JavaScript Questions Part-1

1. What's the difference between `undefined` and `null`?

- `undefined` is the default value of a variable that has not been assigned a specific value. Or a function that has no **explicit** return value ex. `console.log(1)`. Or a property that does not exist in an object. The JavaScript engine does this for us the **assigning** of `undefined` value.
- `null` is **"a value that represents no value"**. `null` is value that has been **explicitly** defined to a variable. In this example we get a value of `null` when the `fs.readFile` method does not throw an error.

2. What is the DOM?

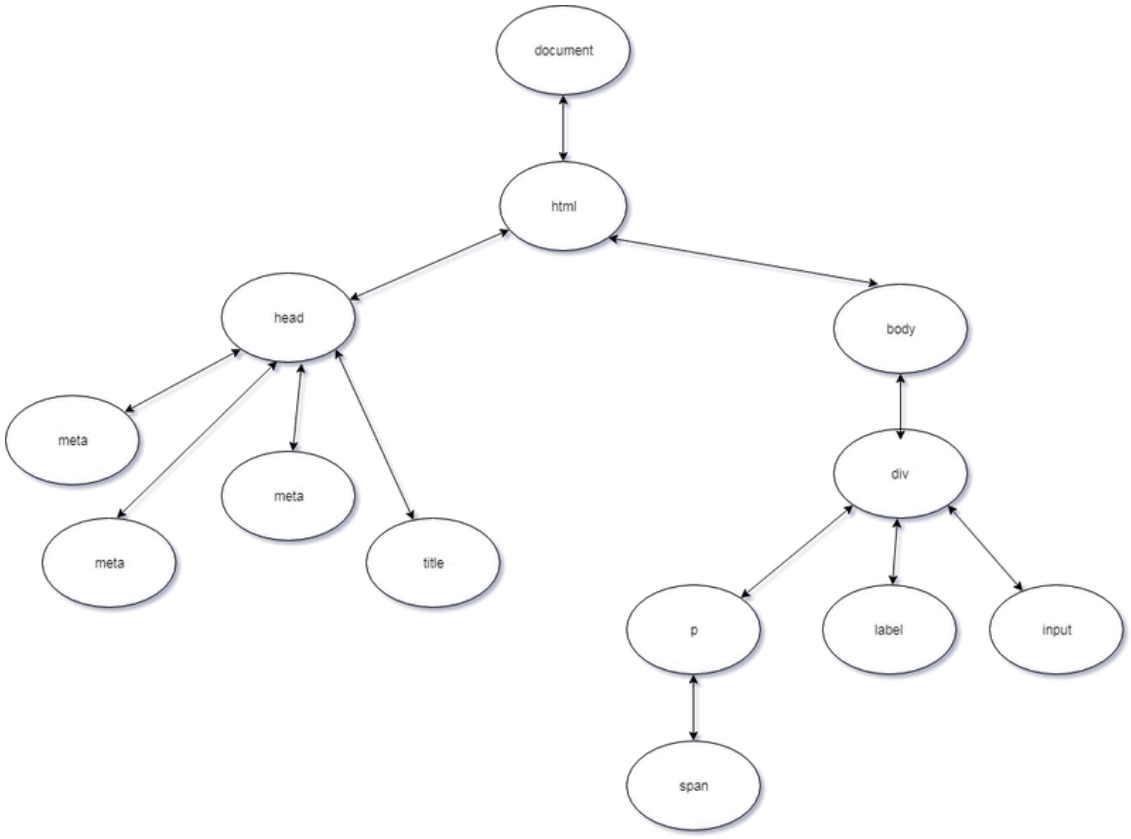
DOM stands for **Document Object Model** is an interface (**API**) for HTML and XML documents. When the browser first reads (*parses*) our HTML document it creates a big object, a really big object based on the HTML document this is the **DOM**. It is a tree-like structure that is modeled from the HTML document. The **DOM** is used for interacting and modifying the **DOM structure** or specific Elements or Nodes.

Imagine if we have an HTML structure like this.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document Object Model</title>
</head>

<body>
  <div>
    <p>
      <span></span>
    </p>
    <label></label>
    <input>
  </div>
</body>
```

