

Javascript

#01.2

Matthieu Nicolas
Licence Pro CIASIE

Plan

- Variables
- Égalités
- Closures
- ES6 Modules

Variables

Javascript

#01.2

Variables

- `var a = 1;`
- `scope` = fonction englobante (et non pas block)
- Possible de re-déclarer plusieurs fois une variable
- si pas `var` => variable GLOBALE

Quand utiliser var?



<http://gph.is/l1oZ8Nw>

Variables

- `let a = 1;`
- `scope` = block englobant (et non pas seulement la fonction)
- Plus possible de re-déclarer une variable

Variables

- `const a = 1;`
- `scope` = block englobant (et non pas seulement la fonction)
- Plus possible de modifier la valeur de la variable
- Par contre, droit de modifier l'objet si ça en est un

Egalités

Javascript

#01.2

2 égalités

- ==

- conversion de type

```
1  0 == "0" .....// true
2  0 == "" .....// true
3
4  "test" == "test" .....// true
5
```

- ===

- pas de conversion

```
1  0 === "0" .....// false
2  0 === "" .....// false
3
4  "test" === "test" .....// true
5
```

Pourquoi ne pas utiliser ==?

[https://www.reddit.com/r/ProgrammerHumor/
comments/88gniv/
old_meme_format_timeless_j
avascript_quirks/](https://www.reddit.com/r/ProgrammerHumor/comments/88gniv/old_meme_format_timeless_javascript_quirks/)



Objets et égalités

- Identité pour les objets
 - Ignore le contenu
 - Un objet n'est égal qu'à lui-même

```
1  const toto1 = { nom: "Toto" }
2  const toto2 = { nom: "Toto" }
3
4  toto1 === toto2 ..... // false
5  toto1 ==  toto2 ..... // false
6  toto1 === toto1 ..... // true
7  toto2 === toto2 ..... // true
8  |
```

Bonnes pratiques

- Ne jamais utiliser `==` !
- Utiliser `===`, sauf pour les objets
 - Utiliser des méthodes spécifiques
- NaN n'est pas égal à lui-même
 - Seul objet à faire cela

```
1  NaN === NaN .....// false
2
3  isNaN(NaN) .....// true
4  |
```

Bonnes pratiques

- Utiliser !! pour transformer une valeur en un VRAI booléen

```
1  !!true.....//true
2  !!false.....//false
3
4  !!0.....//false
5  !!1.....//true
6
7  !!"".....//false
8  !!"hello"....//true
9
10 !!undefined..//false
11 !!null.....//false
12
13 !!{}.....//true
14 |
```

Closures


Javascript

#01.2

Closures

- Une fonction a TOUJOURS accès à ce qui est accessible dans son scope
 - Au scope de sa définition
 - Au scope de son EXECUTION
- Accès par REFERENCES (pas copie)
- Accès gardé MÊME lorsque le scope a disparu...

Scope d'appel

```
1   const c = function () {  
2    ... return b  
3  }  
4  b ..... // undefined  
5  
6  const b = 1  
7  c() ..... // 1  
8
```

