Developpez.com
Club des développeurs

Trois fondamentaux de JavaScript

par Jean-Pierre Vincent (BrainCracking) (Blog)

Date de publication : 22 novembre 2011

Dernière mise à jour :

Après quelques années à écrire dans un langage, on finit facilement par oublier les premières difficultés que l'on avait rencontrées. Et à force de faire de la veille, de l'autoformation et de parler entre experts dans des conférences, j'ai un peu quitté la réalité de la majorité des équipes Web.

Maintenant que je suis consultant indépendant je retourne dans des équipes qui avaient autre chose à faire que de se demander si on a le droit de parler de classe en JavaScript, quelle est la bonne définition d'une *closure*, ou quelles sont les fonctionnalités de *EcmaScript 5* qui auraient dû rester dans *Ecmascript.Next*.

J'avais déjà parlé de **JavaScript et la programmation orientée objet pour les développeurs PHP**, nous allons explorer ici les **trois notions fondamentales** de JavaScript qui sont probablement les plus grosses sources de bogues, d'incompréhension et de frustration pour le développeur Web moyen. Et qui accessoirement sont la base d'une programmation plus évoluée par la suite.

L'article original peut être lu sur le site **BrainCracking** : **JavaScript : 3 fondamentaux**.

JavaScript est différent : apprenez-le....3
La portée des variables....3
  La théorie....3
  La pratique....4
  Créer son espace de travail sécurisé....5
Les fonctions....5
  C'était si simple ?....5
  La fausse bonne idée....5
  Particularités JavaScript....6
    Auto-exécution....7
    Classe ou fonction ?....7
    Orienté objet ?....7
    Slip ou caleçon ?....8
Le contexte ou this....9
  Changement de contexte (natif)....9
  Conservation du contexte (à la main)....10
L'essentiel....10
Références et remerciements....11

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

JavaScript est différent : apprenez-le

Le monde du développement Web semble dominé par les langages dérivés de la syntaxe du C, PHP en tête, avec des paradigmes qui se ressemblent. Forcément en abordant JavaScript dont la syntaxe n'est pas vraiment révolutionnaire, on est tenté d'appliquer la même logique. Mais c'est oublier un peu vite que ce langage a été créé il y a déjà 15 ans, quand *Java* était seulement à la mode et pas encore ultra dominant comme aujourd'hui, et qu'il est principalement l'héritier de langages comme *Erlang* et *Lisp*, aujourd'hui très peu connus. En fait le mot *Java* dans JavaScript a été rajouté pour des raisons commerciales, et seuls quelques concepts comme la syntaxe et je crois la gestion des dates ont contribué à former JavaScript. JavaScript n'est donc qu'un cousin éloigné des langages *mainstream*.

Le maître mot de ses concepteurs semble avoir été la versatilité. Nous allons voir qu'en explorant seulement trois bases *a priori* simples, il est possible d'obtenir à peu près n'importe quoi, ce que les grands maîtres de JavaScript s'amusent tous les jours à faire. En attendant de passer dans la catégorie des maîtres, il faut déjà maîtriser ces bases pour faciliter son travail au quotidien.

Nous allons donc nous baser sur *EcmaScript 3* (le JavaScript de IE6-7-8) dont les trois fondamentaux sont :