```
</html>
```

The **DOM** equivalent would be like this.



The `document` object in **JavaScript** represents the **DOM**. It provides us many methods that we can use to selecting elements to update element contents and many more.

📜 DOM Methods Cheat Sheet

🔍 Selecting Elements

Method	Description
`document.getElementById('id')`	Selects an element by its ID.
`document.getElementsByClassName('class')`	Returns a live HTMLCollection of all elements with that class.
`document.getElementsByTagName('tag')`	Returns all elements with the specified tag name.
`document.querySelector('selector')`	Returns the first element matching the CSS selector.
`document.querySelectorAll('selector')`	Returns a NodeList of all elements matching the CSS selector.

📦 Creating and Inserting Elements

Method	Description
`document.createElement('tag')`	Creates a new HTML element.
`element.appendChild(newElement)`	Appends a child to an element (at the end).
`element.insertBefore(newNode, referenceNode)`	Inserts a node before another inside a parent.
`element.insertAdjacentHTML(position, html)`	Inserts HTML at specific positions: `'beforebegin'`, `'afterbegin'`, `'beforeend'`, `'afterend'`.

🛠️ Modifying Elements

Method	Description
`element.setAttribute('name', 'value')`	Sets an attribute on an element.
`element.getAttribute('name')`	Gets the value of an attribute.
`element.removeAttribute('name')`	Removes an attribute.
`element.classList.add('class')`	Adds a class to the element.
`element.classList.remove('class')`	Removes a class.
`element.classList.toggle('class')`	Toggles a class on/off.
`element.classList.contains('class')`	Checks if a class exists.
`element.style.property = 'value'`	Changes inline styles (e.g., `style.color = 'blue'`).

🧹 Removing Elements

Method	Description
`element.remove()`	Removes the element from the DOM.
`parent.removeChild(child)`	Removes a specified child from its parent.

💬 Working with Content

Method	Description
`element.innerHTML`	Gets/sets HTML content inside the element.
`element.textContent`	Gets/sets plain text inside the element.
`element.innerText`	Similar to `textContent`, but respects CSS styling.
`element.value`	Used with input/textarea to get/set user input.

🎯 Event Handling

Method	Description
`element.addEventListener('event', callback)`	Adds an event listener (e.g., click, input, etc.).

`element.removeEventListener('event', callback)`	Removes an event listener.
`event.preventDefault()`	Prevents default browser behavior (e.g., stop form submission).
`event.stopPropagation()`	Stops the event from bubbling up the DOM.

✅ Example

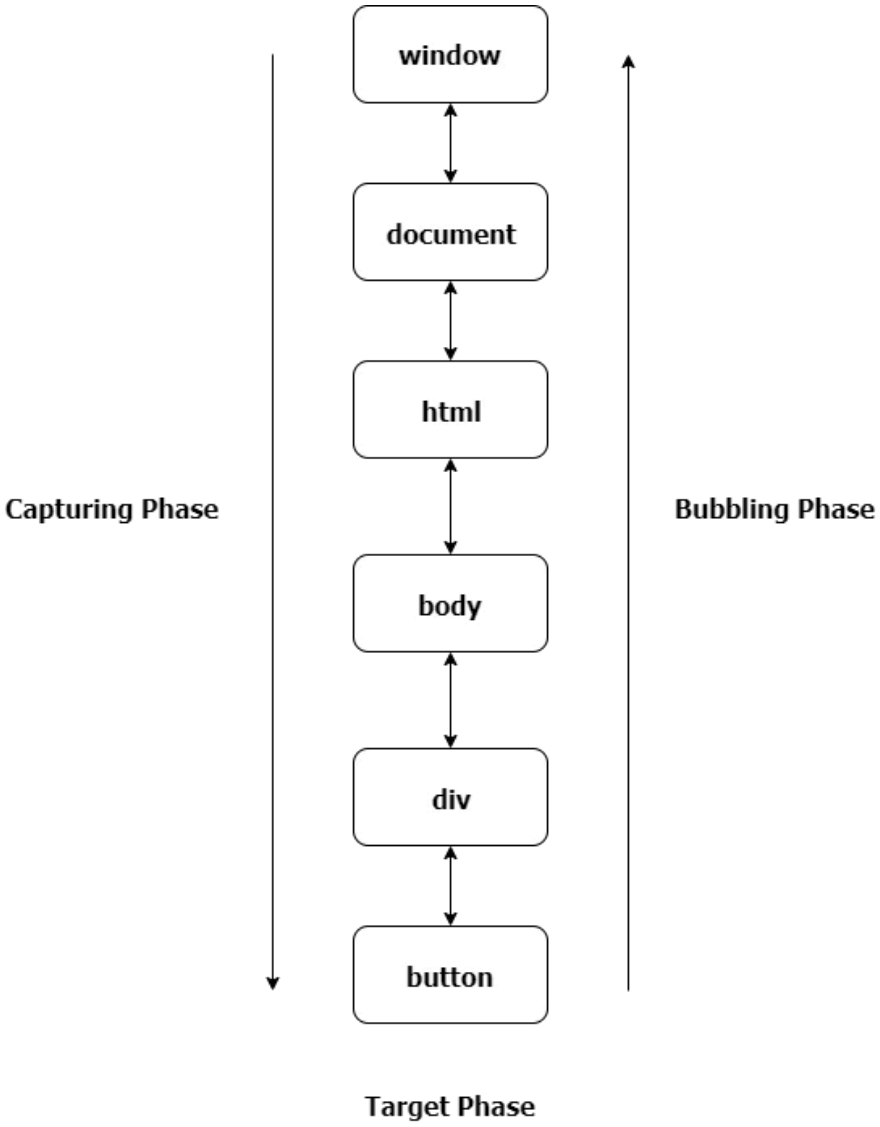
```
const button = document.createElement('button');
button.textContent = "Click Me";
button.classList.add('my-btn');
document.body.appendChild(button);

