

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

↑ Before understanding the differences between `undefined` and `null` we must understand the similarities between them.

- They belong to **JavaScript's 7 primitive types**.

```
let primitiveTypes = ['string', 'number', 'null', 'undefined', 'boolean', 'symbol', 'bigint'];
```

- They are **falsy** values. Values that evaluated to false when converting it to boolean using `Boolean(value)` or `!!value`.

```
console.log(!!null); //logs false
console.log(!!undefined); //logs false

console.log(Boolean(null)); //logs false
console.log(Boolean(undefined)); //logs false
```

Ok, let's talk about the differences.

- `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.

```
let _thisIsUndefined;
const doNothing = () => {};
const someObj = {
  a : "ay",
  b : "bee",
  c : "si"
};

console.log(_thisIsUndefined); //logs undefined
console.log(doNothing()); //logs undefined
console.log(someObj["d"]); //logs undefined
```

- `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.

```
fs.readFile('path/to/file', (e,data) => {
  console.log(e); //it logs null when no error occurred
  if(e){
    console.log(e);
  }
  console.log(data);
});
```

When comparing `null` and `undefined` we get `true` when using `==` and `false` when using `===`. You can read the reason [here](#).

```
console.log(null == undefined); // logs true
console.log(null === undefined); // logs false
```

## 2. What does the && operator do?

↑ The && or **Logical AND** operator finds the first falsy expression in its operands and returns it and if it does not find any falsy expression it returns the last expression. It employs short-circuiting to prevent unnecessary work. I've used this in the `catch` block when closing database connection in one of my projects.

```
console.log(false && 1 && []); //logs false
console.log(" " && true && 5); //logs 5
```

Using **if** statements.

```
const router: Router = Router();

router.get('/endpoint', (req: Request, res: Response) => {
  let conMobile: PoolConnection;
  try {
    //do some db operations
  } catch (e) {
    if (conMobile) {
      conMobile.release();
    }
  }
});
```

Using **&&** operator.

```
const router: Router = Router();

router.get('/endpoint', (req: Request, res: Response) => {
  let conMobile: PoolConnection;
  try {
    //do some db operations
  } catch (e) {
    conMobile && conMobile.release()
  }
});
```

## 3. What does the || operator do?

↑ The || or **Logical OR** operator finds the first truthy expression in its operands and returns it. This too employs short-circuiting to prevent unnecessary work. It was used before to initialize default parameter values IN functions before **ES6 Default function parameters** was supported.

```
console.log(null || 1 || undefined); //logs 1
```

```
function logName(name) {
  var n = name || "Mark";
  console.log(n);
}
```

```
}
```

```
logName(); //logs "Mark"
```

#### 4. Is using the + or unary plus operator the fastest way in converting a string to a number?

↑ According to [MDN Documentation](#) the + is the fastest way of converting a string to a number because it does not perform any operations on the value if it is already a number.

#### 5. 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.

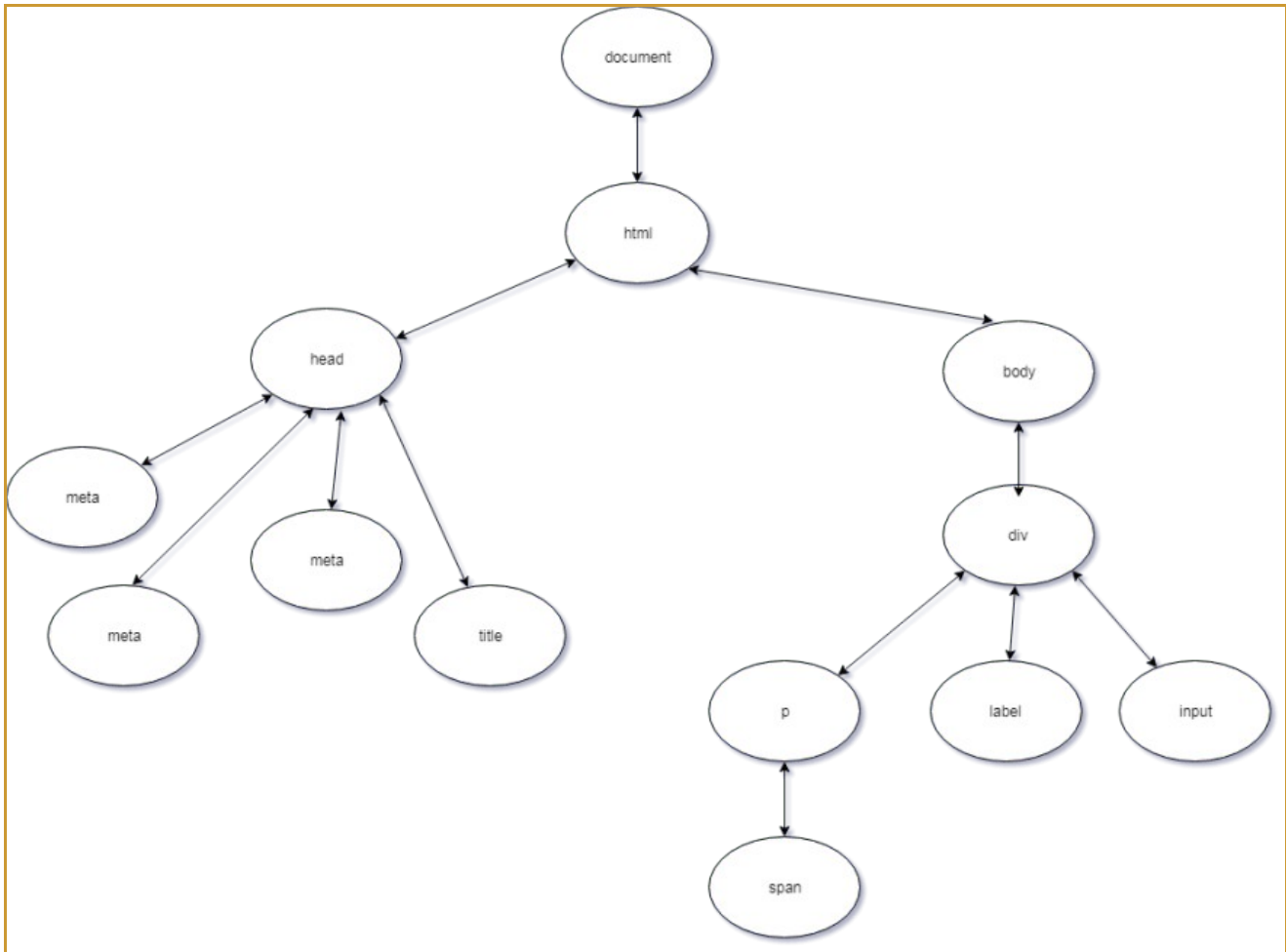
```
<!DOCTYPE html>
<html lang="en">

  <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.



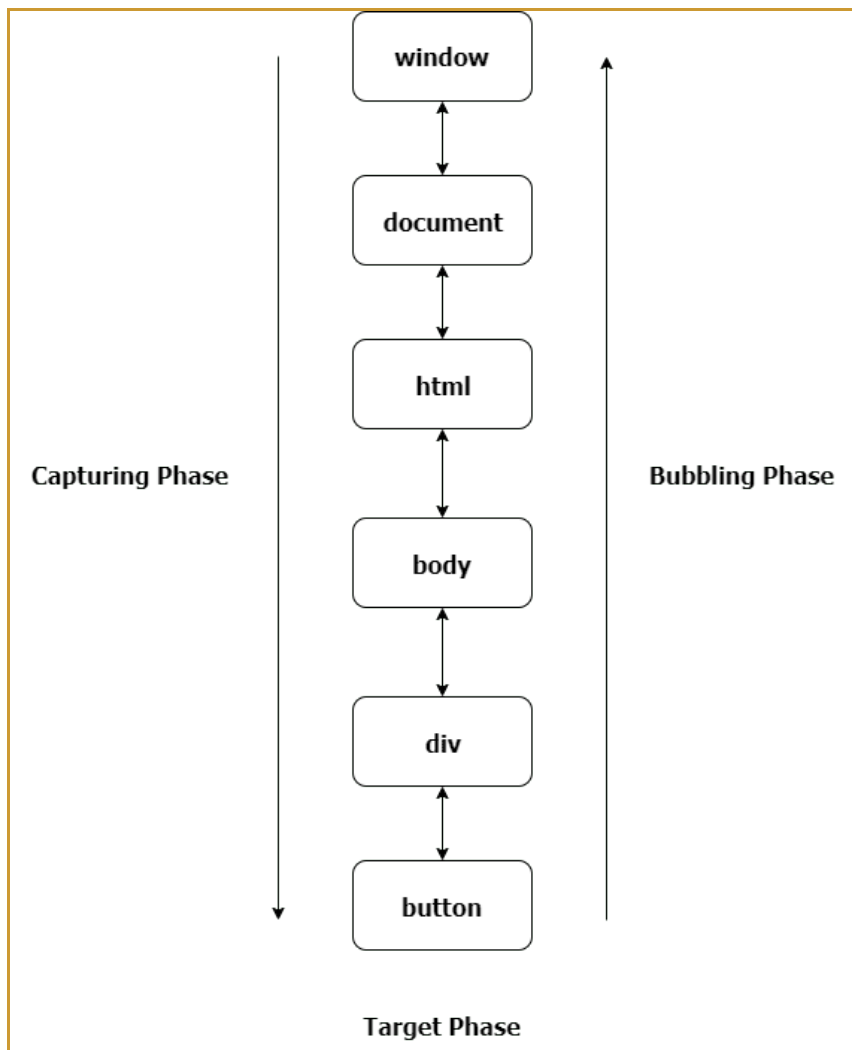
The `document` object in **JavaScript** represents the **DOM**. It provides us many methods that we can use to selecting elements to update element contents and many more.

## 6. 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`.



## 7. What's Event Bubbling?

↑ 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**.

If we have an example markup like this.

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
```

