Async / defer

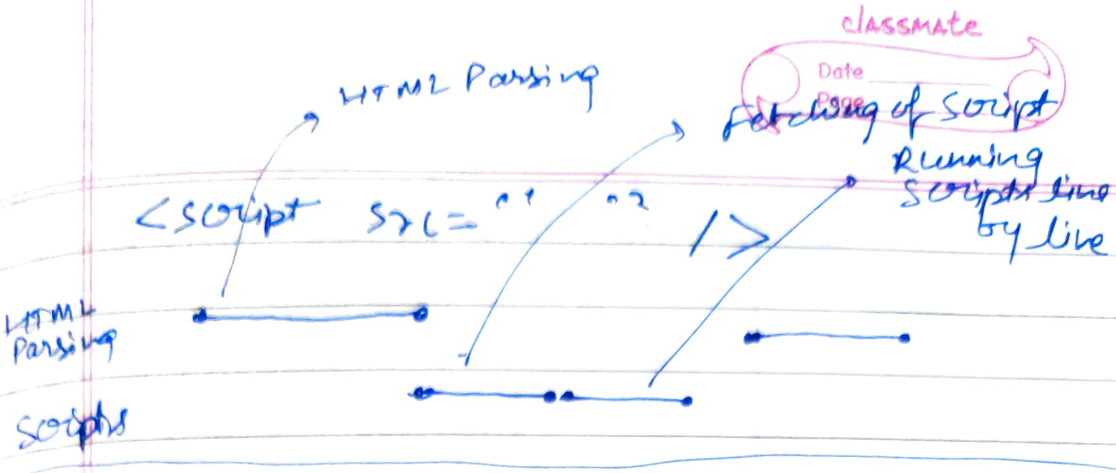
Async and defer are boolean attributes which are used along with script tags to load the external scripts efficiently in our webpage.

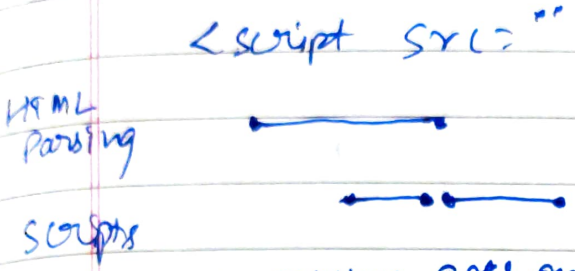
3 Scenerios

1- Without using any attributes
2- using async
3- using defer.

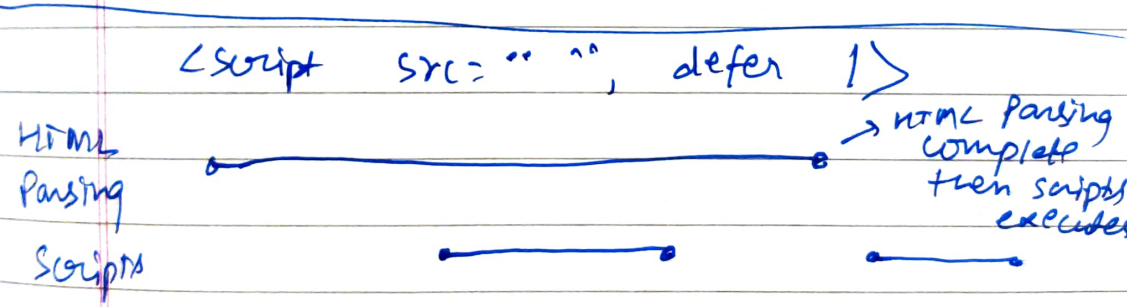
When we load our webpage 2 major things happens

(i) HTML parsing
2- Loading of scripts
   - fetching of script in network
   - actually executing scripts line by line.





HTML parsing goes on and scripts are fetched in parallel. As soon as scripts are fetched and available in the Browser, HTML parsing stops, scripts are executed then HTML parsing continues



* Async attributes does not guarantee the order of execution of scripts but defer does.
If your scripts are dependent on each other, never use async. You can use defer.

* Suppose you have to load some external scripts like Google analytics which are quite modular and independent of normal

Local storage session storage cookies

	Local storage	session storage	cookie
Storage capacity	5-10 mb	5-10 mb	4 kb
Auto Expiry	No	Yes	Yes
Server side accessibility	NO	NO	yes
Data transfer HTTP Request	NO	NO	Yes
Data persistance	Till manually deleted	Till browser tab is closed	As per Expir TTL se

It is always best to store JWTs in http only cookies. Http only cookies are special cookies that cannot be accessed by client side Javascript. This way they are secure against XSS attacks.

CORS - Cross origin Resource Sharing

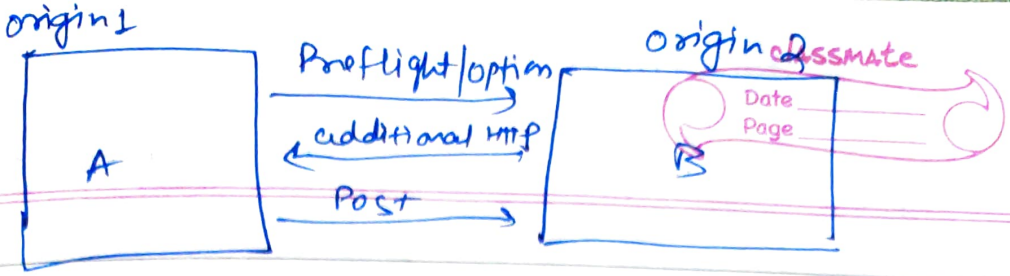
CORS is the mechanism which uses additional http headers to tell the browser whether a specific web app can share resource with another web app when both webapp have different origin (If origin is same, obviously they can share resource easily).

If they dont have same origin, they need to follow CORS Mechanism

Without CORS

https://akshaysaini.in

- → google.com/api/data ⊗ different domain
- → api.akshaysaini.in ⊗ Sub domain
- → akshaysaini.in:5050 ⊗ Port is different
- → http://akshaysaini.in ⊗ protocal is different.