button.addEventListener('click', () => {
  alert('Button Clicked!');
});
```

3.What is Event Propagation?

When an **event** occurs on a **DOM** element, that **event** does not entirely occur on that just one element. In the **Bubbling Phase**, the **event** bubbles up or it goes to its parent, to its grandparents, to its grandparent's parent until it reaches all the way to the `window` while in the **Capturing Phase** the event starts from the `window` down to the element that triggered the event or the `event.target`.

Event Propagation has **three** phases.

1. Capturing Phase – the event starts from `window` then goes down to every element until it reaches the target element.
2. Target Phase – the event has reached the target element.
3. Bubbling Phase – the event bubbles up from the target element then goes up every element until it reaches the `window`.



4. What's the difference between `event.preventDefault()` and `event.stopPropagation()` methods?

The `event.preventDefault()` method **prevents** the default behavior of an element. If used in a `form` element it **prevents** it from submitting. If used in an `anchor` element it **prevents** it from navigating. If used in a `contextmenu` it **prevents** it from showing or displaying. While the `event.stopPropagation()` method stops the propogation of an event or it stops the event from occurring in the bubbling or capturing phase

5.Why does this code `obj.someprop.x` throw an error?

```
const obj = {};
console.log(obj.someprop.x);
```

This throws an error due to the reason we are trying to access a

`x` property in the `someprop` property which have an `undefined` value. Remember **properties** in an object which does not exist in itself and its **prototype** has a default value of `undefined` and `undefined` has no property `x`.

6.What is event.target ?

In simplest terms, the **event.target** is the element on which the event **occurred** or the element that **triggered** the event.

Sample HTML Markup.

```
<div onclick="clickFunc(event)" style="text-align: center;margin:15px;
border:1px solid red;border-radius:3px;">
  <div style="margin: 25px; border:1px solid royalblue;border-radius:3px;">
    <div style="margin:25px;border:1px solid skyblue;border-radius:3px;">
      <button style="margin:10px">
        Button
      </button>
    </div>
  </div>
 </div>
```

Sample JavaScript.

```
function clickFunc(event) {
 console.log(event.target);
}
```

If you click the button it will log the **button** markup even though we attach the event on the outermost `div` it will always log the **button** so we can conclude that the **event.target** is the element that triggered the event.

7.What's the difference between `==` and `===` ?

↑ The difference between `==` **(abstract equality)** and `===` **(strict equality)** is that the `==` compares by **value** after *coercion* and `===` compares by **value** and **type** without *coercion*.

Let's dig deeper on the `==`. So first let's talk about *coercion*.

coercion is the process of converting a value to another type. As in this case, the `==` does *implicit coercion*. The `==` has some conditions to perform before comparing the two values.

Suppose we have to compare `x == y` values.

1. If `x` and `y` have same type. Then compare them with the `===` operator.
2. If `x` is `null` and `y` is `undefined` then return `true`.
3. If `x` is `undefined` and `y` is `null` then return `true`.
4. If `x` is type `number` and `y` is type `string` Then return `x == toNumber(y)`.
5. If `x` is type `string` and `y` is type `number` Then return `toNumber(x) == y`.
6. If `x` is type `boolean` Then return `toNumber(x) == y`.
7. If `y` is type `boolean` Then return `x == toNumber(y)`.
8. If `x` is either `string`, `symbol` or `number` and `y` is type `object` Then return `x == toPrimitive(y)`.
9. If `x` is either `object` and `x` is either `string`, `symbol` Then return `toPrimitive(x) == y`.
10. Return `false`.

Note: `toPrimitive` uses first the `valueOf` method then the `toString` method in objects to get the primitive value of that object.

Let's have examples.

x	y	x == y
5	5	true
1	'1'	true
null	undefined	true
0	false	true
'1,2'	[1,2]	true
'[object Object]'	{}	true

These examples all return `true`.

The **first example** goes to **condition one** because `x` and `y` have the same type and value.

The **second example** goes to **condition four** `y` is converted to a `number` before comparing.

The **third example** goes to **condition two**.

The **fourth example** goes to **condition seven** because `y` is `boolean`.

The **fifth example** goes to **condition eight**. The array is converted to a `string` using the `toString()` method which returns `1,2`.

The **last example** goes to **condition ten**. The object is converted to a `string` using the `toString()` method which returns `[object Object]`.

x	y	x === y
5	5	true

`1`	`'1'`	`false`
`null`	`undefined`	`false`
`0`	`false`	`false`
`'1,2'`	`[1,2]`	`false`
`'[object Object]'`	`{}`	`false`

If we use the `===` operator all the comparisons except for the first example will return `false` because they don't have the same type while the first example will return `true` because the two have the same type and value.

8. What is Hoisting?

Hoisting is the term used to describe the moving of *variables* and *functions* to the top of their *(global or function)* scope on where we define that variable or function.

Ok to understand **Hoisting**, I have to explain the *execution context*.

The **Execution Context** is the "environment of code" that is currently executing. The **Execution Context** has two phases *compilation* and *execution*.

Compilation - in this phase it gets all the *function declarations* and *hoists* them up to the top of their scope so we can reference them later and gets all *variables declaration* **(declare with the var keyword)** and also *hoists* them up and give them a default value of *undefined*.

Execution - in this phase it assigns values to the variables *hoisted* earlier and it *executes* or *invokes* functions **(methods in objects)**.

Note: only **function declarations** and variables declared with the *var* keyword are *hoisted* not **function expressions** or **arrow functions**, `let` and `const` keywords.

Ok, suppose we have an example code in the *global scope* below.

