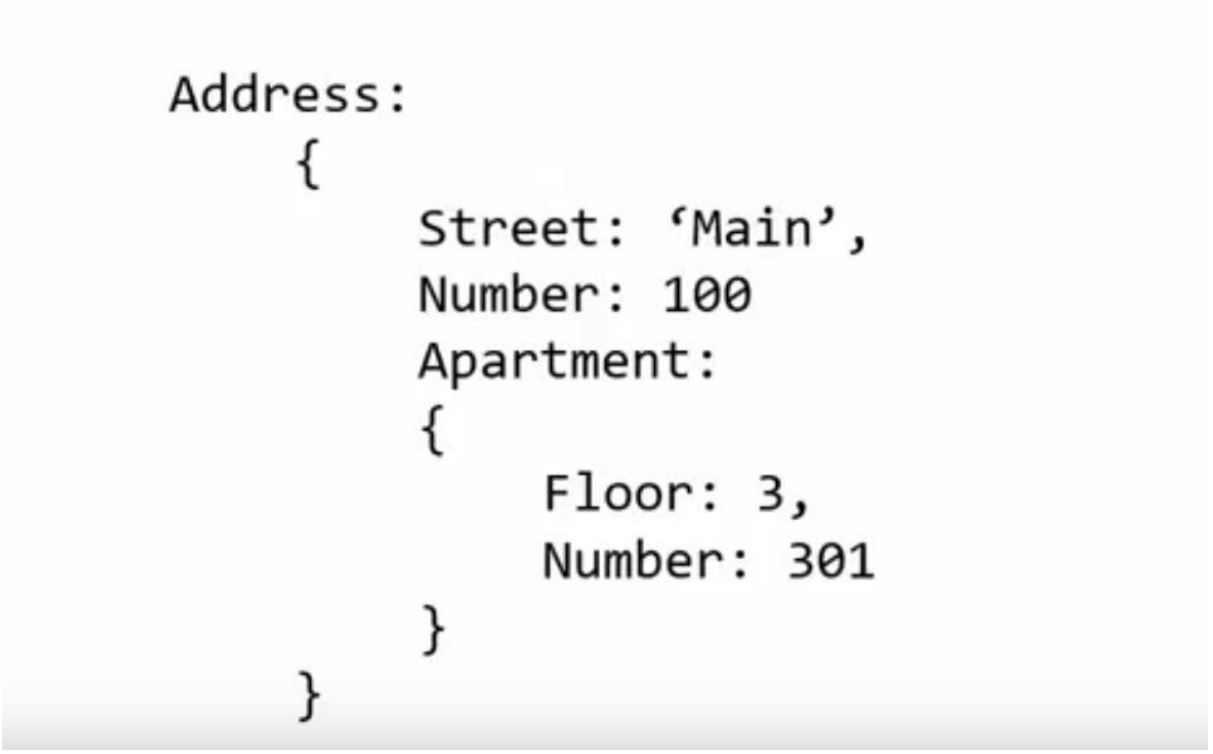
JavaScript Notes

(Part 1)

(make use of BRACKETS.IO as it has its own server and uses chrome)

1. **Frameworks** - Things like Jquery and angular JS developed by the open-source community that can be used to make fab web applications.
2. **Syntax parsers** - A program that reads our code and determines what it does and is it valid or not.
3. **Lexical environment** - The physical place where our code is present.
4. **Execution context** - What part of the code is running right now. There are a lot of lexical environments and which one is working right now is managed by execution context. Wraps the code in the execution context.
5. **Name/Value pairs**- The name that maps to a unique value. The name may be defined more than once but has only one value in specific reference.
6. **Object** - Collection of name-value pairs.



