JAVASCRIPT

CONCEPTS

- 3 ways to apply JavaScript
- Data types and output function
- Variables
- Conditionals
- PopUp boxes
- Strings
- Numbers
- Arrays
- Functions
- Arrow functions
- Higher order functions
- Callback functions
- Pure & Impure functions
- Closures
- Objects and classes
- Object Prototypes
- Diff. btwn expression & Statements
- JS Imports/Exports
- JS Object & array destructuring
- JS Spread operator
- Timing events
- JS audio
- JS Date
- AJAX – Introduction
- HTTP Requests & Response codes
- API & JSON
- AJAX REQUEST SYNATX
- AJAX ERROR HANDLING
- JSON METHODS
- JS PROMISES
- Promise FETCH()
- Async await

3 WAYS TO APPLY JAVASCRIPT

Inline • JS is specified using onload attribute	<body onload="a=10; b=10; alert(a+b);">

Internal
• JS is specified inside <script> tag
	<script>
	var a = 10; // Output comes in console
	console.log(a) // in developer tools
	</script>

External • JS is specified in an independent .js file and linked inside the html file inside <head> or **just before end of <body> tag (preferable)**	<script src = “index.js”></script>

JS BASICS (DATA TYPES & OUTPUT FUNCTION)

Data Types in JS :-	Number, String, Boolean, Undefined, Null, Symbol, Object	

typeof operator :-
• Returns name of datatype of the value passed
	name = ‘jane’;
	typeof(name); // ‘string’
	
	a = 210;
	typeof(a); // ‘number’	

console.log( ) • Output’s data to console	console.log(‘hello’) //hello	
Template Literals • Helps to format strings without need of backslash (\ )	var message = ` this is my ‘first’ message` ; **Output :-** this is my ‘first’ message	name = ‘john’; var message = ` hi ${name} ${2+3}, thank you for joining my mailing list` ; **Output :-** hi john 5, thank you for joining my mailing list`

JS BASICS (VARIABLES)

var keyword:-
• Keyword for declaring variables that have function scope
• Can also be redeclared later
 	
var name = "jane";
var a = 0;
var c ;
var array = [ ];
	function name() {
	  b = 10;
	  a = 5;
	  if (a == 5) {
	    var b = 6;
	    console.log(b); *//6*
	  }
	  console.log(b) *//6*
	}

let keyword :-
• Keyword for declaring variables that have Block scope
• Can be modified later once declared
• Can't be redeclared
 	
let b = 66;
	b = 10;
	a = 5;
	if (a == 5) {
	    let b = 6;
	    console.log(b); *//6*
	}
	console.log(b) *// 10*

const keyword :- • Another Keyword for declaring variables that have Block scope • Can't be modified later once declared	const a = 10
Global variables :- • They can be declared directly to a variable without any keyword	c = 20;

JS BASICS (CONDITIONALS)

For loops & If-else :-

```
for ( let c = 1; c <= 100; c++) {
    if (c%3===0 && c%5===0) {
        a.push("fizzbuzz");
    } else if (c%3===0){
        a.push("fizz");
    } else if (c%5===0){
        a.push("buzz");
    } else {
        a.push(c);
    }
}
```

Switch-case :-

```
switch (new Date().getDay()) {
     case 4:
     case 5:
          text = "Soon it is Weekend";
          break;
     case 0:
     case 6:
          text = "It is Weekend";
          break;
     default:
          text = "Looking forward to the Weekend";
}
```

