

JAVASCRIPT

TELECOM PARISTECH

Slides préparés par [Jean-Claude Dufourd](#) et [Cyril Concolato](#).

JAVASCRIPT

- Historique
- Bases
 - Syntaxe de base, variables, fonctions, expressions, boucles, conditions
- Programmation « avancée »
 - Object, Array et autres objets globaux
 - Function, arguments, call, apply, map
- Particularités importantes:
 - closure,...
- Programmation Orientée Objet:
 - constructeur, héritage, surcharge,...
- Environnements (navigateur, node.js, JSON)

HISTORIQUE

- Langage de programmation créé par Brendan Eich, Netscape, en 1994/1995
 - En quelques semaines
 - Au plus fort de la bataille avec Microsoft ("browsers war", JScript)
 - D'abord nommé Mocha, puis LiveScript, puis JavaScript,
 - Appelé JavaScript pour apaiser une dispute avec Sun, en référence au langage Java
 - Mais très différent de Java: ne pas confondre!
- Standardisé sous le nom ECMA-262, ECMAScript...
 - 1ère version (1997)
 - 3e édition (1999)
 - 5e édition - ES5 (2009)
 - 6e édition - ES6 (2015)
 - 7e édition - en cours

PARTICULARITÉS DU LANGAGE

- Langage interprété (côté client)
- Inspiré des langages Scheme et Self (et Java pour la syntaxe)
- Très puissant, mais avec des erreurs de jeunesse dignes d'une bêta
- Vidéos: [The Good Parts](#) par Douglas Crockford

SYNTAXE DE BASE

■ Syntaxe inspirée du C, du Java

- Utilisation de {} pour séparer les blocs de code
- Utilisation de () pour les fonctions, les if, ...
- Commentaires
 - Sur une ligne avec //
 - Sur plusieurs ligne avec /* */
- Sensibilité à la casse: une variable a est différente de A

■ Quelques particularités:

- L'utilisation de ; après chaque expression n'est pas obligatoire, mais fortement conseillé!

```
a  
=  
3  
console.log(a)
```

équivalent à

```
a = 3;  
console.log(a);
```

LES BASES PAR L'EXEMPLE: LES VARIABLES

■ déclarer une variable

```
var x;
```

- On ne déclare pas le type d'une variable
- La valeur initiale, par défaut est la valeur spéciale: undefined

■ déclarer et assigner une valeur à une variable

```
var y = 0;
```

■ Les valeurs ont un type parmi: boolean, number, string, object, function, ou alors undefined

- vérifiable avec: typeof

■ Le type d'une variable peut changer au cours du temps:

```
x = 0;           typeof x; // nombre entier       -> "number"
x = -0.01;       typeof x; // nombre réel        -> "number"
x = "hello";     typeof x; // chaîne de caractères -> "string"
x = 'Hello world!'; typeof x; // chaîne de caractères -> "string"
x = true;        typeof x; // booléen            -> "boolean"
x = null;        typeof x; // objet null          -> "object"
x = undefined;   typeof x; // valeur non définie  -> "undefined"
x = NaN;         typeof x; // Not-a-Number        -> "number"
```

LES BASES PAR L'EXEMPLE: LES TABLEAUX

- Déclarer un tableau:

```
var primes = [2,3,5,7];
```

- Accéder à une entrée du tableau:

```
primes[0];           // → 2
primes.length;       // → 4
primes[primes.length - 1]; // → 7
primes["2"];          // → 5
primes.2;             // → SyntaxError: unexpected number
```

- Un tableau est dynamique, et peut contenir n'importe quoi:

```
var tableau = [];           // tableau initialement vide
tableau[0] = "test";        // l'entrée 0 est ajoutée et une valeur est
tableau[1] = true;          // on peut stocker des types différents
tableau[2] = {};
tableau[3] = null;
tableau[4] = undefined;
tableau[5] = [];
tableau[18] = 2;            // trou entre les index 5 et 18
```