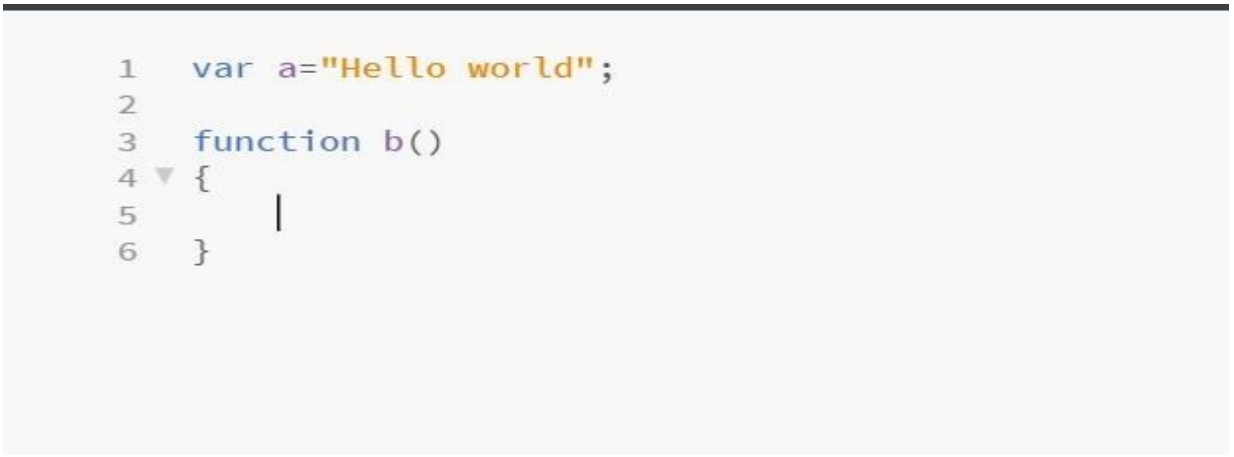
Here address and apartment are the names of the object and street, number and apartment are name-value pairs where the apartment has more name-value pairs.

7. **Global Environment and Global objects** - Base execution context is globally available and can be accessed in any part of the code. And this automatically creates objects(name/value pairs). And it creates a special variable for us known as "this".
When we run an empty js file by attaching it to our HTML and open dev tools and see the console, we can notice it does not have any errors. And if we type 'this' we get the following results.

If we type 'window' we get this:

In the case of node js and others "window" isn't the global object but since we are running our program using brackets.io "**chrome is our global object in this case".** An execution context was created at the global level, which created a global object named

windows and a variable named “this” was created. Code and variables which aren’t inside a function are called global.



Here ‘a’ is a global variable as it is not inside a function.
In this case, as we run our code, a(variable) and b(function) gets embedded inside the global object, in this case, it is ‘window’.
So, if we run
Window.a = Hello world
and if we run ‘a’ we get “Hello world” only. A global variable can be used everywhere in a piece of code.

8. Creation of execution context and “Hoisting”:

   The normal flow of code is like:
   // Function and variables declared and then used afterward
   1.

```
var a = 'Hello world';

function b() {
        console.log("This is b");
}

b();
console.log(a);
```

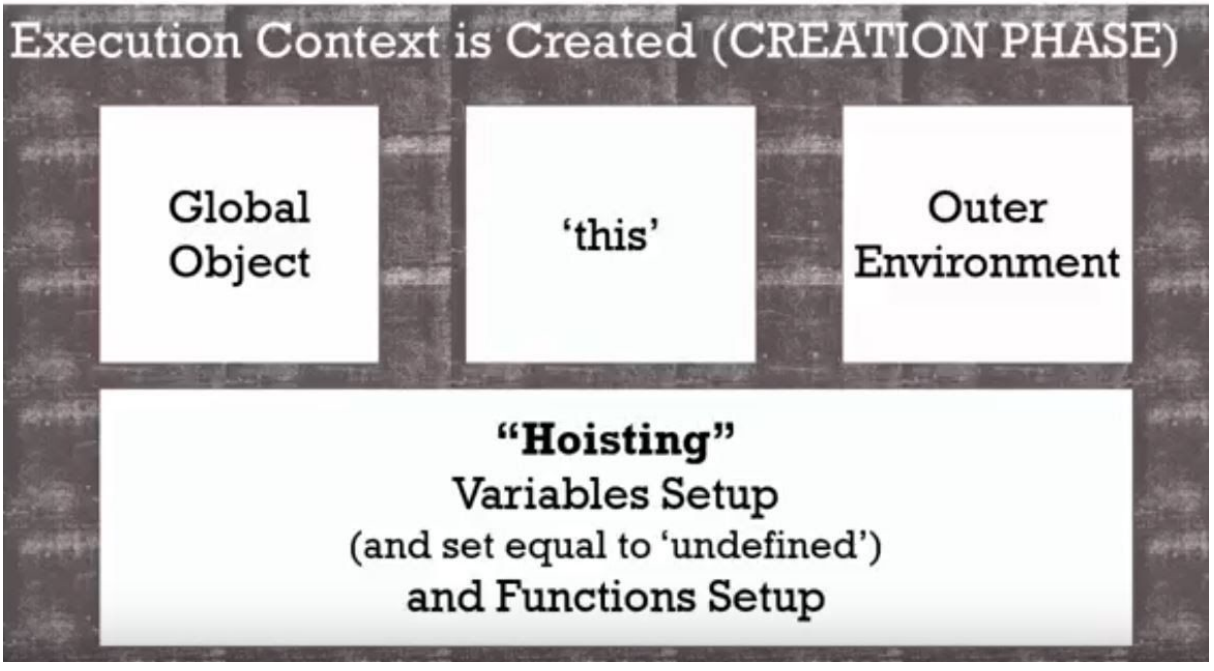
2.

```
b();
console.log(a);
var a = 'Hello world';

function b() {
        console.log("This is b");
}
```

