Introduction to JavaScript CS50

Monday, June 13, 2022 7:39 PM

⇒ Browsers only understand Javascript.

⇒ Java script is Interpreted

i) Each browser has its own Javascript engine, which either interpret the code or compile it in some way

ii) They all follow ECMAScript standard, but may differ for any thing which is not standardise by ECMA.

⇒ Types

i) Dynamic Typing

ii) Primitive types (No methods)

iii) Objects

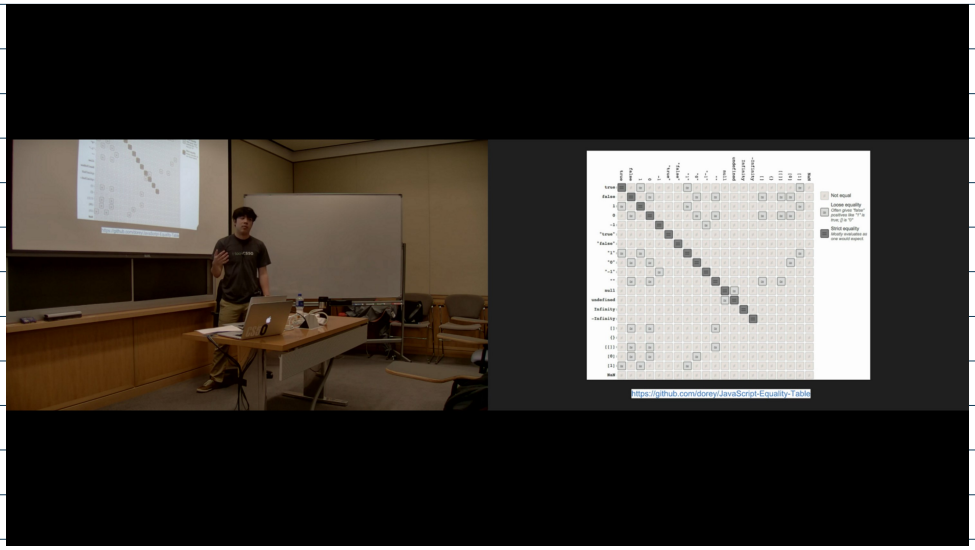
⇒ Type casting ? Coercion ✓

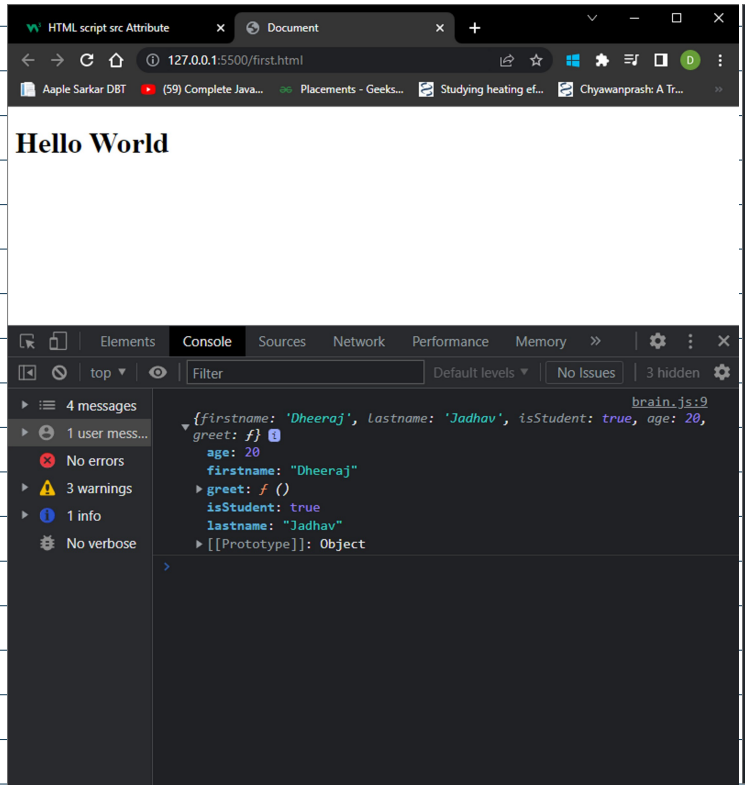
```
var x = 42
var explicit = String(x)        // explicit = "42"
var implicit = x + ""           // implicit = "42"
```

