# Promises VS Observable:

**Both Promises and Observables help us dealing with asynchronous operations. They can call certain callbacks when these asynchronous operations are done.**

|  |  |
| --- | --- |
| **Promise** | **Observable** |
| promise emits a single value | Emits multiple values over a period of time |
| Ex: const numberPromise = new Promise((resolve) => {  resolve(5);  resolve(10);  });  numberPromise.then(value => console.log(value));  // still prints only 5 | const numberObservable = new Observable((observer) => {  observer.next(5);  observer.next(10);  });  numberObservable.subscribe(value => console.log(value));  // prints 5 and 10 |
| A promise is Not Lazy | Observable is Lazy. The "Observable" is slow. It isn't called until we are subscribed to it. |
| A Promise cannot be cancelled | An Observable can be cancelled by using the unsubscribe() method |
|  | An addition Observable provides many powerful operators like map, foreach, filter, reduce, retry, retryWhen etc. |
| having one pipeline | Multiple PipeLines |
| Cannot be **retried**(Promises should have access to the original function that returned the promise to have a retry capability, which is a bad practice) | Helps you run functions asynchronously, and use their return values in a continuous sequence (**multiple times**) when executed. |
| Provides Operators: filter, map , reduce, retry | No operators |

# Why Subject?

 An RxJS Subject is a special type of Observable that allows **values to be multicasted to many Observers**. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast.

## Internal Implementation:

* Internally to the Subject, subscribe does not invoke a new execution that delivers values. It simply registers the given Observer in a list of Observers, similarly to how addListener usually works in other libraries and languages.
* Every Subject is an Observer. It is an object with the methods next(v), error(e), and complete(). To feed a new value to the Subject, just call next(theValue), and it will be multicasted to the Observers registered to listen to the Subject.

### Example

**:******

* Used to multicast for multiple Observer listening, So all subscribed subject will be listened to it.
* This is called a multicasting

## Behavior Subject:

BehaviorSubjects are useful for representing "values over time". For instance, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.

## Replay Subject:

A ReplaySubject records multiple values from the Observable execution and replays them to new subscribers.

Auth Servic.ts

Export function **loadConfig** (authSVC: AuthService) {

return ()=> authSvc.doLogin().catch()

}

In Modules:

Providers: [ Gaurds, Services,{}, {provide: APP\_INITIALIZER, useFactory: **loadConfig,** deps: [AuthService], multi: true}, {}]

doLogin()=>Promise(resolve, reject)

# Promise Example

Promises (like callbacks) allow us to wait on certain code to finish execution prior to running the next bit of code.

Our Promise can have one of three states:

* Pending — Asynchronous operation has not completed yet
* Fulfilled — Operation has completed and the Promise has a value
* Rejected — Operation has completed with an error or failed.

*function delay(t){*

*return new Promise(function(resolve){*

*return setTimeout(resolve, t)*

*});*

*}*

*function logHi(){*

*console.log('hi');*

*}*

*delay(2000).then(logHi);*

# Observer Example

Will See later

# Async/Await

1. The newest way to write asynchronous code in JavaScript.
2. It is non blocking (just like promises and callbacks).
3. Async/Await was created to simplify the process of working with and writing chained promises.
4. Async functions return a Promise. If the function throws an error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

## Await Example:

// Normal Function

*function add(x,y){*

*return x + y;*

*}*

*// Async Function*

***async*** *function add(x,y){*

*return x + y;*

*}*

## Full Example wait and watch

<https://codeburst.io/javascript-es-2017-learn-async-await-by-example-48acc58bad65>