THE COMPLETE JAVASCRIPT COURSE 2020:
FROM ZERO TO EXPERT.

→ Strictly equality operator: === (doesn't perform type coercion)

a === b will only results in true when both values and datatype of a and b are same.

→ Loose equality operator: == (performs type coercion)

a == b results in true if value of a and b is same and not datatype necessarily.

→ Type coercion (JS automatically convert one datatype into another)

→ Type conversion (aka explicit coercion we use following to convert one datatype into another.

i) Number ('23')
ii) String (23)
iii) Boolean

Other operators except '+'.

'23' - '13' - 3 = 10.

'23' + '13' + 3 = 23133.

'23' * '2' = 46.

'23' > '15' = true.

'15' / '3' = 5.

Note.

Plus operator triggers a coercion to strings.

'I am' + 23 + 'years old'.

↓

JS will Convert this number to string.

So in JS whenever there is a operation b/w string and no. the no will be converted into string, in case of '+' operator.

2+3+'5' = 55.

'10'-'4'-'3'-2+'5' = 15.

*

In JS, there are five falsy values. (when we try to convert them into Boolean we get false

i) '' ii) 0 iii) undefined iv) NaN v) null.

↓
Empty String

- Above can be used to find if variable is defined or not.

Const, var, let.

Const should be initialized when declaring else will get error.

let → blocked scoped

Var → function scoped

```
Var a = 10;
Var a = 0
```
} correct

```
Let a = 10;
Let a = 0;
```
} error "can't redeclare variable with let.

- Null & undefined both represent "no value".

undefined: JS way of saying "I can't find a value".

Null: Developer sets the value.

Backward Compatibility

If we try to run older JS in modern web browsers it will run (nothing will be broken).

Forward Compatibility (no fc in JS)

Current browsers don't understand code from future. agr abi browser me JS ka modern version run kre old browsers me to things will be broken.

Q. How to use modern JS today? as browsers users are using might be old.

Ans. For that we need to consider two distinct scenarios

i) development: process when we are building site/application in our computer. To ensure we can use latest feature of JS we should be using up to date google chrome.

ii) Production: When web application is made and we deploy it on internet. Now we can't control which version of google chrome or any browser the user is using.

Sol. Convert modern JS versions back to ES5 using a process called transpiling and polyfilling. (Via Babel).

JS Fundamental part-2

Strict mode: special mode in JS which makes it easier for us to write secure Javascript code. To activate strict mode in JS: use this string at the beginning of script.

file.js

'use strict'; → it has to be very first statement of file.

↳ it avoids us to introduce bugs into our code because.

i) strict mode forbids us to do certain things
ii) it will create visible errors for us in certain situations.

↳ it will let us know about reserved keywords (that maybe used later in language) - eg interface.

---

function declaration

```
function greeting(name)
{
  let val = `Hi ${name}`;
  return val;
}
```

function expression (function without a name) ← aka anonymous function.

```
const a = function (name)
{
  let val = `Hi ${name}`;
  return val;
}

Const res = a("Junaid");
```

* functions are just values like string, boolean

Main difference

We can call function declaration before we define it. but thats not the case with function expression.

Arrow Function (third type of func after function declaration and function expression).

- Special form of func. expression, faster to write.
- example

```
const calcAge = birthYear => 2027 - birthYear;
const res = calcAge(1997);
```

* arrow function does not get 'this' keyword.

↳ no return statement (implicit return)
↳ only in case of one-liner function we can omit return keyword.

Functions calling other function

Arrays

```
const years = [2001, 2002, 2003];
years[0] = 2022; // We can change no matter if it is const, because only primitive values with const are immutable
```

Array is not primitive.
But we cannot redeclare whole array. Only can change elements at different indexes.

Objects

key-value pairs.

```
const a = {
  firstName : 'A',
  age : 24
}
```

object literal syntax.

Dot operator-Notation

`console.log(a.age);` → Is mein dot K sath usi property name hoga jo object me h

Bracket Notation

`console.log(a['firstName']);`

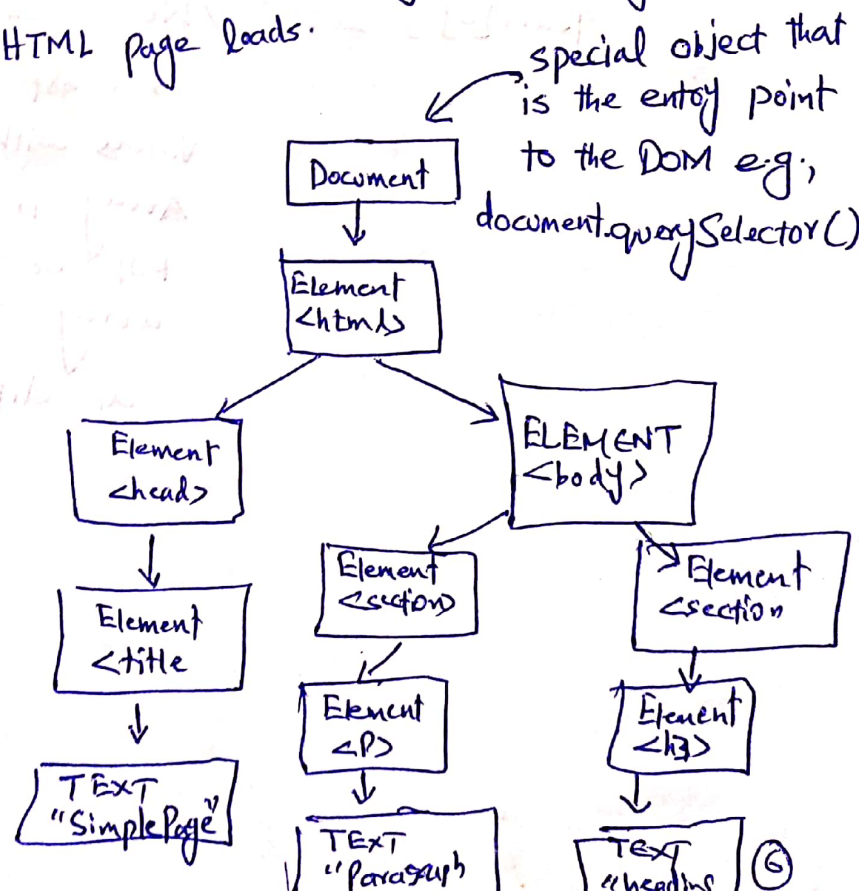
→ Is me hm expression b variable bna skte hain use this when we need to compute the property.

'a representation of all elements in document as a JS object.

DOM: Document object Model: It is a structured representation of HTML documents, Allows Javascript to access HTML elements and styles to manipulate them. DOM can be called as connection point between html documents and Javascript code DOM is automatically created by the browser as soon as the HTML page loads.

```
<html>
 <head>
  <title>Simple Page</title>
 <body>
  <section>
      <p>Paragraph</p>
  </section>

  <section> <h3>heading</h3>
  </section>
 </body>
</html>
```



DOM Tree Structure.

- DOM and DOM methods are part of Web API's (these are like libraries that browser implements and that we can access from our Javascript code. There is an official DOM Specification that browsers implement that's why DOM manipulation works the same in all browsers.

Three types of events for keyboard: i) keyup ii) keypress iii) keydown)

JS features

1) High-level: don't have to manage resources manually. as these language have so called abstraction that take all work from developers and done itself. One disadvantage of high level language is it is very slow and unoptimized as compared to low-level language.
2) Garbage collected: an algo inside JS engine which automatically removes all unused objects from computer memory.
3) JS is interpreted or J-I-T compiled language:

4) Multi-paradigm: JS is multi-paradigm. "Paradigm" can be defined as an approach & mindset of structuring code, which will direct your coding style & technique.

