
JS

1. Qu'est-ce que Javascript

Javascript est un langage de programmation orienté **objet** à **prototype**

1. Qu'est-ce que Javascript

Prototypal

- la programmation orientée prototype est une programmation sans classe ni instance
- la base est un objet, un prototype sur lequel on crée de nouveaux objets

1. Qu'est-ce que Javascript

Interprété

- Javascript est un langage qui nécessite un interpréteur
- un interpréteur lit, analyse et compile le script pour l'exécuter
- il existe plusieurs interpréteurs

1. Qu'est-ce que Javascript

Les interpréteurs

- SpiderMonkey (Mozilla)
- Rhino (Mozilla)
- V8 (Google Chrome)
- Chakra (Internet Explorer)
- Opéra
- ...

2. Historique

Invention

- inventé en 1995 par **Brendan Eich** pour Netscape
- développé à la base pour être un langage côté serveur
- devait s'appeler **LiveScript** mais fut changé pour **Javascript**
- Mars 1996 Javascript est intégré dans Netscape Navigator 2.0
- Août 1996 Microsoft ajoute JScript dans Internet Explorer 3.0
- Juin 1997 Adoption d'un nouveau standard nommé EcmaScript

2. Historique

EcmaScript

- ECMA-262
- EcmaScript version 3 (ES3) sortie 1999 :
  Version minimale de tous les navigateurs
- EcmaScript version 4 (ES4)
  Abandonnée
- EcmaScript version 5 (ES5) sortie 2009 :
  Version actuelle (navigateurs modernes)
- EcmaScript version 6 (ES6)
  Version non officielle

3. Physionomie d'un script

Les instructions

Une instruction représente une étape dans un programme :

```
var x = 5; // instruction 1
var y = 5; // instruction 2

console.log(x, y); // affiche 5 5 dans la console
```

3. Physionomie d'un script

Les instructions

```
var x = 5; // instruction 1
var y = 5; // instruction 2
var z = 5  // instruction 3

console.log(z);
```

- un script peut-être composé de plusieurs instructions
- les instructions se terminent par des points virgules
- le point virgule est optionnel mais fortement encouragé

3. Physionomie d'un script

Les instructions

Javascript permet la déclaration de plusieurs instructions sur une même ligne en les séparants par une virgule :

```
var x = 1, y = 2, z = 3;

console.log(x, y, z); // affiche 1 2 3
```

3. Physionomie d'un script

Bloc d'instructions

Il est possible de définir un bloc d'instructions en les déclarants entre deux accolades :

```
{
    var x = 4; // instruction 1
    var y = 8; // instruction 2
}

console.log(x, y);
```

3. Physionomie d'un script

Les expressions

Une expression est une instruction dont l'évaluation retourne une valeur :

```
var x = 4;  // instruction 1
var y = 8;  // instruction 2

x == y;     // expression qui retourne la valeur true
1 + 1;      // expression qui retourne la valeur 2
```

3. Physionomie d'un script

Les expressions

Instructions, expressions et blocs :

```
var x = 4; // instruction 1
var y = 8; // instruction 2

if (x == 4) {
    console.log("x vaut 4");
} else {
    console.log("x ne vaut pas 4");
}
```

3. Physionomie d'un script

Commentaires

Laissez vos commentaires :

```
// un commentaire en ligne

/*
    Bloc de commentaire
    sur plusieurs lignes ...
*/
```

3. Physionomie d'un script

Commentaires

Insérer des commentaires dans son code :

```
var foo = "bar"; // un commentaire après déclaration
var bar = "foo"; /* un commentaire après déclaration */

if (true) { // un commentaire dans un bloc d'instructions
    console.log(foo);
}
```

3. Physionomie d'un script

Les variables

En javascript, il y a une distinction entre la déclaration et l'assignation :

```
var x;          // déclaration
x = 10;         // assignation
var x = 10;     // déclaration et assignation

console.log(x); // évaluation de l'expression
```

3. Physionomie d'un script

Nommage des variables

- sensible à la casse
- les caractères accentués, espaces et tirets ne sont pas autorisés
- doit impérativement commencer par un caractère compris entre « A-Z », « a-z » et ou « $, _ »
- doit avoir du sens pour une meilleure compréhension du code

