Frontend Development 2 - les 1
Les van Casper Boutens

Javascript leeswijze
Wanneer JS onze code leest gaat hij eerst door de creation phase en daarna door naar de execution phase.

creation phase
- eerst een global object
- dan creation of this
- dan zet hij alle vars en functions in een memory

dan in de *execution phase* gaat hij het lezen zoals wij dat doen. Hij haalt sommige informatie dan al uit de memory

wanneer je een functie in een var stopt krijg je ook undefined

Arguments
```
function skills() {
        console.log(arguments);
}

skills('frontend', 'backend');
```

output:
```
dingen [
        "frontend",
        "backend"
];
```

Hoisting
Declarations van functions en variabelen naar boven worden geplaatst. Maar dus geen assignments.

Scopes (!)
current context of executions; welke variabelen, values etc zijn er zichtbaar en kunnen er gerefereerd worden.

Er zit een verschil tussen
- global scope: Is dus in de browser het window
  > Dingen die niet in functies zitten en dus algemeen zijn
- local scope
  > Is echt lokaal, dus binnen een functie worden er dingen tijdelijk opgeslagen

Uitleggend metafoor:
globaal: de stoppen in je meterkast die invloed hebben over je hele huis
lokaal: lichtknoppen in de kamers

nog een metafoor:
globaal: elke octopus is een inktvis
local: niet elke inktvis is een octopus

Er zijn ook nog verschillende soorten scopes:

- function scope
- block scope

Ook gelijk het verschil tussen var en let;
een '**var'** wordt gelimiteerd aan de '**function scope'** en de **'let'** wordt gelimiteerd aan de **'block scope'**.

Local scope en function scope zijn hetzelfde.

voorbeeld:

```
//Global

function greetings() { //function scope / local scope
	var henk = "henk" // function scope / local scope

	if(true) { // block scope, GEEN nieuwe local scope
		let henk2 = "henk2" // block scope
	}

	console.log(henk); // henk
	console.log(henk2); // ERROR, henk2 not defined
}
```

<u>Closures</u>
Het toegang hebben tot oude functions

```
// global scope
var henk1 = "henk1"

function func1() { //local scope
	var henk2 = "henk2"

	function func2() { //local scope
		var henk3 = "henk3"

		function func3() { //local scope
			var henk4 = "henk4"

			console.log(henk1)
```

```
            }
      }
}
```

Hier onder zie je een goed voorbeeld van hoe zo'n closure nou écht werkt. De local scope wordt namelijk alsnog uitgevoerd ondanks dat de function al is uitgevoerd. Dit heeft ook te maken met de setTimeout. Dus eerst geeft hij 'undefined' en 5 seconden later geeft hij 'Hello henk'.

```
function greetings(name) { //local scope
      setTimeout(function() { //local scope
            console.log('Hello ' + name)
      }, 5000)
}

greetings('henk')

> undefined
*5 seconden later*
> Hello henk
```