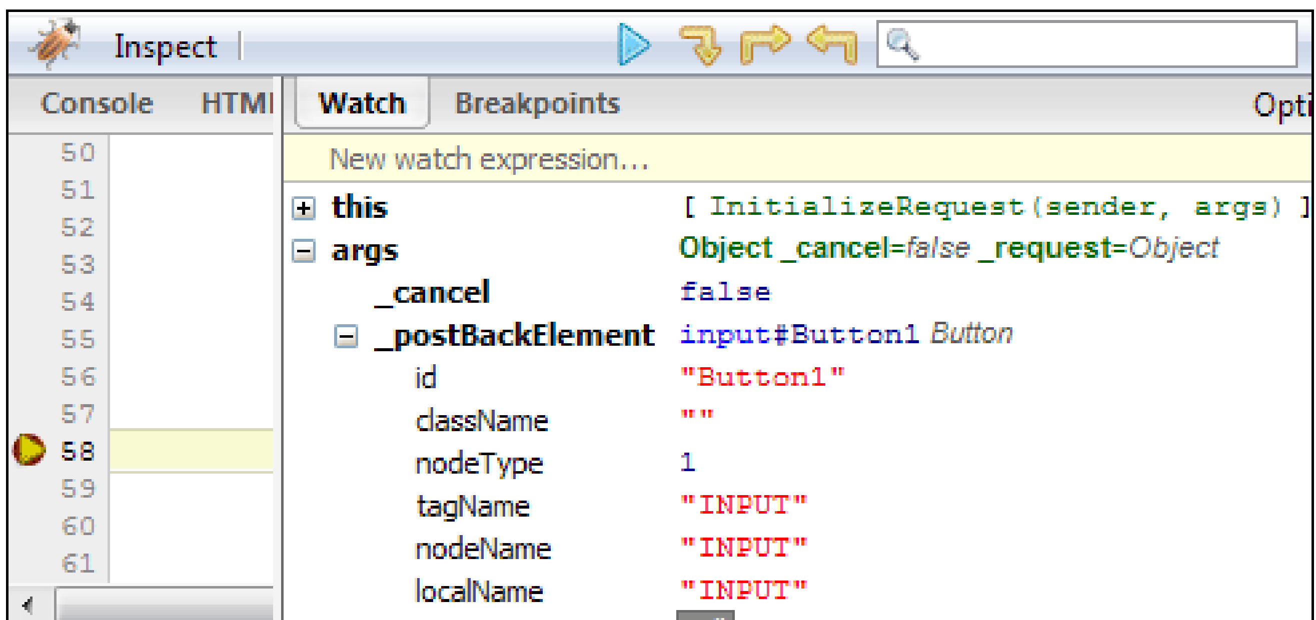
- la portée des variables (var + function) ;
- les fonctions ;
- le contexte (this).

Commençons par la portée.

La portée des variables

La théorie

La portée, c'est ce que voit l'interpréteur à un moment donné de l'exécution du code. Ouvrez un fichier JavaScript, tapez le mot-clé debugger; où bon vous semble, démarrez *Firebug* (ou votre outil préféré) et cherchez l'onglet « Espions » ou « Watch ».



L'onglet Watch ou Espions de Firebug

La règle pour comprendre comment est créée une portée est très simple :

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

1 fonction = 1 portée

Concrètement vous avez la portée globale où tout se trouve (dans un navigateur, c'est window), puis à chaque fonction que vous créez, vous créez une nouvelle portée, et celles-ci peuvent s'imbriquer à l'infini. Chaque fois que vous précédez une assignation par le mot-clé var, la variable n'est visible qu'à partir de cette portée.

```
test1 = function() {
    var x = 1;
    test2 = function() {
        var x = 2;
        test3 = function() {
            var x = 3;
            console.log(x); // 3
        }();
    }();
    console.log(x); // 1
}();
console.log(x); // undefined
```

Si vous exécutez le code précédent dans votre console JavaScript, vous constaterez trois choses :

- le console.log le plus profond affiche bien sûr la dernière valeur de x, à savoir 3 : dans ce troisième niveau, on a écrasé les valeurs précédentes de x ;
- malgré cet écrasement à un niveau inférieur, le deuxième console.log (qui affiche 1) montre bien que les nouvelles valeurs de x ne sont pas visibles depuis la première fonction ;
- enfin le dernier console.log est fait au niveau global, en dehors des définitions de fonction : ici x n'existe même pas.