Output:
This is b
Undefined

// here function and variables were used before being declared. This worked in the case of the function (due to hoisting but failed in the case of variable). Remember hoisting means using a function before being declared and is not valid for variables.



9. **Not defined vs Undefined**

1.

```
Var a;
```

console.log(a);

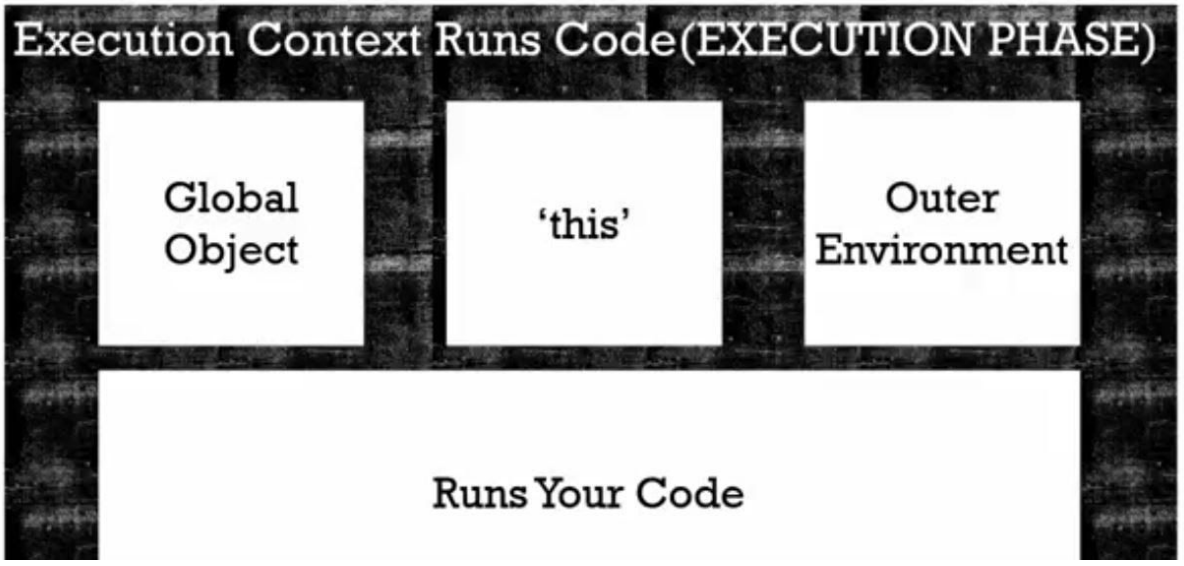
Error: Undefined

2.
console.log(a);

Error: Not defined (during context creation it wasn't declared to be a variable).