LES BASES PAR L'EXEMPLE: LES OBJETS

- Un objet est un ensemble de **propriétés**, c'est-à-dire de couples (nom, valeur)
- Déclarer un objet (**object literal expression**):

```
var book = {  
  topic: "JavaScript",  
  fat: true,  
  "major version": 1 // espace dans le nom: à éviter !!!  
};
```

- Accéder à une propriété de l'objet:

```
book.topic; // → "JavaScript", syntaxe pointée  
book["fat"]; // → true, un objet est un tableau de propriétés
```

- On peut assigner une propriété à tout moment:

```
book.author = "Jean Dupont"; // ajout de la propriété 'author'  
book.contents = {}; // ajout de la propriété 'contents'  
book["title"] = "JavaScript pour les nuls"; // ajout de la propriété  
book["langue principale"] = "français"; // propriété avec un espace
```


LES BASES PAR L'EXEMPLE: TABLEAUX ET OBJETS

```
var empty = []; // tableau vide
empty.length; // → 0

var points = [ // tableau d'objets
  {x: 0, y: 1},
  {x: 1, y: 1},
  {x: 1, y: 1, z: 2}
];

var data = { // objet contenant des objets
  p1: {x: 0, y: 1},
  p2: {x: 1, y: 1}
};

var trials = {
  trial1: [[1, 2], [3, 4]], // tableau de tableau ~ matrice
  trial2: [[1, 2], [4, 6]]
};
```

LES BASES PAR L'EXEMPLE: ARITHMÉTIQUE

```
3 + 2;           // → 5
3 * 2;           // → 6
3 - 2;           // → 1
3 / 2;           // → 1.5
3 % 2;           // → 1    // modulo
"3"+"2";         // → "32"  // concaténation
"3"-"2";         // → 1    // conversion de type
var count = 0;
count++;
count--;
++count;
--count;
count += 2;
count -= 4;
count *= 5;
+"21";           // → 21    // convertit "21" en nombre 21
+"21toto";       // → NaN   // conversion impossible
parseInt("21");   // → 21    // convertit "21" en nombre 21
parseInt("21toto"); // → 21  // cherche un entier au début
parseFloat("21.5toto"); // → 21.5 // cherche un float au début
```

LES BASES PAR L'EXEMPLE: (IN-)ÉGALITÉS

```
var x=3, y=2;
x == y           // → false
x != y           // → true
x < y            // → false
x >= y           // → true
"two" == "three" // → false
"two" > "three"   // → true (ordre lexicographique)
false == (x < y)  // → true
(x == 2) && (y == 3) // → true
(x == 2) || (y == 4) // → true
!(x == y)         // → true
```

LES BASES PAR L'EXEMPLE: TESTS ET IF

```
if (b == 0) x = 4;

if (b == 0) { x = 4; }

if (b == 0) { x = 4; y = 2; }

if (b == 0) {
    x = 4;
    y = 2;
}
```

```
if (b == 0) {
    x = 4;
    y = 2;
} else {
    x = -4;
}

if (b == 0) {
    x = 4;
    y = 2;
} else if (b == 1) {
    x = -4;
}
```

ATTENTION AUX TESTS D'ÉGALITÉ

- == (resp. !=) teste l'égalité (resp. la non-égalité) après d'éventuels changements de type

```
2 == 2           // → true
"2" == 2         // → true !!!attention
"2" != 2         // → false !!!attention
```

- === (resp. !==) teste l'égalité (resp. la non-égalité) stricte sur les types d'origine

```
"2" === 2        // → false
"2" !== 2        // → true
```

Utiliser cette syntaxe le plus souvent possible

- inégalité et conversion

```
"22" > "3"       // → false
+"22" > "3"      // → true
```

TABLEAUX D'ÉGALITÉS

ATTENTION AUX TESTS SANS ÉGALITÉ

- Les tests suivants sont évalués à `false` (ce sont les seuls 6 cas) et donc le code dans la boucle ne s'exécute pas

```
if (false) { ... }  
if (0) { ... }  
if ("") { ... }  
if (null) { ... }  
if (undefined) { ... }  
if (NaN) { ... }
```