Three Popular paradigms: 1) Procedural programming 2) Object-oriented programming 3) Functional programming.

5) Prototype-based object-oriented.

6) First-class functions: means functions are simply treated as variables. We can pass them into other functions and return from functions

7) Dynamic: means dynamically typed. We don't assign datatypes to variables

```
Let x = 23;
Let y = 19;   ← No datatype definitions. Types become known at runtime
x = "abc"
```
↳ Datatype of variable is automatically changed.

8) Single-threaded

Concurrency model: How JS engine handles multiple tasks happening at the same time. Why do we need concurrency model because

* → JS runs in one single thread, so it can do only thing at a time.

* → A Thread is like a set of instructions that is executed in computer's CPU. ~~Basically, thread is where our code is~~

Event Loop: → "Essential for non-blocking concurrency model"
They take long running tasks, executes them in the "background" and puts them back in the main thread once they are finished.

JS-ENGINE: program that executes Javascript code.

- Every browser has its own JS Engine.
- Most well known is 'Google's V-Eight".

"Any JS engine always contains call stack and heap".

i) call-stack: Where our code is executed using ~~execution~~ execution context.

ii) heap: unstructured memory pool which stores all the objects that our application needs.

Q. How code is compiled into machine code? → it can be done by compilation or interpretation.

A. First few concepts,

i) Compilation: Entire code is converted into machine code at once, and written to a binary file (Portable File) that can be executed by a computer.

ii) Interpretation: Interpreter runs through the source code and executes it line by line.

* Modern JS engine uses a mix b/w compilation and interpretation which is called JIT.

→ Entire code is converted into machine code at once, then executed immediately.

Modern Javascript Just-in-time compilation

All these 4 steps happen in special thread that we can't access from our code

JS ENGINE

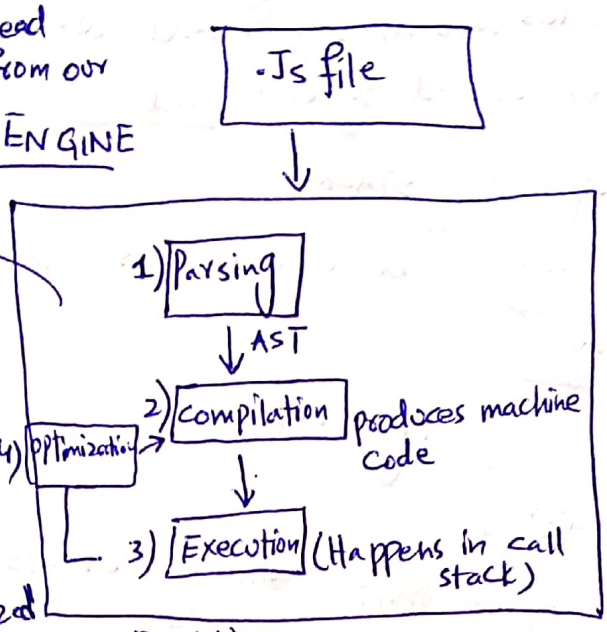


Fig 1.1)

start mein ik unoptimized version of machine code is code which is optimized later and recompiled later. and unoptimized code is removed.

Parsing: means to read the code. During this process, the code is parsed into data structure called Abstract syntax tree (AST). It works by first splitting up each line of code into pieces that are meaningful to the language like keywords const, function etc. Then saving all those piece into the tree in a structured way. this steps also checks if there are any syntax error. Later the resulting tree will be used to generate machine code

Javascript Runtime : Browser

We can imagine JS Runtime as a big box/container which includes all the things that we need in order to use JS, in this case, in the browser. Heart to any JS Runtime is always a JS engine, In order to work properly, we also need access to the Web APIs (DOM, Timers, Fetch API ---)

These are essentially functionalities provided to the engine but actually are not part of JS language itself). JS simply gets access to these APIs through the global window object)

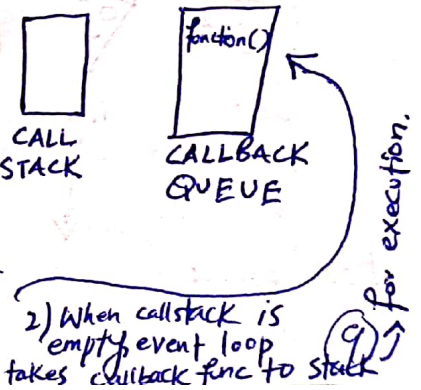
Typical JS runtime also includes a so called "callback queue". Basically it is a data structure that contains all the callback functions that are ready to be executed.

Example

script.Js

```
const element = document.querySelector('.myElement');
element.addEventListener('click', function() { });
```

callback function (1) after event callback function is put into callback queue)



Note : JS can exist outside of browsers e.g Node.JS. So, in Node.JS Javascript runtime there are no Web APIs as we don't have browser now (browser provides Web APIs). Instead, we have multiple C++ bindings and a so called thread pool.

• EXECUTION CONTEXT & CALL STACK

Suppose the code is ready to be executed as in Fig 1.1)

Parsing
↓
compilation

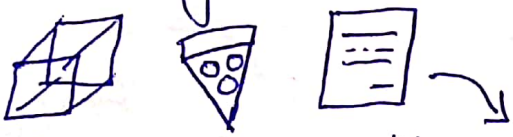
Fig 1.2)

↓ After that following events occur.

So-called global execution context is created for the top level code. Top-level code is basically code that is not inside any function. So in the beginning only the code that is outside of functions will be executed.

Execution Context can be defined as an environment in which a piece of JS is executed. stores all the necessary information for some code to be executed. Such as local variables or arguments passed into a function.
So "Javascript code always runs inside an execution context".

Analogy. We deliver pizza, it comes within a box alongwith a bill.



Box — Pizza — Bill → Execution Context Contains necessary things like bill which is required to eat a pizza.

Pizza "Execution Code". ↑ ↓ JS code

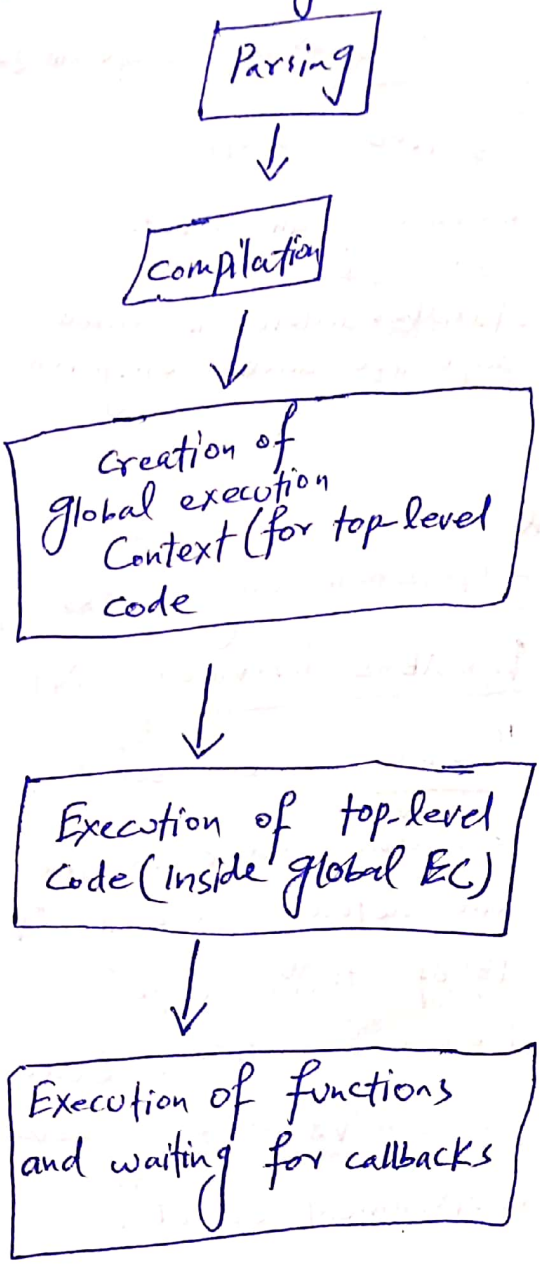
1) top-level code so it will be executed in global execution context

```
const name = 'abc';
const first = () =>
  { return "Hi";
  }
function second()
{ retor "Bye";
}
```

2) Next we've two functions, one expression and one declaration. These will also be declared so that they can be called later. But the code inside the functions will only be executed when they're called.