3. Physionomie d'un script

Les constantes

- une constante est un identifiant qui représente une valeur simple
- la valeur d'une constante ne peut jamais être modifiée pendant la durée d'exécution d'un script
- les constantes sont rendu possible depuis la version EcmaScript >= ES6

3. Physionomie d'un script

Les constantes

Déclaration et utilisation d'une constante :

```
const MA_CONSTANTE = 40;   // déclaration

console.log(MA_CONSTANTE); // utilisation
```

4. Structures de données

Types

- Javascript fourni 5 types primitifs
- tout le reste est Objet

4. Structures de données

Types primitifs

- string : chaine de caractères
- number : entier
- boolean : vrai ou faux
- null
- undefined
- les types *boolean*, *string* et *number* peuvent être encapsulés dans des objets primitifs Javascript

4. Structures de données

Truthy, Falsy

- Javascript défini une valeur comme étant un **équivalent boolean**

4. Structures de données

Falsy

- false
- 0
- ""
- null
- undefined
- NaN : Not an Number

4. Structures de données

Truthy

- tout le reste y compris un objet et array vide

4. Les opérateurs

Opérateurs arithmétiques

Table des opérateurs arithmétiques élémentaires :

Exemple	Nom
-a	Négation
a + b	Addition
a - b	Soustraction
a * b	Multiplication
a / b	Division
a % b	Modulo

4. Les opérateurs

Opérateurs arithmétiques

Exemple d'utilisation avec le modulo :

```
// le modulo retourne le reste d'une division par un entier

0 % 2;    // résultat 0    |  reste 0
1 % 2;    // résultat 1    |  reste 1
2 % 2;    // résultat 0    |  reste 0
3 % 2;    // résultat 1.5  |  reste 1

// les opérandes du modulo sont convertie en entier
```

4. Les opérateurs

Opérateurs d'assignation

Affectation d'une valeur à une variable :

```
var b;              // déclaration
b = 10;             // assignation avec l'opérateur « = »

b  = b + b;         // assignation avec expression
b += 10;            // raccourci équivalent à (b = b + b)

c  = "Hello";       // affectation d'une chaîne de caractères
c += " World";      // affection raccourci avec concaténation

console.log(c)      // Hello World
```

4. Les opérateurs

Opérateurs raccourcis

Incrémentation d'un nombre avec un raccourci :

```
var a = 1;

a++;                // est égale à l'expression « a = a + 1 »
console.log(a);     // vaut 2

a++;
console.log(a);     // vaut 3

a--;                // on décrément de 1
console.log(a);     // vaut 2
```

4. Les opérateurs

Opérateurs raccourcis

POST incrémentation :

```
var a = 1;

console.log(a++);   // affiche 1
console.log(a);     // affiche 2

console.log(a--)    // affiche 2
console.log(a);     // affiche 1
```

4. Les opérateurs

Opérateurs raccourcis

PRE incrémentation :

```
var a = 1;

console.log(++a);   // affiche 2
console.log(a);     // affiche 2

console.log(--a)    // affiche 1
console.log(a);     // affiche 1
```

4. Les opérateurs

Opérateurs sur les bits

Usage	Opérateur	Description
a & b	And (et)	Les bits positionnés à 1 dans **a** ET **b** retourne 1
a | b	Or (ou)	Les bits positionnés à 1 dans **a** OU **b** retourne 1
a ^ b	Xor (ou exclusif)	Les bits positionnés à 1 dans **a** OU **b** mais pas dans les deux retourne 1
~ a	Not (non)	Les bits positionnés à 1 dans **a** retourne 0, et vice-versa
a << b	Décalage à gauche	Décale les bits de **a**, **b** fois sur la gauche (chaque décalage équivaut à une multiplication part 2).
a >> b	Décalage à droite	Décale les bits de **a**, **b** fois par la droite (chaque décalage équivaut à une division par 2).