La pratique

Nous avons vu comment le couple var + function permet de définir une portée et empêche le code en dehors de la portée de voir certaines variables. Pourquoi ne pas tout mettre au niveau global ? Pour des raisons d'organisation du code et de maintenabilité ! Imaginez le code suivant :

```
function genericFunctionName() {
    for(i = 0; i < myArray.length; i++) {
        console.log(i);
    }
}
for(i = 0; i < 10; i++) {
    genericFunctionName();
}
```

Nous avons simplement une fonction qui parcourt un tableau fictif, puis nous appelons cette fonction 10 fois. N'exécutez surtout pas cela dans votre navigateur, car **c'est une boucle infinie** !

En fait les deux boucles for utilisent le même nom de variable (i) pour compter les tours. C'est un nom extrêmement commun et ça ne poserait pas de problème si les deux boucles ne pointaient pas sur **la même variable** ! Fixons cela.

```
function genericFunctionName() {
    for( var i = 0; i < myArray.length; i++) {
        console.log(i);
    }
}
for(i = 0; i < 10; i++) {
    genericFunctionName();
}
```


Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

Dans la fonction, nous avons rajouté le mot-clévar pour spécifier que la variable n'appartenait qu'à cette fonction. Les deux bouclent ne pointent plus sur la même variable même si leurs noms sont les mêmes. C'est très important pour le développeur de savoir qu'il est le seul maître sur les variables qu'il crée pour créer un code particulier. Ceci nous donne la première application concrète des portées.

Créer son espace de travail sécurisé

En tant que développeur, vous allez créer ou modifier du code sur des pages dont vous ne maîtrisez pas l'environnement. Entre l'historique du code, vos collègues et les publicités ou les widgets, beaucoup de variables sont créées ou écrasées au niveau global sans que vous puissiez le prévoir. Au milieu de cette tourmente, je vous propose de vous créer un petit havre de paix :

```
(function() {
}())
```

Je vous conseille de démarrer tout fichier JavaScript ou tout code entre balises <script> par la première ligne et de toujours terminer par la dernière. Ce bout de code vous permet de poser les bases de votre portée et de créer des variables qui n'appartiennent qu'à vous, sans pour autant s'isoler du reste du monde.

```
(function() {
    var privateVariable = true;
    window.init = function() {
        console.log( privateVariable );
    }
}())
init(); // true
console.log(privateVariable); // undefined
```

Exécutez ce code, et vous verrez que privateVariable n'est pas accessible de l'extérieur, mais que le code défini à l'intérieur permet d'y accéder. Très pratique en environnement hostile. Vous remarquerez aussi que l'on rattache la fonction init directement à la portée globale, afin qu'elle soit elle aussi visible de l'extérieur.

Les fonctions

C'était si simple ?

Il est facile de créer une fonction et de l'exécuter. Les syntaxes que vous utilisez au quotidien sont celles-ci :

```
// création v1
function action() {}
// création v2
action = function() {};
// exécution
action();
```

Simple non ?

La fausse bonne idée

Au fait quelle est la différence entre les deux manières de déclarer une fonction ? Pour la première (function nom()) le compilateur JavaScript extrait toutes les définitions et les met à disposition au niveau global AVANT de commencer l'exécution. Concrètement vous pouvez même exécuter la fonction avant qu'elle ne soit créée dans la source.

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

```
// exécution avant la définition !
action();
// création
function action() {}
```

Cool ? Pas vraiment : c'est systématiquement la dernière déclaration dans la source qui est prise en compte par les interpréteurs JS. Vous ne pouvez pas conditionner la définition des fonctions.

Par exemple le code suivant ne va pas s'exécuter comme vous pourriez le penser :

```
if( 1 == 1 ) {
    // l'exécution du code passe ici ...
    function action() {
        console.log('a == 1');
    }
    } else {
        // ... pourtant le compilateur se souvient de cette fonction
        function action() {
        console.log('a != 1');
    }
}
action(); // a != 1
```

Lors de l'exécution, l'interpréteur passe bien dans le premier if et pourtant c'est la seconde définition qui est retenue, uniquement parce qu'elle arrive en dernier dans la source.