While loop :-

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

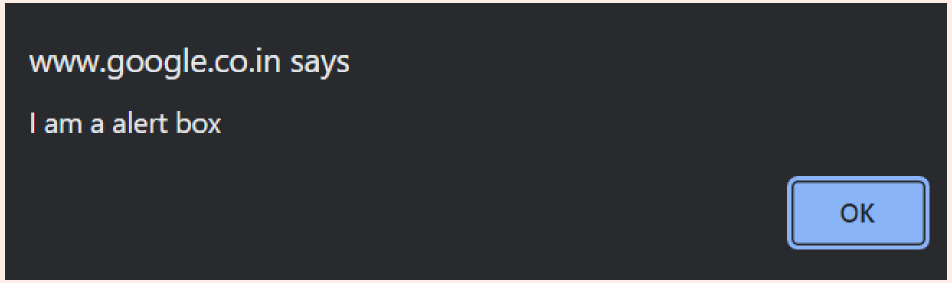
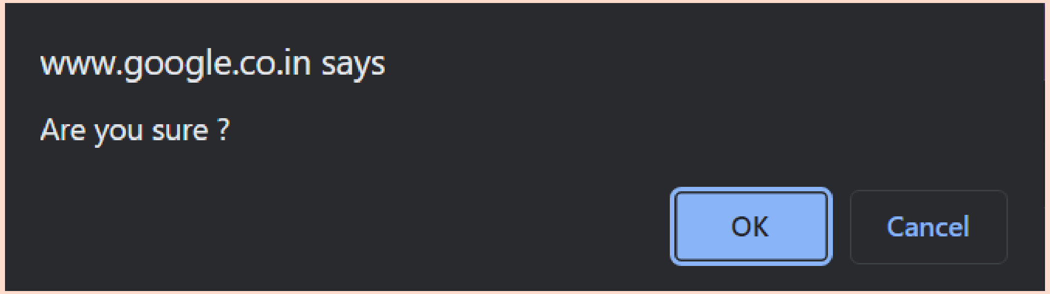
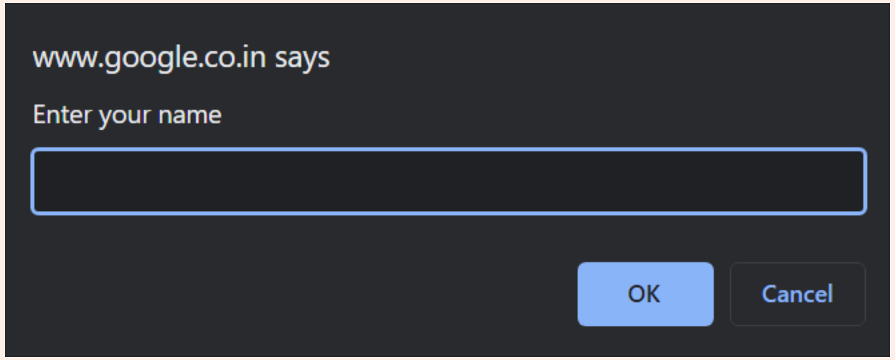
do-while loop :-

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

Ternary Operator :- Condition? Do if true : Do if false

```
if( IsloggedIn === true )? RenderPage() : LogIn()
```

JS BASICS (POPUP BOXES)

Alert Box:- • Used to make sure information comes through the user alert("I am a alert box");	 www.google.co.in says I am a alert box
Confirm Box :- • Used if you want the user to verify or accept something. confirm("Are you sure ?");	 www.google.co.in says Are you sure ?
Prompt box :- • used if you want the user to input a value prompt("Enter your name");	 www.google.co.in says Enter your name

JS BASICS (STRINGS)

Declare string →	var name = "jane";
String concatenation →	console.log(" your name is " + name + " ;) "); // your name is jane ;)
Display length of string →	name.length; //4

String slicing/truncate →	name.slice(0, 2); //ja
	name.substring(0, 2); //ja

Changing string case to upper or lower →	name.toUpperCase(); //"JANE" name.toLowerCase(); //"jane"

Changing string case to title case →	a = name.slice(0,1).toUpperCase();
	b = name.slice(1,name.length).toLowerCase();
	a+b;
	//"Jane"

JS BASICS (NUMBERS)

floor( ) →	Math.floor(3.4); // 3
ceil( ) →	Math.ceil(3.4); // 4
round( ) →	Math.round(3.5); // 4
pow( ) →	Math.pow(3, 2); // 9
random() → • generate 16 decimal random number & never crosses 1	Math.random(); // 0.28325463198997125

JS BASICS (ARRAYS)

Declare array →	var eggs = [1, 2, 3, 4]; console.log(eggs); // [1,2,3,4]
Display length of array →	eggs.length; //4
Check element →	eggs.includes(3) //true
Finds index of element →	eggs.indexOf(3); //3
Insert at end →	eggs.push(6) // 5
Remove from end →	eggs.pop() // 6
Insert at front →	eggs.unshift(6) // 5
Remove from front→	eggs.shift() // 6
Display random element from array →	eggs[Math.floor(Math.random() * eggs.length)];

Loop through an array easily ->	eggs.forEach(function(egg)) {
	  console.log(egg)
	}

JS BASICS (ARRAYS)

