Javascript

- Roadside Coder

Variable Scope :-

→ A scope is a certain region of program, where a defind variable exist & can be recognized. Beyond that it can't be recognized.

→ var is functional scope, that it can be accessible outside the function scope, block scope etc. ✗

→ But let & const are block, where it declared, it can only accessed within that block or scope.

Ex:-

```
{ var x = 10;
}
print(x);   // possible
```

```
{ let x = 10;
  const y = 11;
}
print(x & y);
// not possible.
```

Variable shadowing :-

```
function test() {
    let a = "Hello";
    if (true) {
        let a = "Hii";
        console.log(a);
    }
    console.log(a);
}
```

→ This is called shadowing it will work absolutely fine.

```
function test() {
    var a = "Hello";  ✓
    let b = "by";  ×
    if (true) {
        let a = "Hii";  ✓
        var b = "Bye";  ×
        console.log(a);
        console.log(b);
    }
}
```

→ This is called illegal shadowing.

Declaration :-

→
```
var a;  ✓
var a;  ✓
```

. it is possible to re declare var variable as many time as you want.

→
```
let a;  ✓      const b;  ✓
let a;  ×      const b;  ×
```

. in case of let & const, it is not possible to redeclare a variable.

Declaration without initialization :-

→ It is possible in case of var & let

→ But not possible in case of const.

Ex:-
```
let a;                    const a;  ×
var b;  a=10;  b=5;
```

Re-initialization :-

→ We can re-initialize the value of var & let

→ But we can't in const.

Ex:- let a = 10;

var b = 20;

a = 12;

b = 22;

Const x = 10;

x = 20; ✗ (Not possible).

Hoisting :-

→ Storing before executing.

Map, Filters & reduce :-

Map() :-

→ The map method is used for creating a new array from existing one, by applying a function to each of the element of the array.

Ex:-

```
const arr = [1, 2, 3, 4];

const multiplyTwo = arr.map((ele, i, arr) => {
    return ele * 2;
});
console.log(multiplyTwo);
```

O/P

[2, 4, 6, 8]

filter() :-

→ It takes each element of array ~~of~~ & it applies a conditional statement against it, if the condition returns true, then the element get pushed inside the array, otherwise it doesn't push the element.

Ex:-

```
const nums = [2, 4, 6, 8];

const greaterThanFour = nums.filter((ele) => {
    return ele > 4;
});
console.log(greaterThanFour);
```

O/P

[6, 8]

reduce() :-

→ This method reduces the array of values down to just 1 value.

Syn:- num.reduce(callbackFn, initialvalue);

Ex:- const nums = [1, 2, 3, 4]

const sum = nums.reduce((acc, curr, i, arr) => {?, 0);

→ if there is no initial value, it takes first element of array as value for accumulator.

polyfills :-

1) polyfill for map().
2) polyfill for filter().
3) polyfil for reduce().

map Vs forEach :-

→ Both are arrays function to loop through items of the array.

forEach :-

→ It doesn't return anything just like map does.

→ It has different use case.

→ Ist we can modify the original array value, which is also be done by map. but the map can also return the value, which create new array by that method.

→ we can chain other methods in map, but not in forEach, because it doesn't return anything.

Declare an Array in Js :-

→ Array is one of the most commonly used data-types, which stores multiple values & elements in one variable.

→ In Js it can be of any datatype - meaning we can store a string, number, boolean & others in a single array.

Ex:- let arr = [12, "Hello", 2.44, 'c'];

→ other ways using array constructor

Ex:- let arr = new Array("Hello Ji", 12, true);

→ Declaration,

let myArray=[];

Functions in Javascript :-

① Function Declaration :-

Syn:-
```
function fname(arg){
    statements
}
```

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

→ This can also be called function defination or function statement.

② Function Expression :-

```
const square = function(num){
    return num*num;
};
```

→ This is called an anonymous function, the function which has no name.

→ It can be assigned to a variable or passed as a call back.

→ we can call this function as normal function like

```
square();
```

→ we can assign a function to a variable, and function acts like a value, this is known as function expression.

→ The major difference betn Declaration & Expression is hoisting, we can't call function expression until initialization.