Et malgré cela, contrairement à d'autres langages, la déclaration d'une fonction n'est valable que pour la portée en cours et ses descendantes. Dans le code suivant, la fonction action n'est pas visible au niveau global :

```
(function() {
 function action() {
  console.log('action');
 }
}())
action(); // undefined
```

La manière traditionnelle de déclarer les fonctions est donc assez trompeuse et de ce fait tombe doucement en désuétude. Pour des débutants on utilisera plutôt la seconde méthode qui a le mérite de pouvoir **explicitement** choisir la portée de ses fonctions, au même titre qu'une variable

```
var action = function() {};
```

En ce qui concerne le compilateur, cette syntaxe est une variable avec pointeur sur une fonction anonyme. Le désavantage est que lorsque vous utilisez un débogueur pas à pas, la pile d'appels (*call stack*) ne vous affiche pas le nom des fonctions (l'anonymat sur Internet existe donc). Vous pouvez retrouver le nom de la variable assignée simplement en cliquant sur la fonction et en regardant dans la source.

Particularités JavaScript

JavaScript a rajouté plusieurs particularités aux fonctions qui les rendent terriblement puissantes et flexibles. En fait toutes les constructions de code un peu élaborées telles que l'orientation objet ou l'application de *design pattern* se basent sur ces spécificités. Nous entrons donc dans le cœur de JavaScript.

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

Auto-exécution

En rajoutant simplement une paire de parenthèses après une déclaration, la fonction va immédiatement être exécutée.

```
var autoInit = function() {
    console.log('hello world');
}();
// hello world
```

Comme nous l'avons vu tout à l'heure, cela vous permettra principalement de créer des portées, pour protéger l'ensemble de votre script, et nous allons voir dans une minute que vous pouvez aussi l'utiliser dans certains cas particuliers.

Classe ou fonction ?

En JavaScript, tout est objet. Un objet n'est jamais qu'une collection de clés et de valeurs, et les valeurs peuvent être tout et n'importe quoi y compris d'autres objets. Lorsqu'une fonction est créée, automatiquement JavaScript y rajoute la propriété prototype. Tapez le code suivant dans votre console JavaScript :

```
var myFunction = function() {};
console.log( myFunction.prototype ); // Object
```

Nous avons donc un objet myFunction.prototype et nous pouvons directement y accéder pour y rajouter des valeurs, comme d'autres fonctions par exemple. Ensuite pour accéder à ces fonctions, nous allons utiliser le mot-clénew.

```
// cette innocente fonction devient un constructeur
var myClass = function () {
    this.publicVariable = 0;
};
// accès au prototype
myClass.prototype = {
    decrement:function() {
        console.log( --this.publicVariable );
    },
    increment:function() {
        console.log( ++this.publicVariable );
    }
};

myObject = new myClass();
myObject.decrement(); // -1
myObject.decrement(); // -2

myObject2 = new myClass();
myObject2.increment(); // 1
myObject2.increment(); // 2
```

Entre nous, nous venons de créer quelque chose qui ressemble furieusement à une classe, avec une instanciation (new myClass), des variables propres à chaque instance (this.publicVariable) et un constructeur (le corps de la fonction).

Orienté objet ?

Si vous trouvez bizarre qu'une fonction se transforme soudainement en classe puis en objet, attendez de voir la suite. Toujours grâce à la notion du tout objet de JavaScript, nous pouvons également faire de l'héritage ! Là encore, en

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

utilisant .prototype, nous allons créer une nouvelle fonction (ou classe) qui va hériter des méthodes de la première. Outre la déclaration de la nouvelle fonction, cela ne va prendre que deux lignes.

```
mySubClass = function() {
    // maintenant on part de 10 au lieu de 0
    this.publicVariable = 10;
};
mySubClass.prototype = myClass.prototype;
mySubClass.prototype.constructor = mySubClass;

myObject2 = new mySubClass();
myObject2.increment(); // 11
myObject2.increment(); // 12
```

La première ligne après la déclaration de fonction fait pointer l'objet .prototype de notre nouvelle fonction vers l'objet .prototype de la classe mère, empruntant ainsi toutes ses propriétés. La seconde ligne est là pour rectifier un défaut de l'écrasement de .prototype : on refait pointer .prototype.constructor (encore une propriété automatiquement créée pour toutes les fonctions) vers le bon constructeur.

