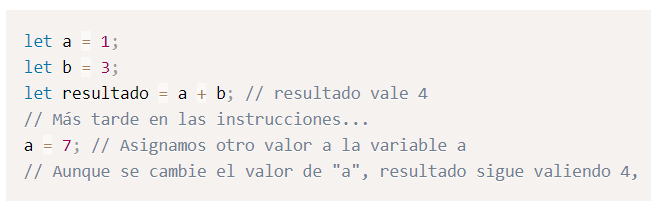
# Programación tradicional Vs Programación Reactiva

**En la programación tradicional las instrucciones se ejecutan una tras la otra**. Por lo tanto, si realizamos un cálculo con dos variables y obtenemos un resultado, **aunque las variables usadas para hacer el cálculo cambien en el futuro, el cálculo ya se realizó y por lo tanto el resultado no cambiará**



En el ejemplo anterior se muestra como es el modo de trabajo de la programación tradicional. **En la programación reactiva la variable resultado habría actualizado su valor al alterase las variables con las que se realizó el cálculo**

### Programación reactiva y flujos de datos

En la programación reactiva se utilizan se utilizan intensamente los flujos de datos (**la programación reactiva es la programación con flujos de datos asincrónicos**)

En la programación reactiva se pueden crear flujos de datos (*streams)* a partir de cualquier cosa, como, por ejemplo:

* Los valores que tome una variable a lo largo del tiempo
* Clics sobre un botón
* Cambios en una estructura de datos
* Una consulta para traer un JSON de un servidor
* Un feed RSS
* Un listado de tuits
* Etc.

Cuando trabajamos con programación reactiva lo que hacemos es generar sistemas capaces de consumir esos flujos de datos, filtrando los datos que nos interesan, combinándolos con otros, etc.

Como objetivo final, la programación reactiva se ocupa de lanzar distintos tipos de eventos sobre flujos de datos:

* La aparición de algo interesante dentro del flujo de datos
* La aparición de un error dentro del flujo de datos
* La finalización del flujo de datos

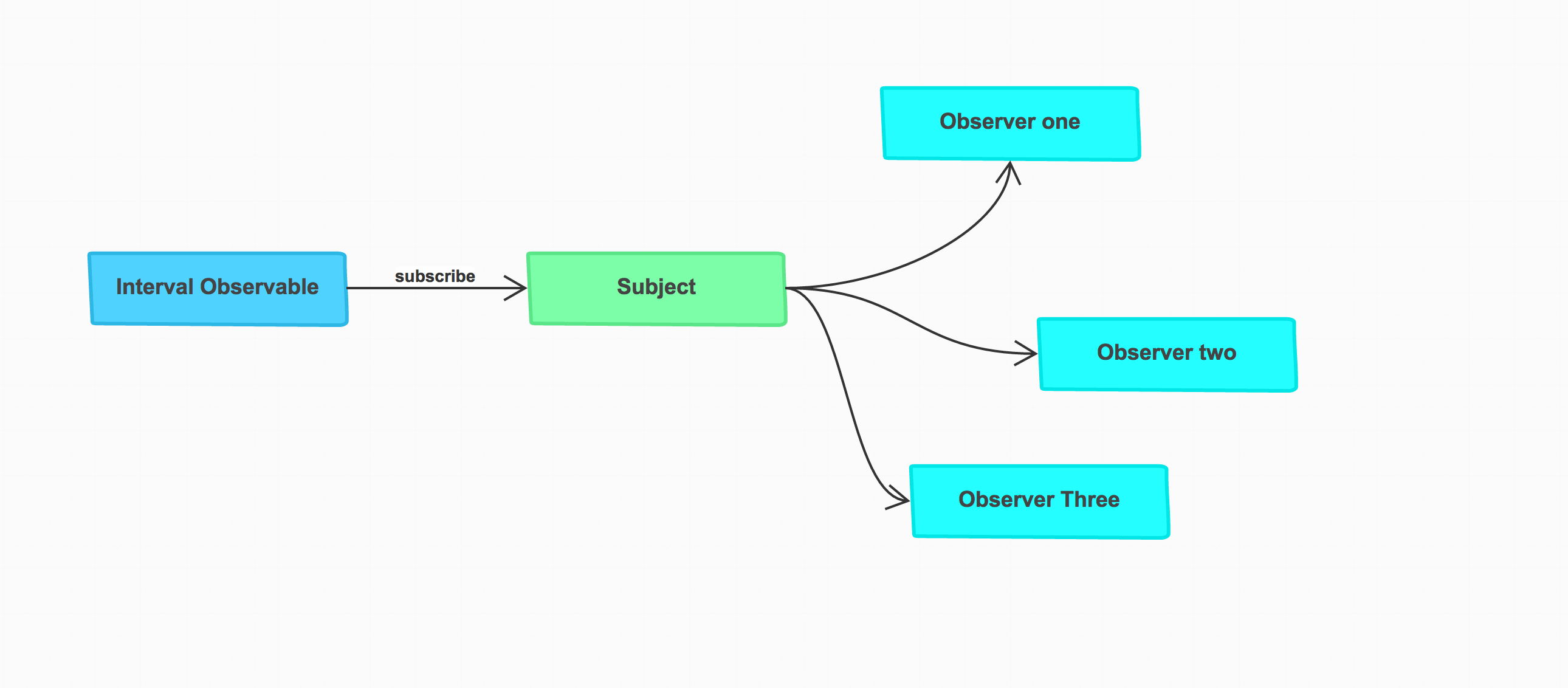
Lo que vamos hacer es especificar que es lo que va a ocurrir cuando se produzcan cualquiera de estos eventos