(3) First class function

→ When a function can be treated as like variable, these function can be called as first class function.

→ In these cases functions can be passed into another function can be used, manipulated, & returned from those function, basically everything a variable can do.

Ex:-

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

function displaySquare(fn) {
    console.log("Square is " + fn(5));
}

displaySquare(square);
```

→ When a function is passed as an argument, which is treated like another variable. i.e first class function.

(4) IIFE (Immediately invoked function expression)

Ex:-

```
function square(num) {
    console.log(num*num);
}
square(); // Normal way

(function square(num) {
    console.log(num*num);
})(5);          // IIFE
```

⑤ Function scope :-

→ if a variable doesn't exist in the function, but declared outside, we can still use them.

→ It is possible by the lexical environment of the javascript.

Parameters Vs Arguments :-

```
function square(num) {        → parameters
    console.log(num * num);
}
square(5);                    → Arguments
```

→ when we pass a value, while function calling i.e Arguments.

→ when that arguments is riecieved by function i.e parameters.

Spread operator :-

Ex:-

```
function multiply(num1, num2) {
    console.log(num1 * num2);
}
var arr = [5, 6];
multiply(...arr);
```

→ rest operator

or (...nums)

or

```
(nums[0] * num[1]);
```

→ Spread operator

→ It should be written in our last of our parameter

Ex:-

```
const Fn = ( a, x, y, ...numbers ) => {
    console.log( x, y, numbers);
}
Fn( 5, 6, 7, 8, 9, 10 );
```

→ [8, 9, 10]

⑥ Callback Function :-

→ A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Ex:-

```
function saymyname( name ) {
    console.log( "Hello" + name);
}
function takeInput ( cb ) {
    let name = prompt ( "Enter name" );
    cb(name);
}
takeInput( saymyname);
```

⑦ Arrow Function :-

→ Converting an arrow function to regular function or vice-versa.

Ex:-

```
const add = function ( num1, num2) {
    return num1+ num2;
};
```

(Normal function)

```
const add = (num1, num2) => {
    return num1 + num2;
};
```
(Arrow function)

→ For one liner,

```
const add = (num1, num2) => num1 + num2;
```

→ Differences from normal function.

① Syntax is different from normal function.

② Implicit return keyword.

③ Arguments keyword. don't work in arrow function.

④ this keyword,

⑧ Anonymous Function :-

→ A function which doesn't have any name is called anonymous function.

→ ~~when a~~ These functions only be used as values

```
function () {
    ...
}
```
↳ we can't declare like this

```
var b = function () {
    ...
}
```
↳ Here we can

⑨ Named function expression :-

Ex:-
```
var b = function xyz() {
        console.log("Hii");
}
```

→ when you give a name to function, in function expression, it is known as named function expression.

- `b();` ⟶ It can called like this
- `xyz();` ⟶ It is not possible to call like this

→ The ability of a function, which is used ~~as~~ as value & can be passed as an argument to another function & can be returned from function is known as first class function.

→ Ability to be used like values make the function first class citizen.

→ Functions are first class citizen in javascript, when we pass a function as an argument i.e the call back function.

→ It is useful in asynchronous javascript.

Roadside Coder

Objects :-

- Object is one of javascript data types, which is a collection of properties.
- a property is an association betn a name (key) and a value.
- A property value can be different type of data, function or another object etc., in case of function the property is known as method.
- It can be created by using two methods Object() constructor & literal syntax i.e using curly braces.

Literal syntax

```
const user = { }
```

this is object, now empty object

Normally declared with const

Ex:-

```
const user = {
    name : "Tahir",
    age : 24,
};

print( user.name); // Access

user.name = "js"; // Modify

delete user.age; // Delete
```

* Delete keyword only used to delete the properties of an object, not the local variables.

Giving multiple word key :-

```
const user = {
    "like this video" : true;
}
```

→ For accessing multiple word key, we can't use the (.) operator, so

* print (user["like this video"]);

→ We can use this syntax for accessing single word key also.

Adding Dynamic property to object :-

```
const property = "name";
const value = "Tahir";

user user = {
    [property] : value;
}
```

→ To add dynamic property, we just need to wrap inside a square bracket.

→ There is also other ways.

Loop through Object :-

→ By using for in loop we can do that.

Ex:-

```
for ( key in user ) {
    print ( key );          // property name
    print ( user[key] );    // property value
}
```

Questions :-

① what is the output ?