```
console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));

function greet(name){
  return 'Hello ' + name + '!';
}

var y;
```

This code logs `undefined`, `1`, `Hello Mark!` respectively.

So the *compilation* phase would look like this.

```
function greet(name) {
  return 'Hello ' + name + '!';
}

var y; //implicit "undefined" assignment

//waiting for "compilation" phase to finish

//then start "execution" phase
/*
console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));
*/
```

for example purposes, I commented on the *assignment* of variable and *function call*.

After the *compilation* phase finishes it starts the *execution* phase invoking methods and assigns values to variables.

```
function greet(name) {
  return 'Hello ' + name + '!';
}

var y;

//start "execution" phase

console.log(y);
y = 1;
console.log(y);
console.log(greet("Mark"));
```

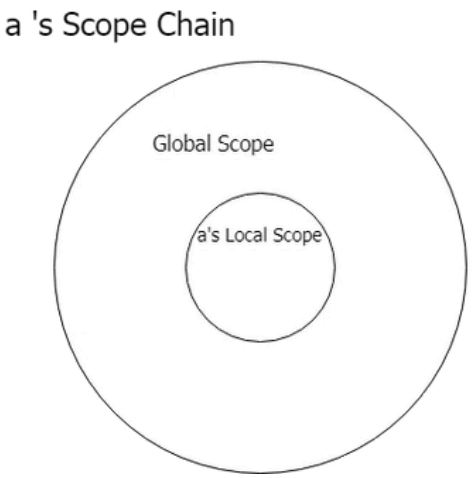
9.What are Closures?

Closures is simply the ability of a function at the time of declaration to remember the references of variables and parameters on its current scope, on its parent function scope, on its parent's parent function scope until it reaches the global scope with the help of **Scope Chain**. Basically it is the **Scope** created when the function was declared.

Examples are a great way to explain closures.

```
//Global's Scope
var globalVar = "abc";