"In any JS project, there is only one global execution context."
"It's always there as default context, and it's where top-level code will execute."

So after creation of global execution context fig 1.2) becomes



Step 5 Explanation

When top level code finished execution, functions finally start to execute as well. its working is as follows.

- for each and every function call, a new execution context will be created containing all the information that is necessary to run exactly that function. And same goes for "methods". Now All these execution Contexts together make up the call stack. When all function done execution, JS engine will keep waiting for callback functions to arrive, so that they can be executed. Ar in functions ko event loop lata ha callback queuese

Things inside execution context

i) Variable environment
   - let, const, var declarations
   - Functions
   - arguments object.
ii) Scope chain → basically consists of references to variables that are located outside of the current function.
iii) this Keyword

All these 3 content of EC is generated during "creation phase" right before execution.

Note - "Arrow functions EC doesn't have arguments object & this keyword"

- Scoping controls how our programs variables ( ) accessed by the JS engine.
- Lexical scoping: Scoping is controlled by placement of functions and blocks in the code.

Three types of scope in JS

1) GLOBAL SCOPE
- outside of any function or block
- Variables declared in global scope are accessible everywhere.

2) FUNCTION SCOPE
- Variables are accessible only inside function, NOT outside.
- Also called local scope.

3) BLOCK SCOPE
- Variables are accessible only inside block (block scoped)
- only variables declared with let, const are restricted to the block in which they were created.
- Functions are also block scoped (only in strict mode)
  ↓
  it means functions that are declared inside block is only accessible within that block.

* Scope chain only works upwards, not sideways.

VARIABLE ENVIRONMENT HOISTING

Hoisting: Makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".

Before Execution: Code is scanned for variable declarations and for each variable, a new property is created in the "variable environment object".

	HOISTED?	INITIAL VALUE	SCOPE
1) Function declarations	YES	Actual function	Block (only in strict mode else function)
2) var variables	YES	undefined	Function
3) let & const variables	No (They are hoisted but there value is set to uninitialized)	no value to work with. They are placed in Temporal dead zone. means we can't access variable blw beginning of scope and to place where variables are declared.	BLOCK
4) function expressions and arrows		Depends whether created with let, var or const	

"Window is the global object of JS in browser.

•Variables declared with var will create a property on the global window object. Same is not true for let and const.

"mutual recursion"

hoisting Purpose

This Keyword/variable : Special variable that is created for every execution context (every function). It takes the value of (points to) the "owner" of the function in which the "this" keyword is used. This is not static. It depends on how the function is called. and its value is assigned only when function is actually called.

↳ Method me 'this' object ko point krta jo is function ko call krta ha.

↳ Simple functional call me this undefined hota agr hm strict mode use kr re else 'this' will point to global object (which is window object in case of browser). (global parent scope)

↳ Arrow functions: they don't get their own "this" keyword. Now, if we use 'this' in arrow function it will be 'this' keyword of surrounding function. (lexical this). Means its simply gets picked up from the outer lexical scope of the arrow function.

↳ Event Listener: Here if the function is called as an event listener then 'this' keyword will always point to the DOM element that the handler function is attached to.

Note : i) 'this' will never point to the function in which we are using it.

ii) Also, 'this' will never point to the variable environment of function.

Method borrowing

```
Const A = {
    name = "first";
    sayName: function()
    { Console.log(this.name);
    }
}
```

```
Const B = {
    name = B,
}
```

B.sayName = A.sayName // Method borrowing

B.sayName; // Here this will point to object B.

*We should never use arrow function as a method

JS Primitive data-types

1) Number
2) String
3) Boolean
4) Undefined
5) NULL
6) Symbol
7) BigInt

Everything else are objects (they are stored in heap)
- object literal
- arrays
- Functions

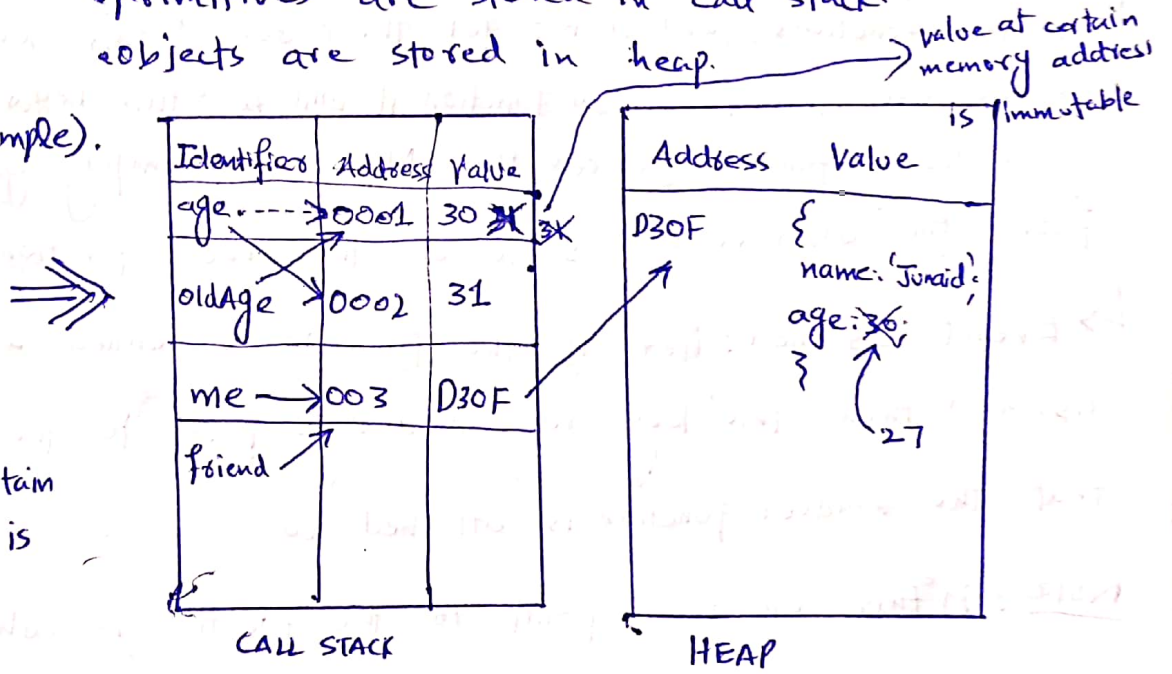
"Now when we're talking about memory and memory management primitives are called primitive types and objects as reference types because of the different way in which they are stored in memory.

- Primitives are stored in call stack.
- objects are stored in heap.

Code (Primitive example).

```
let age = 30;
let oldAge = age;
age = 31;
Console.log(age);
Console.log(oldAge);
```

*Value at a certain memory address is immutable.