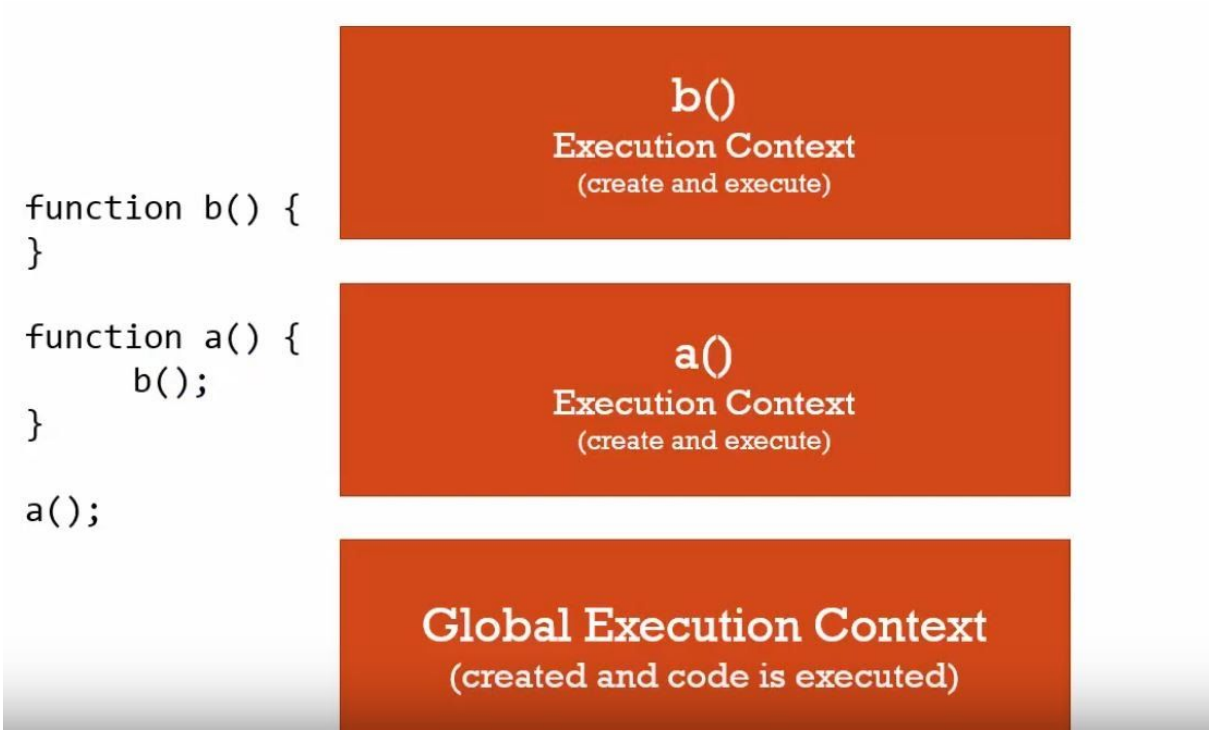
As a programmer never set a variable equal to undefined.

10. **Execution phase**



JavaScript program = Creation phase(setting up variables)+ Execution phase(running code)

11. Single-threaded - One command is executed line by line.
12. Synchronous - One at a time. And in order.
13. **JavaScript is single-threaded and synchronous.**
14. **Function execution and execution stack** -
    Invocation: Running a function