<table>
<tr><td rowspan="2"><b>Declare array →</b></td><td colspan="2">var no = [3, 56, 2, 48, 5];</td></tr>
<tr><td><b>Using function() :-</b></td><td><b>Using forEach loop :-</b></td></tr>
<tr><td><b>map()</b>
• Creates a new array by performing operation with each item in an array.</td><td>const SqNo = no.map(function(n){
  return n*n;
})

console.log(SqNo)
//[9, 3136, 4, 2304, 25]</td><td>const SqNo = [];
no.forEach(function (n) {
  SqNo.push(n * n);
});
console.log(SqNo);
//[9, 3136, 4, 2304, 25]</td></tr>
<tr><td><b>filter()</b>
• Creates a new array by keeping the items that specify a given condition</td><td>const ENo = no.filter(function(n){
   return (n%2 === 0)
})

console.log(ENo)
//[56, 2, 48]</td><td>const ENo = [];
no.forEach(function(n){
  if (n % 2 === 0) {
    ENo.push(n);
  }
});
console.log(ENo);
//[56, 2, 48]</td></tr>
<tr><td><b>reduce()</b>
• Accumulate a value by doing something to each item in an array.</td><td>var total = no.reduce(function(sum, n){
  return sum + n;
});

console.log(total); //114</td><td>var sum = 0;
no.forEach(function(n){
   sum += n;
})
console.log(sum) //114</td></tr>
</table>

JS BASICS (ARRAYS)

find()
• find the first item that matches a given condition from an array..
	const num = no.find(function(n) {
	  return n > 10;
	});
	console.log(num); //56

findIndex()
• find the index of the first item that matches a given condition from an array
	const index = no.findIndex(function (i) {
	  return i > 10;
	});
	console.log(index); //1

JS BASICS (FUNCTIONS)

Function :-

- JavaScript takes functions to top of code declared in following manner →

```
root(9);    // 3

function root(n){
     console.log(Math.sqrt(n));
     return;
}

root(16);  //4
```

Function Hoisting :-

- JavaScript does not take functions to top of code declared in following manner →

```
sqrt_n(9);  //TypeError

var sqrt_n = function(n){
     console.log(Math.sqrt(n));
     return;
}

sqrt_n(9);    //3
```

JS FUNCTIONS (ARROW FUNCTIONS)

Arrow functions :-

- provides a new and shorter way to write anonymous function expressions and are always anonymous

```
// normal function
const sum = function (a, b) {
        return console.log(a + b);
   };
sum(5, 5);     //10
```

```
// Writing the above function as an arrow function
const sum = (a, b) => {
        return console.log(a + b);
      };
 sum(5, 5);   //10
```

Parenthesis syntax :-

- if function has 0 or more than 1 parameters then we need parentheses.
- But if function has 1 parameter only then no need of parentheses

```
// here name is only 1 parameter so no need of parenthesis ( )

const greet = name => {
        return console.log(`hey ${name}`);
      };
greet("user");
```

JS FUNCTIONS (ARROW FUNCTIONS)

Concise function body :-

- In arrow function we write 1 line body functions as 1 line expression.
- The return keyword is included in 1 line expressions

```
// normal arrow function
    const game = () => {
      return "Sonic";
     };
console.log(game());    // Sonic
```

```
// above function can be reduced to 1 line expression
const game = () => "Sonic";
console.log(game());   // Sonic
```

- **Value of this :-**

- In regular functions --> this represents the object that calls the function
- In arrow functions --> this represents the owner of the function & value of this depends on surrounding scope.

```
let game3 = {
      title: "Sonic the hedehog",
      related: ["Sonic 2", "Sonic 3"],

regularFunction() {
   console.log(`the game is called ${this.title}`);
    // the game is called Sonic the hedgehog
},
```

JS FUNCTIONS (ARROW FUNCTIONS)

- **Value of this :-**

- In regular functions --> this represents the object that calls the function
- In arrow functions --> this represents the owner of the function & value of this depends on surrounding scope.

```
let game3 = {
        title: "Sonic the hedehog",
        related: ["Sonic 2", "Sonic 3"],

 arrowFunction: () => {
    console.log(`the game is called ${this.title}`);
     // the game is called undefined
},
```

```
showRelated: function () {
 this.related.forEach((relatedGame) => {
   console.log(`Related game of ${this.title} - ${relatedGame}`);

// Related game of Sonic the hedehog - Sonic 2
// Related game of Sonic the hedehog - Sonic 3
```

JS FUNCTIONS (HIGHER ORDER FUNCTIONS)

Higher order functions :- • Are functions which receive a function has an argument or return the function as an output. • Major advantage is that they can be reused dynamically.	Examples of higher order functions include :- 1. map() function in array methods 2. addEventListener() in DOM

map() :-
• map method takes an array and maps those values to new array
• In the following Eg. map method takes double() function as an parameter
	const double = n => n*2;
	let nums = [1,2,3,4,5];
	let result = nums.map(double);
	console.log(result); //[2, 4, 6, 8, 10]

addEventListener() :-
• In the following Eg. one of its arguments is an function
	<p id="p"></p>
	<button id="btn">Click me</button>
	p = document.getElementById('p')
	btn = document.getElementById('btn’)
	
	btn.addEventListener('click', () =>{
	    p.innerText = 'Button was clicked'
	})