CORS preflight mech - A preflight option call is made before the actual api call. Suppose A want to post data to B. So first a preflight call will be made. then CORS mech uses additional http headers to verify this request. Browser makes Preflight/ options call. server verify whether this call is valid or not. If valid, B will set some additional http header which will let the client know this is safe and then actual call is made

→ Preflight call is not required. They are not made for all requests.

→ most common headers set by servers:

Accept - Control - Allow - origin: *

→ * → any domain can access this

→ : https://akshaysaini.in then only call from this url will be allowed.

The Net Ninjas

Object oriented JS

(1)

"Everything in JS is an Object"

var names = ['ryu', 'crystal', 'mario']

names array is an object with properties and methods.

Ex- → length (property)

→ sort (method)

Another Example of builtin object - Window.

Primitives types are Not object -

1→ Number
2→ String
3→ Undefined
4→ Boolean
5→ Null

var name = "mario"

It's not object but can actually behave like object. JS wraps them in an object in the background.

We can use nam.length.;

JS wraps this string primitive types inside an object temporaly.

var name = "mario"

Enter → Just give mario

var name2 = new String ('ryu')

Enter → 0 : "r"
1 : "y"
2 : "u"
length: 3
Prototype → contain all methods and properties related to String.

typeOf (name) → String
typeOf (name2) → Object.

(2) Object literals

```
var userOne = {
email: 'ryu@ninjas.com',
name: 'Ryu',
login() {
console.log(this.email, 'has logged in');
},
logout() {
console.log(this.email, 'has logged out');
}
};
```

(1)

```
var userOne = {
email: 'ryu@ninjas.com',
name: 'Ryu',
login: function() {
console.log(this.email, 'has logged in')
},
logout: function() {
console.log(this.email, 'has logged out');
}
};
```

Encapsulation - we are encapsulating all userOne properties and methods inside the object.

(3) Updating properties

userOne.name = 'Ryu'

if you wanna update -

userone.name = "Yashi".

I same.

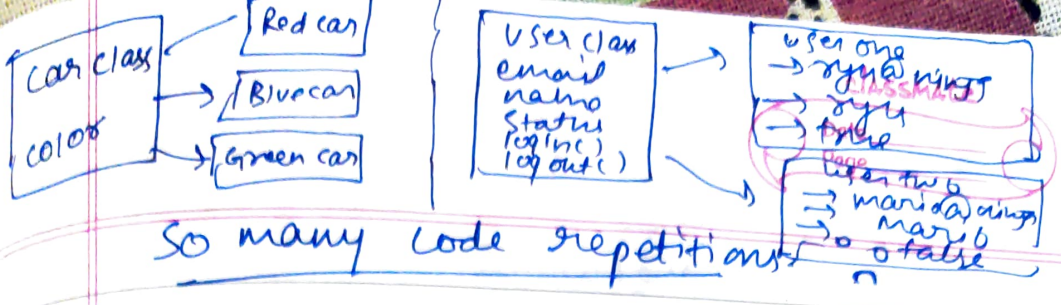
userone['name'] = 'Yashi'

4
Classes

What we do if we have multiple objects ?

```
var userOne = {
email: 'ryu@ninjas.com',
name: 'Ryu',
login() {
console.log(this.email, 'has logged in');
},
logout() {
console.log(this.email, 'has logged out');
}
};

var userTwo = {
email: 'yoshi@ninjas.com',
name: 'yoshi',
login() {
console.log(this.email, 'has logged in');
}
logout(){
console.log(this.email, 'has logged out');
}
```



So many code repetitions

(5) Class Constructors

```
Class User {
constructor (email, name) {
this.email = email;
this.name = name;
}
}
```

```
var userOne = new User ('ryu@ninja', 'Ryu');
var userTwo = new User ('Yoshi@ninja', 'Yoshi');
```

The 'this' keyword —

(i) creates a new empty object {}
(ii) sets the value of 'this' to be new empty object
(iii) calls the constructor method.

6

Class Methods

```
Class User {
constructor (email, name) {
 this.email = email;
 this.name = name;
}
login () {
console.log (this.email, 'just logged in');
 }
logout () {
console.log (this.email, 'just logged out')
}
}
Var userone = new user ('xyz@xyz', 'Ryu');
 userone.login();
```

7

Method chaining

what if you do - userOne.login().logout()
out -- cannot read property logout of undefined
why-? ∴ When we don't return any thing from functions, JS automatically return undefined. login() will return undefined and we are calling logout() after that, that's why.

Soln?

→ return this from every method

```
login() {
console.log("login");
return this;
}

logout() {
console.log("---");
return this;
}

updateScore() {
this.score++;
console.log("---");
return this;
}

userOne.login().updateScore().updateScore()
       .logout()
```

Works fine

8

Class inheritance

```
Class User {
constructor (email, name) {
this.email = email
 this.name = name;
this.score = 0;
 }
 login() {
console.log("logged in");
 return this;
 }

logout() {
console.log(this.email, "logged out");
    return this;
}

 updateScore() {
  this.score++;
console.log(this.email, "score is", this.score);
 return this;
 }
 }
```

```
class Admin extends User {
  deleteUser(user) {
    users = users.filter(u => {
      return u.email != user.email;
    })
  }
}

var userOne = new User('ryu@ninja', 'Ryu');
var userTwo = new User('yoshi@ninja', 'Yoshi');
var admin = new Admin('shaun@ninja', 'shaun');

var users = [userOne, userTwo, admin]

admin.deleteUser(userTwo);

console.log(users) // [User, Admin]
```

Admin →

email: "shaun@ninja"
name: "shaun"
score: 0
--proto--
→ deleteUser
--proto--
→ login()
logout()
updateScore()

(9)

constructors (under the hood)

ES6 classes (which we learnt previously) are just syntaxital sugar on top of JS prototype model.