Code (Reference Values example)

```
Const me = {
    name : 'Junaid',
    age: 25
};
Const friend = me;
friend.age = 27;

Console.log(friend);
Console.log(age);
```

- To copy objects we can use object.assign;

```
Const a = { age : 27, };

Const b = Object.assign({}, a);
  b.age = 10;
    cl(a); // age = 27;
    cl(b) // age = 10;
```

*But with object.assign only a shallow copy is made. it only works on first level. Means if we have an object inside the object then inner object will still be the same. shallow copy will only copy the properties in the first level. so for that we need deep clone.

New Section

Array destructuring: ES6 feature which is basically a way of unpacking values from an array or an object into separate variables.

(used to switch variable)

e.g.

```
const arr = [1,2,3]

// Destructing   cons [x,y,z] = arr;
```

* original array is not affected.

What we can do with array destructuring:

i) switch variables without using 'temp' variable

ii) return multiple values from function.

Destructuring: break larger data structure into small data structure.

Destructuring objects

```
Const obj = { 'a' : 1,
              'b' : 2,
              'c' : 3,
            };

//destructuring object, names of variables should be same like properties names
   const {a, b, c} = obj;

we can give variables the names of our choice. In that case,
   const {a:myVar, b:myVar1, c:myVar2} = obj;
```

Nested Array destructuring

```
const nested = [2,4,[5,6]];

const [i,j,K] = nested;
```

i → 2, j → 4, K → [5,6]

```
const [i, j, [k,l]] = nested
```

i → 2, j → 4, k → 5, l → 6

Practical application of Destructuring objects

Many times in JS, we've functions with alot of parameters so it is ~~order~~ hard to know the order of parameters. So instead of defining the parameters manually we can just pass an object into the functions as an argument and the function will then immediately destructure that object.

Nested objects destructuring

```
const a = { b:{ c:{d=11,
                   e=12,
                  }}};

Const {c:{d:new1(optional), e:new2(optional)}} = a.b;
```

The Spread Operator ("...")

Let's say we've an array, const a = [1, 2, 3]

Now we want an array like [-1, 0, ---remaining a elements]

so, const newArr = [-1, 0, ... a].

* Difference b/w spread op and destructuring is that spread op. takes all the elements from the array and doesn't create new variables.

- Spread op. i) merge two arrays ii) creates shallow copies of arrays.
- Spread op. works on iterables (arrays, maps, sets, string but not objects).
- spread op. works on objects too (ES2018)

We can only use spread operator when building an array or when we pass values into a function.

Spread operator is used where multiple values are separated by a comma.

spread op. with objects. (also make shallow copy of objects).

Rest Patterns & Rest parameters

By Spread operator we can expand an array into individual elements.

Rest pattern collects multiple elements & collect them into an array.

- When (...) are used on right side of assignment operator it is called spread operator.
- When (...) are used on left " " " " " " " rest pattern.

* Spread operator is used where we would otherwise write values, separated by commas

* Rest pattern is basically used where we would other write variable names separated by commas.

3-properties of logical operators in JS.
i) they can use any datatype ii) they can return any datatype iii) They do short circuiting

~~* We can use logical operators to check wether properties exist in object or not without writing if block inside of object.~~

~~const obj = { 'name': 'Abc', 'age': 25, };~~

~~(obj.name && obj.name = 26)~~

~~should not be using this approach always.~~

*We can use OR operator to set default values.
*We can use AND operator to execute code in the second operand if the first one is true.

NULLISH COALESCING OPERATOR ??

↳ it works with nullish values i.e., NULL and UNDEFINED. only.

For-of loop

```
const arr = [1,2,3,4];
for (const item of arr) console.log(item);
```

- We can use continue and break in ~~break of~~ for-of loop.

Enhanced-object Literal

ES6 introduced three ways to write object literals.

```
Const childObj = { a:1, b:2, c:3};
```

1) `Const parent = { A:10, B:13, C:15, childObj };`

2) We can ~~write~~ method inside obj like this now: → `Const obj = {a:1, b:2, c:3, sum(a,b) {return a+b;}}`
So we omit 'function' keyword and (:) semi-colon.

3) Now, we can compute properties name instead of writing them manually. example → `Const arr = ["sun", "mon", "tue"];`

```
const obj = {
    [arr[0]] : { open: 10,
                 close: 12, } }
```

Optional chaining (ES2020)

- If a certain property doesn't exist in an object then 'undefined' is returned immediately.

e.g. cl(restaurant.openingHours.mon?.open);

* optional chaining operator is generally used with nullish Coalescing operator.

Looping objects, object keys, values, Entries

Object.keys(objName) ⟹ return array containing no, of properties in objName.

object.values(objName) ⟹ return array containing no, of values in objName.

object.entries(objName) ⟹ return array containing both properties and values of objName.

SETS: (Introduced in ES6)

→ it is a collection of unique values. i.e., doesn't contain duplicate values.

```
Const nameSet = new Set( ) → Pass an iterable like array
                          → ['Junaid', 'Mohammad', 'Junaid']);
console.log(nameSet); // Set(2) {'Junaid', 'Mohammad'}
```

→ duplicates are removed only unique values in output we get.

Set is different from as it contains unique elements and order of element is irrelevant. Also there is no index in set. We don't get values out of a set. If we want and retrieve data then array is the suitable data structure to use.

Use case for sets in normal code base

In normal code base, main use case of sets is actually to remove duplicate values of arrays.

Maps : (Introduced in ES6)

↳ is a data structure that we can use to map values to keys.

↳ Just like objects data is stored in key value pair.

Main difference b/w objects and maps is that in maps, the keys can have any type and in objects keys are always strings.

```
const myMap = new Map();
myMap.set('name', 'Junaid');
```

To get value from map we use get method as:-

```
myMap.get('name');
```

↳ datatype of key matters.

Another way of Populating a map

```
const myMap = new Map([ ['question', 'what's your name?'],
                        [1, 'c'],
                        [2, 'Java'],
                        [3, 'JavaScript'],
                        ['correct', 3],
                        [true, 'correct'],
                        [false, 'Try again'] ]);
```

Change object to map

```
const myObj = { 'name': 'Junaid',
                age: 25,
              };
const myMap = new Map(Object.entries(myObj));
```

Convert map to array

```
const arr = [...myMap];
```

Which data structures to use?

First, there are three sources of data.

1) From the program itself: Data written directly in source code (eg status msgs)
2) From the UI: Data input from the user or data written in DOM (eg task in todo app)
3) From external sources: Data fetched for example from Web API.

For a simple list of values use

i) Array ii) sets

Here, we have simply values without any description.

For key value pairs use

i) Map ii) object

Here, we've a way of describing the values by using the key.

Data from web API usually comes in a special data format called JSON (Javascript object Notation). JSON can be converted into JS objects.

Other built in datastructure in JS

→ Weak Map
→ Weak Set

NON-BUILT IN

→ Stacks
→ Queues
→ Linked Lists
→ Trees
→ Hash tables.

ARRAYS VS SETS

- use when you need ordered list of values (contain duplicates)
- use when you need to manipulate data.

```
tasks = ['code', 'Eat', 'code'];
```

- use when you need to work with unique values.
- use when high-performance is really important.
- use to remove duplicates from array.

```
tasks = new Set(['code', 'Eat'])
```

OBJECTS VS MAPS

- More "traditional" key/value store ("abused" objects)
- Easier to write and access values with . and []

```
task = {
  task: 'code',
  date: 'today',
  repeat: true,
};
```

- Better performance
- Keys can have any data type
- Easy to iterate
- Easy to compute size.

```
task = new Map([
  ['task', 'code'],
  ['date', 'today'],
  [false, 'start coding']
]);
```

conclusion

i) Object
→ use when you need to include functions (methods)
→ use when working with JSON (can convert to map)

ii) Map
→ use when you simply need to map key to values
→ use when you need keys that are not string.