function a(){
//testClosures's Scope
  console.log(globalVar);
}

a(); //logs "abc"
/* Scope Chain
   Inside a function perspective

   a's scope → global's scope
*/
```

In this example, when we declare the `a` function the **Global Scope** is part of `a's` *closure*.



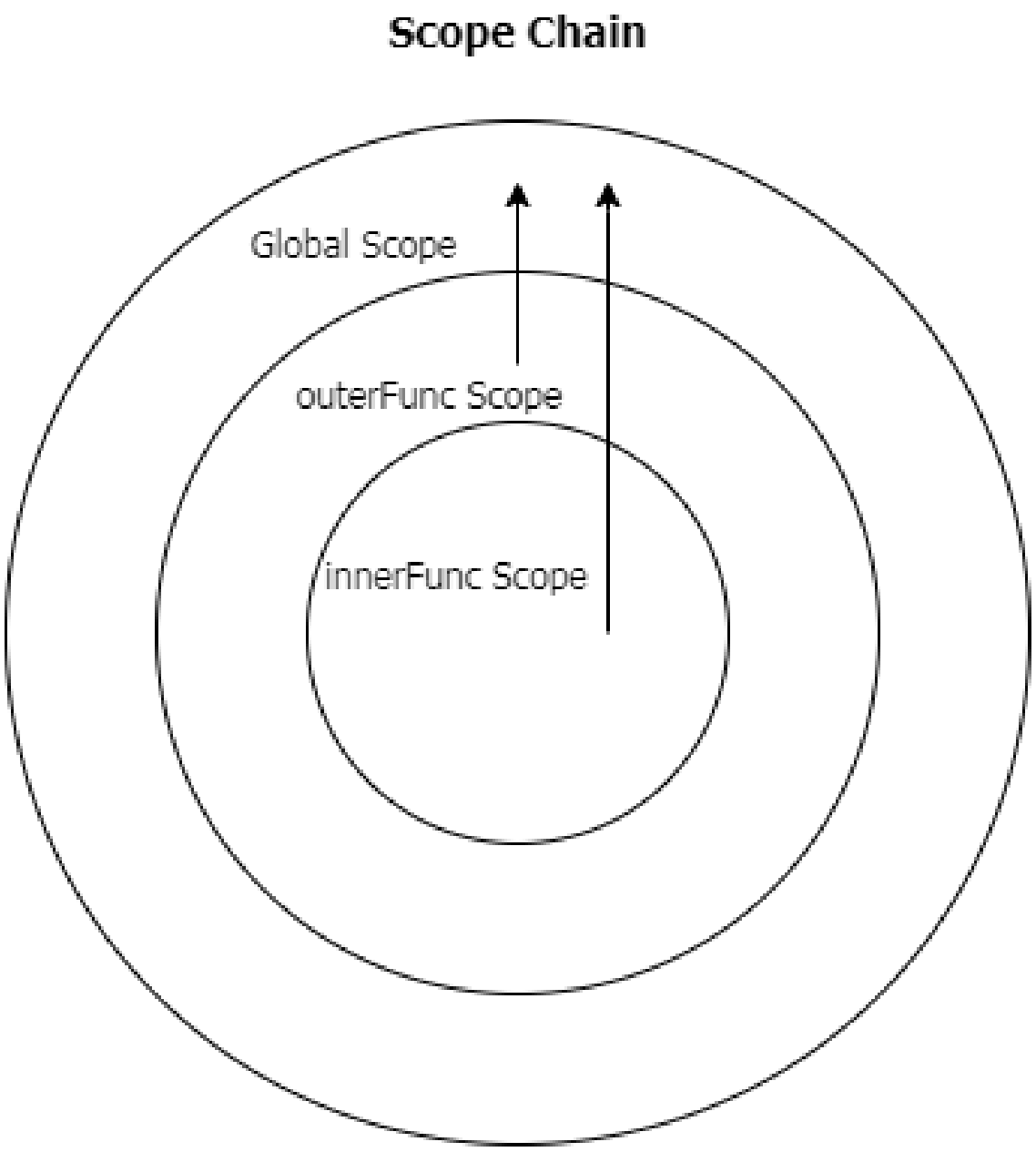
The reason for the variable `globalVar` which does not have a value in the image because of the reason that the value of that variable can change based on **where** and **when** we invoke the `a` function.

But in our example above the `globalVar` variable will have the value of **abc**.

Ok, let's have a complex example.

```
var globalVar = "global";
var outerVar = "outer"

function outerFunc(outerParam) {
  function innerFunc(innerParam) {
```

```
    console.log(globalVar, outerParam, innerParam);
  }
  return innerFunc;
}

const x = outerFunc(outerVar);
outerVar = "outer-2";
globalVar = "guess"
x("inner");
```



This will print "guess outer inner". The explanation for this is that when we invoke the `outerFunc` function and assigned the returned value the `innerFunc` function to the variable `x`, the `outerParam` will have a value of **outer** even though we assign a new value **outer-2** to the `outerVar` variable because

the reassignment happened after the invocation of the `outer` function and in that time when we invoke the `outerFunc` function it's look up the value of `outerVar` in the **Scope Chain**, the `outerVar` will have a value of **"outer"**. Now, when we invoke the `x` variable which have a reference to the `innerFunc`, the

`innerParam` will have a value of **inner** because thats the value we pass in the invocation and the `globalVar` variable will have a value of **guess** because before the invocation of the `x` variable we assign a new value to the `globalVar` and at the time of invocation `x` the value of `globalVar` in the **Scope Chain** is **guess**.

We have an example that demonstrates a problem of not understanding closure correctly.

```
const arrFuncs = [];
for(var i = 0; i < 5; i++){
  arrFuncs.push(function (){
    return i;
  });
}
console.log(i); // i is 5

for (let i = 0; i < arrFuncs.length; i++) {
  console.log(arrFuncs[i]()); // all logs "5"
}
```

This code is not working as we expected because of **Closures**.

The `var` keyword makes a global variable and when we push a function

we return the global variable `i`. So when we call one of those functions in that array after the loop it logs `5` because we get

the current value of `i` which is `5` and we can access it because it's a global variable.
Because **Closures** keeps the **references** of that variable not its **values** at the time of it's creation. We can solve this using **IIFES** or changing the `var` keyword to `let` for block-scoping.

10.What's the value of `this` in JavaScript?

Basically, `this` refers to the value of the object that is currently executing or invoking the function. I say **currently** due to the reason that the value of **this** changes depending on the context on which we use it and where we use it.