When we create class using new ES6 class keyword, JS still works the same way as it did the way to create these classes using prototype model.

(CONSTRUCTOR FUNCTION)

```
function User(email, name){
  this.email = email;
  this.name = name;
  this.online = false;
  this.login = function(){
    console.log(this.email, 'has logged in');
  }
```

```
var userOne = new User('ryu@ninja', 'Ryu');
var userTwo = new User('yoshi@ninja', 'Yoshi');
```

this is the way we create classes without using class keyword.

But when we want to attach methods in emulated class like this, we don't put those methods inside constructor functions, instead we use prototype property

we can call prototype only on constructor functions (user.prototype) not the instances of the object (userOne.prototype) ✗

(10)

Prototype ; instances have --proto-- which points to prototype.

```
function User (email, name){
this.email= email;
this.name = name;
this.online = false;
}
User.prototype.login = function(){
  this.online = true;
console.log(this.email, 'has logged in');
}
User.prototype.logout = function(){
 this.online = false;
console.log(this.email, 'has logout');

var userOne = new User('ryu@ninja', 'Ryu')
```

userOne ↴

email : "ryu@ninja"
name : "Ryu"
online : false
--proto-- → login()
logout()

If we use class keyword, all the methods are added to prototype automatically. (lecture 8)

(11)

Prototype Inheritance

we will create an admin class with all properties and methods of user class along with some more admin related things (role & delete user)

```
function User(email, name){
this.email = email;
this.name = name;
this.online = false;
}
User.prototype.login = function(){
  this.online = true;
console.log(this.email, 'has logged in');
}

User.prototype.logout = function(){
this.online = false;
  console.log("...");
}
function Admin(...args){
  User.apply(this, args);
  this.role = "super admin";
}
var user1 = new User('ryu@ninja', 'Ryu');
var admin = new Admin('shaun@ninja', 'shaun');
console.log(admin) → email: "shaun@ninja"
                     name: "shaun"
                     online: false
                     role: "super admin"
                     __proto__:
```

inside --proto-- of admin, login & logout is not available. How to add?

```
Admin.prototype = Object.create(User.prototype);

Admin.prototype.deleteUser = function(){
    // code to delete user
};
```

Admin →
email
name
online
role
--proto-- : User
↳ deleteUser
↳ --proto--
↳ login()
↳ logout()

→ We can also use User.call().

→ 'this' refer to new object which we created. We need to past this context while calling User constructor function. Now In constructor function User,
this.email refer to email of newly created admin object.

(11)

Prototype Inheritance

We will create an admin class with all properties and methods of user class along with some more admin related things (role & deleteUser)

```
function User(email, name){
this.email = email;
this.name = name;
this.online = false;
}
User.prototype.login = function(){
this.online = true;
console.log(this.email, "has logged in");
}
User.prototype.logout = function(){
this.online = false;
console.log("...");
}
function Admin(...args){
User.apply(this, args);
this.role = "Super admin";
}
var user1 = new User("ryu@ninja", "Ryu");
var admin = new Admin("shaun@ninja", "Shaun");
console.log(admin) → email: "shaun@ninja"
                     name: "Shaun"
                     online: false
                     role: "Super admin"
                     --proto--:
```

inside --proto-- of admin, login & logout is not available. How to add?

```
Admin.prototype = Object.create(User.prototype);
Admin.prototype.deleteUser = function(){
  // code to delete user
};
```

Admin →
- email
- name
- online
- role
- --proto-- : User
  - deleteUser
  - --proto--
    - login()
    - logout()

we can also use User.call().

'this' refer to new object which we created. we need to pass this context while calling User constructor function. Now in constructor function User,

this.email refers to email of newly created admin object.

```
function User(email, name) {
  this.email = email;
  this.name = name;
  this.online = false;
}

User.prototype.login = function() {
  this.online = true;
  console.log("---");
}

User.prototype.logout = function() {
  this.online = false;
  console.log("---");
}

function Admin(...args) {
  User.apply(this, args)
  this.role = "Super admin";
}

Admin.prototype = Object.create(User.prototype)

Admin.prototype.deleteUser = function() {
// code for delete user;
}
```

```
var userOne = new User("raymond wingo", "Ray")
var admin = new Admin("Shaun Wingo", "Shaun")
```

Deepak Gupta Medium

Everything about JS Objects

Nearly everything in JS is an object other than SIX things -

1- null
2- undefined
3- strings
4- numbers
5- boolean
6- Symbols

Everything that is not a primitive value is an Object. That includes arrays, functions, constructors & objects themselves.

Ways to create objects

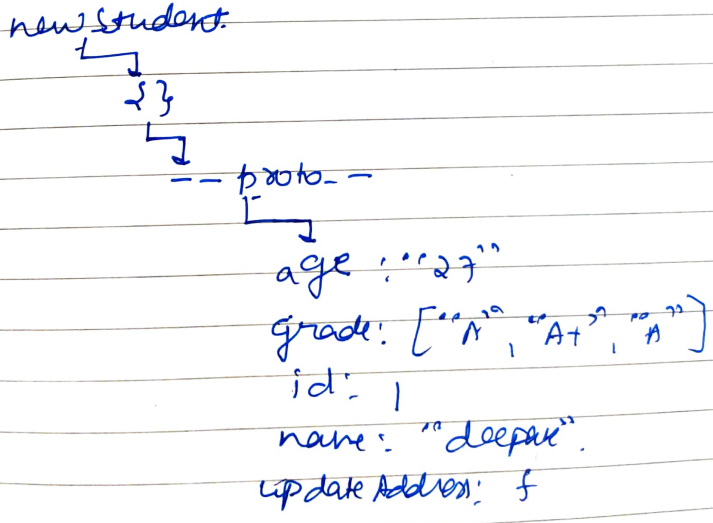
1- Object literal

```
Var student = {
  id: 1,
  name: "deepak",
  age : "27",
  updateAddress : () => {
    // logic to update address
  },
  grade: [ 'A', 'A+', 'A' ]
}
```

