Javascript

Execution stack

---

1) Un peu de théorie – Execution stack (pile d'exécution) :

La compréhension du contexte d'exécution et de la pile d'exécution est essentielle pour comprendre d'autres concepts JavaScript tels que le hoisting (fiche 5), Scope (fiche 5 - idem) et les Closures (fiche 5.2).

Qu'est-ce qu'un contexte d'exécution?

Chaque fois qu'un code est exécuté en JavaScript, il est exécuté dans un contexte d'exécution et ajouté dans la pile d'exécution. Voici les contextes :

Contexte d'exécution global

L'environnement par défaut dans lequel ton code est exécuté pour la première fois. Tout le code qui n'est à l'intérieur d'une fonction devient un objet global appelé objet window et est **poussé en bas d'une pile d'exécution**. Sa valeur est ensuite définie au moment de l'exécution de cette pile.

Contexte d'exécution fonctionnel

Toujours en balayant ton script du haut vers le bas, dès que Javascript trouve un appel de fonction, il crée un contexte d'exécution fonctionnel pour cette fonction et le pousse **vers le haut de cette pile**. Chaque fonction a son propre contexte d'exécution, mais ce contexte est créé seulement lorsque la fonction est appelée.

Les Variables d'Objet (ou VOs ou Variable Objects) sont les informations récupérées pendant la création de la pile d'exécution, à savoir : **les arguments de fonction, les déclarations de variable et de fonction**

Voici un exemple :

```
var name = "Vanessa";

function first() {
var a = "Hello";
second();
var x = a + name
}

function second() {
var b = "Hi";
third();
var y = b + name
}

function third() {
var b = "Hey";
var z = c + name
}

first();
```

Comment Javascript lit ce script :

<table>
<tr><td>1</td><td colspan="2">VO – Variable Object / Objets des variables</td></tr>
<tr><td>Scope
THIRD()</td><td>VO third
c | undefined

z | undefined</td><td></td></tr>
<tr><td>Scope
SECOND()</td><td>VO second
b | undefined

y | undefined</td><td></td></tr>
<tr><td>Scope
FIRST()</td><td>VO first
a | undefined

x | undefined</td><td></td></tr>
<tr><td>Scope
GLOBAL()</td><td>VO global
first()
second()
third()
name | undefined (au moment de la création du ES)/ Vanessa (au moment de l'exécution du ES)</td><td>Javascript balaya ton code et ajoute toutes les variables globales, arguments et fonctions dans la pile d'éxecution (voir hoisting, fiche 5)</td></tr>
<tr><td colspan="3">EXECUTION STACK (ou ES ou contexte d'exécution)</td></tr>
</table>

```
var name = "Vanessa";

function first() {
var a = "Hello";
second();
var x = a + name
}

function second() {
var b = "Hi";
third();
var y = b + name
}

function third() {
var b = "Hey";
var z = c + name
}

first();
```

2	**VO – Variable Object / Objets des variables**	
Scope THIRD()	VO third c | undefined z | undefined	
Scope SECOND()	VO second b | undefined y | undefined	
Scope FIRST() *(peut aussi accéder aux VOs de Scope GLOBAL)*	VO first a | undefined (au moment de la création du ES)/ Hello (au moment de l'exécution du ES) Création du ES de second() x | undefined	Javascript entre dans la première fonction. Il récupère la déclaration de variable a dans un premier temps, ensuite il créé le ES de second() et la déclaration de variable de x. Une fois celle fait, il remplie le variable a avec sa valeur
Scope GLOBAL()	VO global first() second() third() name | Vanessa	
EXECUTION STACK (ou ES ou contexte d'exécution)		

```
var name = "Vanessa";

function first() {
var a = "Hello";
second();
var x = a + name
}

function second() {
var b = "Hi";
third();
var y = b + name
}

function third() {
var b = "Hey";
var z = c + name
}

first();
```