```
const carDetails = {
  name: "Ford Mustang",
```

```
  yearBought: 2005,
  getName(){
    return this.name;
  },
  isRegistered: true
};

console.log(carDetails.getName()); // logs Ford Mustang
```

This is what we would normally expect because in the **getName** method we return `this.name` , `this` in this context refers to the object which is the `carDetails` object that is currently the "owner" object of the function executing.

Ok, Let's some add some code to make it weird. Below the `console.log` statement add this three lines of code

```
var name = "Ford Ranger";
var getCarName = carDetails.getName;

console.log(getCarName()); // logs Ford Ranger
```

The second `console.log` statement prints the word **Ford Ranger** which is weird because in our first `console.log` statement it printed **Ford Mustang**. The reason to this is that the `getCarName` method has a different "owner" object that is the `window` object. Declaring variables with the `var` keyword in the global scope attaches properties in the `window` object with the same name as the variables. Remember `this` in the global scope refers to the `window` object when `"use strict"` is not used.

```
console.log(getCarName === window.getCarName); //logs true
console.log(getCarName === this.getCarName); // logs true
```

`this` and `window` in this example refer to the same object.

One way of solving this problem is by using the `apply` and `call` methods in functions.

```
console.log(getCarName.apply(carDetails)); //logs Ford Mustang
console.log(getCarName.call(carDetails));  //logs Ford Mustang
```

The `apply` and `call` methods expects the first parameter to be an object which would be value of `this` inside that function.

IIFE or **Immediately Invoked Function Expression**, Functions that are declared in the global scope, **Anonymous Functions** and Inner functions in methods inside an object has a default of **this** which points to the **window** object.

```
(function (){
  console.log(this);
```

```
})(); //logs the "window" object

function iHateThis(){
  console.log(this);
}

iHateThis(); //logs the "window" object

const myFavoriteObj = {
 guessThis(){
   function getThis(){
    console.log(this);
   }
   getThis();
 },
 name: 'Marko Polo',
 thisIsAnnoying(callback){
  callback();
 }
};


myFavoriteObj.guessThis(); //logs the "window" object
myFavoriteObj.thisIsAnnoying(function (){
 console.log(this); //logs the "window" object
});
```

If we want to get the value of the `name` property which is **Marko Polo** in the `myFavoriteObj` object there are two ways to solve this.

First, we save the value of `this` in a variable.

```
const myFavoriteObj = {
 guessThis(){
    const self = this; //saves the this value to the "self" variable
    function getName(){
     console.log(self.name);
    }
    getName();
 },
 name: 'Marko Polo',
 thisIsAnnoying(callback){
  callback();
```

```
  }
};
```

In this image we save the value of `this` which would be the `myFavoriteObj` object. So we can access it inside the `getName` inner function.

Second, we use **ES6 Arrow Functions**.

```
const myFavoriteObj = {
  guessThis(){
      const getName = () => {
        //copies the value of "this" outside of this arrow function
        console.log(this.name);
      }
      getName();
  },
  name: 'Marko Polo',
  thisIsAnnoying(callback){
    callback();
  }
};
```

Arrow Functions does not have its own `this`. It copies the value of `this` of the enclosing lexical scope or in this example the value of `this` outside the `getName` inner function which would be the `myFavoriteObj` object. We can also determine the value of `this` on how the function is invoked.

11. What are Higher Order Functions?

Higher-Order Function are functions that can return a function or receive argument or arguments which have a value of a function.

```
function higherOrderFunction(param,callback){
   return callback(param);
}
```

12. Why are functions called First-class Objects?

Functions in JavaScript are **First-class Objects** because they are treated as any other value in the language. They can be assigned to **variables**, they can be **properties of an object** which are called **methods**, they can be an **item in array**, they can be **passed as arguments to a function**, and they can be **returned as values of a function**. The only difference between a function and any other value in **JavaScript** is that **functions** can be invoked or called.

13. Implement the `Array.prototype.map` method by hand.