- Tout le reste s'évalue à `true` et donc le code dans la boucle s'exécute

```
if (true) { ... }  
if (1) { ... }  
if (-1) { ... }  
if ("true") { ... }  
if ("false") { ... }  
if ("1") { ... }  
if ("0") { ... }  
if (Infinity) { ... }  
if ([]) { ... }  
if ({}) { ... }  
if ([0]) { ... }  
if ([1]) { ... }
```

LES BASES PAR L'EXEMPLE: BOUCLES

```
while (...) {  
    ...  
}  
  
do {  
    ...  
} while (...);  
  
var i;  
for (i=0; i<10; i++) {  
    ...  
}  
  
// déclaration de la variable d'itération dans la boucle  
for (var j=0; i<10; i++) {  
    ...  
}  
  
// énumération des propriétés d'un objet  
for (var a in obj) {  
    ...  
}
```


LES BASES PAR L'EXEMPLE: SWITCH

```
// test d'egalité stricte
switch(type) {
    case "a": // string
        ...
        break;
    case 1: // number
        ...
        break;
    default:
        ...
}
```

LES FONCTIONS

FONCTIONS ET ARGUMENTS

- Les arguments sont séparés par des virgules, et utilisés dans l'ordre
- Les arguments non spécifiés ont la valeur undefined
- On peut accéder aux arguments via le tableau arguments

```
function f(x,y) {  
    console.log("x: "+x+", y: "+y+", z: "+arguments[2]);  
}  
  
f(1,2,3);    // x: 1, y: 2, z: 3  
f(1,2);      // x: 1, y: 2, z: undefined  
f(1);        // x: 1, y: undefined, z: undefined
```

PORTÉE DES VARIABLES

- en Java ou en C, les variables ont une portée bloc
- en JS une **portée fonction** (sauf en utilisant le mot clé ES6 let)

```
function test(o) {  
  var i = 0;  
  if (o !== null) {  
    var j = 0;  
    for(var k=0; k < 10; k++) {  
      console.log(k, i);  
    }  
    console.log(k); // la variable k est toujours accessible  
  }  
  console.log(j); // la variable j est toujours accessible  
}
```

VARIABLES ET PILE D'APPELS

- Comment déterminer quelle variable utiliser quand plusieurs variables locales à des fonctions imbriquées ont le même nom?

```
var currentScope = 0; // global scope
(function () {
  var currentScope = 1, one = 'scope1';
  alert(currentScope);
  (function () {
    var currentScope = 2, two = 'scope2';
    alert(currentScope);
    (function () {
      var currentScope = 3, three = 'scope3';
      alert(currentScope);
      alert(one + two + three);
   })();
  })();
})();
```

- Dans la pile d'appels, la variable utilisée est cherchée:
 - dans la fonction courante
 - dans la fonction appelante (en remontant)
 - dans le code en dehors des fonctions

VARIABLE NON-DÉCLARÉE

- Une variable non déclarée est automatiquement déclarée avec comme portée la fonction englobante (sauf si une variable de portée plus globale existe)

```
function maFonction() {  
    a = 0;  
}
```

équivalent à:

```
function maFonction() {  
    var a;  
    a = 0;  
}
```

mais la variable n'est pas assignée

```
function maFonction() {  
    console.log(a); // → undefined  
    a = 0;  
    console.log(a); // → 0  
}
```

PORTÉE DES FONCTIONS

- une déclaration de fonction est une déclaration de variable de type "function"

```
function run(obj) { ... }  
run(a);
```

est (à peu près) équivalent à:

```
run = function (obj) { ... } // variable dont la valeur est une fonction  
run(a);
```