4. Les opérateurs

Opérateurs sur les bits

- les opérateurs binaire permettent de manipuler les bits dans un entier
- un bit ne peut avoir comme seule valeur 0 ou 1
- la comparaison binaire d'un entier se fait bit par bit

Opérateurs sur les bits

Table de conversion d'un nombre décimal en bits :

3	2	1	0	Puissance
8	**4**	**2**	**1**	**Nombre**
0	**1**	**1**	**1**	**7**
0	0	**1**	**1**	**3**
1	0	**1**	0	**10**

4. Les opérateurs

Opérateurs sur les bits

Exemple de conversion avec Javascript :

```
// conversion d'un nombre décimal en équivalent binaire

    Number(4).toString(2); // 100

// conversion de bits en équivalent décimal

    parseInt("100", 2);   //  4
```

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec le symbole & :

```
var a = 4;
var b = 5;

console.log(a & b); // affiche 4
```

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec ET (4 & 5) :

3	2	1	0	**Puissance**
8	**4**	**2**	**1**	**Nombre**
0	**1**	0	0	**4**
0	**1**	0	**1**	**5**
0	**1**	0	0	**Résultat**

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec OU (4 | 5) :

```
var a = 4;
var b = 5;

console.log(a | b); // affiche 5
```

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec OU (4 | 5) :

3	2	1	0	**Puissance**
8	**4**	**2**	**1**	**Nombre**
0	**1**	0	0	**4**
0	**1**	0	**1**	**5**
0	**1**	0	**1**	**Résultat**

Opérateurs sur les bits

Exemple de comparaison avec le OU exclusif (4 ^ 5) :

```
var a = 4;
var b = 5;

console.log(a ^ b); // affiche 1
```

Opérateurs sur les bits

Exemple de comparaison avec le NOT :

```
var a = 4;
var b = 5;

// les bits positionnés à 1 dans a et b
// sont positionnés à 0, et vice-versa.

console.log(a & ~b); // affiche 0
console.log(b & ~a); // affiche 1
```

Opérateurs sur les bits

Exemple de comparaison avec le OU exclusif (4 ^ 5) :

3	2	1	0	**Puissance**
8	**4**	**2**	**1**	**Nombre**
0	**1**	0	0	**4**
0	**1**	0	**1**	**5**
0	0	0	**1**	**Résultat**

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec le NOT (4 & ~5) :

3	2	1	0	**Puissance**
8	**4**	**2**	**1**	**Nombre**
0	**1**	0	0	**4**
0	**1**	0	**1**	**5**
0	0	0	0	**Résultat**

Opérateurs sur les bits

Exemple de comparaison avec le NOT (5 & ~4) :

3	2	1	0	Puissance
8	**4**	**2**	**1**	**Nombre**
0	**1**	0	0	**4**
0	**1**	0	**1**	**5**
0	0	0	**1**	**Résultat**

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec le décalage vers la gauche :

```
var a = 2;

console.log(a << 1); // affiche 4  soit 2*2     : 2²
console.log(a << 2); // affiche 8  soit 2*2*2   : 2³
```

4. Les opérateurs

Opérateurs sur les bits

Exemple de comparaison avec le décalage vers la droite :

```
var a = 8;

console.log(a >> 1); // affiche 4  soit 2/2
console.log(a >> 2); // affiche 2  soit 8/2/2
```

4. Les opérateurs

Opérateurs sur les bits

Exemple d'un cas d'utilisation :

```
const ERR_NOTICE  = 1;
const ERR_WARNING = 2;
const ERR_FATAL   = 4;
const ERR_ALL     = ERR_NOTICE | ERR_WARNING | ERR_FATAL;

// simulation d'une erreur provoquée par le système
var err = 2;

if (err & ERR_WARNING) {
    console.log("La méthode 'foo' n'existe pas");
}
```

4. Les opérateurs

Opérateurs de comparaison

Table des opérateurs de comparaison :

Exemple	Nom
a == b	Egal
b === b	Identique en valeur et en type
a != b	Différent
a !== b	Différent en valeur et en type
a < b	Plus petit que
a > b	Plus grand que
a <= b	Inférieur ou égal
a >= b	Supérieur ou égal

4. Les opérateurs

Opérateurs de comparaison

Comparaison de valeur et type :

```
// déclaration et assignation
var a = "4";
var b = 4;

// comparaison des valeurs et transtypage
a == b;  // true

// comparaison de la valeur et du type
a === b; // false
```