```
const obj = {
    a : "one",
    b : "two",
    a : "three",    // Here a is replaced
}
print ( obj );
```

O/p [a : "Three", b : "Two"]

② Create a function, multiplies all numeric property by 2.

```
const user = {
    a : 100,
    b : 200,
    title : "My nums",
};
```

③ What is the output ;

```
const a = { };
const b = { key : "b" };
const e = { key : "c" };

a[b] = 123;
a[c] = 456;

print (a[b]);
```

JSON.stringify :-

→ It will convert an ~~string~~ object to complete string.

```
strobj = JSON.stringify (obj);
```

JSON.parse :-

→ To convert it back to object, we use this method.

```
JSON.parse (strobj);
```

④ What is the output ?

```
print ([... "Lydia"]);
```

It will spread the each character

⑤

```
const sett = {
    username : "piyush",
    level : 19,
    health : 90,
}
const data = JSON.stringify(sett, ["level", "health"])
```

It will only stringify level & health property.

⑥

```
const shape = {
    radius : 10,
    diameter() {
        return this.radis * 2;
    },
    perimeter : () => 2 * Math.PI * this.radius
}
```

This will reference to shape object

This will reference to window object.

```
print(shape.diameter());      // 20
print(shape.perimeter());     // NaN
```

Destructuring in object :—

```
let user = {
    name : "Tahir",
    age : 22,
}
const { name } = user
```

This is destructuring

Object Referencing :-

Ex:-

```
let c = { greeting : "Hey! " };

let d = c;

d.greeting = "Hello";

print ( c.greeting )   // Hello
```

→ Simply here, it will be not creating a copy of 'c' and assigning to 'd', instead it will reference 'c' to 'd'.

→ So whatever change we do to the 'd', it will also affect 'c'.

Shallow copy & Deep copy :-

'this' Keyword :-

→ 'this' is used reference an object in javascript

→ ~~It depends on context~~

→ It depends upon the context in which the 'this' Keyword is used.

Ex:-
```
this.a = 10;
print(this.a);        // Now it is pointing to global object i.e window
```

→ But inside an object, it will reference to the current parent.

Ex:-
```
const user = {
    name: "KK",
    age: 22,
    func: function() {
        print(this.age);    // 22
    }
};
```

Now this is pointing to user object.

→ In case of arrow function, the case is different.

Ex:-
```
const obj = {
    age: 22,
    func: () => print(this.age);
};
```

Now this is pointing to global object

Ex-2 :-

```
const user2 = {
    name : "KK",
    func : function ( ) {
        const nested = () => print(this.name);
    },
};
```

Now this is pointing to user2 object.

→ In case of normal function, it will point to immidiate parent.

→ In Arrow function, it will point to global object, Because arrow function takes reference from it's parent function.

→ In example-1, there is no parent function for arrow function, But in example-2, there is a parent function.

Call, Bind & Apply :-

→ This 3 method can be applied to all the javascript function.

Call() :-

Ex:-

```
var obj = { name : "khan" };

function sayHello( ) {
    return "Hello" + this.name;
}
print ( sayHello() );
```

→ There is no such thing as 'name' in global object

→ Right now, 'this' is pointing to global context, we can change that & point it to the whatever object we want, By using this call() method.

Ex:-

```
print ( sayHello.call(obj) );   // Now Done.
```

→ Call methods takes argument first an object & List other arguments

Apply() :-

→ It is similar as call, but the list of arguments should be passed in the form of array.

Ex:-

```
sayHello.apply( obj, [12, "Engineer"] );
```

Bind() :-

→ It can provide a reusable function, that we can use later with different argument.

Ex:-

```
const bindFunc = sayHello.bind(obj);

bindFunc(12, "Engineer");
bindFunc(44, "Youtuber");
```

→ Basically, it will also change content, but it returns whole function, which is later used with different argument.