- on peut définir une fonction à l'intérieur d'une fonction, elle ne sera pas accessible de l'extérieur (comme pour toute autre variable)

```
function run(obj) {  
  function myPrint(x) {  
    console.log(x);  
  }  
  myPrint(obj.a);  
  myPrint(obj.b);  
}  
run({a: 1, b: "toto"}); // affiche: 1 puis toto  
myPrint(2); // ReferenceError: myPrint is not defined
```

TYPAGE AVANCÉ

- Types **primitifs**: contient une valeur simple, pas de méthode
 - boolean: true, false
 - number: nombres entiers et réels (IEEE 754 attention à la précision), +Infinity, -Infinity, NaN

```
0.1 + 0.2; // → 0.30000000000000004
```

- string
- null
- undefined

```
1.toString(); // Uncaught SyntaxError: Unexpected token ILLEGAL(...)
true.toString(); → "true"
null.toString(); // Uncaught SyntaxError: Unexpected token ILLEGAL(...)
"toto".toString(); → "toto"
undefined.toString(); // Uncaught SyntaxError: Unexpected token ILLEGAL(...)
```


TYPAGE AVANCÉ

- Types complexes, c'est-à-dire avec des méthodes prédéfinies
 - Object et ses types dérivés
 - Boolean, Number, String, Array, Math, Date, RegExp, Function, Set, JSON ...

```
var b = new Boolean(true);  
b.toString(); // → "true"  
var n = new Number(3.14);  
n.toString(); // → "3.14"
```

ATTENTION AUX TYPES

- Les variables de type primitif ne sont pas des objets, ce qui peut donner lieu à des comportements bizarres (**type coercion**)

```
var s = "hello, world";  
typeof s; // → "string"  type primitif, pas un objet  
s.x = 15; // → 15        pas d'erreur car équivalent (new String(s))  
typeof s; // → "string"  s n'a pas changé de type  
s.x       // → undefined problème car l'objet String temporaire a disparu
```

- Dans ce cas, utiliser directement une variable de type Object, plus précisément de type String

```
s = new String("hello, world");  
typeof s; // → object: s est un objet à part entière  
s.x = 15; // → 15  
s.x;      // → 15 la propriété est persistente
```

STRING

```
var s = new String("hello, world");  
s.charAt(0);  
s.charAt(s.length-1);  
s.substring(1,4);  
s.slice(1,4);  
s.slice(-3);  
s.indexOf("l");  
s.lastIndexOf("l");  
s.indexOf("l", 3);  
s.split(",");  
s.replace("h", "H");  
s.toUpperCase();
```

MATH

```
Math.pow(2,53);  
Math.round(.6);  
Math.ceil(.6);  
Math.floor(.6);  
Math.abs(-5);  
Math.max(x,y,z);  
Math.min(x,y,z);  
Math.random();  
Math.PI;  
Math.E;  
Math.sqrt(3);  
Math.pow(3, 1/3);  
Math.sin(0);  
Math.log(10);  
Math.log(100);  
Math.LN10;  
Math.log(512);  
Math.LN2;  
Math.exp(3);
```

ARRAY

```
a = new Array();    // constructeur non recommandé
a = [];             // constructeur recommandé
a = [1, 2, 3];
a.length;           // → 3
a.push(4);
b = a.pop();         // → 4
delete a[1];         // a[1] vaut maintenant undefined
1 in a;              // → false
a.length;           // → 3 : longueur non modifiée par delete
a.join();            // → "1,2,3"
a.join(" ");         // → "1 2 3"
a.reverse();
a.sort();
a.concat(...);
a.slice(...);        // -1 est le dernier element, -3 l'antepenultième.
a.splice(...);       // chirurgie complexe
a.shift();           // = pop à gauche
a.unshift();         // push à gauche
a.toString();        // comme join()
```