FIRST CLASS Vs HIGH-ORDER FUNCTIONS

→ JS treats functions as first-class citizens
→ This means that functions are simply values
→ Functions are just another "type" of object.

As JS has first-class functions, it makes it possible for us to use & write higher-order function (A function that receives another function as an argument, that returns a new function, or both).

Q Why are callback functions used in JS ?

A : i) it makes it easy to split up our code into more reusable and interconnected parts.

ii) * Callback functions allow us to creat abstraction (we hide the details of code implementation. Also it allows us to think about problems at a higher or more abstract level.)

CALL & APPLY METHODS on functions

We can manually set which objects 'this' keyword should point. As in regular function call this points to undefined so this problem can also be eliminated., with the help of call method

* apply works exactly as call but there is only a difference in the arguments.

books.call(objName, arg1, arg2); book.apply(objName, array) → this array contains arguments.

In modern JS, apply is not used generally. Subsitute method is using call with spread operator.

book.call(objName, ...array);

The Bind Method

Like call and apply, bind method also allows us to manually set 'this' keywords for any function call. Difference b/w bind and other two methods is that Bind doesn't immediately call the function instead it returns a new function where 'this' keyword is boond.

* Bind returns a new function.

* Partial application: means that a part of the arguments of the orginal function are already applied(set).

CLOSURES

- it is not a feature that we explicitly use

"Any function always has access to the variable environment of the execution context in which the function was created."

Closure is basically "variable environment attached to the function, exactly as it was at the time and place the function was created."

"Closure has ~~property~~ Priority over the scope chain".

Closure in action

i) When a function returns a function.

ii) When we reassign function even without returning.

iii) Timer.

"functions with memories are called closures".

ARRAY METHODS

~~Const arr = ['a', 'b']~~

Const arr = [1,2,3,4,5,6];

i) cl(arr.slice(2)); //[3,4,5,6]
cl(arr.slice(2,4)); //[3,4]
cl(arr.slice(-1)); // [6]
slice doesn't mutate original array.

→ start at this index.

arr.slice(2) // [3,4,5,6]
arr.slice(2,4) // [3,4].

↓ end parameter not included in output.

ii) Splice (mutate original array)
cl(arr.splice(2)); // [3,4,5,6].
cl(arr.splice(2,4)); //[3,4,5,6]
cl(arr.splice(2,3)); //[3,4,5];

starting index ↑

arr.splice(2) // [3,4,5,6]
arr.splice(2,4) //[3,4,5,6].

↙ How many elements to extract after starting index.

iii) Reverse (mutate original array)
cl(arr.reverse()); //[6,5,4,3,2,1].

iv) CONCAT
Const arr2=[7,8]
Const arr3 = arr.concat(arr2) // [1,2,3,4,5,6,7,8];

V) JOIN
Console.log(arr.join('-')); // [1-2-3-4-5-6]

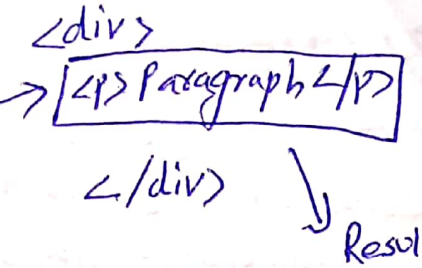
Foreach method ——→ break and continue doesn't work in foreach.

Const arr = [100, 200, -300, 400, -100];
arr.forEach(function(element){ if(element>0) cl('+ive'); else{cl('-ve');}});

*foreach mutates original array.

Here forEach is higher order function as it contains a callback function. That callback function gets executed by forEach method during each iteration. During each iteration, current array element is passed as an argument to that callback function along with its index and the entire array that we're looping. ⇒ arr.forEach(function(element, index, array) ---

document.getElementByTag('div').insertAdjacentHTML
('afterbegin', ' Paragraph ');



Result.

• Data Transformations with Map, filter & Reduce

1) MAP Method

→ another way we can use to loop over arrays.

→ it will return a new array. This new array will contain in each position the results of callback function applied to the elements of original array.

```
const arr1 = [1,2,3,4,5];
const newArr = arr1.map(function(x) { return x * 2 });
// [2,4,6,8,10]
```

2) FILTER Method

→ creates a new array with all elements that pass the test implemented by the provided function (callback function).

3) REDUCE Method

→ executes a reducer function for array element. that function returns a single value: the function's accumulated result.

```
const arr = [1,2,3];
const finalNo = arr.reduce((acc,cur,i,arr) => acc+cur), 0);
```

accumulator initial value.

CHAINING METHOD

```
const mov = [400, 200, -300, -700, 100];

Console.log(mov.filter(curr => curr > 0) // remove -ive number
  .map(curr => curr * 2)  // Multiply +ive no.s with 2
  .reduce((acc, curr) => acc+curr) // Sum all the numbers
);
```

Result = 1400.

4) FIND METHOD

- Returns the first element of the array based on some condition.
- doesn't return new array as the other arrays (previous section).

5) FIND INDEX METHOD

- Returns the Index of the element based on condition.
- findIndex() is similar to IndexOf() but it can contains complex expressions and many operators while IndexOf() only allows to enter the value that need to be searched in array.

6) Some

```
const transactions = [100, 200, -50, -100];
const anyDeposits = transactions.some(mov => mov > 0);
```

↑ return true if there is/are positive no/nos.

Works similar to includes but in includes we can't use condition and also we need to include exact number in argument as it is in the array. So includes only check for equality.

7) Every

array.Every methods return true only if each and every element in the array satisfy the given condition in callback function.

8) FLAT and 9) FLATMAP

Both were introduced in ES2019.

```
const arr = [[1,2,3],[4,5,6]]
console.log(arr.flat()); //[1,2,3,4,5,6]
```

depthArg. Removes nested array.

arr.flat(1) → second level of nesting.
arr.flat(2)

9) Sort

→ mutates the original array.

→ does sorting based on strings.

→ sort method takes a callback function 'Compare' as a parameter.

FlatMap

→ combines map and flat method

→ Better for performance.

→ It only goes one level deep & we cannot change it.

How to create arrays programmatically

```
const x = new Array(7); ⇒ Array constructor function
console.log(x) // creates new array with seven empty elements.
x.fill(1); // [1,1,---1]. ← fill
```

same result

Array.from

```
Array.from({length:7}, () ⇒ 1);
const z = Array.from({length:4}, (_, i) ⇒ i+1);
```

callback func same like map().

= {1,2,3,4}.

★ Array.from() was initially introduced into JS in order to create arrays from array like structures (iterables strings, maps, sets).

use-case

querySelectorAll() returns a NodeList which is something like an array which contains all the selected elements. But its not a real array so it doesn't have methods like map(). In short, querySelectorAll() returning array like structure doesn't have most of the array methods like map() or reduce(). In order to use array methods on that NodeList() we will convert this into array using Array.from() function. Instead of Array.from we can achieve the same using spread operator, but for that we're to do mapping separately.

Which array method to use?

available on codepen.

OOP In JS

- In JS, all objects are linked to a certain prototype object. We say that each object has a prototype. The prototype object contains methods and properties that all the objects that are linked to that Prototype can access and use. This behaviour is called prototypal inheritance

In general OOP, one class was inherited from another class. Whereas, in "JS an instance inherit from a class" - Prototypal inheritance is also known as "delegation".