4. Les opérateurs

Opérateurs de comparaison

Comparaison de valeur et type :

```
// déclaration et assignation
var a = "4";
var b = 4;

// comparaison des valeurs et transtypage
a != b;  // false

// comparaison de la valeur et du type
a !== b; // true
```

4. Les opérateurs

Opérateurs de comparaison

Comparaison hiérarchique de valeur :

```
var a = 4;
var b = 9;

a > b;   // false
a < b;   // true
a >= b;  // false
a <= b;  // true
```

4. Les opérateurs

Opérateurs de comparaison

- false, 0 et "" sont ==
- null et undefined ne sont == qu'avec eux même
- NaN n'est égale avec personne, même pas avec lui même (IEEE 754)

Opérateurs de comparaison

Exemple de comparaisons étranges :

```
!!"0";          // true
!!0;            // false
"0" == 0;       // true

![];            // true
[] == false;    // true
```

4. Les opérateurs

Opérateurs logiques

Opérateur	Usage	Description
ET (&&)	a && b	Renvoie true si les deux opérandes valent true
OU (ll)	a ll b	Renvoie true si au moins une des opérandes valent true
NOT (!)	!a	Renvoie false si son opérande peut-être convertie en true, sinon il renvoie true

Opérateurs logiques

Exemple :

```
// ET
var a = true;
var b = 1;
console.log(a && b);      // true

// OU
var foo = false;
var bar = true;
console.log(foo || bar);  // true
```

4. Les opérateurs

Opérateurs logiques

Exemple :

```
// NOT
var a = true;
var b = 0;
console.log(a && !b);  // true

var bar = true;
console.log(!bar);     // false
```

Opérateurs logiques

Retour de valeurs :

```
var foo = 'Rouge' && 'Vert';
console.log(foo); // affiche Vert

var bar = false || 'Valeur par défaut';
console.log(bar); // Valeur par défaut
```

5. Les structures de contrôle

if / else

Structure :

```
if (expression) {
    // bloc d'instructions
} else {
    // bloc d'instructions optionnel
}
```

5. Les structures de contrôle

if / else

Exemple :

```
var score = 70;

if (score == 50) {
    console.log('Super');
} else if (score == 70) {
    console.log('Vous avez gagné une vie !');
} else {
    console.log('Votre score est de ' + score + ' points');
}
```

5. Les structures de contrôle

La boucle while

Structure :

```
while (expression) {
    // bloc d'instructions à répéter
}
```

La boucle while

Exemple :

```
var compteur = 0;

while (compteur < 3) {
    console.log('Le compteur est à : ' + compteur);
    compteur++;
}

// sortie
Le compteur est à : 0
Le compteur est à : 1
Le compteur est à : 2
```

5. Les structures de contrôle

La boucle do-while

- exécute au moins une instruction

5. Les structures de contrôle

La boucle do-while

Structure :

```
do {
    // bloc d'instructions à répéter
    // exécuté au moins une fois
} while (expression)
```

La boucle do-while

Exemple :

```
do {
    console.log('Je suis exécuté');
} while (false)
```

5. Les structures de contrôle

La boucle for

- permet de fournir les informations de contrôle en une seule fois

5. Les structures de contrôle

La boucle for

Structure :

```
for (initialisation; condition; incrementation) {
   // bloc d'instructions à répéter
};
```

La boucle for

Exemple :

```
for (var i = 0; i < 4; i++) {
    console.log('Exécution ' + i);   // sera exécuté 4 fois
};

// sortie
Exécution 0
Exécution 1
Exécution 2
Exécution 3
```

5. Les structures de contrôle

L'instruction switch

Exemple :

```
var langue;
switch (langue) {
    case 'en':
      console.log('Hello');
      break; // si absent passe au niveau suivant
    case 'fr':
    default:
      console.log('Bonjour');
};
```

5. Les structures de contrôle

Conditions ternaires

Structure :

```
Expression vrai ou faux ? alors : sinon
```

Conditions ternaires

Exemple :

```
var poids   = 21;
var couleur = poids > 20 ? 'rouge' : 'vert;

// Si poids supérieur à 20 alors couleur vaut rouge sinon vert

console.log(couleur); // affiche vert
```