3	**VO – Variable Object / Objets des variables**	
THIRD()	VO third c | undefined z | undefined	
SECOND() *(peut aussi accéder aux VOs de Scope GLOBAL + Scope FIRST)*	VO second b | undefined / ensuite Hi y | undefined	Javascript entre dans la deuxième fonction. Il récupère la déclaration de variable b dans un premier temps, ensuite il créé le ES de third() et la déclaration de variable de y. Une fois celle fait, il remplie le variable b avec sa valeur
FIRST()	VO first a | Hello x | undefined	
GLOBAL()	VO global first() second() third() name | Vanessa	
EXECUTION STACK (ou ES ou contexte d'exécution)		

```
var name = "Vanessa";

function first() {
var a = "Hello";
second();
var x = a + name
}

function second() {
var b = "Hi";
third();
var y = b + name
}

function third() {
var b = "Hey";
var z = c + name
}

first();
```

<table>
<tr><td>4</td><td colspan="2">VO – Variable Object / Objets des variables</td></tr>
<tr><td>THIRD()
(peut aussi accéder aux VOs de Scope GLOBAL + Scope FIRST + Scope SECOND)</td><td>VO third
<table><tr><td>c</td><td>undefined / ensuite Hey</td></tr></table>
<table><tr><td>z</td><td>Undefined / ensuite Hey
+ Vanessa</td></tr></table></td><td>Javascript entre dans la troisième fonction. Il récupère la déclaration de variable c dans un premier temps, et la déclaration de variable de z. Une fois celle fait, il remplie le variable c avec sa valeur. Désormais, Javascript peut executer cette fonction et remplir la valeur de la variable z</td></tr>
<tr><td>SECOND()</td><td>VO second
<table><tr><td>b</td><td>Hi</td></tr></table>
<table><tr><td>y</td><td>undefined</td></tr></table></td><td></td></tr>
<tr><td>FIRST()</td><td>VO first
<table><tr><td>a</td><td>Hello</td></tr></table>
<table><tr><td>x</td><td>undefined</td></tr></table></td><td></td></tr>
<tr><td>GLOBAL()</td><td>VO global
first()
second()
third()
<table><tr><td>name</td><td>Vanessa</td></tr></table></td><td></td></tr>
<tr><td colspan="3">EXECUTION STACK (ou ES ou contexte d’exécution)</td></tr>
</table>

5	**VO – Variable Object / Objets des variables**	
THIRD()	VO third c — Hey z — Hey Vanessa	Une fois crée et exécutée, cette fonction est enlevée de la pile d'exécution
SECOND()	VO second b — Hi y — undefined / ensuite Hi Vanessa	Et Javascript exécute la fonction second(), crée déjà dans la pile
FIRST()	VO first a — Hello x — undefined	
GLOBAL()	VO global first() second() third() name — Vanessa	
EXECUTION STACK (ou ES ou contexte d'exécution)		

6	**VO – Variable Object / Objets des variables**	
SECOND()	VO second b — Hi y — Hi Vanessa	Une fois crée et exécutée, cette fonction est enlevée de la pile d'exécution
FIRST()	VO first a — Hello x — undefined / ensuite Hello Vanessa	Et Javascript exécute la fonction first(), crée déjà dans la pile
GLOBAL()	VO global first() second() third() name — Vanessa	
EXECUTION STACK (ou ES ou contexte d'exécution)		

7	VO – Variable Object / Objets des variables	
FIRST()	VO first a | Hello x | Hello Vanessa	Une fois crée et exécutée, cette fonction est enlevée de la pile d'exécution
GLOBAL()	VO global first() second() third() name | Vanessa	Et revient à scope global, avant que le contexte d'exécution globale soit enlevé à son tour de la pile
EXECUTION STACK (ou ES ou contexte d'exécution)		

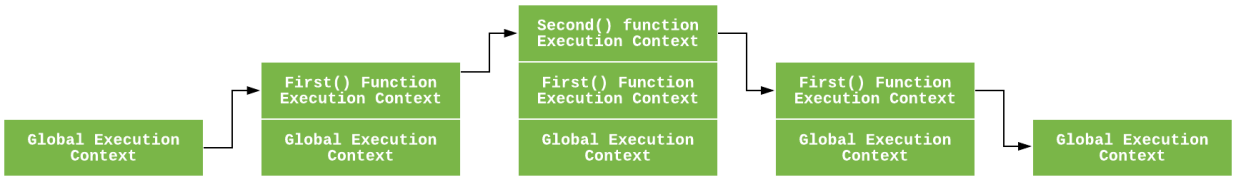
Voici un schéma du contexte d'exécution du code suivant :

```
//contexte global
var a = 'Hello World!';

//contexte fonctionnelle 1
function first() {
  console.log('Inside first function');
  second();
  console.log('Again inside first function');
}

//contexte fonctionnelle 2
function second() {
  console.log('Inside second function');
}

//contexte global
first();
console.log('Inside Global Execution Context');
```