3 Ways of implementing prototypal inheritance in JS

1) Constructor functions	2) ES6 classes	3) Object.create()
→Technique to create objects from a function. →This is how built in objects like arrays, Maps or sets are actually implemented.	→Modern alternative to constructor function syntax. →"syntactic sugar": behind the scenes, ES6 classes work exactly like constructor functions. →ES6 classes don't behave like classes in "classical OOP".	→most easiest and straightforward way of linking an object to a Prototype object.

→Four principles of oop (Abstraction, Encapsulation, Inheritance and polymorphism) are still valid in JS.

① const Person = function (firstName, birthYear) { };

new Person ('Junaid', 1997);

↓ when we call a function with new operator following four things happen.

1) New empty object is created.
2) function is called and this keyword will be set to newly created empty object in ①
3) Newly created object is linked to a prototype (function prototype property) by adding __proto__ property
4) function automatically returns that empty object from constructor function.

Prototypes

Each and every function in JS automatically has a property called Prototype.

② ES6 Classes

→ classes are special type of functions.

→ class declaration :

```
class Person {
    constructor()
}
```

class expression.

```
const Person = class { }
```

→ All the methods that we write in the class, outside of the constructor, will be on prototype of the objects and not on the objects themselves.

→ classes are not hoisted.

→ classes are first-class citizens, means we can pass them into functions and return them from functions.

→ Body of classes are always executed in strict mode.

SETTERS & GETTERS

→ common to all objects in Javascript.

→ objects can have special properties also known as accesor properties while the more normal properties are called data properties.

→ getters & setters are basically function that gets and sets a value.

→ classes also have them.

→ getters & setters can actually be very useful for data validation.

STATIC METHODS

→ static methods are not available on instances of class as they are not included in ~~propert~~ prototype property of Constructor function.

→ Static methods are some kind of helper function about a class, or about a constructor function.

example array.from not available on arrays.

③ · Object.create

→ no prototype properties involved
→ no constuctor function.
→ no new operator
→ By object.create we can manually set the prototype of an object to any other object that we want.

```
Const PersonProto = { calcAge() { console.log('HI'); }, };

Const steve = Object.create(PersonProto);
```

✓ Inheritance between classes constructor functions

✓ Inheritance b/w classes ES6 classes

✓ Inheritance b/w classes using Object.create

MICROTASKS QUEUE

callbacks registered with promises don't go into callback queue instead they move into a special queue called microtasks queue Microtasks queue has the priority over callback queue.

BUILDING A PROMISE

new Promise(executer function).

- As Promise constructor run executer function gets executed

→ executer function takes two parameters
i) resolve, ii) reject.

Consuming Promise with Async/Await

→ Introduced in ES2017, a better way to consume promise.

→ We add 'async' word before the function to make it asynchronous function.

→ Inside async function we can have one or more await statements.

→ Async/await is only about consuming promises, the way we built them is not influenced in any way.

ES6 Modules

→ Importing of module is done synchronously while downloading is done asynchronously.

In ES6, there are two types of exports

i) Named Exports (Put export in front of anything that we want to export) → Export needs to happened in top level code.

We can export multiple things from module using Named Exports.

ii) Default Exports

usually we use Default Exports when we only want to export one thing Per module.

* During "export" we export value.
* During "import" we give it only desired name.

* Imports are not copies of the export.

Tailwind 5 breakpoints

i) "sm" : "640px"
ii) "md" : "768px"
iii) "lg" : "1024px"
iv) "xl" : "1280px"
v) "2xl" : "1536px"

* By default these 5 break points are defined as min-width media queries. It means if we target the small breakpoint its going to effect anything at 640px or higher.

---

agr kisi component pe classes bahut zyada apply ho hui ho to hm uski ik component class class bna src/style.css me add kr skte with @apply

for example ; a class = "btn - - - - - - - - - - " - represents class.

```
@tailwind base;
@tailwind components;
@tailwind utilities;
@layer component {
    .btn {@apply - - - --
    }
}
```

Flex-basis rules

- Content → width → Flex-basis (limited by max-width & min-width)
- if flex-basis > max-width set max-width else set flex-basis.
- if flex-basis < min-width set min-width else set flex-basis.
- if width is set then it is flex-basis (not set explicitly)
- if we set flex-basis then no need to set width.

Min-width, Max-width, Width

Min-width :

If content is smaller than the minimum width, the minimum width will be applied.

- If the content is larger than the minimum width, the min-width property has no effect.

Note : This prevents the value of width property from becoming smaller than min-width property.

Max-width

- If content is larger than the maximum width, it will automatically change the height of the element.
- If the content is smaller than the maximum-width the max-width has no effect.

Note : This prevents the value of width property from becoming larger than max-width. The value of max-width overrides the width property.

Rules

① if (width > max-width use max-width)
if (width < max-width use width)

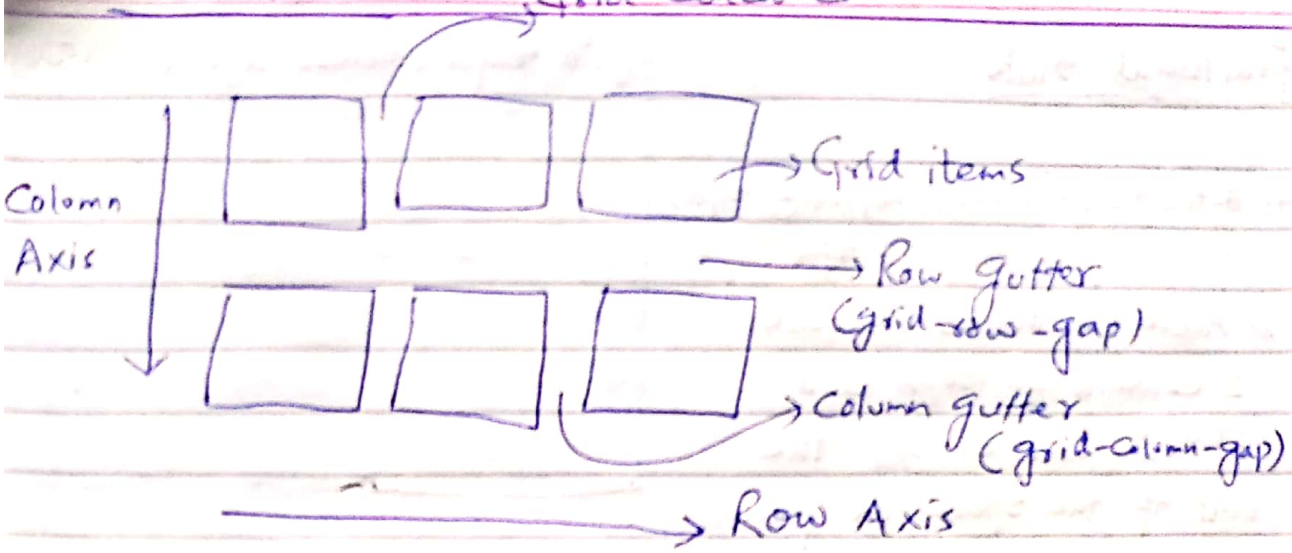
② if (width < min-width use min-width)
if (width > min-width use width)

★ min-width overrides max-width

if (mi-w > ma-w use mi-w
else there is a range.

2:10
4:40
1:40

CSS-GRID



display: grid;

"Vertical & Horizontal lines that divide up the grid and separate the cols & rows are called grid lines"

/* Parent Contain Properties */

display: grid;

flex: 0 1 auto → default

flex: 1 → flex-grow: 1, flex-shrink: 1, flex-base: 0%.

grid-template-rows: 150px 150px

means make 2 rows each of 150px height

grid-template-columns: 150px 150px.