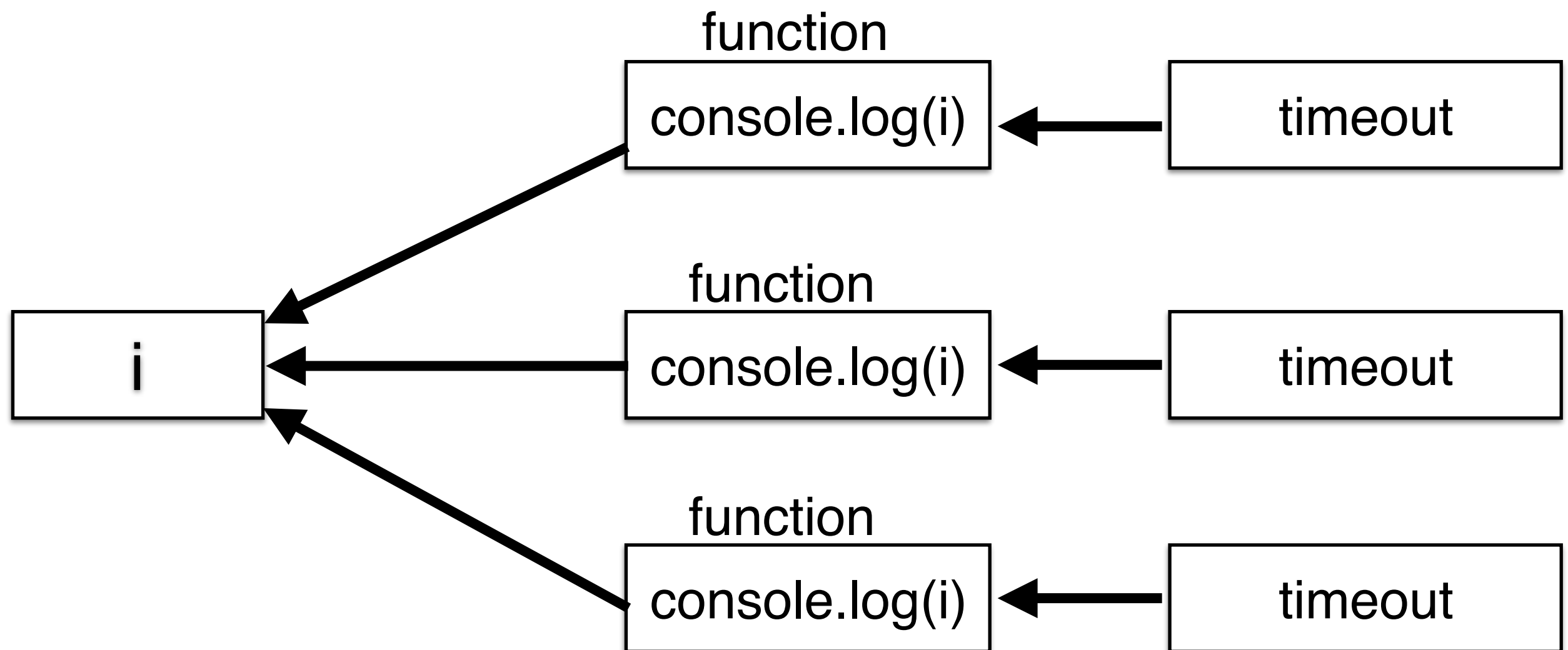

Scope de définition

```
1  const c = function() {  
2    ... let a = 1  
3    ... const f = function() {  
4      ... return a ... // Conserve une réf vers a  
5    ... }  
6    ... a = 3  
7    ... return f  
8  }  
9  
10 const d = c() ... // d est une fonction  
11 d() ... // accès à a, 3  
12
```

Par référence

```
1  let i
2  for(i = 0; i < 10; i++) {
3    . . . setTimeout(function () {
4      . . . | . . . console.log(i)
5      . . . }, 1000)
6  }
7
```

```
1  let i
2  for(i = 0; i < 10; i++) {
3    ... setTimeout(function () {
4    ... | ... console.log(i)
5    ... }, 1000)
6  }
7
```



Correction

```
1  let i
2  for(i = 0; i < 10; i++) {
3      ... (function (e) {
4          ... setTimeout(function () {
5              ... console.log(e)
6              ... }, 1000)
7          ... })(i)
8  }
9
```

Anonymous
wrapper

Passage d'un
entier
en paramètre :
passage
par COPIE

Utilisation des closures

- Variables privées
- Fonction immédiate
- Memoization
- Module

Variables privées

```
1  const f = function () {  
2      |  
3      |   const a = 1  
4      |  
5      |   return {  
6      |       |   getA: function() { return a }  
7      |       |   }  
8      |   }  
9  
10 const obj = f()  
11 obj.getA() // 1  
12
```

Fonction immédiate (IIFE)

```
1  const myObj = (function () {  
2    ... const a = 1  
3  
4    ... return {  
5      ... | ... getA: function() { return a }  
6      ... }  
7  })()  
8  
9  myObj.getA() ... // 1  
10
```

Memoization (mise en cache)

```
1  const f = (function() {  
2    ... let cache  
3  
4    ... const f = function() {  
5      ... if (!!cache) {  
6        ... cache = calculLourdEtLong()  
7      ... }  
8    ... return cache  
9    ... }  
10  
11  ... return f  
12  })()  
13  
14  f() // Effectue calculLourdEtLong()  
15  f() // Renvoie directement cache  
16
```


Module

```
1  const Module = (function () {  
2  
3      ... const obj = {}  
4      ... obj.prop = 1  
5      ... obj.method = function () { return 2 }  
6  
7      ... return obj  
8  })()  
9  
10 Module.prop ... // 1  
11  
12 Module.method() ... // 2  
13 |
```

ES6 Modules

Javascript
#01.2

But des modules

*Good authors divide their books into chapters and sections;
good programmers divide their programs into modules.*

— Preethi Kasireddy

- Décomposer notre application
 - Maintenabilité
 - Isolation (namespace)
 - Réutilisation

Design pattern

Module

- Permet de définir des variables, des objets ou des fonctions dans un scope isolé
- Permet de rendre accessible uniquement ce que nous souhaitons via un objet global
- ... mais complexe...