```
function map(arr, mapCallback) {
  // First, we check if the parameters passed are right.
  if (!Array.isArray(arr) || !arr.length || typeof mapCallback !== 'function') {
    return [];
  } else {
    let result = [];
    // We're making a results array every time we call this function
    // because we don't want to mutate the original array.
    for (let i = 0, len = arr.length; i < len; i++) {
      result.push(mapCallback(arr[i], i, arr));
      // push the result of the mapCallback in the 'result' array
    }
    return result; // return the result array
  }
}

const numbers = [1, 2, 3, 4];
const doubled = map(numbers, function(num) {
  return num * 2;
});
console.log(doubled); // [2, 4, 6, 8]
```

As the MDN description of the `Array.prototype.map` method.

The map() method creates a new array with the results of calling a provided function on every element in the calling array.

other way

Custom Implementation of `map`:

```
Array.prototype.myMap = function(callback) {
  // Ensure 'this' is an array
  if (!Array.isArray(this)) {
    throw new TypeError('Called on non-array');
  }

  // Ensure callback is a function
  if (typeof callback !== 'function') {
    throw new TypeError(callback + ' is not a function');
```

```
  }

  const result = [];

  for (let i = 0; i < this.length; i++) {
    // Check if the index exists (to skip holes in sparse arrays)
    if (i in this) {
      result.push(callback(this[i], i, this));
    }
  }

  return result;
};
```

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

const doubled = numbers.myMap(function(num) {
  return num * 2;
});

console.log(doubled); // [2, 4, 6, 8]
```

14. Implement the `Array.prototype.filter` method by hand.

```
function filter(arr, filterCallback) {
  // First, we check if the parameters passed are right.
  if (!Array.isArray(arr) || !arr.length || typeof filterCallback !== 'function')
  {
    return [];
  } else {
    let result = [];
    // We're making a results array every time we call this function
    // because we don't want to mutate the original array.
    for (let i = 0, len = arr.length; i < len; i++) {
      // check if the return value of the filterCallback is true or "truthy"
      if (filterCallback(arr[i], i, arr)) {
      // push the current item in the 'result' array if the condition is true
        result.push(arr[i]);
      }
    }
    return result; // return the result array
  }
```

```
}

const numbers = [1, 2, 3, 4, 5];
const evens = filter(numbers, function(num) {
 return num % 2 === 0;
});
console.log(evens); // [2, 4]
```

As the MDN description of the `Array.prototype.filter` method.

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

15. Implement the `Array.prototype.reduce` method by hand.

```
function reduce(arr, reduceCallback, initialValue) {
 // First, we check if the parameters passed are right.
 if (!Array.isArray(arr) || !arr.length || typeof reduceCallback !== 'function')
 {
  return [];
 } else {
  // If no initialValue has been passed to the function we're gonna use the
  let hasInitialValue = initialValue !== undefined;
  let value = hasInitialValue ? initialValue : arr[0];
  // first array item as the initialValue

  // Then we're gonna start looping at index 1 if there is no
  // initialValue has been passed to the function else we start at 0 if
  // there is an initialValue.
  for (let i = hasInitialValue ? 0 : 1, len = arr.length; i < len; i++) {
   // Then for every iteration we assign the result of the
   // reduceCallback to the variable value.
   value = reduceCallback(value, arr[i], i, arr);
  }
  return value;
 }
}

const numbers = [1, 2, 3, 4];
const sum = reduce(numbers, function(acc, curr) {
 return acc + curr;
}, 0);
console.log(sum); // 10
```

As the MDN description of the `Array.prototype.reduce` method.

The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

16.What are Classes?

Classes is the new way of writing *constructor functions* in **JavaScript**. It is *syntactic sugar* for using *constructor functions*, it still uses **prototypes** and **Prototype-Based Inheritance** under the hood.

```
//ES5 Version
function Person(firstName, lastName, age, address){
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.address = address;
}

Person.self = function(){
 return this;
}

Person.prototype.toString = function(){
 return "[object Person]";
}

Person.prototype.getFullName = function (){
 return this.firstName + " " + this.lastName;
}

//ES6 Version
class Person {
    constructor(firstName, lastName, age, address){
        this.lastName = lastName;
        this.firstName = firstName;
        this.age = age;
        this.address = address;
    }

    static self() {
      return this;
    }

    toString(){
```

```
        return "[object Person]";
      }

      getFullName(){
        return `${this.firstName} ${this.lastName}`;
      }
  }
```

Overriding Methods and **Inheriting from another class.**

```
//ES5 Version
Employee.prototype = Object.create(Person.prototype);

function Employee(firstName, lastName, age, address, jobTitle, yearStarted) {
  Person.call(this, firstName, lastName, age, address);
  this.jobTitle = jobTitle;
  this.yearStarted = yearStarted;
}

Employee.prototype.describe = function () {
  return `I am ${this.getFullName()} and I have a position of ${this.jobTitle} and I started at ${this.y
earStarted}`;
}

Employee.prototype.toString = function () {
  return "[object Employee]";
}

//ES6 Version
class Employee extends Person { //Inherits from "Person" class
  constructor(firstName, lastName, age, address, jobTitle, yearStarted) {
    super(firstName, lastName, age, address);
    this.jobTitle = jobTitle;
    this.yearStarted = yearStarted;
  }

  describe() {
    return `I am ${this.getFullName()} and I have a position of ${this.jobTitle} and I started at ${this.
yearStarted}`;
  }

  toString() { // Overriding the "toString" method of "Person"
    return "[object Employee]";
  }
```

```
}
```