2 columns each of 150px height.

grid-row-gap: 10px // specifying row gutter

grid-column-gap: 10px // " column //

grid-gap: 30px // same gutter for both rows and columns.

means create 2 rows of height 150px

Fractional Units

```
grid-template-columns: repeat(2, 150px) 1fr;
```

// Above line means create 2 columns of 150px and 3rd col will occupy the rest of the space in parent Container

```
grid-template-rows: repeat(2, 150px);
```

→ Positioning GRID ITEMS (child properties)

shorthand

```
grid-row-start: 2;
grid-row-end: 3;      → grid-row: 2/3;

grid-column-start: 2;
grid-column-end: 3;   → grid-column: 2/3;
```

Another way.

```
grid-area: 1/3/2/4;
```

- 1 → line number to start (actually row no)
- 3 → Column no where to start
- 2 → row number to end
- 4 → column number to end.

→ Spanning GRID ITEMS

→ Naming Grid lines

grid-template-rows: [header-start] 100px [header-end box-start]

200px [box-end main-start] ---.

→ Naming Grid Area.

grid-template-rows: 100px 200px 300px 100px;

grid-template-columns: repeat(3, 1fr) 200px;

grid-template-area: " head head head head"

" box box box side"

" main main main side"

" foot foot foot foot".

.header { grid-area: head; }

.footer { grid-area: foot; }

.side { grid-area: side; }.

Note * for Smaller layouts (4x4, 5x5)

Named Grid areas should be used to
else Named Number lines are the one to go with.

Advanced CSS Continued

Explicit vs Implicit Grid

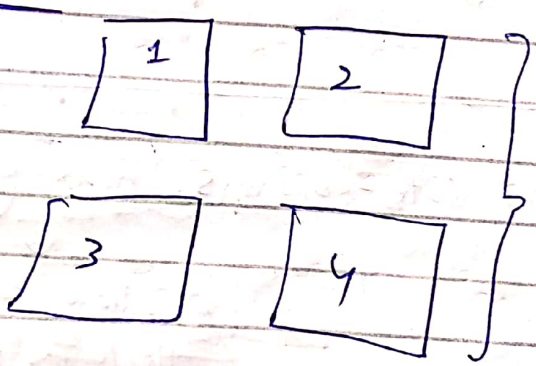
•html

```
<div ..
    div 1
    div 2
    div 3
    div 4
    div 5
    div 6
    div 7
    div 8
</div
```

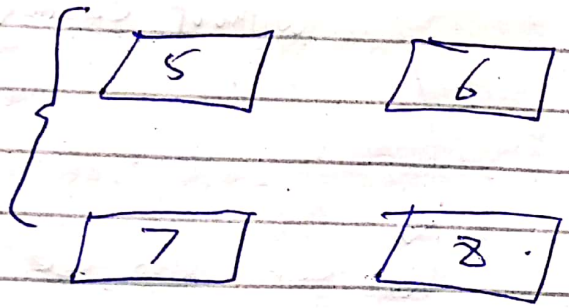
•css

```
display: grid;
~~grid-row~~
grid-template-rows: repeat(2, 150px);
grid-~~colum~~template-columns: repeat(2, 1fr);
```

output



Implicit grid



or grid-auto-columns;

- Property to style implicit grid is grid-auto-rows.

grid-auto-flow: row/column (decided in which direction an implicit grid will be added)

dense: followed by dense to remove holes.

Aligning grid items on grid areas.

Parent property

align-items: center/stretch/start/end

justify-items: center; ↑//

child property { align-self
justify-self

Aligning tracks on inside grid Container

Justify-Content: center;/stretch/start/end

---

grid-template-columns: max-content 1fr 1fr;

↓

allows col to have width so that its content get accomoda without causing any line bre-

We can also use min-content which is used to make column track takes the largest width that is needed to fit the content without overflowing.

Auto-fit vs Auto-fill

<div Container
< 1
< 2
:
< 8
</

```
.Container
{ display: grid;
  width: 1000px;
  grid-template-rows: 150px 150px
  grid-template-columns: repeat(auto-fill, 100px)
```

↳ This will create 10 col tracks as $\frac{1000}{100} = 10$

auto-fit does the same i.e create 10 col. tracks but collapses the unused ones.

↓ give width of 0.

↓ we can used them if we want

Responsive Layouts with minmax and auto-fit.

div Container

8 { div : div

```
.Container { display: grid;
  grid-template-rows: ~~repeat(2,150px)~~
```

↳ repeat(2, minmax(150px, min content))

```
grid-template-columns: repeat(autofit, minmax(200px, 1fr));
```

INTRO to GIT & GITHUB

→ git clone url (to clone repo on local machine from github.

→ ls -la (show all files including files).

→ git status (shows all the files that were created/updated/deleted) but haven't been saved in a commit yet.

→ git add . (track all the files listed in the directory).
or git add file_name/folder

→ git commit -m "Add new file" -m (optional) "description"

Till now, code is saved locally, to make commit live on github, use command git push-

→ git push origin master → branch that we want to push to
↳ location of our git repo.

SSH Keys

→ connect local machine to github account, we use SSH Keys.

Generate SSH Key locally by

→ ssh-keygen -t rsa -b 4096 -C "email"
after generating key search for it by

→ ls | grep testkey
res: testkey → Private Key (Keep it secure)
testkey.Pub → (We're going to upload it to github interface)

cat testkey-Pub
// show content of Public key in console.

Local non-git repo

→ create a folder with a README.md file inside of it.
→ To make it a git repo run following command
git init

To push the above repo live we've to do the following.
→ Make empty repository on Github.
→ copy HTTPS/SSH url
then run following command.
git remote add origin (paste url here)

→ Now if we run → git remote -V
we'll get all remote repositories that we have connected to our local repo.
→ then git push origin master.

GIT BRANCHING

→ git branch // which branch you are currently at.

→ git checkout -b feature-name // make new branch with feature-name

→ git checkout master // switch between branches (master & feature)

★ **Changes made in one branch is not visible in other branch.**

git diff branch_name // show changes made to the file

How to merge two branches locally?

i) First create PR
ii) merge pull request.

PULL REQUEST

→ its basically a request to have your code pulled in into another branch. e.g from feature branch to master branch. For that we make a PR from feature branch to master branch. after PR we delete feature branch.

→ git pull origin master (brought changes locally too).

Asynchronous Javascript Promises, AsyncAwa... and AJAX

- Goal of asynchronous Javascript is to deal with long running tasks that basically run in the background.
- Most common use case of async JS is to fetch data from remote servers in so-called AJAX calls.

~~- Most of the~~

- Synchronous code means the code is executed line by line in the exact order of execution that we defined in our code. Each line of code waits for previous line to finish.

* Execution thread is part of execution Context that actually executes the code in computer's Processor.

Example (1)

```
const p = document.querySelector('.p')
setTimeout(() => {
    p.textContent = 'I am text'
}, 5000)
p.style.color = 'red'
```

Here the callback function in setTimeout func is asychronous Javascript (because async code is executed after a task that runs in a background" finishes, in this it is timer)

- Asynchronous code is non-blocking. example Timer
- Synchronous code is blocking e.g alert.

