*) Asyne javascript

1) Callback functions : fn. passed into another fn. as a parameter then invoked by the other fn.

```
function callback() {
    console.log("coming from callback");
}

function higherorder(fn) {
    console.log("p1");
    fn();
    console.log("p2");
}
```

higher order fn. → a fn. that accepts another fn. as parameter

*) callbacks used for: a) advanced array methods.
b) browser events.
c) AJAX requests.
d) React development

```
function sendMessage(msg, callback) {
    return callback(msg);
}

sendMessage("Msg. for alert", alert);
```

*) common practice is to operate with anonymous fn. in the arg. of the fn.

```
greet("Tim", function(name) {
    return name.toUpperCase();
});
```

*) okay to specify more arguments, than what the callback might actually operate, on, but for them to be used, they need to available in the defn.

↓

if they are used in defn, but not send through the high order fn., then the variable operates through undefined.

↓

apt checks can be used for this case.

2) forEach → forEach (arr, function) takes that as argument.

↓

callback. ← operative fn. on array element.

↓

```
forEach ( arr, function (number) {

    console.log (number*2);

});
```

↓

```
function forEach (array, callback) {

}
```

↓

```
function callback (curElement,
    urrentIndex, array) {

}
```

↓

```
arr.forEach ((str, index, arr) => {

    if (index === arr.length - 1)
    {
        output += (str + "!!!");
    }
```

(i) callback
(ii) higher order fn.
(iii) asynchronous fn.

```
else
{
    output += str;
}
```

});

class method.

first index to satisfy this condition

3) findIndex → arr.findIndex((element) => {

```
if (element % 2 === 0) return true;
else return false;
```

important else it will return undefined.

});

↓

```
function findIndex(arr, callback){
    for (let i=0; i<arr.length; i++)
    {
        if (callback(arr[i], i, arr)) { return i; }
    }
    return -1;
}
```

if element not found.

4) setTimeout & setInterval → facilitate asynchronous code

↓

setTimeout(callback, delay);

delay in ms.

invokes callback after delay.

setInterval(callback, ~~delay~~ repeat); → function that continually invokes a callback after x ms.

returned by setTimeout.

```
var timerId = setTimeout(function(){
    console.log("30secs");
}, 30000);

setTimeout(function(){
    console.log("cancelling the first
        setTimeOut", timerId);
    clearTimeout(timerId);
}, 2000);
```

clearTimeout(timerId);
↓
to not execute the delayed function which could be conditional.

*) setInterval runs till the program lifetime
↓
clearInterval(intervalId);
can be used to stop setInterval.
↓
function id is returned by the invocation, which can be stored in variable & then used accordingly.

*) event loop and the queue. (callback queue)
[
functionality in the javascript runtime that checks the queue when stack is empty.
↓
if stack is empty, front of queue is placed in the stack.

Javascript
↓
single threaded language

```
function square(n) {
    return n*n;
}

setTimeout(function() {
    console.log("callback is placed on
                 the queue");
}, 0);

console.log(square(2));
```

→ puts the callback on the queue

→ does not run immediately.

Used to defer some operation till the callstack is clear

queue: ~~function()~~

~~function()~~
~~console.log()~~;
~~square(2)~~
st: ~~setTimeout~~
~~main~~

↓

once main is exhausted, then event loop runs, and the fn. on queue is picked.

single callstack.

↑

JS is a single threaded language → code that is running cannot be interrupted by something else going in the program.

eager: starts doing whatever task is assigned. (constructor invoked)

↑

*) promise: object that represents a task that will be completed in the future

the async task decides which to call

↓

i) creating a promise:

```
var p1 = new Promise(function(resolve, reject) {
    resolve([1, 2, 3, 4]);
});

p1.then(function(arr) {
    console.log('p1', arr);
});
```

resolve ↖

- .catch() is used to handle reject.

↓

```
var p1 = new Promise(function(resolve, reject){
        reject("Error");
});
p1.then(function(data){
        console.log(data);
}).catch(function(data){
        console.log(data);
});
```

then → resolution.

catch → rejection

catch will be invoked when reject parameter is called

↓

Opt conditionals can be used in the in the promise object to call the req. callback.

*) Wrapping setTimeout with Promise

```
var promise = new Promise(function(resolve, reject){
        setTimeout(function(){
                var randInt = Math.floor(Math.random()*10);
                resolve(randInt);
        }, 4000);
});

promise.then(function(data){
        console.log("resolve");
});
```

*) promises are eager, meaning that a promise will start doing whatever task you give it as soon as the constructor is invoked.

V8 : JS runtime.
↓
chrome's engine.

*) handling asynchronous behaviour

*) promise chaining

*) .catch works like the try-catch statement, and hence should only be req. once.

nested async callbacks → i) code is hard to read.
ii) logic is difficult to reason around.
iii) modularity hit.

promise chaining : multiple .then operatives to a promise.

If an internal .then() returns a promise, then the next chained resolve, then, is called, and operated with.

↓

asynchronous tasks synchronously.

↓

values returned in previous resolve is passed as parameter to the next resolve.

↓

```
var promise = new Promise (function(resolve, reject) {
    resolve(5);
});

promise.then (function(data){
    return data * 2;
}).then (function(data) {
    return data + 20;
}).then (function(data) {
    console.log(data)
});
```

*) to chain catch, we need to re-raise.
↓
or throw.

*) error print may return a call stack trace.

(state of the stack,
when the error occurred)

*) async. calls or runs on returning are pushed to the callback queue (so that it doesn't just pop out of nowhere). once the callstack gets empty. then callback queue is checked for any operation.

↓

this is how web api's operate with a JS runtime.

↓

Now even if the response never arrives, or code executes till the call stack is not empty.

*) promises in practice: useful to understand how promises work but in practice, we will often use promises that are returned to us.