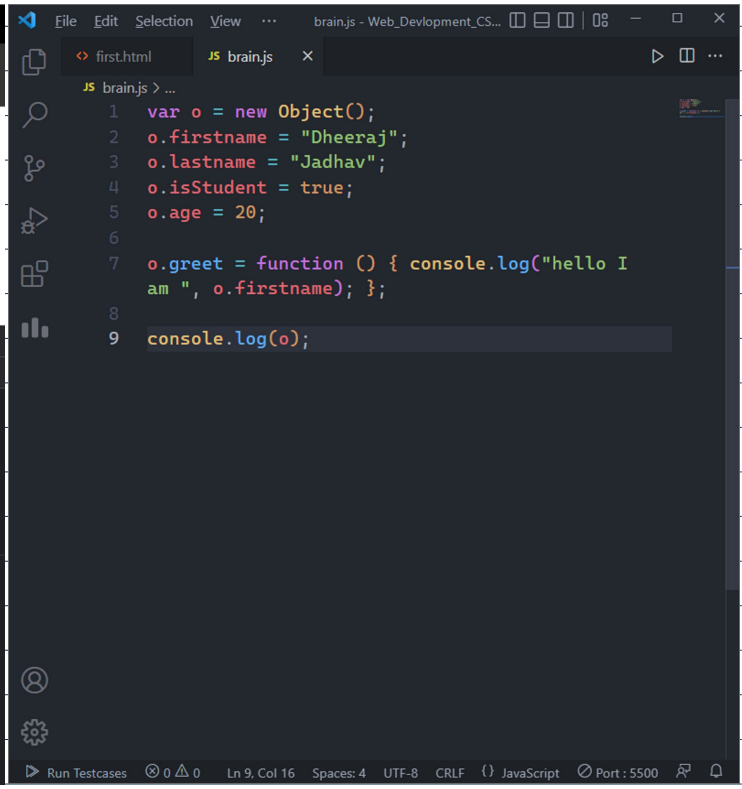
⇒ == vs ===

→ == coeres the types

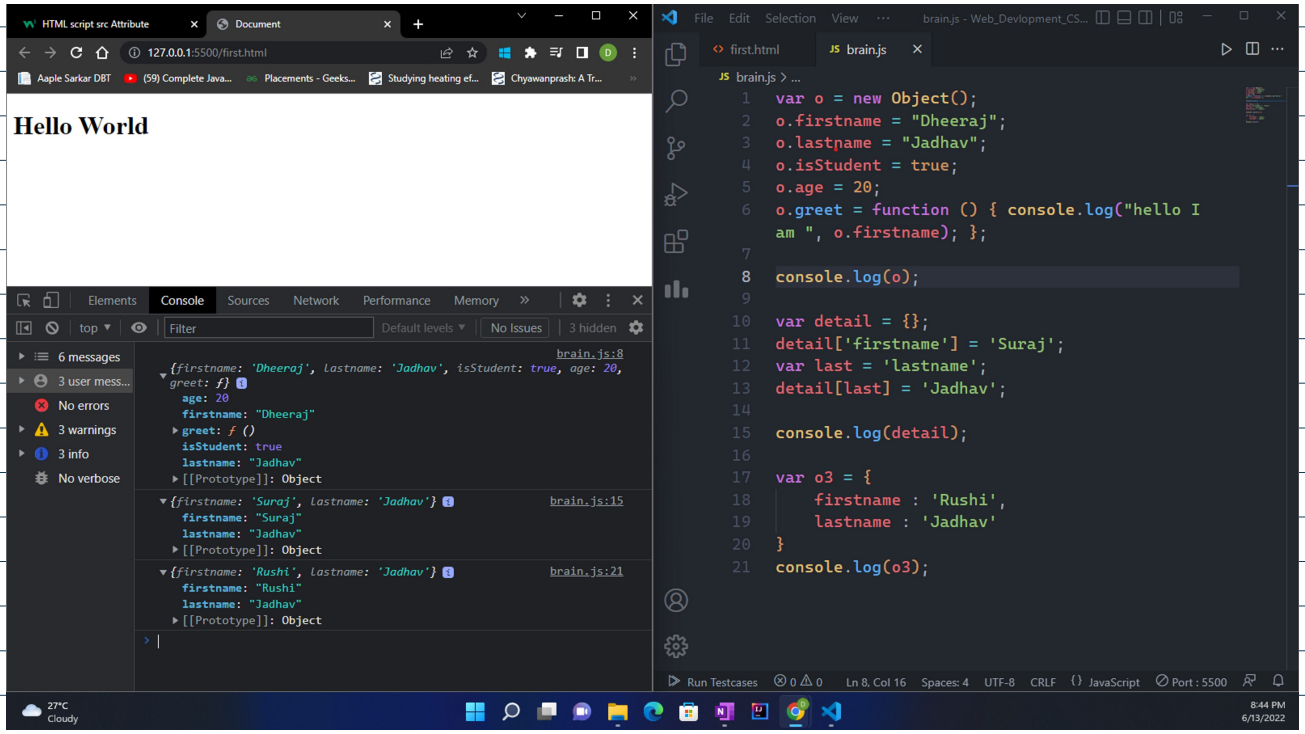
→ === requires equivalent types



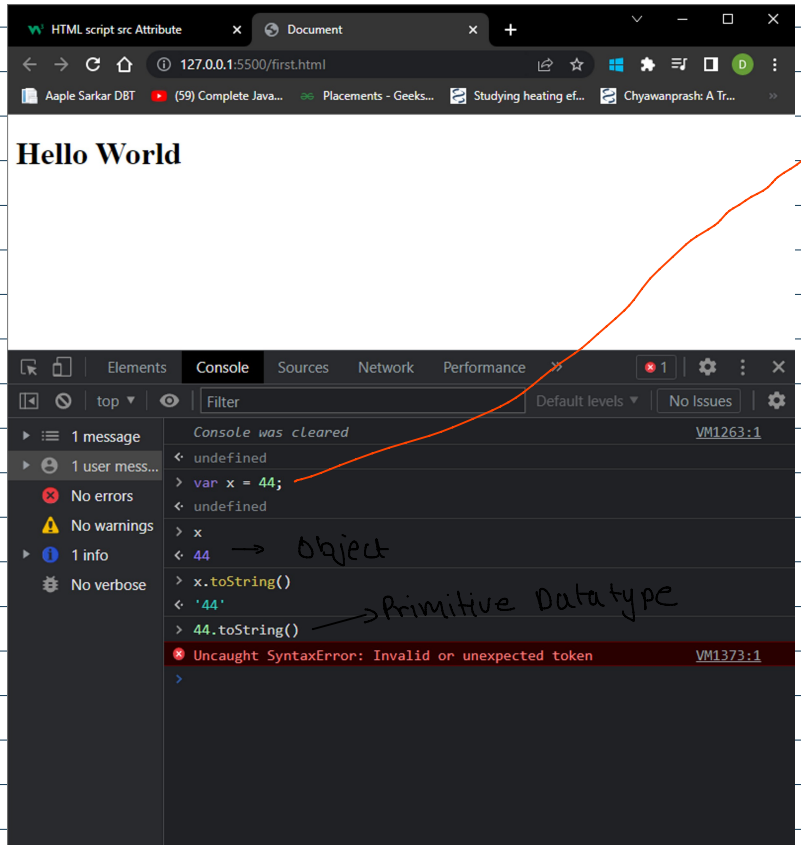




➤ Different ways to create an object



Prototyping



=> Objects have methods associated with them.

=> Primitives have no methods associated with them.

=> Now x is an instance of an object.

-> If x has its own method i.e.

```
var x = []
x.test = function () {}
```

-> As x is an array object lets assume that array has a its own test method

x → test()