ARRAY EN ES 5

```
a.forEach(f);           // appliquer une fonction
a.map(f);                // + retour tableau résultats
a.filter(f);             // selection selon prédicat
a.every(f);              // && sur prédicat
a.some(f);               // || sur prédicat
a.reduce(f,i);           // applique f à tous les éléments de gauche à droite
                        // et retourne le résultat accumulé
a.reduceRight(f,i);      // de droite à gauche
a.indexOf(i);
a.lastIndexOf(i);
Array.isArray(a);
```

THIS

- `this` est un mot-clé, similaire mais différent des autres langages (tel que Java ou C++)
- Le code JavaScript s'exécute (presque) toujours avec un `this` défini:
 - En dehors d'une fonction, `this` représente le contexte d'exécution global, c'est-à-dire:
 - L'objet `window` dans les navigateurs

```
this === window; // true
```

- L'objet `global` dans NodeJS

```
this === global; // true
```

- Dans une fonction:
 - l'objet sur lequel la fonction a été appelée (s'il existe),
 - le `this` du contexte d'exécution global (si pas d'objet appelant) en mode normal ou
 - `undefined` en "strict mode".

THIS - EXAMPLES

```
var A = {};  
function f() { return this === A; }  
A.g = function () {  
  this.z = 2;  
  return this === A;  
}  
f();           // → false  
A.g();         // → true  
A.z;          // → 2: la propriété z est bien assignée sur A  
this.z;       // → undefined: z n'est pas connu en dehors de A
```

```
function h() { this.x = 2; }  
var B = new h();           // toute fonction peut être un constructeur  
                           // dans ce cas, dans h: this === B → true  
B.x;                      // → 2
```


MODIFIER THIS

CALL, APPLY ET BIND

- Possibilité de donner un this explicite/différent de l'objet normal

```
var A = {  
  x: 2,  
  f: function (y, z) { console.log(this.x+y+z); }  
};  
A.f(1,2);           // 5: f est appelée avec this valant A  
var B = { x: 3 };  
B.f(1,2);           // TypeError: undefined is not a function  
A.f.call(B, 1, 2);   // 6: f est appelée avec this valant B  
A.f.apply(B, [1, 2]); // 6: f est appelée avec this valant B
```

- Possibilité de créer une fonction avec un this différent pour l'appeler plus tard

```
var g = A.f.bind(B, 1, 2); // création d'une fonction  
                             // this et les arguments sont prépositionnés  
g();                         // f est appelée avec this=B et les valeurs 1 et 2
```

THAT/SELF/ME

CLOSURES

- Particularité très importante de JS, à la base de nombreuses bibliothèques JS
 - Une fonction javascript peut retourner une valeur de type "function" (**high-order function**)
 - Les variables d'une fonction mise dans la pile d'appels restent disponibles après que cette fonction ait été dépilée

```
function checkscope() {  
  var scope = "local scope";  
  function f() { return scope; } // 'f' réalise une closure autour de  
  return f;  
}  
var x = checkscope(); // x est une fonction  
x(); // → "local scope"
```

- Permet d'éviter d'exposer des variables tout en permettant leur manipulations
 - Mécanisme d'**encapsulation**

CLOSURES: EXEMPLE D'ERREUR CLASSIQUE

```
function f() {  
  var a = [];  
  for (var i=0; i<3; i++) {  
    a[i] = function(){ console.log(i); };  
  }  
  return a;  
}  
  
var b = f();
```

```
b[0](); // → 3  
b[1](); // → 3  
b[2](); // → 3
```

■ Explication du problème:

- f n'est appelée qu'une seule fois: une seule variable i est présente dans l'espace réservé à f dans la pile d'appels
- Au moment d'appeler b[n], dans cet espace, i vaut 3

CLOSURES: CORRECTION

```
function g(j) {  
    return function() { console.log(j); };  
}  
  
function f() {  
    var a = [];  
    for (i=0; i<3; i++) {  
        a[i] = g(i);  
    }  
    return a;  
}  
  
var b = f();  
  
b[0](); // → 0  
b[1](); // → 1  
b[2](); // → 2
```

■ Solution:

- On résoud le problème en appelant n fois une autre fonction
- qui va créer n espaces de stockage dans la pile d'appels pour j

CLOSURES: CORRECTION (2)

```
function f() {  
  var a = [];  
  for (i=0; i<3; i++) {  
    a[i] = (function (j) {  
      return function() { console.log(j); };  
    })(i);  
  }  
  return a;  
}  
  
var b = f();  
  
b[0](); // → 0  
b[1](); // → 1  
b[2](); // → 2
```

- Plus compact en créant une fonction anonyme, auto-appelée
 - pratique courante dans les bibliothèques JS
 - à éviter

UTILISATION DES CLOSURES

FAIRE DES MODULES SANS CLOSURE

```
var myObject = {  
  nb_get: 0,  
  private_value : 13,  
  public_value : "toto",  
  get : function(){  
    nb_get++;  
    return private_value;  
  }  
  set: function(x) {  
    nb_get = 0;  
    this.private_value = x;  
  }  
}  
var a = myObject.get();  
var x = myObject.private_value;  
myObject.set(1);  
myObject.private_value = 22;
```

- possibilité d'accéder à 'private_value' sans garder 'nb_get' cohérent

UTILISATION DES CLOSURES

FAIRE DES MODULES

```
var exposed = (function () {  
  var private_value = 13;  
  var nb_get = 0;  
  var iface = {};  
  iface.public_value = "toto";  
  iface.set = function (x) {  
    nb_get = 0;  
    private_value = x;  
  };  
  iface.get = function () {  
    nb_get++;  
    return private_value;  
  };  
  return iface;  
})();  
exposed.public_value; // → "toto"  
exposed.get();       // → 13
```

- closure autour de 'private_value' pour cacher la mécanique interne (encapsulation)
- objet retourné comme interface (fonctions et attributs)

JAVASCRIPT

PROGRAMMATION ORIENTÉ-OBJET

- JavaScript est orienté-objet
 - possède la notion d'objet
 - mais pas la notion de "classe" (sauf à partir de ES 6)
 - remplacée par la notion très particulière de **prototype**
 - qui permet de reproduire la notion de "classe"
 - et plus
 - possède la notion d'héritage
 - mais pas de polymorphisme

LES OBJETS

CONSTRUCTEUR ET INITIALISATION

```
var a = {};  
a.x = 2;  
a.f = function () { ... };
```

```
var b = {  
  z : "test",  
  g: function () { ...}  
}
```

```
function h(par1, par2) {  
  this.par1 = par1;  
  this.par2 = par2;  
  this.m = function () { ... };  
}  
// Toute fonction peut devenir un constructeur avec 'new'  
var c = new h(1, "toto");
```

```
var d = new Object();
```

```
var e = Object.create();
```

PROTOTYPE ET HÉRITAGE

- On peut indiquer qu'un objet hérite les propriétés d'un autre objet
- en modifiant la propriété **prototype** du constructeur

```
function A(x) { this.x = x; } // un constructeur
var a = new A(0);           // a vaut { x: 0 }
a.constructor;              // A

A.prototype = { y: 2 };     // tous les objets créés par A
                             // auront maintenant une propriété y
var b = new A(1);           // b vaut { x: 1, y: 2 }
var c = new A(3);           // c vaut { x: 3, y: 2 }
a;                           // a vaut toujours { x: 0 }
                             // car il n'a pas le même prototype

A.prototype.t = "toto";     // un prototype est un objet comme un a
b;                           // b vaut { x: 1, y: 2, t: "toto" }
c;                           // c vaut { x: 3, y: 2, t: "toto" }
a;                           // a vaut toujours { x: 0 }
```

- On peut aussi utiliser `Object.create`

```
var d = Object.create({ y: 2 });
```

PROTOTYPE ET CHAÎNE D'HÉRITAGE

■ Quand on accède à une propriété d'un objet, cette propriété (potentiellement une fonction) est cherchée en remontant la **chaîne de prototype** (héritage):