CommonJS

- Apparition de Node.js en 2009
- Propose un système de modules
 - Permet notamment de faire apparaître les dépendances entre modules

Exemple

```
1  let counter = 1
2
3  function increment() {
4    counter++
5  }
6
7  function decrement() {
8    counter--
9  }
10
11 function value() {
12   return counter
13 }
14
15 module.exports = {
16   counter: counter,
17   increment: increment,
18   decrement: decrement,
19   value: value
20 }
21
```

models/counter.js

```
1  const counter = require('./models/counter')
2
3  counter.increment()
4  console.log(counter.value()) // 2
5
6
7
8  console.log(counter.counter) // WARNING : 1
9
```

main.js

Du côté du navigateur...

- CommonJS n'est pas compatible avec les navigateurs...
- ... mais la communauté se rend compte qu'un vrai système de module, c'est cool !
- Asynchronous Module Definition (AMD) s'impose dans la communauté JS

Compatibilité navigateur - Node.js

- À la base, Node.js devait permettre de partager le code entre le client et le serveur
- Mais comme les systèmes de modules utilisés sont différents, cela échoue...
- Apparition de Universal Module Definition (UMD)
 - Permet de définir des modules à la fois compatibles avec le navigateur et Node.js

ES6 Modules

- CommonJS, AMD et UMD ne sont pas standards...
- ... mais avec ES6, JavaScript dispose enfin d'un système de modules !
- Avantages
 - Chargement asynchrone
 - Gestion des dépendances circulaires

Exemple - 1

main.js

```
1 import { increment, value } from "../models/counter.js"
2 // ... // N'oubliez pas l'extension du fichier
3
4 increment()
5 console.log(value()) // 2
6
```

```
1 export let counter = 1
2
3 export function increment() {
4   counter++
5 }
6
7 export function decrement() {
8   counter--
9 }
10
11 export function value() {
12   return counter
13 }
14
```

models/counter.js

```
1 <!DOCTYPE html>
2 <head>
3   <meta charset="utf8">
4   <title>Example</title>
5 </head>
6 <body>
7   <script type="module">
8     import { increment, value } from "../models/counter.js"
9
10     increment()
11     console.log(value()) // 2
12   </script>
13 </body>
14
```

index.html

Exemple - 2

```
1 export let counter = 1
2
3 function increment() {
4   ...counter++
5 }
6
7 function value() {
8   ...return counter
9 }
10
11 export { increment as inc }
12 export { value as val }
13
```

models/counter.js

```
1 import {
2   ...inc as incCounter,
3   ...val as valCounter
4 } from './models/counter.js'
5 // N'oubliez pas l'extension du fichier
6
7 incCounter()
8 console.log(valCounter()) // 2
9
```

main.js

- Possible de changer le nom lors de l'export ou de l'import

Exemple - 3

```
1  export let counter = 1
2
3  export function increment() {
4    counter++
5  }
6
7  export function decrement() {
8    counter--
9  }
10
11 export function value() {
12   return counter
13 }
14
```

models/counter.js

```
1  import * as counter from './models/counter.js'
2  // N'oubliez pas l'extension du fichier
3
4  counter.increment()
5  console.log(counter.value()) // 2
6
```

main.js

- Possible de regrouper tous les imports sous un namespace

Singletons

- Les ES6 Modules sont des Singletons
 - Ils ne sont chargés qu'une fois
 - Les variables sont partagés entre tous les modules clients

Références

```
1 export let counter = 1
2
3 export function increment() {
4   ... counter++
5 }
6
7 export function decrement() {
8   ... counter--
9 }
10
11 export function value() {
12   ... return counter
13 }
14
```

models/counter.js

```
1 import { counter, increment, value } from "../models/counter.js"
2   ... // N'oubliez pas l'extension du fichier
3
4 increment()
5 console.log(value()) // 2
6
7
8 console.log(counter) // WARNING :: 2
9 counter = 3 // WARNING :: Type Error
10
```

main.js

- Les exports sont passés par référence
 - Contrairement à CommonJS
- Par contre, peut pas modifier leur valeur

Remarque

- Les ES6 modules ne sont pas encore standard dans Node
 - doit ajouter l'option “- - experimental-modules” à la commande
 - Exemple: “node - -experimental-modules main.js”
- Le standard est prévu pour courant octobre

Conclusion - 1

- Enfin un système de module en JS...
- ... mais on risque de mettre du temps à enterrer CommonJS, AMD et UMD

Conclusion - 2

- Faire ses modules avec ES6 Modules
- Des bundlers se chargent de transformer vers d'autres formats

Quelques références

- ES6 & Beyond
 - <https://github.com/getify/You-Dont-Know-JS>
- JavaScript Modules: A Beginner's Guide
 - <https://medium.freecodecamp.org/javascript-modules-a-beginner-s-guide-783f7d7a5fcc>