1. Asynchronous calls without promises:

   a. Create a separate independent folder javascript and execute below command from within the javascript folder:

      npm init -y

   b. Create a file callback.js and complete the steps below

   c. Assume below function makes a REST API call to get the login status of the user:

```
function loginUser(email) {
// Simulate REST API call returns result after 1.5 secs
  setTimeout(() => {
    console.log('Now we have the data');
    if(email === 'abc')
        return {email}
     return {'status':'email not found'}
  }, 1500);
}
console.log(loginUser('abc'))
```

   d. To esecute open terminal from within the javascript folder and execute below command:

      node callback.js

What will be the output ?
log prints undefined since the return will be executed after 1.5 seconds

   e. To handle this situation javascript provides with callback as follows:

```
function loginUser(email, callback) {

  setTimeout(() => {

    console.log('Now we have the data');

    if(email === 'abc')

        callback({email})

    else

        callback({'status':'email not found'})

  }, 1500);

}

loginUser('abc', (msg)=> console.log(msg))
```

   f. Soon this may result in callback hell as follows:

```
console.log('Start');
//COMMENT::Three callback function
function loginUser(email, callback) {
  setTimeout(() => {
    console.log('Now we have the data');
    callback({ email });
  }, 1500);
}

function getUserVidoes(email, callback) {
  setTimeout(() => {
    callback(['HTML', 'CSS', 'JAVASCRIPT']);
  }, 1200);
}

function videosDetails(video, callback) {
  setTimeout(() => {
    callback(`title of the video ${video}`);
  }, 1000);
}
//COMMENT::Callback function execution.
//here we get nested function of 3 callback function which make code not clean.
//this problem is call callback hell.
const user = loginUser('shirshakkandel@gmail.com',  function (user) {
  console.log(user);
  getUserVidoes(user.email, videos => {
    console.log(videos);
    videosDetails(videos[0], title => {
      console.log(title);
    });
  });
});
// console.log(user);
//undefined as user come after 1.5 second
console.log('Finished');
```

To handle above situation, javascipt provides with promises

2. Handle callback with promises:

   Characteristics of Promises:

   a. Single Value: Promises represent a single value that will be resolved or rejected.
   b. Immutable State: Once a Promise is resolved or rejected, its state cannot be changed.
   c. Error Handling: Promises have built-in error handling through .catch() or try...catch blocks.

   Create a file promises.js and add below code. To execute:
   node promises.js

```
function loginUser(email) {
    return new Promise((resolve, reject)=>{
        setTimeout(() => {
            console.log('Now we have the data');
            if(email === 'abc')
                resolve({ email});
            reject({status:'failure'})
          }, 1500);
    })
}
function getUserVidoes(email) {
    return new Promise((resolve, reject)=>{
    setTimeout(() => {
        resolve(['HTML', 'CSS', 'JAVASCRIPT']);
    }, 1200);
})
}
function videosDetails(video) {
    return new Promise((resolve, reject)=>{
  setTimeout(() => {
    resolve(`title of the video ${video}`);
  }, 1000);
})
}
loginUser('abc')
.then(resp =>
    getUserVidoes(resp.email)
    .then(videos =>
        videosDetails(videos[0])
        .then(resp => console.log(resp))
    )
)
loginUser('abc1')
.then(resp =>
    getUserVidoes(resp.email)
    .then(videos =>
        videosDetails(videos[0])
        .then(resp => console.log(resp))
    )
)
// to handle for reject use catch
.catch(err => console.log(err))
```

The above code can also get messy with all the chaining of promises.

3. Handle promises with async and await

```
const details = async()=> {
 const user =   await loginUser('abc','123');
 console.log(user)
 const videos = await getUserVidoes(user.email);
 console.log(videos)
 let detail = await videosDetails(videos[0])
 return detail
}
details().then(data => console.log(data))
```

Observables:

Promises deal with one asynchronous event at a time, while observables handle a sequence of asynchronous events over a period of time.
With observables, you can observe a stream of events or data values and react to each one as they occur. They're useful when you need to handle ongoing processes or data streams, like user interactions, network requests, or data updates, and you want to respond to each event as it happens.

Basic requirements for an object to be an observable in JavaScript

a. **Observable Object:** An observable object is like a storyteller who narrates a story (data or events) over time. It needs to have a way to start telling the story and a way for others to listen to the story as it unfolds.

b. **Start Telling the Story (Subscribe):** The storyteller (observable object) should have a method that allows others (subscribers) to start listening to the story. This method should provide a way for subscribers to react to each part of the story as it's told.

c. **Tell the Story (Emit Data/Events):** As the storyteller (observable object) narrates the story, it needs a way to emit different parts of the story (data or events) to its listeners (subscribers). Each emitted part of the story triggers a reaction from the listeners.

d. **Ability to Stop Listening (Unsubscribe):** Listeners (subscribers) should have the option to stop listening to the story if they're no longer interested. This is important for memory management and performance.