6. Les chaîne de caractères

Création d'une chaîne

- de manière littérale
- via un constructeur

```
var a = "Javascript";          // type string
var b = new String("JS");      // type objet
```

6. Les chaîne de caractères

Propriétés

- constructor
- length
- prototype

6. Les chaîne de caractères

Méthodes

- charAt
- charCodeAt
- indexOf
- lastIndexOf
- match
- replace
- search

6. Les chaîne de caractères

Méthodes (suite)

- slice
- split
- substr
- substring
- toLowerCase, toUpperCase
- toLocaleLowerCase, toLocaleUpperCase

6. Les chaîne de caractères

Méthodes ES6

- contains (Firefox)
- repeat (Firefox)

7. Les objets

Représentation des objets

- les objets en javascript sont des collections de paires clef-valeur
- ils sont similaires aux
    - dictionnaires en Python
    - hashMaps en Java
    - tableaux associatifs en PHP

7. Les objets

Création d'un objet

- via un initialisateur : de manière littérale
- via un constructeur : avec une fonction

```
var a = { attr: "Javascript", moteur: "V8" };

var b = new Object({ attr: "Javascript", moteur: "V8" });
```

7. Les objets

Création d'un objet

Créer son propre constructeur

```
Tv = function (marque) {
    this.marque = marque;
}

var tv = new Tv("Captiva");

console.log(tv);                    // Object { marque="Captiva" }
```

7. Les objets

Type et instance

Tout objet est de type Object et est une instance de son constructeur

```
Tv = function (marque) {
    this.marque = marque;
}

var tv = new Tv("Captiva");

console.log(typeof tv);              // object
console.log(tv instanceof Tv);       // true
console.log(tv instanceof Object);   // true
console.log(tv instanceof String);   // false
```

Comparaison avec un langage de classe

Langage de classe	Langage de prototype
Les classes et les instances sont deux entités distinctes	Tous les objets sont des instances
Une classe est la définition d'un objet. On instancie une classe avec une méthode appelées constructeur	On définit et on crée un ensemble d'objets avec des fonctions qui sont des constructeurs
On crée un objet grâce à l'opérateur new	Identique aux langages de classe
On construit une hiérarchie d'objets grâce à la notion d'héritage en étendant la définition d'une classe existante	On construit une hiérarchie d'objet en assignant un prototype à un objet dans le constructeur de cet objet
Les objets héritent des propriétés appartenant à la chaîne des classe de la hierarchie	Les objet héritent des propriétés appartenant à la chaîne des prototypes de la hiérarchie
La définition de la classe défini exactement toutes les propriétés de toutes les instances d'une classe. Il est impossible d'ajouter des propriétés dynamiquement pendant l'exécution	Le constructeur ou le prototype défini un ensemble de propriétés initiales. Il est possible d'ajouter ou de retirer des propriétés dynamiquement, pour certains objets en particuliers ou bien pour l'ensemble des objets

Manipuler le prototype

Définir un prototype

```
function Personne(prenom) {
    this.prenom = prenom;
}

Personne.prototype; // Personne{}
Personne.prototype.direBonjour = function () {
    return "Bonjour, je suis " + this.prenom;
}

var p = new Personne('Toto');
p.direBonjour();
```

Manipuler le prototype

Extension du prototype

```
function Etudiant(prenom) {
    this.prenom = prenom;
}

Etudiant.prototype = new Personne();
Etudiant.prototype.lireUnLivre = function () {
    return "Je lis un livre";
}

var e = new Etudiant('John');
e.direBonjour();      // Bonjour, je suis John
e.lireUnLivre();      // Je lis un livre
```

Manipuler le prototype

Extension du prototype

```
// Résultat Etudiant

e.direBonjour();          // Bonjour, je suis John
e.lireUnLivre();          // Je lis un livre

e instanceof Object;      // true
e instanceof Personne;    // true
e instanceof Etudiant;    // true
```

Manipuler le prototype

Extension du prototype

```
// Résultat Personne

p.direBonjour();          // Bonjour, je suis Toto
p.lireUnLivre();          // N'a pas de méthode « lireUnLivre »

p instanceof Object;      // true
p instanceof Personne;    // true
p instanceof Etudiant;    // false
```