2) Object.create() - This method creates a new object with the specified prototype and properties of an old object.

var newStudent = Object.create(Student)

this create a new object ({}) with old object added in its prototype chain



we can add new properties and data to new Student object.

same as way 1 behind the scene

3→ Object Instance

```
const newObj = new Object();
newObj.name = 'Deepak';
newObj.location = 'Delhi';
```

If you have to create multiple objects of same kind, 3rd way will cause duplication so? 1

4→ Object constructor

```
function Vehicle(name, model) {
this.name = name;
this.model = model;
}
let car1 = new Vehicle('Fiesta', '2019');
let car2 = new Vehicle('Dc avanti', '2018');
```

5→ Object.assign() - this is another method to create a new object from other objects.

It copies the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

```
var obj = { a: 1 };
var copy = Object.assign({}, obj);
console.log(copy);
out = { a: 1 }
```

→ How to delete key from object?

```
delete a.city
```

→ How to iterate on object properties?

for in loop.

```
for (const key in a) {
  console.log(key, a[key]);
}
```

For in loop also iterate over the prototype chain and will return parent keys as well, so don't be suprised if you see more keys. TO avoid seeing more keys we can do a `hasOwnProperty` check to get the current object keys only.

Few methods of object

1- Object.keys() — Returns an array of string keys

```
const keys = Object.keys(a)
["name", "first name", "city", ""]
```

2-> Object.values() → Returns an array of values.

```
const keys = Object.values(a);
["deepak", "new york", "dope"]
```

3-> Object.entries() → Returns an array of [key, value] pairs.

```
const data = Object.entries(a)
[["", "dope"], ["name", "deepak"],
["first name", "deepak"]]
```

How to check property exist in an object?

1- Using hasOwnProperty

```
console.log(a.hasOwnProperty(1)) //true
```

this method returns a boolean indicating whether the object has the specified property as its own property, not the parent/ inheriting property.

2) Using in operator. - The in operator returns true if the specified property is in the specified object or its prototype chain i.e, inside in its parent.

```
console.log(1 in a) //return true
```

hasOwnProperty only checks for current object property whereas in operator check for current + parent properties

Asynchronous JavaScript
NetNinja youtube Lec-4

Callback

```
const getTodos = (callback) => {

const request = new XMLHttpRequest();

request.addEventListener('readystatechange', () => {

if(request.readyState === 4 && request.status === 200) {

callback(undefined, request.responseText);

} else if(request.readyState === 4) {

callback('could not fetch data', undefined);

}

});

request.open('GET', 'http://jsonplaceholder...');

request.send();

};
```

```
getTodos((err, data) => {
console.log('callback fired');
if(err){
 console.log(err);
}
else{
console.log(data);
 }
});
```

Callback hell

```
const getTodos = (resource, callback) => {
const request = new XMLHttpRequest();
request.addEventListener('readystatechange',
  () => {
if(request.readyState === 4 && request.status
                    === 200) {
const data = JSON.parse(request.responseText);
callback(undefined, data);
}
else if(request.readyState === 4) {
  callback('could not fetch data', undefined);
  }
});

request.open('GET', resource);
request.send();
};
```

```
getTodos('todos/luigi.json', (err, data) => {
  console.log(data);
getTodos('todos/mario.json', (err, data) => {
  console.log(data);
  getTodos('todos/shaun.json', (err, data) =>
  {
    console.log(data);
  });
});
});
```

My Analysys

```
function getTodos(callback) {
  fetch("https://jsonplaceholder")
  .then(data => data.json())
  .then(res => {
    callback(null, res)
  })
};

function getList() {
  getTodos((err, data) => {
    console.log(data);
  })
}
```

You can call getList() in HTML on button click.

Promise

dummy promise Example

```
const getSomething = () => {
return new Promise((resolve, reject) => {
// fetch something
resolve('some data'); } anyone at a time.
reject ('Some Error');
});
};

getSomething().then(data => {
console.log(data);
}).catch(err => {
console.log(err);
})
```

previous callback example using promise.

```
const getTodos = (resource) => {
return new Promise((resolve, reject) => {
const request = new XMLHttpRequest();
request.addEventListener('readystatechange', () => {
if(request.readyState === 4 && request.status
                                    === 200) {
const data = JSON.parse(request.responseText);
resolve(data);
}
else if(request.readyState === 4) {
reject('error getting resources');
}
});
request.open('GET', resource);
request.send();
});
};
```

```
getTodos('todos/luigis.json').then(data => {
console.log('promise resolved', data);
}).catch(err => {
console.log('promise rejected', err);
});
```

Promise Chaining

```
getTodos('url1').then(data => {
console.log('promise 1 resolved:', data);
return getTodos('url2');
}).then(data => {
console.log('promise 2 resolved:', data);
return getTodos('url3');
}).then(data => {
console.log('promise 3 resolved:', data);
}).catch(err => {
console.log('promise rejected:', err);
});
```

Async await

```
const getTodos = async() => {
};

const test = getTodos();
console.log(test);
```

out - promise

→ async function always return promise.

```
const getTodos = async() => {
const response = await fetch('url');
const data = await response.json();
return data;
};
console.log(1);
console.log(2);

getTodos().then(data => console.log('resolved', data));

Console.log(3);
Console.log(4);
```

Out:
1
2
3
4
resolved: [...]