And our js code.

```
function addEvent(e1, event, callback, isCapture = false) {
  if (!e1 || !event || !callback || typeof callback !== 'function') return;
  if (typeof e1 === 'string') {
    e1 = document.querySelector(e1);
  }
  e1.addEventListener(event, callback, isCapture);
}
```

```

}

addEventListener(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
  });

  addEvent(window, 'click', function (e) {
    console.log('window');
  });

});

```

The `addEventListener` method has a third optional parameter **useCapture** with a default value of `false` the event will occur in the **Bubbling phase** if `true` the event will occur in the **Capturing Phase**. If we click on the `child` element it logs `child`, `parent`, `grandparent`, `html`, `document` and `window` respectively on the **console**. This is **Event Bubbling**.

## 8. What's Event Capturing?

↑ When an **event** occurs on a **DOM** element, that **event** does not entirely occur on that just one element. In **Capturing Phase**, the event starts from the `window` all the way down to the element that triggered the event.

If we have an example markup like this.

```
<div class="grandparent">
```

```

    <div class="parent">
      <div class="child">1</div>
    </div>
  </div>

```

And our js code.

```

function addEvent(e1, event, callback, isCapture = false) {
  if (!e1 || !event || !callback || typeof callback !== 'function') return;
  if (typeof e1 === 'string') {
    e1 = document.querySelector(e1);
  };
  e1.addEventListener(event, callback, isCapture);
}

```

```

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

```

```

    addEvent(child, 'click', function (e) {
      console.log('child');
    }, true);

```

```

    addEvent(parent, 'click', function (e) {
      console.log('parent');
    }, true);

```

```

    addEvent(grandparent, 'click', function (e) {
      console.log('grandparent');
    }, true);

```

```

    addEvent(document, 'click', function (e) {
      console.log('document');
    }, true);

```

```

    addEvent('html', 'click', function (e) {
      console.log('html');
    }, true)

```

```

    addEvent(window, 'click', function (e) {
      console.log('window');
    }, true)

```

```

  });

```

The `addEventListener` method has a third optional parameter **useCapture** with a default value of `false` the event will occur in the **Bubbling phase** if `true` the event will occur in the **Capturing Phase**.

If we click on the child element it

logs window, document, html, grandparent and parent and child respectively on the **console**. This is **Event Capturing**.

9. 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 context menu it **prevents** it from showing or displaying. While the `event.stopPropagation()` method stops the propagation of an event or it stops the event from occurring in the bubbling or capturing phase.

10. How to know if the `event.preventDefault()` method was used in an element?

↑ We can use the `event.defaultPrevented` property in the event object. It returns a `boolean` indicating if the `event.preventDefault()` was called in a particular element.

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

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

↑ Obviously, 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`.

12. 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>
```



```
</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.

### 13. What is **event.currentTarget**?

↑ The **event.currentTarget** is the element on which we attach the event handler **explicitly**.

Copying the markup in **Question 12**.

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>
```

And changing our the **JS** a little bit.

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

If you click the button it will log the outermost **div** markup even though we click the button. In this example, we can conclude that the **event.currentTarget** is the element on which we attach the event handler.

### 14. 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  $x == \text{toNumber}(y)$ .
5. If  $x$  is type string and  $y$  is type number Then return  $\text{toNumber}(x) == y$ .
6. If  $x$  is type boolean Then return  $\text{toNumber}(x) == y$ .
7. If  $y$  is type boolean Then return  $x == \text{toNumber}(y)$ .
8. If  $x$  is either string,symbol or number and  $y$  is type object Then return  $x == \text{toPrimitive}(y)$ .
9. If  $x$  is either object and  $x$  is either string,symbol Then return  $\text{toPrimitive}(x) == y$ .
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

x	y	x === y
'[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.

## 15. Why does it return **false** when comparing two similar objects in JavaScript?

↑ Suppose we have an example below.

```
let a = { a: 1 };
let b = { a: 1 };
let c = a;
```

```
console.log(a === b); // logs false even though they have the same property
console.log(a === c); // logs true hmm
```

**JavaScript** compares objects and primitives differently. In primitives it compares them by **value** while in objects it compares them by **reference** or the **address in memory where the variable is stored**. That's why the first `console.log` statement returns `false` and the second `console.log` statement returns `true`. `a` and `c` have the same reference and `a` and `b` are not.

## 16. What does the **!!** operator do?

↑ The **Double NOT** operator or `!!` coerces the value on the right side into a boolean. basically it's a fancy way of converting a value into a boolean.

```
console.log(!!null); //logs false
console.log(!!undefined); //logs false
console.log(!!''); //logs false
console.log(!!0); //logs false
console.log(!!NaN); //logs false
console.log(!!' '); //logs true
console.log(!!{}); //logs true
console.log(!![]); //logs true
console.log(!!1); //logs true
console.log(!![].length); //logs false
```

## 17. How to evaluate multiple expressions in one line?

↑ We can use the `,` or comma operator to evaluate multiple expressions in one line. It evaluates from left-to-right and returns the value of the last item on the right or the last operand.

```
let x = 5;
```

```
x = (x++ , x = addFive(x), x *= 2, x -= 5, x += 10);
```

```
function addFive(num) {  
  return num + 5;  
}
```

If you log the value of `x` it would be **27**. First, we **increment** the value of `x` it would be **6**, then we invoke the function `addFive(6)` and pass the 6 as a parameter and assign the result to `x` the new value of `x` would be **11**. After that, we multiply the current value of `x` to 2 and assign it to `x` the updated value of `x` would be **22**. Then, we subtract the current value of `x` to 5 and assign the result to `x` the updated value would be **17**. And lastly, we increment the value of `x` by 10 and assign the updated value to `x` now the value of `x` would be **27**.

## 18. 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"));

```

## 19. What is **Scope**?

↑ **Scope** in JavaScript is the **area** where we have valid access to variables or functions. JavaScript has three types of Scopes. **Global Scope**, **Function Scope**, and **Block Scope(ES6)**.

- **Global Scope** - variables or functions declared in the global namespace are in the global scope and therefore is accessible everywhere in our code.

```

//global namespace
var g = "global";

function globalFunc(){
    function innerFunc(){
        console.log(g); // can access "g" because "g" is a global variable
    }
    innerFunc();
}

```

- **Function Scope** - variables, functions and parameters declared within a function are accessible inside that function but not outside of it.

```
function myFavoriteFunc(a) {
    if (true) {
        var b = "Hello " + a;
    }
    return b;
}
myFavoriteFunc("world");
```

```
console.log(a); // Throws a ReferenceError "a" is not defined
console.log(b); // does not continue here
```

- **Block Scope** - variables (**let, const**) declared within a block {} can only be access within it.

```
function testBlock(){
    if(true){
        let z = 5;
    }
    return z;
}
```

```
testBlock(); // Throws a ReferenceError "z" is not defined
```

**Scope** is also a set of rules for finding variables. If a variable does not exist in the **current scope** it **look ups** and searches for a variable in the **outer scope** and if does not exist again it **looks up** again until it reaches the **global scope** if the variable exists then we can use it if not it throws an error. It searches for the **nearest** variable and it stops **searching** or **looking up** once it finds it. This is called **Scope Chain**.

```
/* Scope chain
   Inside inner function perspective

   inner's scope -> outer's scope -> global's scope
*/
```