- soit elle est sur l'objet: c'est sa **propre** propriété (`obj.hasOwnProperty()`)
- si son prototype est non-null, on cherche récursivement sur l'objet prototype
- sinon la propriété n'existe pas

```
function A() {  
  this.print = function () { console.log("hello"); };  
  this.imprime = function () { console.log("bonjour"); }  
}  
var a = new A();  
function B() {  
  this.print = function () { console.log("hello world!"); };  
}  
B.prototype = new A();  
var b = new B();  
b.print();           // hello world!  
b.imprime();         // bonjour  
b instanceof A;      // true  
a instanceof B;      // false
```

■ Quand on appelle une fonction héritée, `this` représente l'objet (pas le prototype)

OBJET

PROPRIÉTÉ D'INSTANCE ET PROPRIÉTÉ STATIQUE

Propriété de toutes les instances

```
function F(x) { this.x = x; }  
var f = new F(10);  
f.x;  
var g = new F(12);  
g.x;
```

Propriété d'une instance

```
function F(x) { this.x = x; }  
var f = new F(10);  
f.z = 2;  
var g = new F(12);  
g.z; // undefined
```

Propriété statique

```
function F(x) { this.x = x; }  
F.w = 3;  
var f = new F(10);  
f.w; // undefined;  
F.w; // 3
```

AJOUTER DES MÉTHODES À UNE CLASSE

- JavaScript est dynamique
- On peut ajouter des méthodes à un prototype, y compris d'une classe "système" (attention!)

```
Array.prototype.remove = function (obj) {  
    var i = this.indexOf(obj);  
    if (i >= 0) {  
        this.splice(i, 1);  
    }  
};  
String.prototype.trim = String.prototype.trim || function() {  
    if (!this) return this;  
    return this.replace(/^\\s+|\\s+$/g, "");  
};
```

ATTENTION: FOR/IN

- Les boucles for (a in b) { ... } incluent les propriétés héritées
- Sauf si on utilise hasOwnProperty()

```
for(a in b){  
    if (b.hasOwnProperty(a)) {  
        ...  
    }  
}
```

JAVASCRIPT STRICT

- Possibilité d'utiliser une version plus stricte de JavaScript

```
"use strict";
```

```
function f() {  
    "use strict";  
}
```

- var n'est pas optionnel
- fonctions appelées sans this: this est undefined, au lieu d'être l'objet global
- des erreurs "silencieuses" font un throw
- eval() ne crée rien dans global (pas de variables, pas de fonctions)
- pas de with

"EVAL IS EVIL"

- Possibilité d'évaluer une chaîne de caractères comme étant du JavaScript

```
eval("3+2"); // → 5
```

- eval est une fonction qui s'exécute là où elle est appelée, dans le contexte local
- à éviter: empêche les optimisations

NETTOYER SON CODE

- JSHint
- JSLint

JS DANS LE SERVEUR

- `node.js`: environnement d'exécution de programmes JavaScript (moteur V8) en ligne de commande
 - permet d'utiliser JavaScript en dehors du navigateur
 - usage similaire à plein d'autres langages (java.exe, perl, python, ...)
- démon http inclus, par défaut
 - permet de déployer un serveur web facilement
 - de développer la logique serveur en JavaScript
 - équivalent de J2EE, Apache Tomcat et les servlets en Java
- Dispose d'un système de modules
 - possibilité d'importer une bibliothèque développée par quelqu'un d'autre
 - il existe un gestionnaire de modules/package: npm

JSON

- Utilisation de la syntaxe littérale JS pour transmettre des données
- Lecture/écriture:

```
obj = JSON.parse(line);  
line = JSON.stringify(obj);
```

- Exemple:

```
JSON.stringify(book) // →  
'{"topic":"JavaScript","fat":true,"author":"Jean Dupont","content":...}'
```

- disponibilité
 - extension courante de ES3
 - natif dans ES5

NOUVEAUTÉS ES 6