1. Steps to implement a simple Observable:

   Create a file observable.js within javascript folder and add below code.
   To execute :
   node observable.js

```
// Define an Observable class
class Observable {
  constructor(subscribe) {
    console.log(subscribe)
   this._subscribe = subscribe;
  }
 // Subscribe to the observable
  subscribe(observer) {
   return this._subscribe(observer);
  }
  // Create an observable from an array
  static fromArray(array) {
   return new Observable(observer => {
    array.forEach(item => observer.next(item));
    observer.complete();
   });
  }
 }
 // Example usage
 const observable = new Observable(observer => {
  console.log(observer)
  // Emit some values
  observer.next(1);
  observer.next(2);
  observer.next(3);
  // Emit an error
  // observer.error('Something went wrong');
  // Complete the observable
  observer.complete();
  // Cleanup function (optional)
  return {unsubscribe : () => {
   console.log('Observer unsubscribed');
  }}
 });
 // Subscribe to the observable
 const subscription = observable.subscribe({
  next: value => console.log('Received:', value),
  error: err => console.error('Error occurred:', err),
  complete: () => console.log('Observable completed')
 });
 console.log(subscription)
 // Unsubscribe (cleanup)
 subscription.unsubscribe();
```

2. To use 3rd party library rxjs execute below command from within the javascript folder:

   npm i rxjs

   Purpose: RxJS is a library for reactive programming using Observables. It provides a powerful set of tools for working with asynchronous data streams.
   Key Concepts:

   a. Observable: Represents a stream of data or events that can be observed over time. It can emit multiple values asynchronously.
   b. Observer: Consumes the values emitted by the Observable.
   c. Operators: Functions for manipulating the data emitted by Observables.

3. Create a file rxjsdemo.js and add below code:

```
// Example using RxJS
import { Observable } from 'rxjs';
// Create an Observable
const observable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.next('World');
  subscriber.complete();
});
// Subscribe to the Observable
observable.subscribe({
  next: value => console.log(value),
  complete: () => console.log('Observable completed')
});
```

4. Another example with rxjs

```
import { interval } from 'rxjs';
import { take } from 'rxjs/operators';

// Create an Observable that emits a value every second
const observable1 = interval(1000).pipe(
    take(5) // Take only 5 values
  );
  // Subscribe to the Observable
  observable1.subscribe(value => console.log(value));
```

   In this example, interval(1000) creates an Observable that emits a value every second, and take(5) limits it to emitting only 5 values. The subscribe method is used to listen for these emitted values.

5. Commonly used rxjs operators:

   https://www.learnrxjs.io/learn-rxjs/operators

   a. Pipe: the pipeable operator is a function that takes observables as an input and returns another observable. The previous observable stays unmodified.

   b. map: This operator is used to transform the data emitted by an observable. For example, if you have an observable that emits a stream of numbers, you can use the map operator to double the value of each number:
```
import { from} from 'rxjs';
import { map } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
const doubled = source.pipe(map(value => value * 2));
```

```
doubled.subscribe(result => console.log(result));
```

c. Filter Operator: Selective Data Emission The filter operator allows you to emit values based on a specified condition. In this example, we filter out values less than 3:

```
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5]);
const filtered = source.pipe(filter(value => value >= 3));

filtered.subscribe(result => console.log(result));
// Output: 3, 4, 5
```

d. Tap: One of the superpowers of tap is its utility in debugging. When things aren't going as planned with your observable, instead of tearing apart your chain or inserting numerous logs, simply sprinkle in some tap operators. It's like adding checkpoints in a video game, helping you swiftly pinpoint issues without disrupting the main flow.

However, a word of caution: remember that
Tap is solely for side effects. If you find yourself tempted to modify data within a tap, it's generally best to resist. That's not its purpose, and you're better off with map or other transformational operators in these cases.

```
 import { of } from 'rxjs';
 import { tap, map } from 'rxjs/operators';

 const source = of(1, 2, 3, 4, 5);
 // transparently log values from source with 'tap'
 const example = source.pipe(
   tap(val => console.log(`BEFORE MAP: ${val}`)),
   map(val => val + 10),
   tap(val => console.log(`AFTER MAP: ${val}`))
 );

 //'tap' does not transform values
 //output: 11...12...13...14...15
 const subscribe = example.subscribe(val => console.log(val));
```

e. Concat Operator: Concatenating Observables The concat operator concatenates multiple observables, sequentially emitting values. Here's an example:

```
import { concat, of } from 'rxjs';

const source1 = of(1, 2);
const source2 = of(3, 4);
const concatenated = concat(source1, source2);

concatenated.subscribe(result => console.log(result));
// Output: 1, 2, 3, 4
```

f. Merge Operator: Merging Observables The merge operator combines multiple observables into a single observable, emitting values concurrently. Consider this example:

```
import { merge, interval } from 'rxjs';
import { take } from 'rxjs/operators';

const source1 = interval(1000).pipe(take(3));
const source2 = interval(500).pipe(take(6));
const merged = merge(source1, source2);
```

```
merged.subscribe(result => console.log(result));
// Output: 0, 0, 1, 1, 2, 2, 3, 4, 5
```