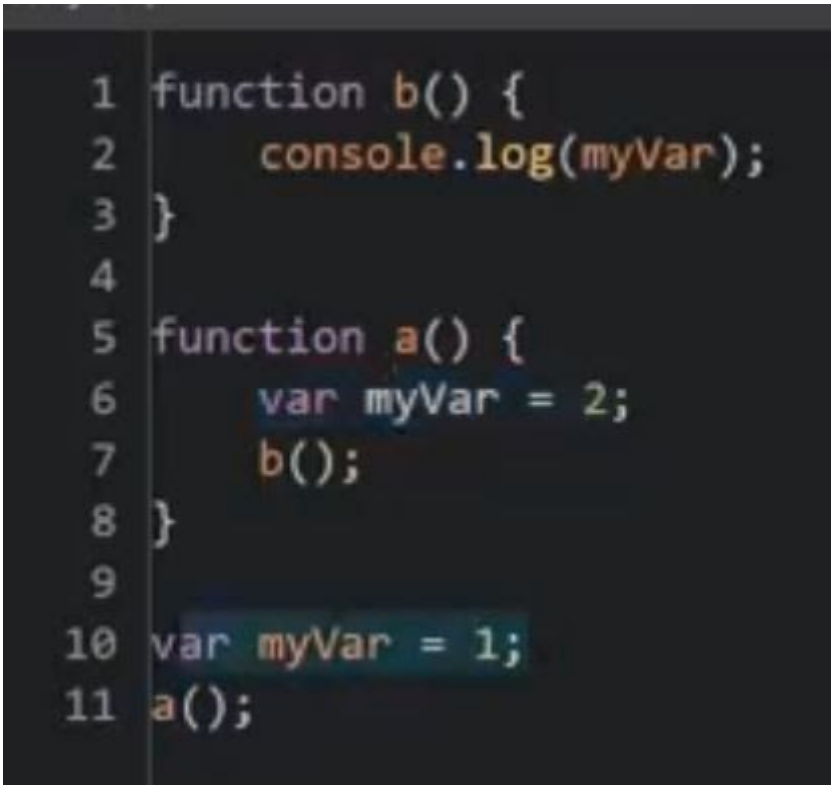


The orange boxes here form an execution stack while the code is getting executed. First Global execution context then a() and then b() [which was inside a()]

15. **Variable environment** - Where the variables live

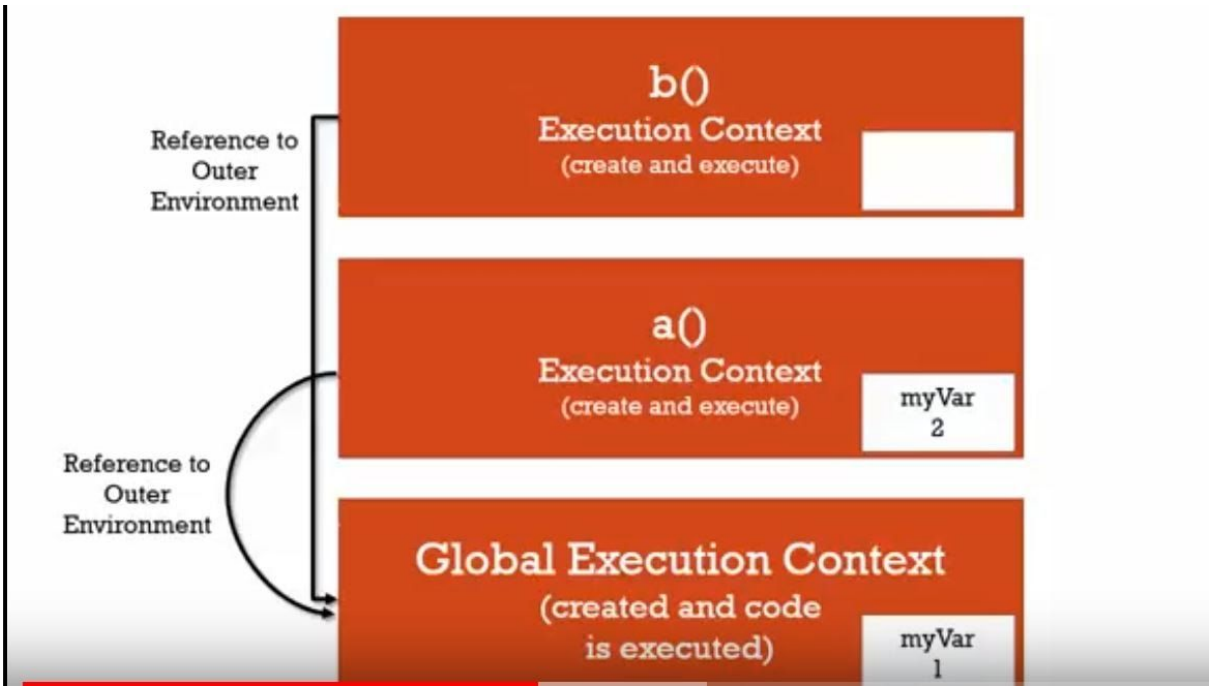
In this code, if myvar is invoked again in the global execution context we again get 1 as this was the value declared in the global context.

