable = new Observable(function subscribe(subscriber) {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();
  subscriber.next(4); // Esta notificación no se emite, ya que supondría una violación
});
```

Suele ser una buena práctica el "envolver" cualquier código que haya en el `subscribe` con un bloque `try/catch` que emita una notificación "Error" en el caso de que capture una excepción:

```
import { Observable } from "rxjs";

const observable = new Observable(function subscribe(subscriber) {
  try {
    subscriber.next(1);
    subscriber.next(2);
    subscriber.next(3);
    subscriber.complete();
  } catch (err) {
    subscriber.error(err); // Emite un error si se captura alguno
  }
});
```

Cancelando Ejecuciones Observables

Ya que las Ejecuciones Observables pueden ser infinitas, y es muy común que un Observador quiera abortar la ejecución en un tiempo finito, necesitamos una API para poder cancelar una ejecución. Dado que cada ejecución pertenece exclusivamente a un único Observador, una vez que el Observador ha terminado

de recibir valores, tiene que tener una forma de parar la ejecución, para evitar el gasto de recursos de memoria o de computación.

Cuando se hace una llamada a `observable.subscribe`, el Observador se vincula a la ejecución Observable recién creada. Este llamada también devuelve un objeto, la `Subscription`:

```
const subscription = observable.subscribe((x) => console.log(x));
```

La Suscripción representa la ejecución en curso, y tiene una API minimalista que permite cancelar dicha ejecución. Se puede leer más acerca del tipo Subscription aquí. Con `subscription.unsubscribe()` se puede cancelar la ejecución en curso:

```
import { from } from "rxjs";

const observable = from([10, 20, 30]);
const subscription = observable.subscribe((x) => console.log(x));
// Más tarde:
subscription.unsubscribe();
```

> *Al suscribirse a un Observable, se obtiene una Suscripción, que representa la ejecución en curso. Para cancelar la ejecución, simplemente hay que llamar a `unsubscribe()`.*

Cada Observable es responsable de definir cómo deshacerse de los recursos de su ejecución cuando creamos el Observable mediante `create()`. Podemos hacerlo mediante el retorno de una función `unsubscribe` personalizada desde la función `subscribe()`.

Por ejemplo, así es como limpiamos una ejecución intervalo creada con `setInterval`:

```
const observable = new Observable(function subscribe(subscriber) {
  // Utilizando el recurso interval
  const intervalId = setInterval(() => {
    subscriber.next("hi");
  }, 1000);

  // Proporcionamos una forma de cancelar el recurso interval
  return function unsubscribe() {
    clearInterval(intervalId);
  };
});
```

Al igual ocurre con el parecido entre `observable.subscribe` y `new Observable(function subscribe() {...})`, el `unsubscribe` que retornamos desde `subscribe` es conceptualmente equivalente al `subscription.unsubscribe`. De hecho, si se eliminan los tipos de ReactiveX que rodean estos conceptos, quedaría un código JavaScript básico.

```
function subscribe(subscriber) {
  const intervalId = setInterval(() => {
    subscriber.next("hola");
  }, 1000);

  return function unsubscribe() {
    clearInterval(intervalId);
  };
}

const unsubscribe = subscribe({ next: (x) => console.log(x) });

// Más tarde:
unsubscribe(); // Se gestiona la eliminación de los recursos
```

La razón por la que se utilizan tipos de Rx como el *Observable*, *Observador* o *Subscription* es para garantizar la seguridad (el Contrato Observable) y la componibilidad con Operadores.