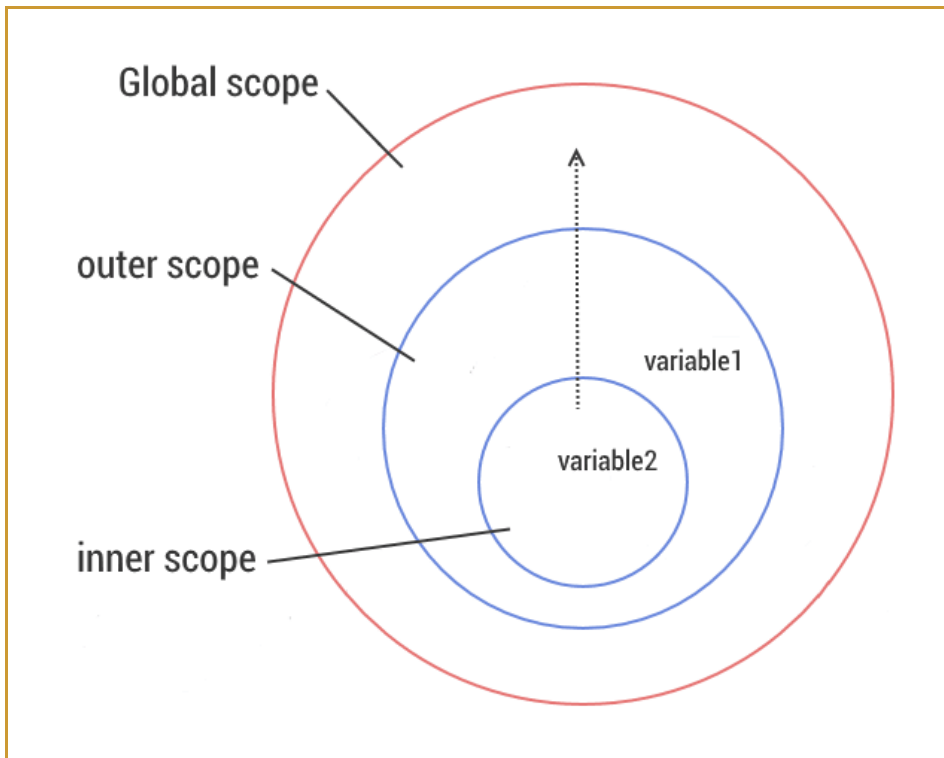
```
//Global Scope
var variable1 = "Comrades";
var variable2 = "Sayonara";

function outer(){
//outer's scope
    var variable1 = "world";
    function inner(){
//inner's scope
        var variable2 = "Hello";
        console.log(variable2 + " " + variable1);
    }
    inner();
}
```

```

}
outer();
// logs Hello world
// because (variable2 = "Hello") and (variable1 = "world") are the nearest
// variables inside inner's scope.

```



## 20. What are **Closures**?

‡ This is probably the hardest question of all these questions because **Closures** is a controversial topic. So I'm gonna explain it from what I understand.

**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";

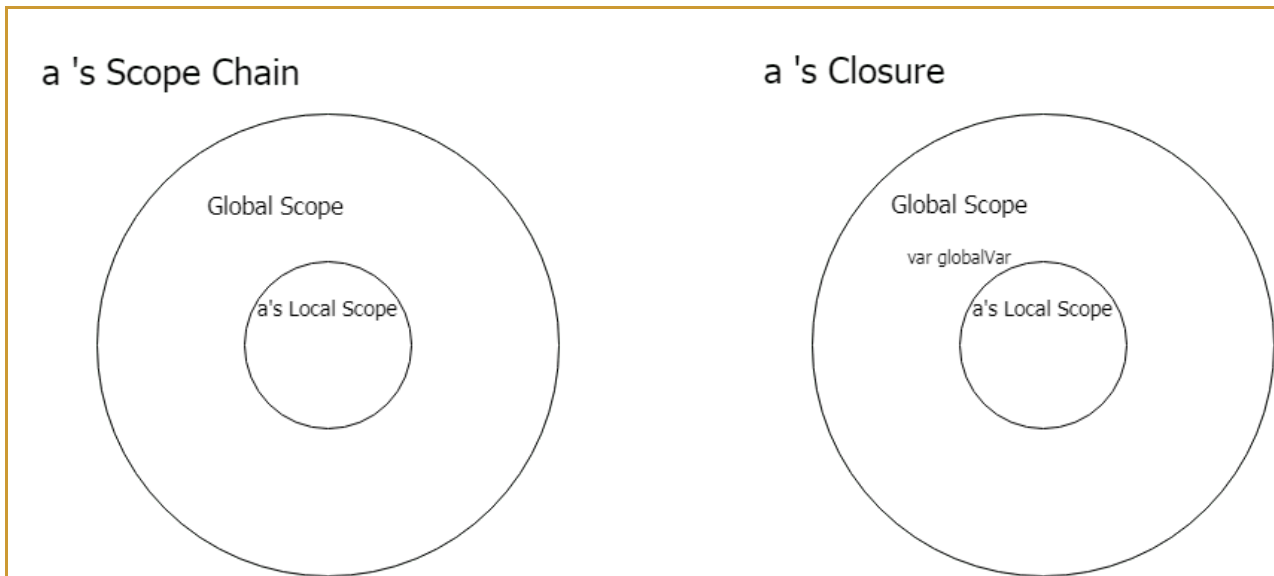
function a(){
  //testClosures's Scope
  console.log(globalVar);
}

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`.

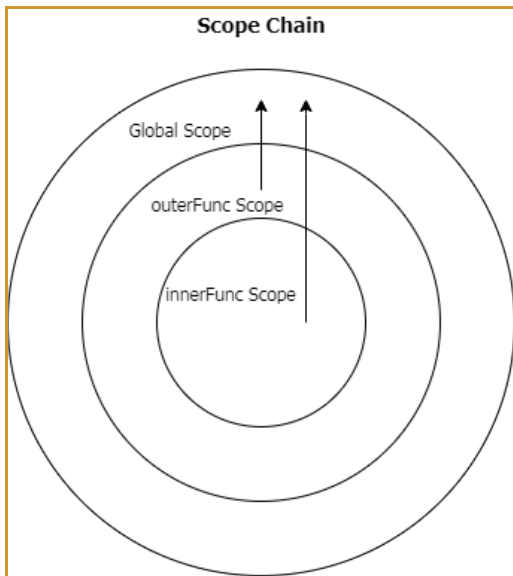
Ok, let's have a complex example.

```
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.

## 21. What are the **falsy** values in JavaScript?

↑

```
const falsyValues = ['', 0, null, undefined, NaN, false];
```

**falsy** values are values that when converted to boolean becomes **false**.

## 22. How to check if a value is **falsy**?

↑ Use the **Boolean** function or the Double NOT operator **!!**

## 23. What does "use strict" do?

↑ "use strict" is a ES5 feature in **JavaScript** that makes our code in **Strict Mode** in functions or entire scripts. **Strict Mode** helps us avoid **bugs** early on in our code and adds restrictions to it.

Restrictions that **Strict Mode** gives us.

- Assigning or Accessing a variable that is not declared.

```
function returnY(){  
  "use strict";  
  y = 123;  
  return y;  
}
```

- Assigning a value to a read-only or non-writable global variable;

```
"use strict";  
var NaN = NaN;  
var undefined = undefined;  
var Infinity = "and beyond";
```

- Deleting an undeletable property.

```
"use strict";  
const obj = {};
```

```
Object.defineProperty(obj, 'x', {  
  value : '1'  
});
```

```
delete obj.x;
```

- Duplicate parameter names.

```
"use strict";
```

```
function someFunc(a, b, b, c){  
  
}
```

- Creating variables with the use of the **eval** function.

```
"use strict";
```

```
eval("var x = 1;");
```

```
console.log(x); //Throws a Reference Error x is not defined
```

- The default value of **this** will be undefined.

```
"use strict";
```

```
function showMeThis(){
  return this;
}
```

```
showMeThis(); //returns undefined
```

There are many more restrictions in **Strict Mode** than these.

## 24. 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.

## 25. What is the prototype of an object?

↑ A **prototype** in simplest terms is a blueprint of an object. It is used as a fallback for **properties** and **methods** if it does exist in the current object. It's the way to share properties and functionality between objects. It's the core concept around JavaScript's **Prototypal Inheritance**.

```
const o = {};
console.log(o.toString()); // logs [object object]
```

Even though the `o.toString` method does not exist in the `o` object it does not throw an error instead returns a string `[object object]`. When a property does not exist in the object it looks into its **prototype** and if it still does not exist it looks into the **prototype's prototype** and so on until it finds a property with the same in the **Prototype Chain**. The end of the **Prototype Chain** is the **Object.prototype**.