Manipuler le prototype

Remplacement de prototype

```
Personne.prototype.direBonjour = function () {
    return "Bonjour, je me nomme " + this.prenom;
}
e.direBonjour();   // Bonjour, je me nomme John
p.direBonjour();   // Bonjour, je me nomme Toto
```

Manipuler le prototype

Remplacement de prototype

```
Personne.prototype = {
    direBonjour: function() {
        return "Je veux pas dire bonjour";
    }
}
e.direBonjour();                    // Bonjour, je me nomme Toto
p.direBonjour();                    // Bonjour, je me nomme John

joe = new Etudiant("Joe");
joe.direBonjour();                  // Bonjour, je me nomme Joe

bar = new Personne("Bar");
bar.direBonjour();                  // Je veux pas dire bonjour
```

Héritage prototypal

Cascade

```
Personne = function (prenom) {
    this.prenom = prenom;
    this.age    = 90;
}

Troll = function (name) {
    this.name = name;
}

Troll.prototype = new Personne();

t = new Troll("Igor");
t.age;  // 90
```

Héritage prototypal

Surcharge

```
Personne.prototype.estMajeur = function() {
    return this.age >= 18;
}

t.estMajeur(); // true;
```

Héritage prototypal

Surcharge

```
Troll.prototype.age = 15;
Troll.prototype.growUp = function() {
    this.age++;
    return this;
}

t.estMajeur();          // false;
t.growUp();             // 15 + 1
t.growUp().growUp();    // 16 + 1 + 1
t.estMajeur();          // true
```

7. Les objets

Faut-il utiliser « new »

L'utilisation de l'opérateur **new** reste conceptuel et il donne l'impression au développeur, de créer une instance de classe.

8. Scope et this

Scope

Le scope est la partie du code dans lequel un identifiant n'a qu'une seul liaison

- un scope est défini par une fonction
- sinon est dans de scope global
- une variable est dans un scope que si elle précédé du mot clé var
- l'endroit où est initiée une variable dans le scope n'a pas d'importance

8. Scope et this

Scope

Exemple :

```
var x;                  // déclaration de x dans le scope global
x = 2;                  // initialisation de x dans le scope global

var f = function() {
    x = 7;              // x est défini dans le scope de la fonction f
    return x;
    var x;              // déclaration de x dans le scope locale
}

console.log(f());       // affiche 7
console.log(x);         // affiche 2
```

8. Scope et this

Scope

Pas de scope de bloc :

- {} un bloc d'instruction ne défini pas un scope
- idem pour les instructions

8. Scope et this

Scope

Pas de scope de bloc :

```
for (var i = 0; i < 5; i++) {
    var j = 2;
    console.log(i*j);
}
console.log(j);

// le code ci-dessus est l'équivalent à :

var i, j = 2;
for (i = 0; i < 5; i++) {
    console.log(i*j);
}
```

8. Scope et this

Scope

Le Hoisting – exemple 1 :

```
'use strict';

go();

function go() {
    console.log("Let's go !");
}
```

8. Scope et this

Scope

Le Hoisting – exemple 1 :

```
'use strict';

go();

function go() {
    console.log("Let's go !");
}

// affiche Let's go !
```

8. Scope et this

Scope

Le Hoisting – exemple 1 :

- la fonction déclarée et son contenu sont hissé « hoisted », permettant l'appel de la fonction accessible avant sa déclaration

8. Scope et this

Scope

Le Hoisting – exemple 1 :

```
// après passage de l'interpréteur ...

function go() {
    console.log("Let's go !");
}

go();
```

8. Scope et this

Scope

Le Hoisting – exemple 2 :

```
'use strict';

go();

var go = function() {
    console.log("Let's go again");
}
```

8. Scope et this

Scope

Le Hoisting – exemple 2 :

```
'use strict';

go();

var go = function() {
    console.log("Let's go again");
}

// TypeError: go is not a function
```

8. Scope et this

Scope

Le Hoisting – exemple 2 :