JS FUNCTIONS (CALLBACK FUNCTIONS)

Callback functions :-

- Is a function that is passed into another function as an argument.
- They are to be executed later, after the outer function is executed or some event is triggerd

```
// Here respondTokey() is an Callback function
document.addEventListener("keypress", respondTokey(event));

function respondTokey(event){
   console.log("key pressed");
}

// See the event which triggered the callback function
document.addEventListener("click", function(event){
   console.log(event);
})
```

JS FUNCTIONS (PURE & IMPURE FUNCTIONS)

Pure functions :-

- Are functions that for some given arguments always produces the same outcome.
- & has no side effects (when a function changes something outside of itself)

```
// The following is an pure function

const sum = (n1,n2) => console.log(n1+n2);
sum(5, 2);    //will always return 7 & will never change
```

Impure functions :-

- Are opposite of pure functions

```
// The following is an impure function
const randNum = () => console.log(Math.random())
randNum();    //output will always change.
```

```
//The following function changes the value of Result variable
let Result = 0
console.log(Result)      //0

const add = (n1, n2) => {
    const sum = n1 + n2;
    Result = sum;
    return sum;
}

console.log(add(5,5));
console.log(Result)       // 10
```

JS FUNCTIONS (CLOSURES)

Lexical Environment :-

- Every scope has it’s own lexical environment, it consists of :-
- Inner scope – variables declared within it’s scope
- Outer scope – reference to outer lexical environments & variables declared in them
- **In the following example the inner() function has access to it’s scope as well as outer scope**

```
const outer = () => {
  let OUT = "outer";

  function inner() {
    let IN = 'inner'
    console.log(OUT, IN);  //outer inner
  }
  inner()
}
outer()
```

Closures :-

- Are functions that references variables in the outer scope from its inner scope
- They functions can also be invoked from anywhere
- **In the following example the inner() function is a closure( ) & can also be called from outside without calling it from inside**

```
const outer2 = () => {
  let OUT = "outer";

  // inner( ) function is a closure
  function inner() {
    console.log(OUT);
  }
  return inner;
}
let myFunc = outer2();
myFunc();  //outer
```

JS OBJECTS

Creating objects in JSON format :-
• JSON – JavaScript object notation
• Allows to create objects without defining class
• **this** keyword is used to access object attributes inside object or class
	var bird = {
	// bird object attributes
	x:100,
	y:20,
	color: "blue",
	eggs: [1,2,3,4],
	// bird object methods
	fly:function fly(){
	console.log("bird is flying", this.x, this.y);
	}
	}	

Acess object attributes & methods :-	bird.color; // "blue" bird.fly(); // bird is flying 100 20	
Change particular attribute of object :-	bird.x = 120;	

Looping over bird.egg array	for (let i = 0; i < bird.eggs.length; i++) {
	element = bird.eggs[i];
	console.log(element);
	}
		bird.eggs.forEach(function(*val*){
		console.log(*val*);
		});

JS OBJECTS

Iterating over objects:-

- We use the for..in loop to iterate over objects

-
  ```
  for (key in object) {
   console.log(`${key} : ${object[key]}`);
  }
  ```

```
for (key in bird) {
  console.log(`${key} : ${bird[key]}`);
}
```

```
// output
x : 100
y : 20
color : blue ..So-on
```

JS OBJECTS

Another way of creating objects :-

```
 function fruit(taste, color) {
   this.color = color;
   this.taste = taste;
 }

 // new keyword to create objects
 let mango = new fruit("sweet", "yellow");
 let orange = new fruit("sour", "orange");
```

Class keyword to create objects :-

<u>Class declaration →</u>

```
class Fruitclass{
     constructor(taste, color){
       this.color = color;
       this.taste =taste;
     }
 }

let kiwi = new Fruitclass("sour", "green");
```

<u>Class expression →</u>

```
let fruitclass2 = class{
     constructor(taste, color){
       this.color = color;
       this.taste =taste;
     }
 }

let kiwi2 = new fruitclass2("sour", "green");
```

JS OBJECTS (PROTOTYPES)