Traversy Media Youtube

```
const posts = [
{ title: 'Post one', body: 'this is post one'},
{ title: 'post two', body: 'this is post two'}
];

function getPosts() {
setTimeout(() => {
let output = '';
posts.forEach((post, index) => {
output = output + `<li> ${post.title}</li>`
});
document.body.innerHTML = output;
}, 1000);
}

function createPost(post){
setTimeout(() => {
posts.push(post)
}, 2000)
}

createPost({title: 'post 3', body: 'this is post 3'})
getPost();
```

Same

Out:-
- post one
- post two

Callbacks

```
function createPost (post, callback){
setTimeout (() => {
posts.push(post);
callback();
}, 2000);
}

createPost ({title: 'Post three', body: 'this is post three'}, getPosts);
```

out :
- post one
- post two
- post three

Promise

```
function createPost(post) {
return new Promise((resolve, reject) => {
setTimeout (() => {
posts.push (post);
const error = false;
if(!error) {
resolve(); }
else {
reject ("Error");
}
}, 2000);
});
}

createPost({title: 'Post 3', body: 'Post 3'})
.then(getPosts)
.catch(err => console.log(err));
```

Promise.all

Promise.all is actually a function which takes an array of promises and returns a promise. It gets resolved when all the promises gets resolved or any one of them gets rejected.

```
Promise.all([Promise1, Promise2...])
.then(result => {
console.log(result)
})
.catch(error => console.log("Error"));
```

```
const fetchUser = fetch('userUrl');
const fetchColor = fetch('colorUri');

Promise.all([fetchUser, fetchColors]).
then(values => {
  console.log(values);
  });
```

values will be an array

* If one of the promises fails, then all the rest of the promises fail. Then promise.all gets rejected.

Medium

Abdul Manaz

Promise.all

```
const promise1 = promise.resolve(3);
const num = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('foo'), 1000);
});

promise.all([promise1, num, promise3])
.then((values) => {
  console.log(values);
});
```

out:- [3, 42, "foo"]

Promise.any is used when you have multiple async operations that needs to be run in parallel and you have to perform some operation once any of the input promises resolve irrespective of the status of other input promises.

* The returned promise will be resolve as soon as one of the input Promise resolves
* The returned promise will not reject if any of the input promises reject However, if none of the input promise resolves, the returned promise will reject

Promise.race - unlike promise.any which waits for any of the input promises to resolve, this method resolve/reject as soon as one of the input promise resolves/rejects

Promise.any - functional opposite of Promise.all

```
const promise1 = Promise.reject(3);
const promise2 = new Promise((resolve, reject)
=> setTimeout(() => resolve("resolved
second"), 2000));
const promise3 = ---.("resolved first",
                      1000.

promise.any([promise1, promise2, promise3])
.then((response) => {
console.log(response);
})
```

Output - after 1 sec resolved first.

Promise.any

```
const P1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('first Promise'), 500);
});

const P2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('second Promise'), 100);
});

Promise.race([P1, P2]).then((value) => {
  console.log(value);
}).catch(error => console.log(error));
```

out: "second Promise.

{change resolve to reject in the second promise and the catch block will be executing.}

Promise.race is used when you have many async operation in // and you need to perform operations as soon as any of the input resolves/rejects.

Closure (from my notes from Kyle)

It came from λ calculus.

> closure is when a function "remembers" its lexical scope even when the function is executed outside that lexical scope.

```
function one() {
  var temp = "val";

  function two() {
    console.log(temp);
  }

  three(two);
}

function three(two) {
  two();          // "val"
}

one();
```

closure is the ability of a function to remember and continue to access the variables surrounding it in lexical scope even if you take function send it else where (pass it or return it) and execute it in entire different scope.

```
function a() {
  var item = "item";
  return function() {
    console.log(item);
  }
}

function b() {
  a()();   // "item"
}

b();
```

Works same with let also

good example to explain closure in interview.

How the function inside timeout able to access variable bar even when it runs a lot longer in the future after the scope should have gone away and which is run by JS Engine which is entire different scope.

```
function foo() {
var bar = "bar";
setTimeout (function () {
console.log (bar);
}, 1000);
}
foo();
```

function which is our timer hook is referencing variables outside its scope. How do the function continue to access the variable even when it runs a lot longer in the future after the scope should have gone away and it is run by JS Engine which is in entirely different scope

Soln - (closes over variable bar)

Closure: Shared scope-

below is the two different function close over the same variable (bar in this case)

```
function foo() {
var bar = 0;
setTimeout ( function() {
 console.log (bar++);
}, 100);
setTimeout ( function() {
 console.log (bar++);
  }, 200);
}
foo();  // 0 1
```

```
for (var i=1; i<=5; i++) {
setTimeout(function () {
console.log(i);
}, i * 1000);
}
```

out = 6
6
6
6
6

Why didn't we get a new i for each iteration? Because we use the var. We made one i attached to the existing scope.
So all 5 of the inner function are closed over exact same i and reveals the exact same ans for i.

How to get a new variable for each iteration of a for loop?

Ans IIFE

```
for (var i=1; i<=5; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i);
    }, i*1000);
  })(i);
}
```

out :
1
2
3
4
5

this i on which we are closing over is this and not the one in the for loop.

We are making a whole new i for each iteration.

```
for (var i=1; i<=5; i++) {
  let j=i;
  setTimeout(function() {
    console.log(j);
  }, j*1000);
}
```

out = 1, 2, 3, 4, 5

```
for (let i=1; i<5; i++){
setTimeout(function(){
console.log(i);
}, i*1000);
}
```

out = 1, 2, 3, 4, 5

It will automatically create whole new i for each iteration.

Namaste JS closure

```
function x() {
for (var i=1; i<=5; i++) {
  setTimeout (function () {
    console.log (i);
  }, i*1000);
}
}
x();
```

out :-
6
6
6
6
6

Why-? Became of closure.

When loop runs the first time, it makes copy of function attaches a timer and also remembers the reference of i. Similarly these 5 copy of functions are refering to same reference of i.
Second thing is JS don't wait for anything. it will run the loop again & again and store the timeout function & move on.
It won't wait for those timers to expire. When the timers expires its too late and i value will be 6 at that time and when the callback functions runs, by that time value of i will be 6.
console.log(i) → i here refering to reference not value of i.

```
function x() {
for( let i=1; i<=5; i++) {
setTimeout( function() {
console.log(i);
}, i*1000);
}
}
x();
```

Output: 1 2 3 4 5

Let has a block scope. Every times the for loop runs, it will create a new, i (copy of) every time. Every time setTimeout is run, the callback function has new copy of i with it.