- la variable est hissé « hoisted », mais seulement la déclaration, pas son contenu
- la fonction n'a pas été assigné au moment de l'appel
- une erreur sera alors provoqué

8. Scope et this

Scope

Le Hoisting – exemple 2 :

```
// après passage de L'interpréteur ...

var go;
go();
go = function() {
    console.log("Let's go again");
}
```

8. Scope et this

Scope

Attention aux closures « fermetures »

- une closure est une fonction qui capture des variables libres dans le scope
- en particulier une fonction définie dans le corps d'une autre fonction
- une closure est lié au scope

8. Scope et this

Le mot clé this

Deux règles détermine la valeur de **this** :

- dépendant du contexte d'exécution
- méthode ou fonction

Le mot clé this

Méthode :

```
var objet = {
    maMethode: function () {
        console.log(this);            // Object { maMethode=function() }
        console.log(this === objet);  // true
    }
}

objet.maMethode();
```

8. Scope et this

Le mot clé this

Méthode :

```
function objet() {}
objet.prototype.maMethode = function () {
    console.log(this);  // Object { maMethode=function() }
}

var obj = new objet();
obj.maMethode();
```

8. Scope et this

Le mot clé this

Fonction :

- **this** fait toujours référence à l'objet window (ou global)

```
console.log(this);          // window

function f() {
    console.log(this);      // window
}

f();
```

8. Scope et this

Le mot clé this

Fonction :

- **this** fait toujours référence à l'objet window même dans un objet

```
var obj = {
    a: function () {
        console.log(this);          // Object { a:function() }
        function f() {
            console.log(this);      // Window
        }
        f();
    }
}
obj.a();
```

8. Scope et this

Le mot clé this

Astuce :

- Récupération de l'objet parent dans une fonction

```
var obj = {
    a: function () {
        self = this;
        function f() {
            console.log(self);      // Object { a=function () }
        }
        f();
    }
}
obj.a();
```

Changer le contexte d'une fonction

Il est possible de changer la valeur de **this** lors de l'appel à une fonction :

```
var f = function () {
    console.log(this);
}

f();                    // Window
f.call();               // Window
f.call({});             // Object
f.call(new Number());   // Number
```

8. Scope et this

call et apply

- **call** prend ses arguments individuellement
- **apply** prends ses arguments via un tableau

```
// call et apply permettent de changer la valeur de this

obj.call(thisArg[, arg1[, arg2[, …]]])

obj.apply(thisArg, [arg1, arg2, ...])
```

8. Scope et this

call et apply

exemple :

```
var f = {
    v: "fun",
    m: function (callback) {
        callback();
    }
}

f.m(function () {
    console.log(this);     // Window
});
```

8. Scope et this

call et apply

exemple :

```
var f = {
    v: "fun",
    m: function (callback) {
        callback.apply(this);
    }
}

f.m(function () {
    console.log(this.v);
    console.log(this);     // Object
});
```

9. Asynchronisme

Temps et vitesse

L'asynchronisme désigne le caractère de ce qui ne se passe pas à la même vitesse, que ce soit dans le temps ou dans la vitesse proprement dite, par opposition à un phénomène synchrone.

```
function test() {
    setTimeout(function () {
        console.log(1);
    });
    console.log(2);
}
test();                                    // affiche 2 puis 1
```

9. Asynchronisme

Temps et vitesse

Callback :

```
boutton = document.getElementById("boutton");

boutton.addEventListener("click", function () {
    // faire quelque chose
});
```

10. Les collections

Créer une collection

En Javascript une collection est un objet de données

```
var collection = {
    a: 1,
    b: 2,
    c: 3
};
```

10. Les collections

for in

Parcourir une collection :

```
for (attr in collection) {
    console.log(attr);                  // a, b, c
    console.log(collection[attr]);      // 1, 2, 3
}
```

10. Les collections

forEach

Parcourir une collection (ES5) :

```
['a', 'b', 'c'].forEach(function(v, i) {
    console.log(v, i);
});

// a 0
// b 1
// c 2
```

10. Les collections

filter

Filtrer une collection (ES5) :

```
var result = [30, 3, 12, 21].filter(function(val) {
    return val >= 20;
});

console.log(result);   // [30, 21]
```