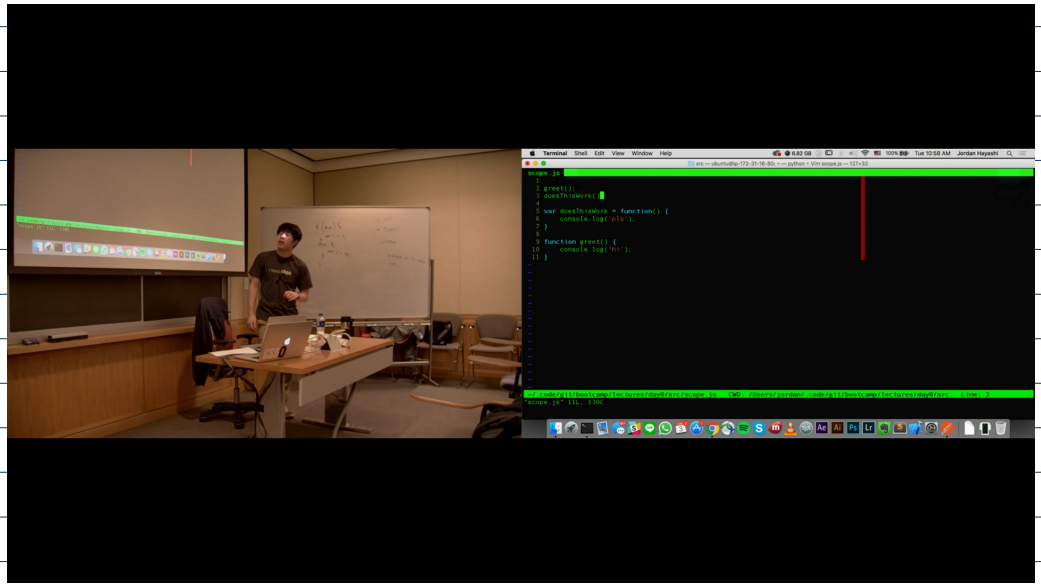
array → test()] => built-in

user-defined

=> now if test() of array is riddable, then user-defined test() will override, built in test().

i.e if we call x.test(), user-defined test() will be called.

How Javascript Engine work?



=> Before execution of code Javascript Interpretor, do following things.

i) If a function is defined it will be saved in the memory.
ii) If a variable is there, then its name will be declared, value will not be stored.

b will not be stored.

```
Console.log(i);
Greet();

Function greet() {
   Console.log('hi');
}

Var I =42;
```

→ Let's say we have this code

=> so, before execution

i) greet will be stored in memory
ii) i will be declared, not initialized.

⇓

Now at the time of execution
=> though greet() is called before its defined, it will be executed error free.
=> As it was already stored in memory before execution

⇓

Now, In case of variable i

=> there will be an error

⇓

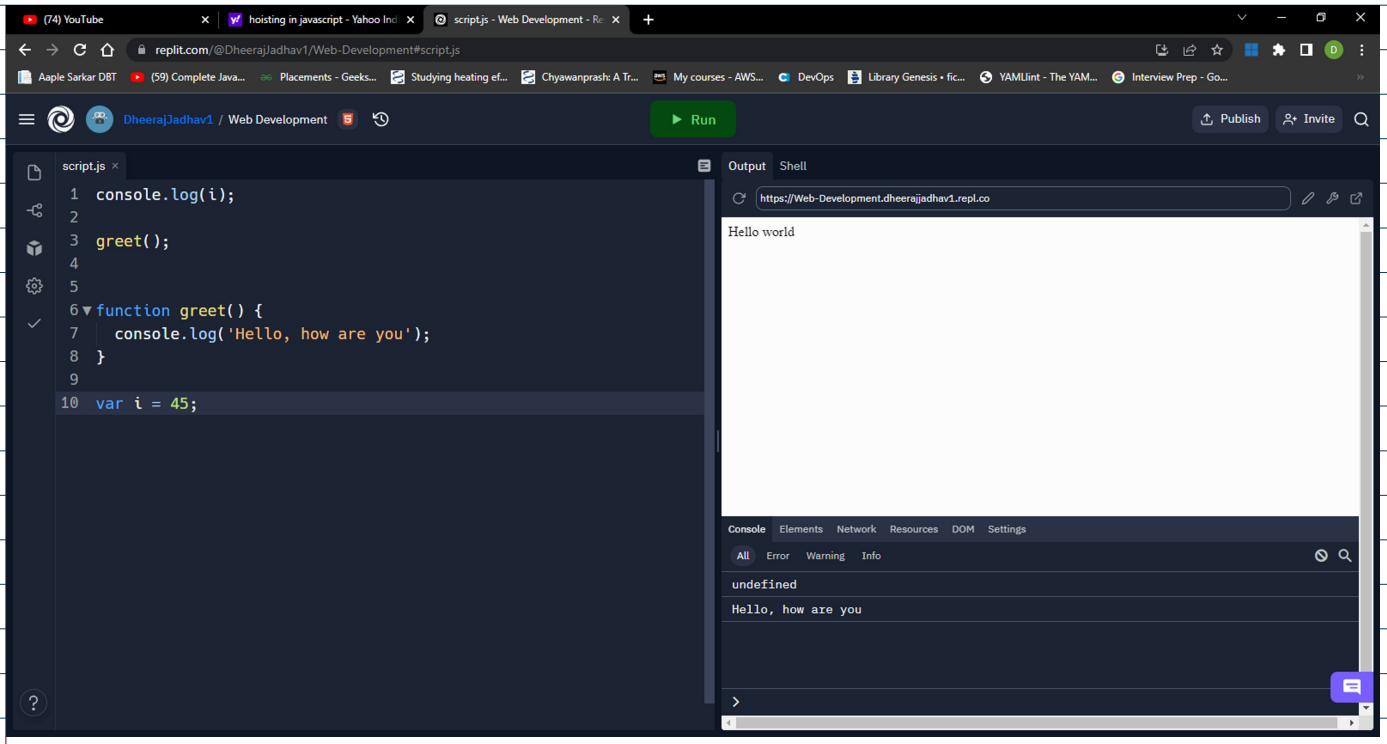
Because, i was declared in memory but it's value was not stored, it's value was defined later in the code