Every time setTimeout method is called the function forms a closure with a new variable of i itself. The copy of i in each iteration is new.

Advantages of closure -
Module pattern
function currying
function memoization
- data hiding and Encapsulation.

→ Suppose you have a variable and you want privacy like other functions should not have access to that variable.

```
var count = 0;
function incrementCounter() {
  count++;
}
```

→ Without data hiding. Anyone can access and change count variable.

```
function counter() {
var count = 0;
return function incrementCounter() {
  count++;
console.log(count);
  }
}
```

// console.log(count) → error count is not defined

```
var count1 = counter()
count1();   out = 1
```

disadvantages

overconsumption of mem
closed variables are not garbage collected
and if not handled properly leads
to memory leaks.

Tech storming youtube

Polyfills

① forEach

```
let arr = [1, 2, 3, 4]

Array.prototype.myForEach = function (cb){
  for(let i=0; i<this.length; i++){
    cb(this[i], i); // optional
  }
}

arr.myForEach((val) => {
  console.log(val+100);
})
```

out = 101, 102, 103, 104

Dry run? How this Referring to arr.

② map

```
let arr = [1,2,3,4]

Array.prototype.myMap = function(cb){
  let arr = [];
  for( let i=0; i<this.length; i++){
    arr.push( cb(this[i], i) ); → optional
  }
  return arr;
}

let ans = arr.myMap((val) => {
    return val*2;
})
console.log(ans)
```

ans: 2, 4, 6, 8

(3) Filter

```
let arr = [1, 2, 3, 4]

Array.prototype.myFilter = function (cb) {
let arr = [];
for(let i=0; i<this.length; i++){
if(cb(this[i], i)){
arr.push(this[i]);
}
}
return arr;
}

let ans = arr.myFilter((val) => {
if(val>2)
return val>2;
})
console.log(ans)
```

out:- [3, 4]

youtube
code along with vishal.

call

```
const test = {
  a : 123,
  b : 456
}

function tester(a,b) {
return `a: ${this.a} and b: ${this.b}
  | other a: ${a} and b: ${b}.`;
}

console.log(tester(8,9))
```

out = a: undefined and b: undefined | other a: 8 and b: 9

---

bind -

```
const bindExecutor =
tester.bind(test, 8, 9);
Console.log(bindExecutor());
```

out = a: 123 and b: 234 | other a: 8 and b: 9

all →

```
Console.log(tester.call(test, 8,9));
```

pply →

```
Console.log(tester.apply(test, [8,9]));
```

Before writing polyfills, we have to understand 1 thing.
for this.a and this.b to point to test.a & test.b, we have to move tester function inside test object with any random key (_this in our case)

```
const test = {
  a : 123,
  b : 456,
  _this : function(a, b) {
return( `a: ${this.a} and b: ${this.b} |
  other a: ${a} and b: ${b}`);
  }
};
```

now this.a will point to 123
this.b → 456

---

polyfills for bind

```
Function.prototype.myBind = function(scope, ...args)
{
scope._this = this;   → tester function
return function() {
return scope._this(...args);
};
};

const bindExecutor = tester.myBind(test, 8, 9)
console.log(bindExecutor());
```

out = a: 123 and b: 234 |
Other a: 8, b: 9

My thoughts :- num of arguments can vary, that's why we use (...args) to collect all the arguments

polyfill for call

```
Function.prototype.myCall = function(scope, ...args){
  scope._this = this;
  return scope._this(...args);
}
```

polyfill for apply

```
Function.prototype.myApply = function(scope, args){
  scope._this = this;
  return scope._this(...args);
};
```

... syntax is called the rest parameter syntax. It is used to represent an indefinite number of arguments as an array.

Suppose if we would have written or in our case have already written -

```
const bindExecutor = tester.myBind(test, 8, 9)
```

and -

```
Function.prototype.myBind = function(scope, ...args)
{
console.log(args) // [8, 9] - array
console.log(scope) // Entire test object.
==
}
```

youtube - codeSikho

classmate
Date
Page

currying

```
function multiply (a, b) {
return a * b;
}
```

if you have to do 3 * 5
2 * 5
6 * 5

You have to write

```
multiply (3,5);
multiply (2,5);
multiply (6,5);
```

currying

```
function multiply (a) {
return (b) => {
return a * b;
}
}

const multiplyWithFive = multiply(5);

multiplyWithFive(2); 10
multiplyWithFive(3); 15
multiplyWithFive(6); 30
```

medium- Suppose

when a function, instead of taking all the arguments at one time, takes the first one and return a new function and takes the second one and returns a new function which takes the third one and so forth until all the arguments have been fulfilled

1) Currying helps to avoid passing the same variable again & again.
2) It helps to create higher order function.

Sum (5) (10) (7) (3)

function Sum (a) {
return function (b) {
return function (c) {
return function (d) {
console.log (a+b+c+d);
}
}
}
}

Ans = 25

Akshay saini

Sum(1)(2)(3)(4)---(n)()

Smaller problem -

Sum(1)(2)()

↓

function sum(a){
return function(b){
return a+b;
}
}

↑ recursive call.

~~function~~(

function sum(a){
return function(b){
base case ← if(b){
return sum(a+b)
}
return a;
}
}

classmate
Date
Page

Throttling by techsith youtube (adjust binding of args & context similar to debouncing)

```
function saveData() {
 console.log("saving...")
}

function better(fn, d) {
 let last = 0;
 return function() {
 const now = new Date().getTime();
 if( now - last < d) {
  return;
 }
 last = now;
 fn();
 }
}

const throttle = better(saveData, 2000)
```

```
<button onClick="throttle()"> click me</button>
```

DOMContentLoaded → document Load
↓
assets
(stylesheet, images, subframes)

Date
Page

Q→ What is the difference between document load and DOMContent loaded events?