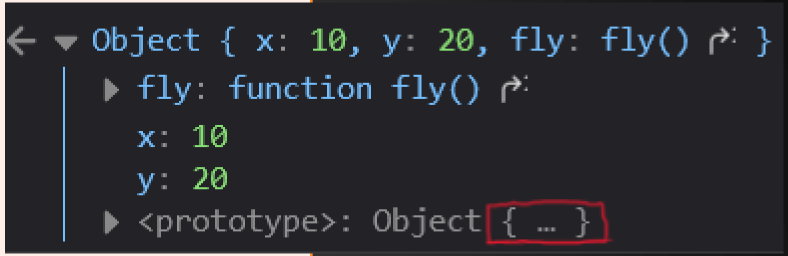
Prototypes :-

- every object type has a prototype.
- it's like a map for a object type as it contains the different functionalities of the object.
- __proto__ will point to prototype.
- functions will be inside the __proto__ property instead of objects directly

// The birdclass class has a fly method

```
function birdclass(x, y){
   this.x = x;
   this.y = y;
   this.fly = function() {
      console.log("bird is flying", this.x, this.y);
   }
 }
```

Output before prototype declaration

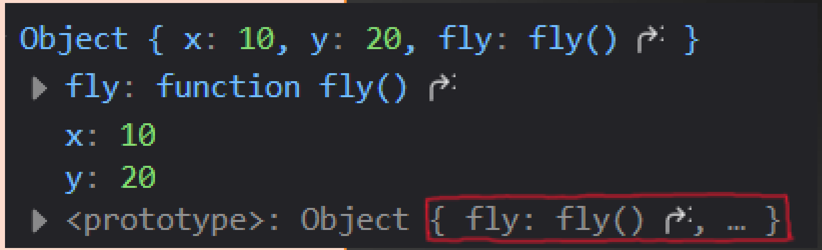


```
← ▾ Object { x: 10, y: 20, fly: fly() ↱ }
    ▸ fly: function fly() ↱
      x: 10
      y: 20
    ▸ <prototype>: Object { … }
```

// we can define fly method on the prototype of user in the following way :-

```
birdclass.prototype.fly = function(){
   console.log("bird is flying", this.x, this.y)
  };
```

Output after prototype declaration



```
Object { x: 10, y: 20, fly: fly() ↱ }
 ▸ fly: function fly() ↱
   x: 10
   y: 20
 ▸ <prototype>: Object { fly: fly() ↱, … }
```

// here's another example :-

```
birdclass.prototype.stoped = function () {
      console.log("bird has stoped flying", this.x, this.y);
  };
```

DIFFERENCE BETWEEN EXPRESSION & STATEMENTS

expression :- • Piece of code that returns a value	**Eg.** const x = 5 const x = sum(2, 3)
Statements :- • piece of code that performs or controls actions but don't result to a value	**Eg.** ifelse, loops, etc.

JS IMPORTS/EXPORTS

export :-
• Will export the specified functions or variable in .js file
	// Export pi variable
	const pi = 3.1415962;
	export default pi;
	
	// Export doublePi(), triplePi() functions
	function doublePi() {return pi * 2;}
	function triplePi() {return pi * 3;}
	export { doublePi, triplePi};
	
	Or
	export function doublePi() {return pi * 2;}
	export function triplePi() {return pi * 3;}

import :-
• Will import the specified functions or variable in from specified .js file
	// Import Specific parts from .js file
	import pi, {doublePi, triplePi} from "./math.js";
	console.log(`${pi} ${doublePi()} ${triplePi()}`)
	
	// Import Everything from .js file
	import * as PI from "./math.js";
	console.log(`${PI.default} ${PI.doublePi()} ${PI.triplePi()}`)

JS ARRAY & OBJECT DESTRUCTURING

It is a way to extract values of array and object keys into new variables

Array, object & nested destructuring :-

- In obj. destructuring names given should match with key names, however alternate names can be provides by key:alternateName syntax

```
// Here red is mapped to 9, green to 132, blue to 227
const [red, green, blue] = [9, 132, 227];
```

```
const animals = [
 { name: "cat", sound: "meow" },
 { name: "dog", sound: "woof", feedingReq: { food: 2, water: 3 } }
];

// array destructuring
const [cat, dog] = animals;

// object destructuring
const { name, sound } = cat;

// providing alternative name to keys of objects
const { name: Catname, sound: Catsound } = cat;

// Give custom values to undefined keys in objects
const { name = "Fluffy", sound = "purr" } = cat;

// nested destructuring - Access object inside object
const {feedingReq: { food, water }} = dog;
```