Slip ou caleçon ?

Pardon je voulais dire : pour les classes, vous êtes plutôt prototype ou closure ? On vient de voir que grâce à .prototype on pouvait émuler une classe en JavaScript. Les fonctions peuvent également renvoyer des objets simples contenant des propriétés et des méthodes, et la manière de les utiliser se rapprochera toujours terriblement d'une classe.

```
myClass = function () {
    // variable privée !
    var privateVariable = 0;
    // méthodes publiques
    return {
        decrement:function() {
            console.log( --privateVariable );
        },
        increment:function() {
            console.log( ++privateVariable );
        }
    }
};
myObject = myClass();
myObject.decrement(); // -1
myObject.decrement(); // -2
myObject2 = myClass();
myObject2.increment(); // 1
myObject2.increment(); // 2
```

Plusieurs différences par rapport à la version prototype :

- notez le return {...} dans la définition de la fonction : vous pouvez y mettre exactement ce que vous auriez mis dans le prototype ;
- vous pouvez utiliser la portée de la fonction constructeur pour avoir des variables privées (ici var privateVariable), alors qu'il n'est **pas possible d'avoir des variables privées avec prototype** ;
- lorsque vous instanciez votre classe, vous n'avez plus besoin d'utiliser new myClass(), un simple appel sans new suffit ;
- le mot-cléthis désigne l'objet renvoyé, alors qu'avec prototype il désignait le corps de la fonction constructeur, généralement avec closure on n'utilise plus du tout this ;
- la création des fonctions est évaluée à chaque fois que vous invoquez la fonction constructeur, tandis que ce n'est fait qu'une seule fois avec prototype, **n'utilisez jamais closure avec des objets qui peuvent être instanciés des centaines de fois**.

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

Voilà pour l'essentiel sur les fonctions : avec cela vous maîtrisez déjà leur portée et vous pouvez commencer à développer orienté objet, ce qui va vous permettre d'éviter beaucoup de bogues et d'avoir un code plus maintenable.

Le contexte ou this

this est le petit mot-clé qui perd beaucoup de développeurs, notamment lorsque l'on est habitué aux dérivés du C comme PHP. En JavaScript comme ailleurs, il fait référence au contexte d'exécution, donc le code suivant s'exécutera probablement comme vous vous y attendez :

```
myClass = function() {
    this.id = 'myClass';
}
myClass.prototype = {
    action:function() {
        console.log( this.id );
    }
};
myObject = new myClass();
myObject.action(); // 'myclass'
```

Ici on a défini dans le constructeur this.id avec une certaine valeur, puis on instancie la classe (new myClass()) et la méthode action() accède tout naturellement à this pour retrouver la valeur de id.

Pourtant lorsque vous voulez exécuter myObject.action au clic sur un élément du DOM, plus rien ne marche comme prévu.

```
document.body.onclick = myObject.action;
// document.body.id
```

Vous pouvez exécuter les deux bouts de code précédents dans votre console JavaScript et cliquer sur la page, vous verrez que this.id n'est plus ‘myClass' mais fait référence à l'id du body, qui n'existe peut-être même pas. Ceci n'est pas particulier au DOM d'ailleurs, imaginez que vous créiez un autre objet qui aurait un événement onfire que vous auriez besoin d'écouter :

```
myEvent = {
    id:'myEvent'
};
myEvent.onfire = myObject.action;
myEvent.onfire(); // 'myEvent' au lieu de 'myClass'
```

Lorsque onfire() s'exécute, il exécute bien le corps de la fonction action, mais comme pour les objets du DOM, this**fait référence à l'objet d'où est exécutée la fonction**, pas forcément à l'objet d'origine de la fonction.