⇒ The DOMContentLoaded event is fired when the initial HTML document has been completely loaded and parsed, without waiting for assets (stylesheet, images and subframes) to finish loading. Whereas the load event is fired when the whole page has loaded including all dependent resources.

Q→ Difference between Null and Undefined

	NULL	Undefined
(i)	Type of null is object	type of is undefined
(ii)	Indicates the absence of a value for a variable	Indicates absence of variable itself
(iii)	Converted to 0 while performing primitive operations	Converted to NaN while performing primitive operation

What is a pure function ?

A pure function is a function where the return value is only determined by its arguments without any side effects.

Impure

```
const array = [1, 2, 3]
function addElementToArray(element){
  array.push(element)
}
4
out = [1, 2, 3, 4]
```

above function is impure because it relies on external info. it's affecting things which are outside the function.

Pure functions takes some inputs and it only going to operate on those inputs in order to create output.

```
const array = [1, 2, 3]
function addElementToArray(a, element) {
  a.push(element)
}
addElementToArray(array, 4)
out = [1, 2, 3, 4]
```

Impure - Our function is only depending on inputs but it's not pure. It has side effects. It is changing the input a.

```
const array = [1, 2, 3]
function addElementToArray(a, element) {
  return [...a, element]
}
```

↓

Pure function

Youtube - Technical Suneja

Memoization in Javascript

```
function calc(n) {
  let sum = 0;
  for (let i=0; i<n; i++) {
    sum += i;
  }
  return sum;
}

function memoize(fn) {
let cache = {};
return function(...args) {
let n = args[0];
if(n in cache) {
  return cache[n];
} else {
let result = fn(n);
cache[n] = result;
return result;
}
}
}
const run = memoize(calc);
run(5);
```

```
obj = { name: abhi
        age: 25
        address: { Permanent: Ballia,
                   Work: 'bangalore
        }
}
```

```
newObj = { name: "newAbhi"
           age: 25,
           address: { Permanent: 'Ballia',
           Work: 'bangalore,
           }
}
```

Shallow & deep copy Youtube - Pepcoding

```
let obj = {
name : 'Adam',
age : 25
}
let newObj = obj
newObj.name = 'steve'
console.log(obj)      = {name: 'steve', age:25}
console.log(newObj) {name: 'steve', age:25}
```

Shallow copy (using spread operator)

```
let obj = {
name: 'Adam',
age: 25
}
let newObj = {...obj}
newObj.name = 'steve'

console.log(obj)    {name: 'Adam', age:25
console.log(newObj) {name: 'steve', age:25}
```

```
let obj = {
name: 'Adam',
age: 25
location: {Permanent: 'Ballia'
           work: 'delhi'}}
let newObj = {...obj}
newObj.name = 'newAbhi'
newObj.address.work = "bangalore"
```

Deep copy

```
let person = {
name: 'Abhi',
age : 22,
hobbies : ['sports', 'coding', 'music']
}

let newPerson = JSON.parse(JSON.stringify(Person));
newPerson.age = 43
newPerson.hobbies[0] = 'dancing'
```

console.log(Person)	console.log(newPerson)
{	{
name : 'Abhi',	name: 'Abhi',
age : 22	age : 43,
hobbies: ['sports', 'coding', 'music']	hobbies: ['dancing', 'coding', 'music']
}	

Object.assign — does shallow copy

```
let newPerson = Object.assign({}, person);
```

Sorting an Array of objects

```
let arr = [{
firstName: "Steven",
lastName: "Hancock",
score: 90
},{
firstName: "Lynette",
lastName: "Tøgensen",
score: 100
},{
firstName: "Andrew",
lastName: "Wilson",
score: 71
},{
firstName: "Annika",
lastName: "Turner",
score: 82
}];

arr.sort(function(a,b){
if(a.lastName.toLowerCase() < b.lastName.toLowerCase())
  return -1;
else if(a.lastName.toLowerCase() > b.lastName.toLowerCase())
  return 1;
Else
return 0;
});
```

out - Hancock
Tøgensen
Turner
Wilson

if a should appear before b return -ve
if a should appear after b, return +ve
else return 0

```
arr.sort(function(a,b){
return a.score - b.score;
})
```

out
71
82
90
100

Fredrik Youtube

classmate
Date
Page

Critical Rendering Path

The critical Rendering Path is the sequence of steps the browser goes through to convert the HTML, CSS and Javascript into pixels on the screen.

Summary

* Document object model (DOM) + Css object model (CSSOM) = Render tree.
* Css blocks rendering.
* Scripts blocks rendering.
* Css blocks script execution.
* async scripts will not block rendering.
* Reduce the number of static files you load.

It doesnot matter how quickly we get the HTML, if Css is not resolved, we can't able to see anything.

→ <link rel="stylesheet" href="/main.css">

my notes from video.

Scripts and Css block rendering.

→ Dom constraction is Incremental. The HTML response turns into tokens which turns into nodes which turn into the DOM Tree.

The browser is rendering out the web page for every complete node it finds. This means that we will see content in the order it is discovered

→ CSS is render blocking because rules can be overwritten, so the content can't be rendered until the CSSDOM is complete.

The CSS will block rendering and construct the CSSDOM in one operation. This makes it important for us to:

- place the CSS link correctly
- Load as little CSS as possible.
- Load as few network requests as possible

CSS blocks script execution

→ The browser continues to parse HTML making requests and building the DOM, until it gets to the end, at which point it constructs the CSS object model.

Javascript can effect the CSSOM. The browser will try to optimize rendering by halting Javascript execution until the CSSOM is constructed. This means that you need to load your CSS before your JS if you want to be sure that your JS will run

as soon as it is loaded.

async

* The 'async' attribute will execute the script without blocking.
* Put all the scripts at the bottom of the body as a rule.