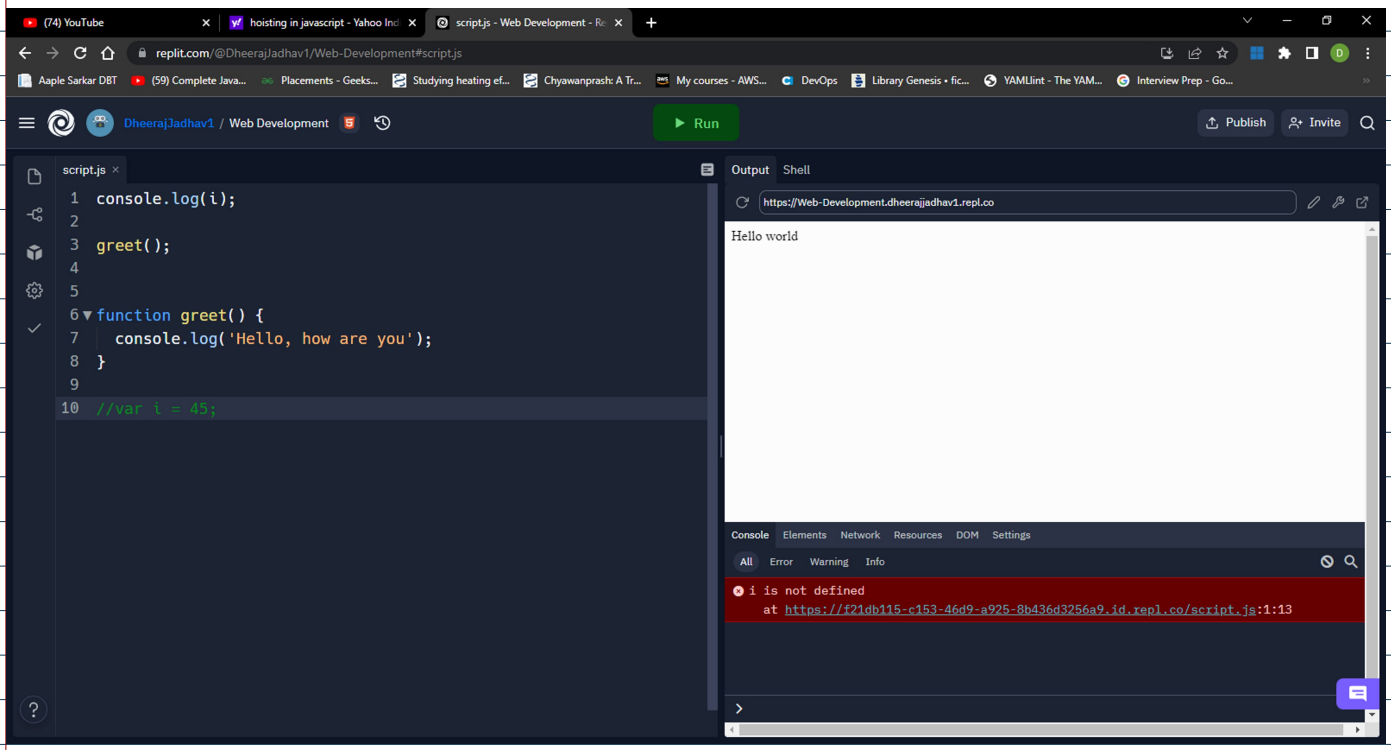
⇓

=> So, when i was called in the first line, i was undefined thats why it will give error

WHAT IF 2

=> we don't declare i later in the code
=> Error will be shown, as it was not even declared in the code.