Ceci est dû à la versatilité de JavaScript qui permet de très simplement copier le corps d'une fonction d'un objet à l'autre, alors que dans les langages traditionnels orientés objet il vous faut explicitement faire de l'héritage. Comment fixer cela ? Il y a au moins deux méthodes.

Changement de contexte (natif)

La première méthode est native à JavaScript et utilise la fonction .call(). Ici seule la dernière ligne a changé :

```
myClass = function() {
    this.id = 'myClass';
}
myClass.prototype = {
```


Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

```
    action:function() {
        console.log(this.id);
    }
};
myObject = new myClass();
myEvent = {
    id:'myEvent'
};
myEvent.onfire = myObject.action;
myEvent.onfire.call( myObject ); // myClass
```

Le premier argument de la fonction .call() est l'objet qui sera utilisé pour this. Ici on redonne à onfire l'objet d'origine de la fonction action.

Le problème avec cette solution, c'est qu'il faudrait que vous soyez certain que TOUS les endroits où votre fonction peut être appelée vont rectifier le contexte d'exécution. Or ne serait-ce que pour les événements natifs du DOM cela n'est pas possible, sauf si vous utilisez une bibliothèque de manière systématique. Donc faire confiance à this n'est raisonnable que si vous maîtrisez parfaitement tout le code qui sera écrit.

Conservation du contexte (à la main)

Pour éviter ces problèmes de maintenance, la solution en vogue est de tout simplement ne jamais utiliser this directement et de préférer utiliser la syntaxe closure et une variable privée pour définir ses classes.

```
myClass = function() {
    this.id = 'myClass';
    // capture du contexte
    var me = this;
    return {
        action:function() {
            console.log( me.id );
        }
    }
};
myObject = myClass();
document.body.onclick = myObject.action;
// 'myClass'
```

On voit donc ici :

- dans le constructeur, on capture le contexte avec une variable privée : var me = this ;
- dans la fonction action on n'utilise plus this mais me.

Peu importe l'endroit d'où est lancée la fonction (ici au clic sur la page), et peu importe la valeur réelle de this, on est maintenant certain que l'on aura toujours un accès à l'objet originel grâce à une variable privée. Bien sûr this reste utilisable normalement, notamment si vous avez besoin de retrouver l'objet DOM cliqué par exemple.

Pour encore plus de souplesse avec les fonctions de JavaScript, **allez voir la deuxième partie de cet article**.

L'essentiel

Voici les quelques points importants à retenir.

- JavaScript a des concepts différents des langages majeurs et devient extrêmement important sur votre CV. Prenez le temps de l'apprendre.
- Les bibliothèques telles que jQuery ne sont pas faites pour couvrir les cas que nous venons de voir. jQuery permet d'utiliser le DOM sereinement, pas d'organiser votre code proprement.

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.

- Lorsque vous créez une classe, choisissez la syntaxe prototype pour la performance ou closure pour les variables privées et la maîtrise du contexte d'exécution.
- N'utilisez plus la syntaxe function maFonction() {}.
- Utilisez systématiquement la fonction anonyme auto-exécutée ((function() { var .. }())) en début et fin de fichier ou de <script>.
- Utilisez systématiquement var pour vos variables, y compris celles qui font référence à des fonctions.

Dernière chose : j'ai essayé d'être pratique, mais lire un article ne suffit jamais pour comprendre des concepts de programmation : codez !

Références et remerciements

L'article original peut être lu sur le site **BrainCracking** : **JavaScript : 3 fondamentaux**.

Je tiens à remercier **ClaudeLELOUP** pour sa relecture attentive de cet article.

Copyright ® 2011 Jean-Pierre Vincent. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.