```
console.log(o.toString === Object.prototype.toString); // logs true
// which means we we're looking up the Prototype Chain and it reached
// the Object.prototype and used the "toString" method.
```

## 26. What is an **IIFE**, what is the use of it?

↑ An **IIFE** or **Immediately Invoked Function Expression** is a function that is gonna get invoked or executed after its creation or declaration. The syntax for creating **IIFE** is that we wrap the function `() {}` inside a parentheses `()` or the **Grouping Operator** to treat the function as an expression and after that we invoke it with another parentheses `()`. So an **IIFE** looks like this `(function(){}())`.

```
(function () {

}());

(function () {

})();

(function named(params) {

})();

(() => {

})();

(function (global) {

})(window);

const utility = (function () {
  return {
    //utilities
  };
})();
```

These examples are all valid **IIFE**. The second to the last example shows we can pass arguments to an **IIFE** function. The last example shows that we can save the result of the **IIFE** to a variable so we can reference it later.

The best use of **IIFE** is making initialization setup functionalities and to avoid **naming collisions** with other variables in the global scope or polluting the global namespace. Let's have an example.

```
<script src="https://cdnurl.com/somelibrary.js"></script>
```

Suppose we have a link to a library `somelibrary.js` that exposes some global functions that we use in our code but this library has two methods that we don't use `createGraph` and `drawGraph` because these methods have bugs in them. And we want to implement our own `createGraph` and `drawGraph` methods.

- One way of solving this is by changing the structure of our scripts.

```
<script src="https://cdnurl.com/somelibrary.js"></script>
<script>
  function createGraph() {
    // createGraph logic here
  }
  function drawGraph() {
    // drawGraph logic here
  }
</script>
```

When we use this solution we are overriding those two methods that the library gives us.

- Another way of solving this is by changing the name of our own helper functions.

```
<script src="https://cdnurl.com/somelibrary.js"></script>
<script>
  function myCreateGraph() {
    // createGraph logic here
  }
  function myDrawGraph() {
    // drawGraph logic here
  }
</script>
```

When we use this solution we will also change those function calls to the new function names.

- Another way is using an **IIFE**.

```
<script src="https://cdnurl.com/somelibrary.js"></script>
<script>
  const graphUtility = (function () {
    function createGraph() {
      // createGraph logic here
    }
    function drawGraph() {
      // drawGraph logic here
    }
    return {
      createGraph,
      drawGraph
    }
  })();
</script>
```

In this solution, we are making a utility variable that is the result of **IIFE** which returns an object that contains two methods `createGraph` and `drawGraph`.

Another problem that **IIFE** solves is in this example.

```
var li = document.querySelectorAll('.list-group > li');
for (var i = 0, len = li.length; i < len; i++) {
  li[i].addEventListener('click', function (e) {
```

```

        console.log(i);
    })
}

```

Suppose we have a `ul` element with a class of **list-group** and it has 5 `li` child elements. And we want to `console.log` the value of `i` when we **click** an individual `li` element.

But the behavior we want in this code does not work. Instead, it logs 5 in any **click** on an `li` element. The problem we're having is due to how **Closures** work. **Closures** are simply the ability of functions to remember the references of variables on its current scope, on its parent function scope and in the global scope. When we declare variables using the `var` keyword in the global scope, obviously we are making a global variable `i`. So when we click an `li` element it logs 5 because that is the value of `i` when we reference it later in the callback function.

- One solution to this is an **IIFE**.

```

var li = document.querySelectorAll('.list-group > li');
for (var i = 0, len = li.length; i < len; i++) {
    (function (currentIndex) {
        li[currentIndex].addEventListener('click', function (e) {
            console.log(currentIndex);
        })
    })(i);
}

```

This solution works because of the reason that the **IIFE** creates a new scope for every iteration and we capture the value of `i` and pass it into the `currentIndex` parameter so the value of `currentIndex` is different for every iteration when we invoke the **IIFE**.

## 27. What is the use `Function.prototype.apply` method?

↑ The `apply` invokes a function specifying the `this` or the "owner" object of that function on that time of invocation.

```

const details = {
    message: 'Hello world!'
};

function getMessage(){
    return this.message;
}

```

```
getMessage.apply(details); // returns 'Hello world!'
```

This method works like `Function.prototype.call` the only difference is how we pass arguments. In `apply` we pass arguments as an array.

```

const person = {
    name: "Marko Polo"
};

function greeting(greetingMessage) {

```



```

    return `${greetingMessage} ${this.name}`;
}

greeting.apply(person, ['Hello']); // returns "Hello Marko Polo!"

```

## 28. What is the use `Function.prototype.call` method?

↑ The `call` invokes a function specifying the `this` or the "owner" object of that function on that time of invocation.

```

const details = {
  message: 'Hello world!'
};

function getMessage(){
  return this.message;
}

getMessage.call(details); // returns 'Hello world!'

```

This method works like `Function.prototype.apply` the only difference is how we pass arguments. In `call` we pass directly the arguments separating them with a comma , for every argument.

```

const person = {
  name: "Marko Polo"
};

function greeting(greetingMessage) {
  return `${greetingMessage} ${this.name}`;
}

greeting.call(person, 'Hello'); // returns "Hello Marko Polo!"

```

## 29. What's the difference between `Function.prototype.apply` and `Function.prototype.call`?

↑ The only difference between `apply` and `call` is how we pass the **arguments** in the function being called. In `apply` we pass the arguments as an **array** and in `call` we pass the arguments directly in the argument list.

```

const obj1 = {
  result: 0
};

const obj2 = {

```

```

    result:0
  };

  function reduceAdd(){
    let result = 0;
    for(let i = 0, len = arguments.length; i < len; i++){
      result += arguments[i];
    }
    this.result = result;
  }

  reduceAdd.apply(obj1, [1, 2, 3, 4, 5]); // returns 15
  reduceAdd.call(obj2, 1, 2, 3, 4, 5); // returns 15

```

### 30. What is the usage of `Function.prototype.bind`?

↑ The `bind` method returns a new function that is bound to a specific `this` value or the "owner" object, So we can use it later in our code. The `call`, `apply` methods invokes the function immediately instead of returning a new function like the `bind` method.

```
import React from 'react';
```

```

class MyComponent extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      value : ""
    }
    this.handleChange = this.handleChange.bind(this);
    // Binds the "handleChange" method to the "MyComponent" component
  }

  handleChange(e){
    //do something amazing here
  }

  render(){
    return (
      <>
        <input type={this.props.type}
          value={this.state.value}
          onChange={this.handleChange}
        />
      </>
    )
  }
}

```

```
}  
}
```

31. What is **Functional Programming** and what are the features of **JavaScript** that makes it a candidate as a **functional language**?

↑ **Functional Programming** is a **declarative** programming paradigm or pattern on how we build our applications with **functions** using **expressions** that calculates a value without mutating or changing the arguments that are passed to it.

JavaScript **Array** has **map**, **filter**, **reduce** methods which are the most famous functions in the functional programming world because of their usefulness and because they don't mutate or change the array which makes these functions **pure** and JavaScript supports **Closures** and **Higher Order Functions** which are a characteristic of a **Functional Programming Language**.

- The **map** method creates a new array with the results of calling a provided callback function on every element in the array.

```
const words = ["Functional", "Procedural", "Object-Oriented"];
```

```
const wordsLength = words.map(word => word.length);
```

- The **filter** method creates a new array with all elements that pass the test in the callback function.

```
const data = [  
  { name: 'Mark', isRegistered: true },  
  { name: 'Mary', isRegistered: false },  
  { name: 'Mae', isRegistered: true }  
];
```

```
const registeredUsers = data.filter(user => user.isRegistered);
```

- The **reduce** method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
const str = ["I", " ", "am", " ", "Iron", " ", "Man"];  
const result = str.reduce((acc, currentStr) => acc + currentStr, "");
```

32. 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);  
}
```

### 33. 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.

### 34. 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
  }
}
```

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.

### 35. 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
}
}

```

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.

### 36. 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;
    }
}

```

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.

### 37. What is the **arguments** object?

↑ The **arguments** object is a collection of parameter values pass in a function. It's an **Array-like** object because it has a **length** property and we can access individual values using array indexing notation `arguments[1]` but it does not have the built-in methods in an array `forEach`, `reduce`, `filter` and `map`.

It helps us know the number of arguments pass in a function.