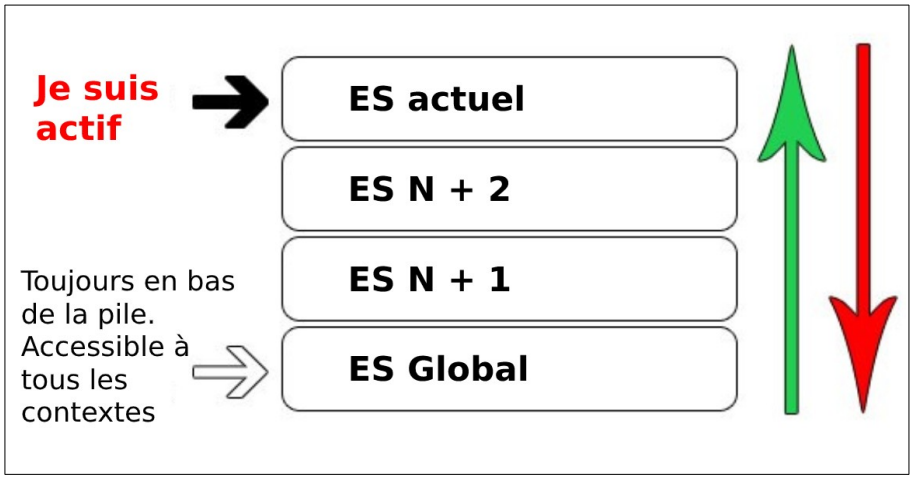
Comment le contexte d'exécution est-il créé et ensuite exécuté ?

Le cycle de vie du contexte d'exécution se compose de trois parties: la phase de création, la phase d'exécution du code et la phase d'achèvement de l'exécution

Phase de création: dans cette phase, le contexte d'exécution crée des VOs, ou variables objets (toutes les appels de fonction, arguments ou déclarations de fonctions), établit des scopes, détermine la valeur de «this» (voir ci-dessus).

Phase d'exécution du code: une fois la création terminée, le code est exécuté : il assigne des valeurs aux variables déclarées et execute les scripts à l'intérieur des fonctions.

Étape d'achèvement: après l'exécution, le contexte d'exécution commence à enlever le scope exécuté, puis la mémoire occupée est libérée



Tous les VOs des scopes en bas de la pile sont accessibles aux scopes supérieurs.

L'opérateur .this

L'objet JavaScript représentant le contexte dans lequel le code courant est exécuté.

Exemples

`console.log(this);`	`// « this » réfère à l'objet global. Donc «this» fait référence à l'objet globale lorsque vous êtes dans la portée globale.`
	`>> console.log(this);`
	`▸ Window about:home`

`function test1(){`
`console.log(this);`
`}`
 	
`test1();`
	`// « this » ici réfère aussi à l'objet global. Donc «this» fait référence à l'objet globale lorsque vous appelez une fonction de cette façon.`
	`>> function test1(){`
	`console.log(this);`
	`}`
	
	`test1();`
	`▸ Window about:home`

`function Personne(nom_complet, age, administrateur) {`
`this.nom_complet = nom_complet;`
`this.age = age;`
`this.administrateur = administrateur;`
`this.afficherThis = function() {`
`console.log(this);`
`}`
`}`
 	
`var personne = new Personne("François chaloux", 32, false);`
`personne.afficherThis();`
	`// Lorsqu'une fonction est appelée en tant qu'une méthode d'objet, «this» fait référence à l'objet qui appelle la méthode.`
	`>> ▸ function Personne(nom_complet, age, administrateur) {`
	`this.nom_complet = nom_complet;`
	`this.age = age;`
	`this.administrateur = administrateur;`
	`this.afficherThis = function() {…`
	`▸ Object { nom_complet: "François chaloux", age: 32, administrateur: false, afficherThis: afficherThis() }  debugger eval code:6:9`

Si le « this » est exécute par une méthode
(une fonction à l'intérieur d'un objet), il point vers l'objet
Sinon, vers l'objet global