JS ARRAY & OBJECT DESTRUCTURING

It is a way to extract values of array and object keys into new variables

Array, object & nested destructuring :-

```
const tesla = {
  model: "Tesla Model 3",
  coloursByPopularity: ["red", "white"],
  speedStats: { topSpeed: 150, zeroToSixty: 3.2 }
};
// Access topSpeed value in speedstats object as teslaTopSpeed
const {speedStats: { topSpeed: teslaTopSpeed }} = tesla;

// Access the 1st color from coloursByPopularity array as teslaTopColour
const {coloursByPopularity: [teslaTopColour]} = tesla;

// Access the 2nd color from coloursByPopularity array as tesla2ndColour
const {coloursByPopularity: [,tesla2ndColour]} = tesla;
```

```
const useAnimal = (animal) => {
  return [animal.name, () => console.log(animal.sound)];
};

const [animal, makeSound] = useAnimal(cat);

// access function inside useAnimal()
makeSound();
```

JS SPREAD OPERATOR

Spread operator with objects & arrays :-

```
// Spread operator with arrays
const citrus = ["lime", "lemon", "Orange"];
const fruits = ["apple", "kiwi", "coconut", ...citrus];

console.log(fruits);
//["apple", "kiwi", "coconut", "lime", "lemon", "Orange"]
```

```
// Spread operator with objects
const fullName = {fname: "james", lname: "Bond"};

const user = {
  ...fullName,
  id: 1, username: "jamesbond007"
};

console.log(user);
// {fname: "james", lname: "Bond", id: 1, username: "jamesbond007"}
```

JS TIMING EVENTS

Allows to control when our function is executed.

e.g. Invoke a function 3sec after function has been triggered
e.g. We may want our function to repeat every second

SetTimeout( ) :-
- It will execute function after specified milliseconds.
	// This function will execute after 1sec (1000ms = 1sec)
	setTimeout(function sub() {
	  console.log("message");
	}, 1000);

	// This function will add 2 & 3 and execute after 2sec
	setTimeout(
	  function add(a, b) {
	    console.log(a + b);
	  },2000,2,3);

	// we can also provide reference to a function in following way.
	function mul(a, b) {
	  console.log(a * b);
	}
	setTimeout(mul, 3000, 3, 6);

clearTimeout( ) :-
- Will clear setTimeout methods.
	// the below defined setTimeout event will not execute because of clearTimeout( )
	let timer = setTimeout(mul, 3000, 3, 8);
	clearTimeout(timer);

JS TIMING EVENTS

setInterval() :- - it will repeat the function over and over after specified milliseconds.	**// This function will multiply 3 & 8 and repeat over & over again after 1sec** let time = setInterval(mul, 1000, 3, 8);

clearInterval() :-
- Will clear setInterval methods
	// the below defined setInterval event will not execute because of clearInterval()
	let time = setInterval(mul, 1000, 3, 8);
	clearInterval(time);

This function will display no. 1-10 with a delay of 1sec between them	function count(start, end) {
	  let timer = setInterval(() => {
	      console.log(start);
	      if (start >= end) {
	          clearInterval(timer)
	      }else{
	          start++
	      }
	
	  }, 1000);}
	
	count(1, 10);

JS AUDIO

play() :-
• it will play the audio object
	// create audio object
	var audio = new Audio('audio_file.mp3’);
	// play audio
	audio.play();

JS DATE

getDay() :-

- It returns a int between 0-6 which specifies the day
- 0 – Sunday
- 1-6 – Monday to Saturday

```
// create Date object
var today = new Date();

// Check the date
if (today.getDay() == 6 || today.getDay() == 0) {
   console.log("It's the weekend");
 } else {
   console.log("It's a weekday");
}
```

toLocaleDateString() :-

- To format dates in specific format
- https://stackoverflow.com/a/34015511/14637765

```
// Options parameter of function
var options = {
   weekday: "long",
   day: "numeric",
   month: "long"
  };

var day = today.toLocaleDateString("en-US", options)
```

AJAX - ASYNCHRONOUS JAVASCRIPT & XML

AJAX :- • It is about updating a web page without reloading the entire web page, this is very useful when only some parts of a web page need to be changed**.** • It can make asynchronous requests (i.e. it can send or receive data **asynchronously**.)	**Eg. :-** In Social media Website when you like a post, The server will store this information & will come back with a response, in which the browser reflects the change & the like count is increased
Asynchronous requests :- • Asynchronous means once a client makes a request**,** the client can work on other operations and does not need to wait for the response • Note :- JS is single-threaded & uses callback mechanisms to perform operations in different order	**E.g. :-** setTimeout() & setInterval(), functions are asynchronous
Synchronous requests :- • Synchronous means once a client makes a request, the execution of other operations stop executing until the response is not received	**E.g. :-** console.log(), for loops & variable declarations are synchronous