We can convert the arguments object into an array using the `Array.prototype.slice`.

```
function one() {  
    return Array.prototype.slice.call(arguments);  
}
```

Note: the **arguments** object does not work on ES6 arrow functions.

```
function one() {  
    return arguments;  
}  
  
const two = function () {  
    return arguments;  
}  
  
const three = function three() {  
    return arguments;  
}
```

```
const four = () => arguments;
```

```
four(); // Throws an error - arguments is not defined
```

When we invoke the function `four` it throws a `ReferenceError: arguments is not defined` error. We can solve this problem if your environment supports the **rest syntax**.

```
const four = (...args) => args;
```

This puts all parameter values in an array automatically.

### 38. How to create an object without a **prototype**?

↑ We can create an object without a prototype using the `Object.create` method.

```
const o1 = {};  
console.log(o1.toString());  
// logs [object Object] get this method to the Object.prototype  
  
const o2 = Object.create(null);  
// the first parameter is the prototype of the object "o2" which in this  
// case will be null specifying we don't want any prototype  
console.log(o2.toString());  
// throws an error o2.toString is not a function
```

### 39. Why does `b` in this code become a global variable when you call this function?

↑

```
function myFunc() {  
    let a = b = 0;  
}
```

```
myFunc();
```

The reason for this is that **assignment operator** or `=` has right-to-left **associativity** or **evaluation**. What this means is that when multiple assignment operators appear in a single expression they are evaluated from right to left. So our code becomes like this.

```
function myFunc() {  
    let a = (b = 0);  
}
```

```
myFunc();
```

First, the expression `b = 0` is evaluated and in this example `b` is not declared. So, the JS Engine makes a global variable `b` outside this function after that the return value of the expression `b = 0` would be `0` and it's assigned to the new local variable `a` with a `let` keyword.

We can solve this problem by declaring the variables first before assigning them with value.

```
function myFunc() {  
    let a, b;  
    a = b = 0;  
}  
myFunc();
```

### 40. What is **ECMAScript**?

↑ **ECMAScript** is a standard for making scripting languages which means that **JavaScript** follows the specification changes in **ECMAScript** standard because it is the **blueprint** of **JavaScript**.

### 41. What are the new features in **ES6** or **ECMAScript 2015**?

↑

- Arrow Functions
- Classes
- Template Strings
- **Enhanced Object literals**
- Object Destructuring
- Promises
- **Generators**
- Modules

- Symbol
- **Proxies**
- Sets
- Default Function parameters
- Rest and Spread
- Block Scoping with `let` and `const`

## 42. What's the difference between `var`, `let` and `const` keywords?

↑ Variables declared with `var` keyword are function scoped.

What this means that variables can be accessed across that function even if we declare that variable inside a block.

```
function giveMeX(showX) {
  if (showX) {
    var x = 5;
  }
  return x;
}

console.log(giveMeX(false));
console.log(giveMeX(true));
```

The first `console.log` statement logs `undefined` and the second `5`. We can access the `x` variable due to the reason that it gets hoisted at the top of the function scope. So our function code is interpreted like this.

```
function giveMeX(showX) {
  var x; // has a default value of undefined
  if (showX) {
    x = 5;
  }
  return x;
}
```

If you are wondering why it logs `undefined` in the first `console.log` statement remember variables declared without an initial value has a default value of `undefined`.

Variables declared with `let` and `const` keyword are block scoped. What this means that variable can only be accessed on that block `{}` on where we declare it.

```
function giveMeX(showX) {
  if (showX) {
    let x = 5;
  }
  return x;
}
```



```
function giveMeY(showY) {
  if (showY) {
    let y = 5;
  }
  return y;
}
```

If we call this functions with an argument of `false` it throws a `Reference Error` because we can't access the `x` and `y` variables outside that block and those variables are not hoisted.

There is also a difference between `let` and `const` we can assign new values using `let` but we can't in `const` but `const` are mutable meaning. What this means is if the value that we assign to a `const` is an object we can change the values of those properties but can't reassign a new value to that variable.

### 43. What are Arrow functions?

↑ **Arrow Functions** are a new way of making functions in JavaScript. **Arrow Functions** takes a little time in making functions and has a cleaner syntax than a **function expression** because we omit the function keyword in making them.

//ES5 version

```
var getCurrentDate = function (){
  return new Date();
}
```

//ES6 version

```
const getCurrentDate = () => new Date();
```

In this example, in the ES5 Version have `function(){} declaration` and `return` keyword needed to make a function and return a value respectively. In the **Arrow Function** version we only need the `()` parentheses and we don't need a `return` statement because **Arrow Functions** have a implicit return if we have only one expression or value to return.

//ES5 version

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

//ES6 version

```
const greet = (name) => `Hello ${name}`;
const greet2 = name => `Hello ${name}`;
```

We can also parameters in **Arrow functions** the same as the **function expressions** and **function declarations**. If we have one parameter in an **Arrow Function** we can omit the parentheses it is also valid.

```
const getArgs = () => arguments
```

```
const getArgs2 = (...rest) => rest
```

**Arrow functions** don't have access to the `arguments` object. So calling the first `getArgs` func will throw an Error. Instead we can use the **rest parameters** to get all the arguments passed in an arrow function.

```
const data = {
  result: 0,
  nums: [1, 2, 3, 4, 5],
  computeResult() {
    // "this" here refers to the "data" object
    const addAll = () => {
      // arrow functions "copies" the "this" value of
      // the lexical enclosing function
      return this.nums.reduce((total, cur) => total + cur, 0)
    };
    this.result = addAll();
  }
};
```

**Arrow functions** don't have their own `this` value. It captures or gets the `this` value of lexically enclosing function or in this example, the `addAll` function copies the `this` value of the `computeResult` method and if we declare an arrow function in the global scope the value of `this` would be the `window` object.

## 44. 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.prototype = 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.yearStarted}`;
}

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

```

## 45. What are Template Literals?

↑ **Template Literals** are a new way of making **strings** in JavaScript. We can make **Template Literal** by using the backtick or back-quote symbol.

```

//ES5 version
var greet = 'Hi I\'m Mark';

//ES6 version
let greet = `Hi I'm Mark`;

```

In the ES5 version, we need to escape the ' using the \ to escape the normal functionality of that symbol which in this case is to finish that string value. In Template Literals, we don't need to do that.

```

//ES5 version

```

```
var lastWords = '\n'
  + '    I  \n'
  + '    Am \n'
  + 'Iron Man \n';
```

```
//ES6 version
let lastWords = `
    I
    Am
    Iron Man
`;
```

In the ES5 version, we need to add this `\n` to have a new line in our string. In Template Literals, we don't need to do that.

```
//ES5 version
function greet(name) {
  return 'Hello ' + name + '!';
}
```

```
//ES6 version
const greet = name => {
  return `Hello ${name} !`;
}
```

In the ES5 version, If we need to add an expression or value in a string we need to use the `+` or string concatenation operator. In Template Literals, we can embed an expression using `${expr}` which makes it cleaner than the ES5 version.

## 46. What is Object Destructuring?

↑ **Object Destructuring** is a new and cleaner way of **getting** or **extracting** values from an object or an array.

Suppose we have an object that looks like this.

```
const employee = {
  firstName: "Marko",
  lastName: "Polo",
  position: "Software Developer",
  yearHired: 2017
};
```

The old way of getting properties from an object is we make a variable that has the same name as the object property. This way is a hassle because we're making a new variable for every property. Imagine we have a big object with lots of properties and methods using this way in extracting properties will be irritating.

```

var firstName = employee.firstName;
var lastName = employee.lastName;
var position = employee.position;
var yearHired = employee.yearHired;

```

If we use **object destructuring** it looks cleaner and takes a little time than the old way. The syntax for object destructuring is that if we are getting properties in an object we use the `{}` and inside that, we specify the properties we want to extract and if we are getting data from an array we use the `[]`.

```

let { firstName, lastName, position, yearHired } = employee;

```

If we want to change the variable name we want to extract we use the `propertyName:newName` syntax. In this example the value of `fName` variable will hold the value of the `firstName` property and `lName` variable will hold the value of the `lastName` property.

```

let { firstName: fName, lastName: lName, position, yearHired } = employee;

```

We can also have default values when destructuring. In this example, if the `firstName` property holds an undefined value in the object then when we destructure the `firstName` variable will hold a default of "Mark".