16. **Scope chain**



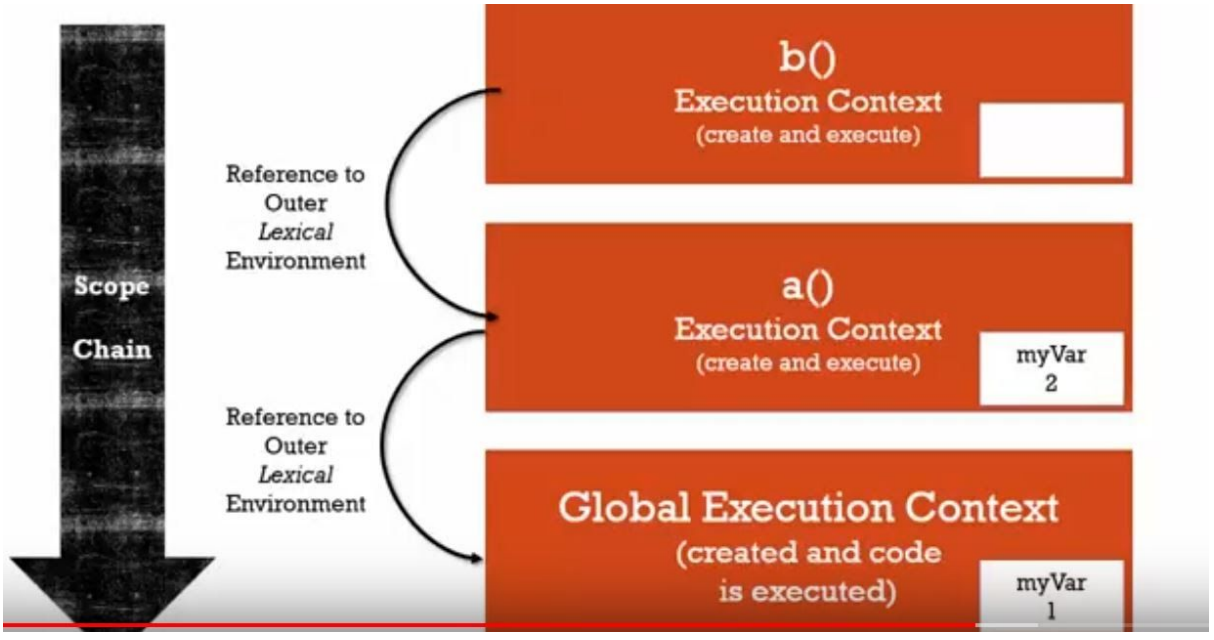
```
1  function b() {
2      console.log(myVar);
3  }
4
5  function a() {
6      var myVar = 2;
7      b();
8  }
9
10 var myVar = 1;
11 a();
```

Answer to the above problem is 1 because function b() sits lexically inside the global reference (window) that's why it takes myVar from the global variable of the window.



This process of finding which lexical environment is currently in use is k/as scope chain.

If we change the lexical environment of b and shift it to the scope of a than the value of myVar inside b as 2.