HTTP REQUESTS & RESPONSE CODES

HTTP :-

- Stands for Hypertext Transfer Protocol
- It’s an underlying protocol that defines how messages are formatted and transmitted.
- It’s a stateless protocol (means the server does not require to maintain information or status about every user for the duration of multiple request)

Steps in HTTP requests processing :-

1. Client opens up connection with server
2. Client makes a request to server
3. Server will process request
4. Server will send response to the client
5. Client will close the connection

HTTP Requests :-

- It indicates the action to be performed on the data transmitted to the server
- **Types of request that you can make using HTTP →**

- **GET** - requests for a data
- **HEAD** - requests for data but without the response body
- **POST** - submits data causing a change in state on the server
- **PUT** – updates existing data
- **DELETE** - deletes the specified data
- **CONNECT** - establishes a tunnel to the server
- **OPTIONS** - describes the communication options for the target data
- **TRACE** - perform message loop-back test along the path to the target data
- **PATCH** - apply partial changes to data

Response Codes :-

- There are different response codes that a server can send to the client to indicate status of a request
- **These request codes are grouped into 5 main classes →**

- **1xx (Informational Resources)** - the request has been received and the process is continuing
- **2xx (Successful Responses)** - the request was successfully received, understood, and accepted
- **3xx (Redirection Messages)** - further action must be taken in order to complete the request
- **4xx (Client Error Responses)** - request contains incorrect syntax or cannot be fulfilled
- **5xx (Server Error Responses)** - server failed to fulfill an apparently valid request

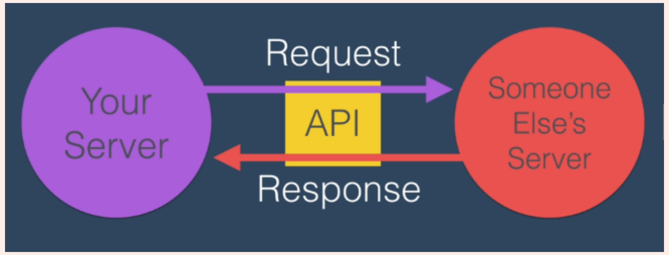
API & JSON

API :-

- Stands for Application programming interface
- It allows 2 applications(client & server) to communicate with each other
- It is a set of commands, functions, protocols, and objects that programmers can use to create software or interact with an external system.

- The starting url of the API is called base URL
- The api_key in the API endpoint serves as means to authenticate
- We can also specify paths and parameters in certain endpoints to extract specific info.
- The first query starts with ? & the remaining with &

// Example of API endpoint – Nasa's APOD :-
// api_key=DEMO_KEY authenticates data to user
https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY



// Example of specifying path to an API endpoint
https://v2.jokeapi.dev/joke/<u>Programming</u>

// Example of specifying parameters to an API endpoint
https://v2.jokeapi.dev/joke/Programming?<u>contains=debugging</u>

JSON :-

- Although 'X' in AJAX stands for XML, JSON is used more than XML
- It is used to exchange data between a browser and a server
- It's syntax is derived from JavaScript object notation, So it is a collection of key: value pairs.
- It's format is text/string only

// Example of JSON

```
{
"date": "2022-07-03",
"hdurl":
"https://apod.nasa.gov/apod/image/2207/Phobos_MRO_3374.jpg",
"media_type": "image",
"title": "Phobos: Doomed Moon of Mars",
"url":
"https://apod.nasa.gov/apod/image/2207/Phobos_MRO_960.jpg"
}
```

AJAX REQUEST SYNTAX

XMLHttpRequest() object :- • it is used to make a request	**// Create object from XMLHttpRequest() to make a request** var xhrRequest = new XMLHttpRequest();
XMLHttpRequest.Open(method, url, async, user, password) :- • **Method** - indicates https requests like GET, POST etc.. • **url** – indicates the API's URL • **async** – is a Boolean attribute, if true then async request else sync request • The open( ) function is used to initialize the request call	**// Initialize the request call** xhrRequest.open("get","https://dog.ceo/api/breeds/image/random", true);
XMLHttpRequest. send(body) • It sends the request to the server. • **Body** – optional parameter & ignored in GET or HEAD	**// Send the request to the server** xhrRequest.send();

XMLHttpRequest.response
• This property returns the response body content
	// Specify the handler
	xhrRequest.onload = function () {
	// log the response received from API in console
	console.log(xhrRequest.response);
	}

AJAX ERROR HANDLING

onerror event listener :-

- The onerror event listener can be used to handle errors

```
// handle errors in case the request fails
XMLHttpRequest.onerror =  function (){
  console.log("Request Failed");
};
```