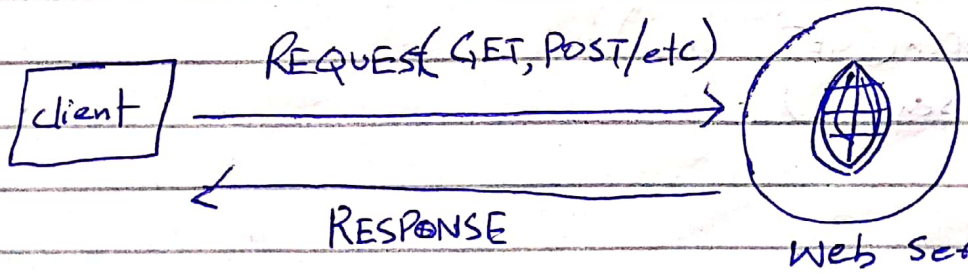
In example (1) when the timer finishes after 5 seconds, the callback function will finally be executed as well. Now in code, callback function was not written in the but it is still executed in the last. Basically, an action was deferred into the future here in order to make code non-blocking.

* Callback functions alone don't make code asynchronous.
* Also Event Listeners // // //
* Like setTimeout, setting the source attribute of any image is asynchronous.

AJAX

(nowadays JSON is used in place of XML as data format)

→ stands for Asynchronous Javascript and XML.
→ Allows us to communicate with remote web servers in an asynchronous way.
→ with AJAX calls we can request data from web servers dynammically.



API

→ Application Programming Interface

→ Piece of software that can be used by another piece of software, in order to allow applications to talk to each other.

There are many types of Web API's

- DOM API
- Geolocation API.
- Own Class API.
- Online API (Application running on a server, that receives requests for data and sends data back as response. or Web API.

JSON: most popular data format. It is basically a JS object converted to a string.

In JS, there are multiple ways of doing AJAX calls.

1) XML HTTP Request → old way of doing ajax calls

```
const request = new XMLHttpRequest();
request.open('GET', 'url');
request.send();
```

CALLBACK TO HELL

- When we have alot of nested callbacks in order to execute asynchronous tasks in sequence

* Promises are used to escape callback hell.

2) FETCH API → modern way of doing AJAX CALLS.

Promise: "An object that is used as a placeholder for the future result of an asynchronous operation."
or "it is like a container for an asynchronously delivered value".

→ Since Promises work with asynchronous operations so they are time sensitive, they change over time so promises can be in different states. This is called the life cycle of a promise.

→ In the beginning we say the promise is pending

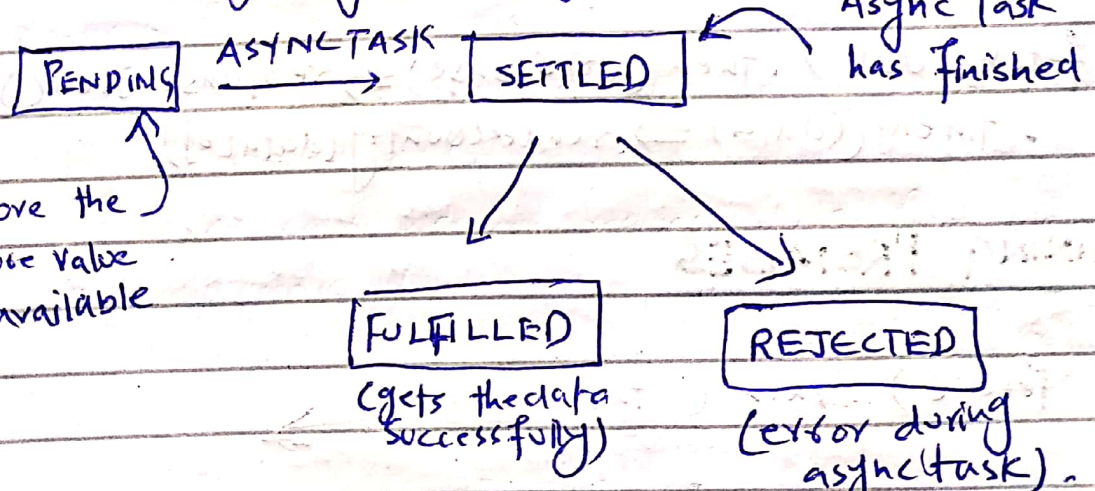
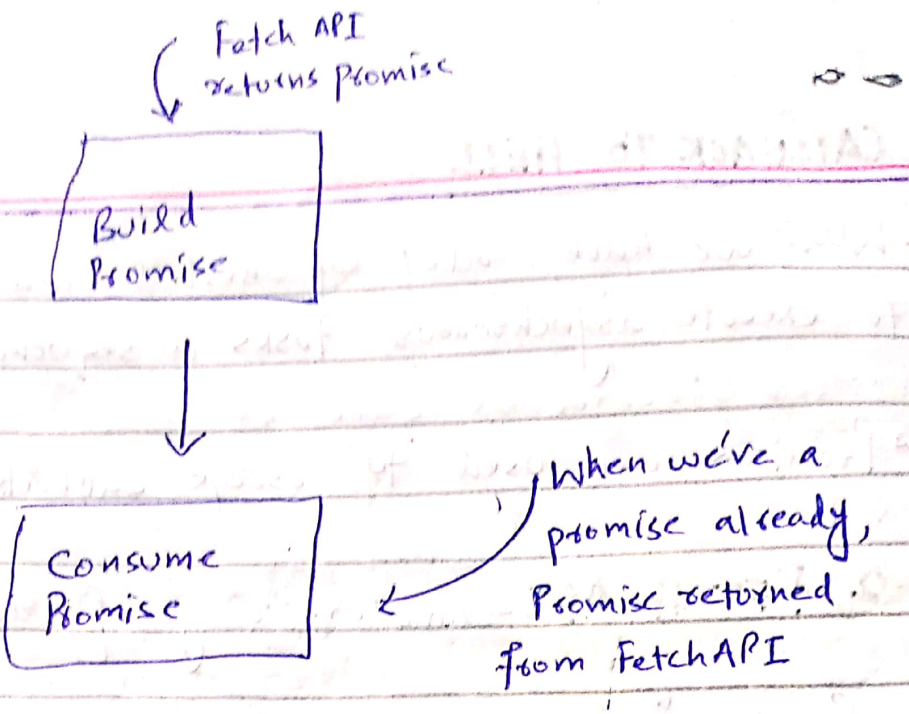


Fig: Lifecycle of a promise.

* A promise is only settled once, so its state then never change forever.



CONSUMING A PROMISE

'then' accepts a callback function to handle state of promise

fetch('url').then() that is executed as soon as the promise is actually fulfilled.

fetch('url').then((response) ⇒ response.json())

Now json() func is also an asynchronous function so it will also return a new promise

fetch('url').then((response) ⇒ response.json())
.then((data) ⇒ renderCountry(data[0]))

CHAINING PROMISES

fetch().then(then()) →

2nd AJAX call inside then method

HANDLING REJECTED PROMISES

Two ways to handle rejected promises

1) Passing a second callback function in then so first callback function will be called for the fulfilled promise and second one for rejected promise.

.catch, .finally (.finally for both rejected & fulfilled Promises. Use case, showing spinner.

.catch → only rejected Promise

.finally → only rejected promise / fullfilled

.catch → doesn't catch 404 error, only catches network error.

~~THROWING ERRORS MANUALLY~~

~~throw new Error~~

2) adding catch at the end of promises chain.

```
fetch('url')
.then(response => response.json())
.then(data => console.log(data))
.catch(err => console.log(err));
```

Throwing errors Manually

```
fetch('url')
.then(response => {

  if(!response.ok)
    throw new Error('country not found')

  return response.json()

})

.then(data => {

})

.catch(err => console.log(err))
```

Immediately terminate the current function

→ Since we are throwing an error in one of then blocks so that then block will return a rejected promise.

PROMISIFYING: It means to convert callback based asynchronous behaviour to promise based.