```

let { firstName = "Mark", lastName: lName, position, yearHired } = employee;

```

## 47. What are ES6 Modules?

⚡ **Modules** lets us split our code base to multiple files for more maintainability and this lets us avoid putting all of our code in one big file (yucksss). Before ES6 has supported Modules there were two popular module systems that were used for Code Maintainability in **JavaScript**.

- CommonJS - **Nodejs**
- AMD (Asynchronous Module Definition) - **Browsers**

Basically, the sytanx for using modules are straightforward,

`import` is used for getting functionality from another file or several functionalities or values while

`export` is used for exposing functionality from a file or several functionalities or values.

### Exporting functionalites in a File or Named Exports

Using ES5 (CommonJS)

```

// Using ES5 CommonJS - helpers.js
exports.isNull = function (val) {
  return val === null;
}

exports.isUndefined = function (val) {
  return val === undefined;
}

exports.isNullorUndefined = function (val) {
  return exports.isNull(val) || exports.isundefined(val);
}

```

Using ES6 Modules

```
// Using ES6 Modules - helpers.js
export function isNull(val){
  return val === null;
}

export function isUndefined(val) {
  return val === undefined;
}

export function isNullorUndefined(val) {
  return isNull(val) || isUndefined(val);
}
```

## Importing functionalities in another File

```
// Using ES5 (CommonJS) - index.js
const helpers = require('./helpers.js'); // helpers is an object
const isNull = helpers.isNull;
const isUndefined = helpers.isUndefined;
const isNullorUndefined = helpers.isNullorUndefined;

// or if your environment supports Destructuring
const { isNull, isUndefined, isNullorUndefined } = require('./helpers.js');
// ES6 Modules - index.js
import * as helpers from './helpers.js'; // helpers is an object

// or

import { isNull, isUndefined, isNullorUndefined as isValid } from './helpers.js';

// using "as" for renaming named exports
```

## Exporting a Single Functionality in a File or Default Exports

Using ES5 (CommonJS)

```
// Using ES5 (CommonJS) - index.js
class Helpers {
  static isNull(val) {
    return val === null;
  }

  static isUndefined(val) {
    return val === undefined;
  }

  static isNullorUndefined(val) {
    return this.isNull(val) || this.isUndefined(val);
  }
}
```

```
}  
}
```

```
module.exports = Helpers;
```

## Using ES6 Modules

```
// Using ES6 Modules - helpers.js
```

```
class Helpers {  
  static isNull(val) {  
    return val === null;  
  }  
  
  static isUndefined(val) {  
    return val === undefined;  
  }  
  
  static isNullOrUndefined(val) {  
    return this.isNull(val) || this.isUndefined(val);  
  }  
}
```

```
export default Helpers
```

## Importing a Single Functionality from another File

### Using ES5 (CommonJS)

```
// Using ES5 (CommonJS) - index.js  
const Helpers = require('./helpers.js');  
console.log(Helpers.isNull(null));
```

### Using ES6 Modules

```
import Helpers from './helpers.js'  
console.log(Helpers.isNull(null));
```

This is the basics of using **ES6 Modules**. I won't explain all about Modules because it's a broad topic and my Post is now really long.

## 48. What is the Set object and how does it work?

↑ The **Set** object is an **ES6** feature that lets you store unique values, **primitives** or **object references**. A value in a Set can only occur **once**. It checks if a value exists in the set object using the **SameValueZero** algorithm.

We can make Set instance using Set constructor and we can optionally pass an **Iterable** as the initial value.

```
const set1 = new Set();  
const set2 = new Set(["a","b","c","d","d","e"]);
```



We can add a new value into the `Set` instance using the `add` method and since the `add` returns the `Set` object we can chain `add` calls. If a value already exists in `Set` object it will not be added again.

```
set2.add("f");
set2.add("g").add("h").add("i").add("j").add("k").add("k");
// the last "k" will not be added to the set object because it already exists
```

We can remove a value from the `Set` instance using the `delete` method, this method returns a `boolean` indicating `true` if a value exists in the `Set` object and `false` indicating that value does not exist.

```
set2.delete("k") // returns true because "k" exists in the set object
set2.delete("z") // returns false because "z" does not exists in the set object
```

We can check if a specific value exists in the `Set` instance using the `has` method.

```
set2.has("a") // returns true because "a" exists in the set object
set2.has("z") // returns false because "z" does not exists in the set object
```

We can get the length of the `Set` instance using the `size` property.

```
set2.size // returns 10
```

We can delete or remove all the elements in the `Set` instance using the `clear`.

```
set2.clear(); // clears the set data
```

We can use the `Set` object for removing duplicate elements in an array.

```
const numbers = [1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 5];
const uniqueNums = [...new Set(numbers)]; // has a value of [1,2,3,4,5,6,7,8]
```

## 49. 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**.

## 50. What are Promises?

↑ **Promises** are one way in handling asynchronous operations in **JavaScript**. It represents the value of an asynchronous operation. **Promises** was made to solve the problem of doing and dealing with async code before promises we're using callbacks.

```
fs.readFile('somefile.txt', function (e, data) {
  if (e) {
    console.log(e);
  }
  console.log(data);
});
```

The problem with this approach if we have another async operation inside the callback and another. We will have a code that is messy and unreadable. This code is called **Callback Hell**.

```
//Callback Hell yucksss
fs.readFile('somefile.txt', function (e, data) {
  //your code here
  fs.readdir('directory', function (e, files) {
    //your code here
    fs.mkdir('directory', function (e) {
      //your code here
    })
  })
})
```

If we use promises in this code it will be more readable and easy to understand and easy to maintain.

```
promReadFile('file/path')
  .then(data => {
    return promReaddir('directory');
  })
  .then(data => {
    return promMkdir('directory');
  })
  .catch(e => {
    console.log(e);
  })
```

Promises have 3 different states.

**Pending** - The initial state of a promise. The promise's outcome has not yet been known because the operation has not been completed yet.

**Fulfilled** - The async operation is completed and successful with the resulting value.

**Rejected** - The async operation has failed and has a reason on why it failed.

**Settled** - If the promise has been either **Fulfilled** or **Rejected**.

The **Promise** constructor has two parameters which are functions `resolve` and `reject` respectively.

If the async operation has been completed without errors call the `resolve` function to resolve the promise or if an error occurred

call the `reject` function and pass the error or reason to it.

We can access the result of the fulfilled promise using the `.then`

method and we catch errors in the `.catch` method. We chain multiple async promise operations in the `.then` method because the `.then` method returns a **Promise** just like the example in the image above.

```
const myPromiseAsync = (...args) => {
  return new Promise((resolve, reject) => {
    doSomeAsync(...args, (error, data) => {
      if (error) {
        reject(error);
      } else {
        resolve(data);
      }
    })
  })
}
```

```
myPromiseAsync()
  .then(result => {
    console.log(result);
  })
  .catch(reason => {
    console.log(reason);
  })
```

We can make a helper func that converts an async operation with a callback to promise. It works like the **promisify** utility function from the node core module `util`.

```
const toPromise = (asyncFuncWithCallback) => {
  return (...args) => {
    return new Promise((res, rej) => {
      asyncFuncWithCallback(...args, (e, result) => {
        return e ? rej(e) : res(result);
      });
    });
  }
}
```

```
const promReadFile = toPromise(fs.readFile);
```

```
promReadFile('file/path')
```

```

.then((data) => {
  console.log(data);
})
.catch(e => console.log(e));

```

## 51. What is async/await and How does it work?

↑ **async/await** is the new way of writing asynchronous or non-blocking code in **JavaScript's**. It is built on top of **Promises**. It makes writing asynchronous code more readable and cleaner than **Promises** and **Callbacks**. But you must learn the basics of **Promises** before using this feature because as I said earlier it is built on top of **Promises** which means it still uses **Promises** under the hood.

Using Promises.

```

function callApi() {
  return fetch("url/to/api/endpoint")
    .then(resp => resp.json())
    .then(data => {
      //do something with "data"
    }).catch(err => {
      //do something with "err"
    });
}

```

Using Async/Await.

**Note:** We're using the old try/catch statement to **catch** any errors that happened in any of those async operations inside the try statement.

```

async function callApi() {
  try {
    const resp = await fetch("url/to/api/endpoint");
    const data = await resp.json();
    //do something with "data"
  } catch (e) {
    //do something with "err"
  }
}

```

**Note:** The **async** keyword before the function declaration makes the function return implicitly a **Promise**.

```
const giveMeOne = async () => 1;
```

```

giveMeOne()
  .then((num) => {
    console.log(num); // logs 1
  });

```

**Note:** The `await` keyword can **only** be used inside an **async function**. Using `await` keyword in any other function which is not an **async function** will throw an error. The `await` keyword **awaits** the right-hand side expression (presumably a **Promise**) to return before executing the next line of code.

```
const giveMeOne = async () => 1;
```

```
function getOne() {
  try {
    const num = await giveMeOne();
    console.log(num);
  } catch (e) {
    console.log(e);
  }
}
```

```
//Throws a Compile-Time Error = Uncaught SyntaxError: await is only valid in an async function
```

```
async function getTwo() {
  try {
    const num1 = await giveMeOne(); //finishes this async operation first before going to
    const num2 = await giveMeOne(); //this line
    return num1 + num2;
  } catch (e) {
    console.log(e);
  }
}
```

```
await getTwo(); // returns 2
```

## 52. What's the difference between **Spread operator** and **Rest operator**?

↑ The **Spread operator** and **Rest parameters** have the same operator `...` the difference between is that the **Spread operator** we **give** or **spread** individual data of an array to another data while the **Rest parameters** is used in a function or an array to **get** all the arguments or values and put them in an array or **extract** some pieces of them.