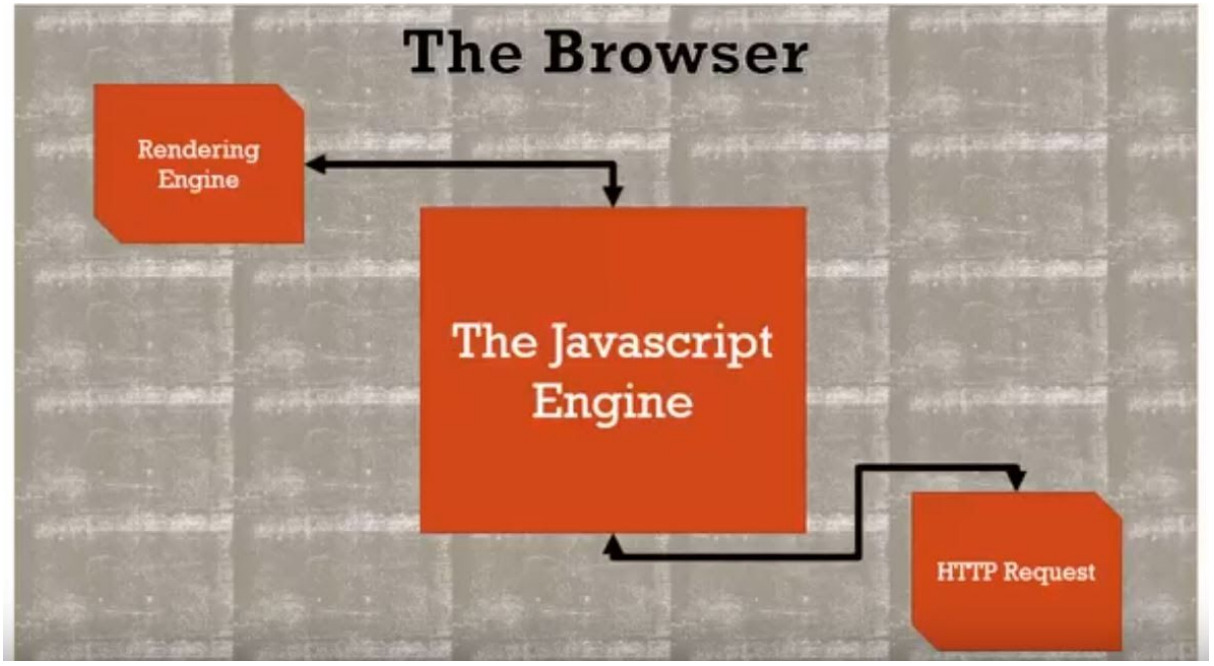
Scope - Place where code is available in the code.

17. ECMAScript 6 or ES6 has introduced a new way of declaring variables using ‘let’.

Block scoping and ‘let’ :

let is always used inside a block(something enclosed by curly braces{}).
for(let myVar=90) is a declaration where the engine has allotted space to our variable in the memory but it can't be used for coding purposes in the for the statement. Variable declared inside a block are available only inside that block.

18. What about asynchronous callbacks?



19. Use of onclicklistener

```
<!DOCTYPE html>
<html>
<body>

<p>This example uses the addEventListener() method to attach a click event to the document.</p>

<p>Click anywhere in the document.</p>

<p><strong>Note:</strong> The addEventListener() method is not supported in Internet Explorer 8 and earlier versions.</p>
```

```
<p id="demo"></p>

<script>
document.addEventListener("click", function(){
  document.getElementById("demo").innerHTML = "Hello World!";
});
</script>

</body>
</html>
```

Output: (click anywhere on the page it will show Hello World!)
Note: innerHTML returns JS stuff back to HTML.

20. Use of date and time

Using only the date() object:
It shows the date and day of the browser's current day and date.
Eg:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript new Date()</h2>

<p id="demo"></p>

<script>
var d = new Date();
document.getElementById("demo").innerHTML = d;
</script>

</body>
</html>
```

Using both day and date to find the number of milliseconds from midnight Jan 1, 1970.

Eg:

```
<!DOCTYPE html>
<html>
<body>

<p>The internal clock in JavaScript starts at midnight January 1, 1970.</p>
<p>Click the button to display the number of milliseconds since midnight, January 1, 1970.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var d = new Date();
  var n = d.getTime();
  document.getElementById("demo").innerHTML = n;
}
</script>

</body>
</html>
```

21. Event Queue v/s Execution Queue:

Event Queue	**Execution Queue**
Executed after the execution stack becomes empty.	Executed first (before an event queue).

Events involved using already built-in objects like onClickListener, date and time	Basic JS code involved.

Eg:

```
function waitforthreeseconds(){
        var ms = 3000 + new Date().getTime();
        while (new Date() < ms) {}
        console.log("finished function");
}

function clickhandler(){
        console.log('Event clicked');
}

document.addEventListener('click',clickhandler);
waitforthreeseconds();
console.log('finished execution');
```

Explanation:

In the above code document.addEventListenter is a part of an event and as the event queue is done after the Event Queue and the rest of functions are a part of the execution queue and are executed first. And remember the event queue would be executed only after clicking in this case(due to onClickListener)

Output:

```
finished function
finished execution
Event clicked
```