JSON METHODS

JSON.parse( ) :- • it is used to convert the JSON string into a JavaScript object.	**// Covert the JSON string into a JavaScript object** var responseJSON = JSON.parse(xhrRequest.response);
JSON.stringify() :- • It is used to convert the JS object into a JSON string	**// Convert the myObj object into a JSON string** var myJSON = JSON.stringify(myObj);

Procedure to access a particular key from JSON & make changes in the document →	**// Use parse to covert the JSON string into JavaScript object**
	var responseJSON = JSON.parse(xhrRequest.response);
	// Extract the message key from the JSON received from the API
	var imageURL = responseJSON.message;
	// Change src attribute of img
	document.querySelector(".Image").setAttribute("src", imageURL)

JS PROMISE

Promise( ) :-

- They are objects used to eventually indicate the success or failure of an asynchronous task.
- **resolve** is called when async request is successful
- **reject** is called when async request failed
- **pending** – when promise is neither resolved nor rejected
- Promises can also be passed inside functions & they return a promise object

```
  // Create a promise
userLoggedIN = true;
var promise = new Promise((resolve, reject) => {
        setTimeout(() => {
          if (userLoggedIN) {
            // Promise is resolved call resolve( )
              resolve("User Logged In");
          } else {
            // Promise is rejected call reject ( )
            reject();
          }
        }, 2000);
} )
```

then(), catch() :-

- Then() part executes when promise was resolved
- catch() part executes when promise was rejected

```
// If  promise is resolved or failed then execute some code
promise
        .then((successMSG) => {
          console.log(successMSG);
        })
        .catch(() => {
          console.log("User Not Logged In");
        });
```

JS PROMISE

Callback hell/ chaining requests :-

- Promises save us from Callback hell / chaining requests
- Callback hell occurs when callbacks are nested within other callbacks thus making the code difficult
- They also occur when we want many asynchronous requests to happen in a chain

```
 // Callback hell in Jquery
$.ajax({
    success: function () {
     $.ajax({
      success: function () {
       $.ajax({});
      },
     });
    },
 });

//callback hell using promises
promise.then().then().catch();
```

Example :-

- Check if a user is logged in.
- If so then fetch user feed
- After that, fetch user friends
- After that, fetch user messages

```
// Chaining request using promises
checkUserLoggedIn()
.then(fetchUserFeed)
.then(fetchUserFriends)
.then(fetchUserMessages);
```

PROMISE FETCH()

fetch(base_url, {method})
• Fetch() is used to retrieve data from API's & it returns a Promise object
• If fetch() is successful in retrieving the data from API, then it would execute the then(), else it would execute the catch()
	// fetch syntax :-
	fetch("URL", {method: "GET"})
	fetch("URL", {method: "POST", body})
	
	// **Error handling using catch()**
	fetch("API_URL")
	  .then(res => res.json())
	  .then(data => {
	    console.log(data)
	    throw Error("I'm an error!")
	  .catch(err => {
	    console.log("Something went wrong! 😭")
	  })

GET request using fetch() →	**// Make a GET request using fetch()**
	// (code to display an img from API)
	fetch("https://dog.ceo/api/breeds/image/random")
	  .then((res) => res.json())
	  .then((data)=> document.querySelector('img').setAttribute("src", data.message))

PROMISE FETCH()

Post request using fetch() →

```
// Make a POST request using fetch()
fetch("https://apis.scrimba.com/jsonplaceholder/todos", {
    method: "POST",

    // JSON data we are sending
    body: JSON.stringify({
        title: "Buy Milk",
        completed: false
    }),

    // This will specify we are sending JSON data
    headers:{
        'Content-Type': "application/json"
    }
})
    .then(res => res.json())
    .then(data => console.log(data))

// output :- {title: "Buy Milk", completed: false, id: 201}
```

ASYNC AWAIT

aync & await :-

- The async await syntax can be used to make asynchronous code appear to be synchronous
- async goes before function
- await goes before a method/function that returns a promise

```
function handleClick() {
   fetch("https://apis.scrimba.com/deckofcards/api/deck/new/shuffle/")
      .then(res => res.json())
      .then(data => {
         remainingText.textContent = `Remaining cards: ${data.remaining}`
         deckId = data.deck_id
         console.log(deckId)
      })
}
```

```
//Changing above code to async function :-
async function handleClick() {
   const res = await
fetch("https://apis.scrimba.com/deckofcards/api/deck/new/shuffle/")
   const data = await res.json()
   remainingText.textContent = `Remaining cards: ${data.remaining}`
   deckId = data.deck_id
   console.log(deckId)
}
```