# Patrón Observables

El patrón observable no es más que un modo de implementación de la programación reactiva, que básicamente pone en funcionamiento diversos actores para producir los efectos deseados: reaccionar ante un flujo de datos en función de los distintos eventos producidos. Mejor dicho: producir dichos eventos y consumirlos de diversos modos.

## Principales componentes del patrón observables

* **Observable:** es aquello que queremos observar, **que será implementado mediante una colección de eventos o valores futuros**. Un observable puede ser creado a partir de eventos del usuario derivado del uso de formularios, una llamada HTTP, un almacén de datos, etc. Mediante el observable nos podemos suscribir a eventos que nos permiten hacer cosas cuando cambia lo que se está observando.
* **Observer:** es el actor que se dedica a observar. Básicamente se implementa mediante una colección de funciones callback que nos permiten escuchar los eventos o valores emitidos por un observable. Las callbacks permitirán especificar código a ejecutar frente a un dato en el flujo, un error o el final del flujo.
* **Subject:** es el emisor de eventos, es capaz de crear el flujo de eventos cuando el observable sufre cambios. Esos eventos son los que consumirán los observers





**Notas:**

* Un observable no se va ejecutar nunca mientras no tenga al menos un *Observer* suscripto. Si queremos que se emitan los datos y que se ejecute los operadores de filtrado necesitamos hacer una suscripción
* Las suscripciones también sirven para cancelar el flujo de datos de ejecución, dado que si nos desuscribimos se corta el flujo de datos.

### Observables

#### Método subscribe()

Los objetos del tipo *observable* tienen un método llamado *subscribre* que toma como parámetro un *observador* y nos devuelve una *subscripción*.

El *observador* se subscribe al *observable* y cada vez que el *observable* emite un valor, el *observador* es notificado

La *subscripción* que nos devuelve el método posee un método llamado *unsubscribe()* que nos permite terminar la relación entre el *observable* y el *observador*

Llevado a un ejemplo de la vida Nisman: cuando sacamos un número en un comercio para ser atendidos, **nos estamos *suscribiendo* a un *stream* de números de atención**. Cada vez que llaman a un número, **estamos pendientes de que si el número *emitido* es el nuestro**.   
Una vez que llega nuestro turno y somos atendidos, dejamos de estar pendientes de los números que vienen a continuación, es decir nos desuscribimos

## Que es RxJS

Rx es *Reactive Extensions* es una librería creada por Microsoft para implementar programación reactiva, es decir para crear aplicaciones que son capaces de utilizar el patrón observable para gestionar operaciones asíncronas.

RxJS es la implementación en JavaScript de Reactive Extensions.

Angular utiliza RxJS para implementar programación reactiva (podemos utilizar RxJS en diversos contactos, entre ellos en una aplicación Angular)

RxJS nos proporciona:

* El tipo *Observable* (llamado *core type*)
* Otros tipos satélites como el *Observer*, el *Subject*, etc.
* Operadores para trabajar con Observables (inspirados en los que se utilizan con Arrays), por ejemplo: *map, filter, reduce, every*, etc.

Reactive X (Rx) existe para muchos lenguajes y frameworks por ejemplo:

* Java: RxJava
* JavaScript: RxJS
* C#: Rx.Net
* Python: RxPY
* PHP: RxPHP
* Etc.

### Operadores RxJS

* Los operadores RxJS son funciones que pueden ser encadenadas. Se sitúan entre medio del *Observable* (productor de la información) y el *Observer* (consumidor de la misma) con el objetivo de filtrar, transformar, o combinar los valores del *Observable*.
* En la mayoría de los casos los operadores reciben un *Observable* y devuelven un *Observable* (es decir que de cierta manera de los operadores actúan como un suscriptor del *Observable*). No modifican el *Observable* original, sino que devuelve un objeto nuevo.

### Encadenar operadores (pipe ())

Mediante el método *.pipe()* de los *Observables* podemos encadenar operadores.

RxJS tiene muchos operadores. Podemos clasificarlos en distintas categorías, por ejemplo: operadores de filtrado, transformación, agregación, etc.

#### Operador Map

Aplica una función a cada uno de los elementos emitidos en el observable