```
function add(a, b) {
  return a + b;
};
```

```
const nums = [5, 6];
const sum = add(...nums);
console.log(sum);
```

In this example, we're using the **Spread Operator** when we call the `add` function we are **spreading** the `nums` array. So the value of parameter `a` will be **5** and the value of `b` will be **6**. So the sum will be **11**.

```
function add(...rest) {  
    return rest.reduce((total,current) => total + current);  
};
```

```
console.log(add(1, 2)); // logs 3  
console.log(add(1, 2, 3, 4, 5)); // logs 15
```

In this example, we have a function `add` that accepts any number of arguments and adds them all and return the total.

```
const [first, ...others] = [1, 2, 3, 4, 5];  
console.log(first); //logs 1  
console.log(others); //logs [2,3,4,5]
```

In this another example, we are using the **Rest operator** to extract all the remaining array values and put them in array `others` except the first item.

## 53. What are Default Parameters?

↑ **Default Parameters** is a new way of defining default variables in **JavaScript** it is available in the **ES6** or **ECMAScript 2015** Version.

```
//ES5 version  
function add(a,b){  
    a = a || 0;  
    b = b || 0;  
    return a + b;  
}  
  
//ES6 version  
function add(a = 0, b = 0){  
    return a + b;  
}  
//If we don't pass any argument for 'a' or 'b' then  
// it's gonna use the "default parameter" value which is 0  
add(1); // returns 1
```

We can also use **Destructuring** in **Default Parameters**.

```
function getFirst([first, ...rest] = [0, 1]) {  
    return first;  
}  
  
getFirst(); // returns 0  
getFirst([10,20,30]); // returns 10  
  
function getArr({ nums } = { nums: [1, 2, 3, 4] }){
```

```

    return nums;
}

```

```

getArr(); // returns [1, 2, 3, 4]
getArr({nums:[5,4,3,2,1]}); // returns [5,4,3,2,1]

```

We can also use the parameters defined first to the parameters defined after them.

```

function doSomethingWithValue(value = "Hello world", callback = () => { console.log(value) })
{
    callback();
}
doSomethingWithValue(); //logs "Hello world"

```

## 54. What are Wrapper Objects?

↑ **Primitive Values** like `string`, `number` and `boolean` with the exception of `null` and `undefined` have properties and methods even though they are not objects.

```

let name = "marko";

```

```

console.log(typeof name); // logs "string"
console.log(name.toUpperCase()); // logs "MARKO"

```

`name` is a primitive `string` value that has no properties and methods but in this example we are calling a `toUpperCase()` method which does not throw an error but returns `MARKO`.

The reason for this is that the primitive value is temporarily converted or coerced to an object so the `name` variable behaves like an object. Every primitive except `null` and `undefined` have **Wrapper Objects**. The Wrapper Objects are `String`, `Number`, `Boolean`, `Symbol` and `BigInt`. In this case, the `name.toUpperCase()` invocation, behind the scenes it looks like this.

```

console.log(new String(name).toUpperCase()); // logs "MARKO"

```

The newly created object is immediately discarded after we finished accessing a property or calling a method.

## 55. What is the difference between Implicit and Explicit Coercion?

↑ **Implicit Coercion** is a way of converting values to another type without us programmer doing it directly or by hand.

Suppose we have an example below.

```

console.log(1 + '6');
console.log(false + true);
console.log(6 * '2');

```

The **first** `console.log` statement logs `16`. In other languages this would throw a compile time error but in **JavaScript** the `1` is converted to a `string` then concatenated with the `+` operator. We did not do

anything, yet it was converted automatically by **JavaScript** for us.

The **second** `console.log` statement logs 1, it converts the `false` to a `boolean` which will result to a 0 and the `true` will be 1 hence the result is 1.

The **third** `console.log` statement logs 12, it converts the `'2'` to a number before multiplying `6 * 2` hence the result 12.

### JavaScript Coercion Rules

While **Explicit Coercion** is the way of converting values to another type where we (programmers) **explicitly** do it.

```
console.log(1 + parseInt('6'));
```

In this example, we use the `parseInt` function to convert the `'6'` to a number then adding the 1 and 6 using the `+` operator.

## 56. What is NaN? and How to check if a value is NaN?

↑ `NaN` means "**Not A Number**" is a value in **JavaScript** that is a result in converting or performing an operation to a number to non-number value hence results to `NaN`.

```
let a;
```

```
console.log(parseInt('abc'));
console.log(parseInt(null));
console.log(parseInt(undefined));
console.log(parseInt(++a));
console.log(parseInt({} * 10));
console.log(parseInt('abc' - 2));
console.log(parseInt(0 / 0));
console.log(parseInt('10a' * 10));
```

**JavaScript** has a built-in method `isNaN` that tests if value is `isNaN` value. But this function has a weird behaviour.

```
console.log(isNaN()); //logs true
console.log(isNaN(undefined)); //logs true
console.log(isNaN({})); //logs true
console.log(isNaN(String('a'))); //logs true
console.log(isNaN(() => { })); //logs true
```

All these `console.log` statements return `true` even though those values we pass are not `NaN`.

In **ES6** or **ECMAScript 2015**, it is recommended that we use `Number.isNaN` method because it really checks the value if it really is `NaN` or we can make our own helper function that check for this problem because in **JavaScript** `NaN` is the only value that is not equal to itself.

```
function checkIfNaN(value) {
  return value !== value;
}
```



## 57. How to check if a value is an **Array**?

↑ We can check if a value is an **Array** by using the `Array.isArray` method available from the **Array** global object. It returns true when the parameter pass to it is an **Array** otherwise false.

```
console.log(Array.isArray(5)); //logs false
console.log(Array.isArray("")); //logs false
console.log(Array.isArray()); //logs false
console.log(Array.isArray(null)); //logs false
console.log(Array.isArray({ length: 5 })); //logs false

console.log(Array.isArray([])); //logs true
```

If your environment does not support this method you can use the polyfill implementation.

```
function isArray(value){
    return Object.prototype.toString.call(value) === "[object Array]"
}
```

## 58. How to check if a number is even without using the % or modulo operator?

↑ We can use the **bitwise AND** operator for this problem. The `&` operates on its operand and treats them as binary values and performs the **AND** operation.

```
function isEven(num) {
    if (num & 1) {
        return false;
    } else {
        return true;
    }
};
```

0 in binary is **000**.

1 in binary is **001**.

2 in binary is **010**.

3 in binary is **011**.

4 in binary is **100**.

5 in binary is **101**.

6 in binary is **110**.

7 in binary is **111**.

and so on...

a	b	a & b
0	0	0
0	1	0
1	0	0

a	b	a & b
1	1	1

So when we `console.log` this expression `5 & 1` it returns `1`. Ok, first the `&` operator converts both numbers to binary so `5` turns to **101** and `1` turns to **001**.

Then it compares every bit (**0's and 1's**) using the bitwise **AND** operator. **101 & 001**. As we can see from the table the result can be only `1` if a **AND** b are `1`.

<b>101 &amp; 001</b>
101
001
<b>001</b>

- So first we compare the left most bit `1&0` the result should be `0`.
- Then we compare the middle bit `0&0` the result should be `0`.
- Then we compare the last bit `1&1` the result should be `1`.
- Then the binary result `001` will be converted to a decimal number which will be `1`.

If we `console.log` this expression `4 & 1` it will return `0`. Knowing the last bit of `4` is `0` and `0 & 1` will be `0`. If you have a hard time understand this we could use a **recursive** function to solve this problem.

```
function isEven(num) {
  if (num < 0 || num === 1) return false;
  if (num == 0) return true;
  return isEven(num - 2);
}
```

## 59. How to check if a certain property exists in an object?

↑ There are three possible ways to check if a property exists in an object.

First, using the `in` operator. The syntax for using the `in` operator is like this `propertyname in object`. It returns `true` if the property exists otherwise it returns `false`.

```
const o = {
  "prop" : "bwahahah",
  "prop2" : "hweasa"
};
```

```
console.log("prop" in o); //This logs true indicating the property "prop" is in "o" object
console.log("prop1" in o); //This logs false indicating the property "prop" is not in "o" object
```

Second, using the `hasOwnProperty` method in objects. This method is available on all objects in JavaScript. It returns `true` if the property exists otherwise it returns `false`.

//Still using the `o` object in the first example.