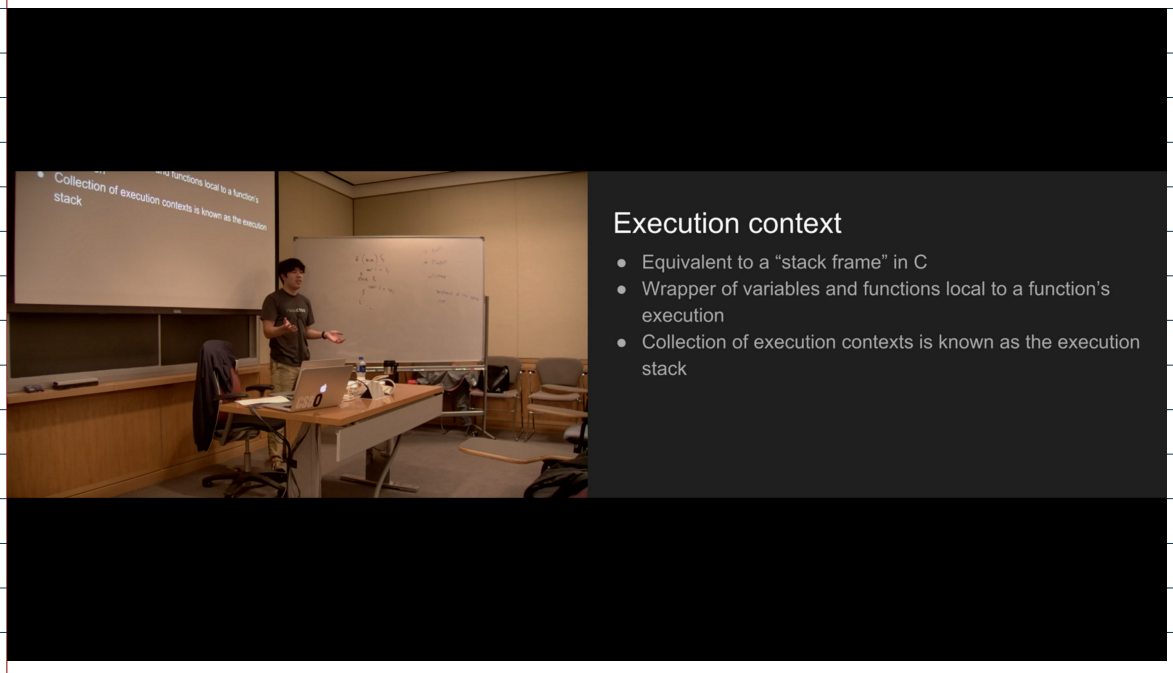




This is called Hoisting in Javascript.

Hoisting :- It allow us to use functions and variables before they are declared.

https://www.freecodecamp.org/news/what-is-hoisting-in-javascript/ => Hoisting in Detail



- Var vs let vs const

APRIL 2, 2020 / #JAVASCRIPT

Var, Let, and Const – What's the Difference?

Sarah Chima Atuonwu

A lot of shiny new features came out with ES2015 (ES6). And now, since it's 2020, it's assumed that a lot of JavaScript developers have become familiar with and have started using these features.

While this assumption might be partially true, it's still possible that some of these features remain a mystery to some devs.

One of the features that came with ES6 is the addition of `let` and `const`, which can be used for variable declaration. The question is, what makes them different from good ol' `var` which we've been using? If you are still not clear about this, then this article is for you.

In this article, we'll discuss `var`, `let` and `const` with respect to their scope, use, and hoisting. As you read, take note of the differences between them that I'll point out.

Var

Before the advent of ES6, `var` declarations ruled. There are issues

let's get to understand `var` more before we discuss these issues.

Scope of var

Scope essentially means where these variables are available for use. `var` declarations are globally scoped or function/locally scoped.

The scope is global when a `var` variable is declared outside a function. This means that any variable that is declared with `var` outside a function block is available for use in the whole window.

`var` is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

To understand further, look at the example below.

```
var greeter = "hey hi";

function newFunction() {
    var hello = "hello";
}
```

Here, `greeter` is globally scoped because it exists outside a function while `hello` is function scoped. So we cannot access the variable `hello` outside of a function. So if we do this:

```
var tester = "hey hi";

function newFunction() {
    var hello = "hello";
}
console.log(hello); // error: hello is not defined
```

outside the function.

var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

```
var greeter = "hey hi";
var greeter = "say Hello instead";
```

and this also

```
var greeter = "hey hi";
greeter = "say Hello instead";
```

Hoisting of var

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that if we do this:

```
console.log (greeter);
var greeter = "say hello"
```

it is interpreted as this:

```
greeter = "say hello"
```

So `var` variables are hoisted to the top of their scope and initialized with a value of `undefined`.