Flatten an object (Think recursively - Akshay saini)

```
let obj = {
name: 'Abhinav',
address: {
permanent: {
city: 'ballia', pin: 1234
}
work: {
city: 'Bangalore', pin: 5678
}
}
}
```

output: { obj_name: 'Abhinav',
obj_address_work_pin: 5678,
obj_address_work_city: 'bangalore',
obj_address_Permanent_pin: 1234,
obj_address_Permanent_city: 'Ballia'
}

Spread operator and Rest Parameter

Spread operator – The spread operator allows us to expand the collected elements of an iterable data type like strings, arrays, or object into individual elements in places. It helps to create a list of elements from a list by spreading out the values of an iterable where zero or more arguments (function calls) or elements (array literals) or key-value pairs (object literals) are expected.

1– The spread operator can be used to take elements of an array and pass them as a list of arguments into a function.

```
pointColors = ([color1, color2, color3]) => {
console.log(color1 + color2 + color3);
};

let colors = ['red', 'blue', 'yellow'];

pointColors(...colors);
```

out - red, blue yellow.

spread operator does shallow copy

classmate Date Page

2- In Array Literals:

Although there are many ways, the spread operator provides us a new quick method for copying and merging arrays.

```
let firstArray = ['red', 'blue', 'yellow'];
let secondArray = [...firstArray];
console.log(secondArray);
out - ['red', 'blue', 'yellow'];
```

```
let firstArray = ['red', 'blue'];
let secondArray = ['green', 'purple'];
let newArray = [...firstArray, ...secondArray]
out :- ['red', 'blue', 'green', 'purple']
```

3→ In object literals -

```
let color1 = {firstColor: 'red'};
let color2 = {secondColor: 'blue'};
let color1Clone = {...color1};
let colors = {...color1, ...color2};
console.log (color1Clone)
out = {firstColor: 'red'};
console.log (colors);
out = {firstColor: 'red', secondColor: 'blue'}
```

Rest Parameter

The rest parameter is the opposite of the spread operator in a way, where the spread operator expands the collected elements of an array into single elements, the rest parameter collects the list of remaining elements into a single array.

```
let colors = ["red", "blue", "yellow"];
let [firstColor, ...otherColors] = colors;
console.log(firstColor); // red
console.log(otherColors); // ["blue", "yellow"]
```

The rest parameter, which is the same syntax as the spread operator can be used for destructuring arrays and objects to unpack their data into different individual variables.

The Rest parameter can only be the last parameter.

The rest parameter can be very useful when we work with an indefinite number of parameters in the functions!

```
function multiply (... numbers) {
  let ans = 1;
  for (let num of numbers) {
    ans = ans * num
  }
  return answer;
}
multiply (1, 2, 3); // 6
multiply (1, 2, 3, 4, 5, 10) // 1200
```

Remember that the arguments object is an array like object but does not support array methods like `map()` and `forEach()` unless you convert it to a real array. Since the rest parameter gives us an array, those array methods can now be easily used.

maybe because of this, while writing polyfill of apply, we didn't used tripple dot in args as we are already passing arguments as array in apply so no need of tripple dot

JavaScript Slice and Splice

Slice

Array.slice(val1, val2)

* Returns a portion of the array as a second array. (returns sub array)
* Does not modify the array.
* First argument specifies starting element.
* Second arguments specifies ending element
* Second argument is optional.

let arr = [4, 5, 6, 7, 8, 9]

→ let arr1 = arr.slice(0,2); // [4, 5]

starting — upto and not including

→ let arr2 = arr.slice(2,3) // [6]

→ let arr3 = arr.slice(4); // [8,9]

Splice

Array.splice(start, delete)

* Modifies the array.
* The first argument specifies the array position for insertion or deletion
* The second argument dictates the number of elements to delete
* The deleted elements are returned as an array
* The second argument is optional
* Each additional argument is inserted into the array.

```
let arr = [4, 5, 6, 7, 8, 9]
arr1 = arr.splice(2,2);
// arr1 = [6,7], arr = [4, 5, 8, 9]

arr2 = arr.splice(4) = [8,9]
```

insert -

start ↑ don't delete anything ↑

```
let arr3 = arr.splice(2, 0, "a", "b");
arr3 = [], arr = [4,5,"a","b",6,7,8,9]
```

Delete & insert at same time

```
let arr4 = arr.splice(2,1, "a", "b");
arr4 = [6]
arr = [4,5,"a","b",7,8,9]
```

IndexOf

```
let a = [1,2,3,4,5]
a.indexOf(4)   out = 3
a.indexOf(6)   out = -1
```

For in and for of loop

Both are used to loop over arrays but they are different in how they iterate over the arrays and what values they return.

* for...in loop iterates over the enumerable properties of an object, including its own properties and those inherited from its prototype chain. In the case of an array, the enumerable properties are the indices of the array.

object

```
let obj = {
        name: "abhinav",
        address: "Delhi"
        }
for ( let item in obj ){
  console.log (item)
}
```

out = name
address

Array

```
const arr = ["apple", "banana", "orange"];

for (let index in arr) {
  console.log(index); // 0, 1, 2
  console.log(arr[index]);
}                     // "apple", "banana",
                      "orange"
```

for of loop is used to loop over the values of an iterable object.
An iterable object is an object has a "Symbol.iterator" method that returns an iterator object. An iterator object is an object that has a "next()" method that returns an object with two properties: 'value' which is the current value in the iteration and "done" which is a boolean value indicating wheather the iteration is complete or not. Arrays are iterable objects and their default iterator returns the values of the array.
If you use for of for object, you will get error - obj is not iterable.

```
const arr = ["apple", "banana", "orange"];
for (let value of arr) {
  console.log(value);
} // "apple", "banana", "orange"
```