g. DebounceTime Operator: Delayed Emission The debounceTime operator introduces a delay before emitting the latest value. Useful for handling scenarios like user input. Example:

```
import { fromEvent } from 'rxjs';
import { debounceTime, map } from 'rxjs/operators';

const input = document.getElementById('input');
const keyup = fromEvent(input, 'keyup').pipe(
  map((event: any) => event.target.value),
  debounceTime(300)
);

keyup.subscribe(result => console.log(result));
// Emits input after 300ms of inactivity
```

h. Reduce Operator: Accumulating Observable Values The reduce operator accumulates values over time, providing a single result. A classic use case is summing:

```
import { of } from 'rxjs';
import { reduce } from 'rxjs/operators';

const numbers = of(1, 2, 3, 4, 5);
const sum = numbers.pipe(reduce((acc, value) => acc + value, 0));

sum.subscribe(result => console.log(result));
// Output: 15
```

i. Retry Operator: Retrying Observable Execution The retry operator allows retrying a failed observable. It's handy for scenarios where transient errors might occur:

```
import { of } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { retry } from 'rxjs/operators';

const request = ajax('https://api.example.com/data');
const retryRequest = request.pipe(retry(3));

retryRequest.subscribe(
  result => console.log(result),
  error => console.error('Retried 3 times, but still failed.')
);
```

j. ConcatMap Operator: Flattening Observables Sequentially concatMap transforms each item emitted by an observable into another observable and flattens them sequentially:

```
import { fromEvent, interval } from 'rxjs';
import { concatMap, take } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(concatMap(() => interval(1000).pipe(take(3))));

result.subscribe(x => console.log(x));
// Output: 0, 1, 2, 0, 1, 2, 0, 1, 2, ...
```

k. SwitchMap Operator: Managing Inner Observables switchMap is used to project each source value to an observable, which is merged in the output observable. If a new value arrives, it switches to a new inner observable:

```
import { fromEvent, interval } from 'rxjs';
import { switchMap, take } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
const result = clicks.pipe(switchMap(() => interval(1000).pipe(take(3))));

result.subscribe(x => console.log(x));
// Output: 0, 1, 2 (for each click), switching to a new inner observable.
```

l. Partition Operator: Splitting Observables partition is used to split one observable into two based on a predicate:

```
import { from } from 'rxjs';
import { partition } from 'rxjs/operators';

const source = from([1, 2, 3, 4, 5, 6]);
const [evens, odds] = source.pipe(partition(val => val % 2 === 0));

evens.subscribe(val => console.log(`Even: ${val}`));
odds.subscribe(val => console.log(`Odd: ${val}`));
// Output: Even: 2, Even: 4, Even: 6, Odd: 1, Odd: 3, Odd: 5
```

m. CombineLatest Operator: Combining Observables combineLatest combines the latest values from multiple observables and emits a new value whenever any input observable emits:

```
import { combineLatest, timer } from 'rxjs';

const first = timer(1000, 2000);
const second = timer(2000, 2000);

const combined = combineLatest([first, second]);
combined.subscribe(value => console.log(value));
// Output: [0, 0], [1, 0], [1, 1], [2, 1], [2, 2], ...
```

n. Skip Operator: Skipping Initial Values skip ignores the first N values emitted by the source observable:

```
import { interval } from 'rxjs';
import { skip } from 'rxjs/operators';

const source = interval(1000);
const example = source.pipe(skip(3));

example.subscribe(value => console.log(value));
// Output: 3, 4, 5, ...
```

o. TakeUntil Operator: Completing Based on Another Observable takeUntil completes the source observable when a second observable emits:

```
import { interval, timer } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

const source = interval(1000);
const timer$ = timer(5000);

const example = source.pipe(takeUntil(timer$));
```

```
example.subscribe(value => console.log(value));
// Output: 0, 1, 2, 3, 4 (until the timer emits at 5 seconds).
```

p. CatchError Operator: Handling Errors catchError intercepts an error in the observable and replaces it with another observable or a default value:

```
import { of } from 'rxjs';
import { catchError } from 'rxjs/operators';

const source = of('1', '2', '3', '4', '5');

const example = source.pipe(
  map(value => parseInt(value)),
  catchError(error => of('Error occurred!'))
);

example.subscribe(
  value => console.log(value),
  error => console.error(error)
);
// Output: 1, 2, 3, 4, 5 (converted to integers)
```

Observables	**Promises**
Emit multiple values over a period of time.	Emit a single value at a time.
Are lazy: they're not executed until we subscribe to them using the subscribe() method.	Are not lazy: execute immediately after creation.
Have subscriptions that are cancellable using the unsubscribe() method, which stops the listener from receiving further values.	Are not cancellable.
Provide the map for forEach, filter, reduce, retry, and retryWhen operators.	Don't provide any operations.
Deliver errors to the subscribers.	Push errors to the child promises

4.

https://medium.com/@avicsebooks/observable-in-javascript-0e64a6d93154