Problem with var

There's a weakness that comes with `var`. I'll use the example below to explain:

```
var greeter = "hey hi";
var times = 4;

if (times > 3) {
    var greeter = "say Hello instead";
}

console.log(greeter) // "say Hello instead"
```

So, since `times > 3` returns true, `greeter` is redefined to `"say Hello instead"`. While this is not a problem if you knowingly want `greeter` to be redefined, it becomes a problem when you do not realize that a variable `greeter` has already been defined before.

If you have used `greeter` in other parts of your code, you might be surprised at the output you might get. This will likely cause a lot of bugs in your code. This is why `let` and `const` are necessary.

Let

`let` is now preferred for variable declaration. It's no surprise as it

let is block scoped

A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.

So a variable declared in a block with `let` is only available for use within that block. Let me explain this with an example:

```
let greeting = "say Hi";
let times = 4;

if (times > 3) {
     let hello = "say Hello instead";
     console.log(hello);// "say Hello instead"
 }
console.log(hello) // hello is not defined
```

We see that using `hello` outside its block (the curly braces where it was defined) returns an error. This is because `let` variables are block scoped .

let can be updated but not re-declared.

Just like `var`, a variable declared with `let` can be updated within its scope. Unlike `var`, a `let` variable cannot be re-declared within its scope. So while this will work:

```
let greeting = "say Hi";
greeting = "say Hello instead";
```

```
let greeting = "say Hi";
let greeting = "say Hello instead"; // error: Identifier 'greeting
```

However, if the same variable is defined in different scopes, there will be no error:

```
let greeting = "say Hi";
if (true) {
    let greeting = "say Hello instead";
    console.log(greeting); // "say Hello instead"
}
console.log(greeting); // "say Hi"
```

Why is there no error? This is because both instances are treated as different variables since they have different scopes.

This fact makes `let` a better choice than `var`. When using `let`, you don't have to bother if you have used a name for a variable before as a variable exists only within its scope.

Also, since a variable cannot be declared more than once within a scope, then the problem discussed earlier that occurs with `var` does not happen.

Hoisting of let

Just like `var`, `let` declarations are hoisted to the top. Unlike `var` which is initialized as `undefined`, the `let` keyword is not initialized. So if you try to use a `let` variable before declaration, you'll get a `Reference Error`.

Variables declared with the `const` maintain constant values. `const` declarations share some similarities with `let` declarations.

const declarations are block scoped

Like `let` declarations, `const` declarations can only be accessed within the block they were declared.

const cannot be updated or re-declared

This means that the value of a variable declared with `const` remains the same within its scope. It cannot be updated or re-declared. So if we declare a variable with `const`, we can neither do this:

```
const greeting = "say Hi";
greeting = "say Hello instead";// error: Assignment to constant va
```

nor this:

```
const greeting = "say Hi";
const greeting = "say Hello instead";// error: Identifier 'greetin
```

Every `const` declaration, therefore, must be initialized at the time of declaration.

This behavior is somehow different when it comes to objects

```
const greeting = {
    message: "say Hi",
    times: 4
}
```

while we cannot do this:

```
greeting = {
    words: "Hello",
    number: "five"
} // error:  Assignment to constant variable.
```

we can do this:

```
greeting.message = "say Hello instead";
```

This will update the value of `greeting.message` without returning errors.

Hoisting of const

Just like `let`, `const` declarations are hoisted to the top but are not initialized.

So just in case you missed the differences, here they are:

- `var` declarations are globally scoped or function scoped

scope; `let` variables can be updated but not re-declared; `const` variables can neither be updated nor re-declared.

- They are all hoisted to the top of their scope. But while `var` variables are initialized with `undefined`, `let` and `const` variables are not initialized.
- While `var` and `let` can be declared without being initialized, `const` must be initialized during declaration.

Got any question or additions? Please let me know.

Thank you for reading :)

Sarah Chima Atuonwu

I am a software engineer that is interested in making the web accessible for all. I love sharing knowledge so I write about things I learn and things I need to learn.