```
console.log(o.hasOwnProperty("prop2")); // This logs true
```

```
console.log(o.hasOwnProperty("prop1")); // This logs false
```

Third, using the bracket notation `obj["prop"]`. If the property exists it returns the value of that property otherwise this will return undefined.

//Still using the o object in the first example.

```
console.log(o["prop"]); // This logs "bwahahah"
```

```
console.log(o["prop1"]); // This logs undefined
```

## 60. What is **AJAX**?

↑ **AJAX** stands for **Asynchronous JavaScript and XML**. It is a group of related technologies used to display data asynchronously. What this means is that we can send data to the server and get data from the server without reloading the web page.

Technologies use for **AJAX**.

- **HTML** - web page structure
- **CSS** - the styling for the webpage
- **JavaScript** - the behaviour of the webpage and updates to the **DOM**
- **XMLHttpRequest API** - used to send and retrieve data from the server
- **PHP,Python,Nodejs** - Some Server-Side language

## 61. What are the ways of making objects in JavaScript?

↑ Using **Object Literal**.

```
const o = {  
  name: "Mark",  
  greeting() {  
    return `Hi, I'm ${this.name}`;  
  }  
};
```

```
o.greeting(); //returns "Hi, I'm Mark"
```

Using **Constructor Functions**.

```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.greeting = function () {  
  return `Hi, I'm ${this.name}`;  
}  
  
const mark = new Person("Mark");
```

```
mark.greeting(); //returns "Hi, I'm Mark"
```

Using **Object.create** method.

```
const n = {
  greeting() {
    return `Hi, I'm ${this.name}`;
  }
};

const o = Object.create(n); // sets the prototype of "o" to be "n"

o.name = "Mark";

console.log(o.greeting()); // logs "Hi, I'm Mark"
```

## 62. What's the difference between `Object.seal` and `Object.freeze` methods?

↑ The difference between these two methods is that when we use the `Object.freeze` method to an object, that object's properties are immutable meaning we can't change or edit the values of those properties. While in the `Object.seal` method we can change those existing properties.

## 63. What's the difference between the `in` operator and the `hasOwnProperty` method in objects?

↑ As you know both of these features check if a property exists in an object. It will return `true`/`false`. The difference between them is that the `in` operator also checks the objects' **Prototype Chain** if the property was not found in the current object while the `hasOwnProperty` method just checks if the property exists in the current object ignoring the **Prototype Chain**.

```
// we'll still use the object in the previous question.
console.log("prop" in o); // This logs true;
console.log("toString" in o); // This logs true, the toString method is available in this
object's prototype which is the Object.prototype

console.log(o.hasOwnProperty("prop")); // This logs true
console.log(o.hasOwnProperty("toString")); // This logs false, does not check the object's
prototype
```

## 64. What are the ways to deal with **Asynchronous Code** in JavaScript?

↑

- Callbacks

- Promises
- async/await
- Libraries like async.js, bluebird, q, co

## 65. What's the difference between a **function expression** and **function declaration**?

↑ Suppose we have an example below.

```
hoistedFunc();
notHoistedFunc();

function hoistedFunc(){
  console.log("I am hoisted");
}

var notHoistedFunc = function(){
  console.log("I will not be hoisted!");
}
```

The `notHoistedFunc` call throws an error while the `hoistedFunc` call does not because the `hoistedFunc` is hoisted while the `notHoistedFunc` is not.

Read **Hoisting** [here](#).

## 66. How many ways can a function be invoked?

↑ There are 4 ways that a function can be invoked in **JavaScript**. The **invocation** determines the value of this or the "owner" object of that function.

- **Invocation as a function** - If a function isn't invoked as a method, as a constructor or with the `apply`, `call` methods then it is **invoked as a function**. The "owner" object of this function will be the `window` object.

//Global scope

```
function add(a,b){
  console.log(this);
  return a + b;
}
```

`add(1,5);` // logs the "window" object and returns 6

```
const o = {
  method(callback){
    callback();
  }
}
```

```
o.method(function (){
```

```
    console.log(this); // logs the "window" object
  });
```

- **Invocation as a method** - If a property of an object has a value of a function we call it a **method**. When that **method** is invoked the **this** value of that method will be that object.

```
const details = {
  name : "Marko",
  getName(){
    return this.name;
  }
}
```

```
details.getName(); // returns Marko
```

```
// the "this" value inside "getName" method will be the "details" object
```

- **Invocation as a constructor** - If a function was invoked with a new keyword before it then it's called a function constructor. An empty object will be created and **this** will point to that object.

```
function Employee(name, position, yearHired) {
  // creates an empty object {}
  // then assigns the empty object to the "this" keyword
  // this = {};
  this.name = name;
  this.position = position;
  this.yearHired = yearHired;
  // inherits from Employee.prototype
  // returns the "this" value implicitly if no
  // explicit return statement is specified
};
```

```
const emp = new Employee("Marko Polo", "Software Developer", 2017);
```

- **Invocation with the apply and call methods** - If we want to explicitly specify the **this** value or the "owner" object of a function we can use these methods. These methods are available for all functions.

```
const obj1 = {
  result:0
};
```

```
const obj2 = {
  result:0
};
```

```
function reduceAdd(){
  let result = 0;
  for(let i = 0, len = arguments.length; i < len; i++){
```

```

        result += arguments[i];
    }
    this.result = result;
}

```

```

    reduceAdd.apply(obj1, [1, 2, 3, 4, 5]);    //the "this" object inside the "reduceAdd"
function will be "obj1"
    reduceAdd.call(obj2, 1, 2, 3, 4, 5); //the "this" object inside the "reduceAdd" function
will be "obj2"

```

## 67. What is memoization and what's the use it?

↑ memoization is a process of building a function that is capable of **remembering** its previously computed results or values.

The use of making a memoization function is that we avoid the computation of that function if it was already performed in the last calculations with the same arguments. This saves time but has a downside that we will consume more memory for saving the previous results.

## 68. Implement a memoization helper function.

↑

```

function memoize(fn) {
    const cache = {};
    return function (param) {
        if (cache[param]) {
            console.log('cached');
            return cache[param];
        } else {
            let result = fn(param);
            cache[param] = result;
            console.log(`not cached`);
            return result;
        }
    }
}

```

```
const toUpper = (str='')=> str.toUpperCase();
```

```
const toUpperMemoized = memoize(toUpper);
```

```
toUpperMemoized("abcdef");
toUpperMemoized("abcdef");
```

This memoize helper function only works on a function that accepts one argument. We need to make a memoize helper function that accepts multiple arguments.

```

const slice = Array.prototype.slice;
function memoize(fn) {
  const cache = {};
  return (...args) => {
    const params = slice.call(args);
    console.log(params);
    if (cache[params]) {
      console.log('cached');
      return cache[params];
    } else {
      let result = fn(...args);
      cache[params] = result;
      console.log(`not cached`);
      return result;
    }
  }
}

const makeFullName = (fName, lName) => `${fName} ${lName}`;
const reduceAdd = (numbers, startingValue = 0) => numbers.reduce((total, cur) => total + cur, startingValue);

const memoizedMakeFullName = memoize(makeFullName);
const memoizedReduceAdd = memoize(reduceAdd);

memoizedMakeFullName("Marko", "Polo");
memoizedMakeFullName("Marko", "Polo");

memoizedReduceAdd([1, 2, 3, 4, 5], 5);
memoizedReduceAdd([1, 2, 3, 4, 5], 5);

```

## 69. Why does `typeof null` return `object`? How to check if a value is `null`?

↑ `typeof null == 'object'` will always return `true` because this was the implementation of `null` since the birth of **JavaScript**. A fix was proposed to change `typeof null == 'object'` to `typeof null == 'null'` but was rejected because it will lead to more bugs.

We can use the `===` or **strict equality** operator to check if a value is `null`.

```

function isNull(value){
  return value === null;
}

```



## 70. What does the `new` keyword do?

↑ The `new` keyword is used with constructor functions to make objects in **JavaScript**.

Suppose we have an example code below.

```
function Employee(name, position, yearHired) {  
    this.name = name;  
    this.position = position;  
    this.yearHired = yearHired;  
};  
  
const emp = new Employee("Marko Polo", "Software Developer", 2017);
```

The `new` keyword does 4 things.

- Creates an empty object.
- Assigns that empty object to the `this` value.
- The function will inherit from **`functionName.prototype`**.
- Returns the `this` if no Explicit return statement is used.

In the above image, it will first create an empty object `{}` then it will the `this` value to that empty object `this = {}` and add properties to that `this` object. Because we don't have a explicit `return` statement it automatically returns the `this` for us.

Thanks guys for reading this post.