So how do we know that it uses *prototypes* under the hood?

```
class Something {

}

function AnotherSomething(){

}
const as = new AnotherSomething();
const s = new Something();

console.log(typeof Something); // logs "function"
console.log(typeof AnotherSomething); // logs "function"
console.log(as.toString()); // logs "[object Object]"
console.log(as.toString()); // logs "[object Object]"
console.log(as.toString === Object.prototype.toString);
console.log(s.toString === Object.prototype.toString);
// both logs return true indicating that we are still using
// prototypes under the hoods because the Object.prototype is
// the last part of the Prototype Chain and "Something"
// and "AnotherSomething" both inherit from Object.prototype
```

17.What is a Callback function?

A **Callback** function is a function that is gonna get called at a later point in time.

```
const btnAdd = document.getElementById('btnAdd');

btnAdd.addEventListener('click', function clickCallback(e) {
   // do something useless
});
```

In this example, we wait for the `click event` in the element with an id of **btnAdd**, if it is `clicked`, the `clickCallback` function is executed. A **Callback** function adds some functionality to some data or event. The `reduce`, `filter` and `map` methods in **Array** expects a callback as a parameter. A good analogy for a callback is when you call someone and if they don't answer you leave a message and you expect them to **callback**. The act of calling someone or leaving a **message** is the **event or data** and the **callback** is the **action that you expect to occur later**.

18. What are Higher Order Functions?

Higher-Order Function are functions that can return a function or receive argument or arguments which have a value of a function.

```
function higherOrderFunction(param,callback){
   return callback(param);
}`
```

19.What is a Callback function?

A **Callback** function is a function that is gonna get called at a later point in time.

```
const btnAdd = document.getElementById('btnAdd');

btnAdd.addEventListener('click', function clickCallback(e) {
   // do something useless
});
```

In this example, we wait for the `click event` in the element with an id of **btnAdd**, if it is `clicked`, the `clickCallback` function is executed. A **Callback** function adds some functionality to some data or event. The `reduce`, `filter` and `map` methods in **Array** expects a callback as a parameter. A good analogy for a callback is when you call someone and if they don't answer you leave a message and you expect them to **callback**. The act of calling someone or leaving a **message** is the **event or data** and the **callback** is the **action that you expect to occur later**.

20.What are ways of handling asynchronous operations in javascript ?

In JavaScript, handling **asynchronous operations** is essential when working with tasks like API calls, file reading, timers, or any operations that take time to complete. JavaScript provides several ways to manage async code efficiently:

1. Callbacks

A **callback** is a function passed into another function to be called once an operation is complete.

```
javascript
CopyEdit
function fetchData(callback) {
  setTimeout(() => {
```

```
    callback('Data received');
  }, 1000);
}

fetchData((data) => {
  console.log(data); // Logs after 1 second
});
```

> ❌ Can lead to callback hell (deeply nested callbacks), which is hard to manage.

2. Promises

A **Promise** represents a value that will be available now, later, or never.

```
javascript
CopyEdit
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Data received');
    }, 1000);
  });
};

fetchData().then(data => console.log(data));
```

> ✅ Cleaner than callbacks
>
> ❌ Can still get messy with multiple `.then()` calls

3. Async/Await

Introduced in ES2017, `async/await` makes asynchronous code look synchronous.

```
javascript
CopyEdit
const fetchData = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Data received');
    }, 1000);
```

```
  });
};

async function getData() {
  const result = await fetchData();
  console.log(result);
}

getData();
```

> ✅ Very clean and readable
>
> ❌ Must be used inside an `async` function

4. Promise.all / Promise.allSettled / Promise.race

Used for handling **multiple asynchronous operations** in parallel.

`Promise.all` (waits for all promises)

```
javascript
CopyEdit
Promise.all([fetchData1(), fetchData2()])
  .then(([res1, res2]) => {
    console.log(res1, res2);
  });
```

`Promise.race` (resolves/rejects as soon as the first promise settles)

`Promise.allSettled` (returns results of all, regardless of success/failure)

5. Observable (Advanced)

Used in **RxJS** (Reactive Programming) for handling complex async data streams (e.g., Angular apps).

```
javascript
CopyEdit
import { of } from 'rxjs';

of('Hello').subscribe((data) => {
  console.log(data);
});
```

> ✅ Powerful for real-time data streams
>
> ❌ Requires learning RxJS