



Concepteur Développeur en Informatique

Développer des composants d'interface

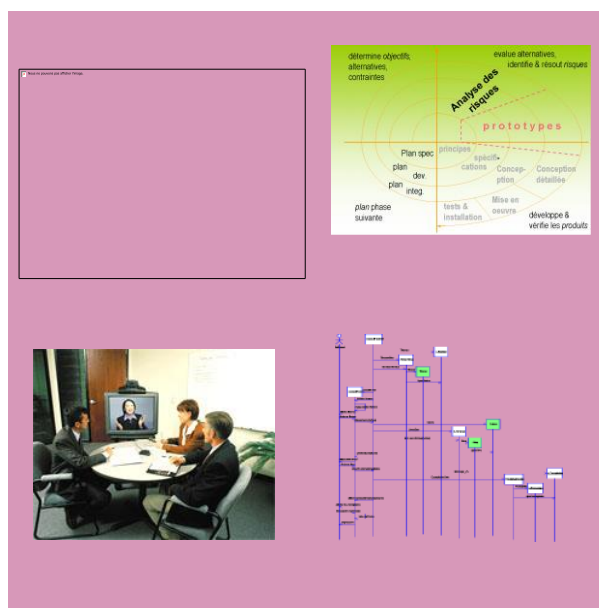
Javascript Bonnes pratiques

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E05-S02

1.	Introduction	3
2.	Règles de codage	3
2.1.	Eviter les variables de portée globale	3
2.2.	Les variables doivent être déclarées.....	3
2.3.	Terminer les instructions	5
2.4.	Utiliser le mode de codage strict	5
2.5.	Utiliser l'égalité stricte	6
2.6.	Spécifier la base lors de l'utilisation de parseInt	7
2.7.	Comparaisons avec rvalue à gauche	7
2.8.	Mise en cache des résultats d'une méthode	8
2.9.	Spécifier la base lors de l'utilisation de parseInt	10
2.10.	Externaliser les scripts	10
2.11.	Consulter régulièrement la documentation... ..	11

1. Introduction

Javascript est un langage qui peut poser de nombreux problèmes de mise au point de par sa nature très permissive. Il vous faudra donc opter pour des règles strictes de codage afin de minimiser les phases de mise au point qui peuvent d'avérer très fastidieuses.

Nous allons voir ici quelques bonnes pratiques qu'il convient d'intégrer afin de faciliter la mise au point de nos scripts.

Pour compléter ce document d'introduction, vous pouvez vous rendre sur le site de référence en matière de développement web, MDN (Mozilla Developer NetWork).

2. Règles de codage

2.1. *Eviter les variables de portée globale*

Cette première règle n'est pas spécifique à JavaScript. Elle est dans ce contexte encore plus prégnante : le recours aux variables globales doit être évité.

Une variable globale, qu'elle que soit le type de données ou de fonction, a une portée globale à l'application. Elle peut donc être réaffectée par toute portion de code de l'application.

L'usage de variables globales induit :

- un couplage fort entre les différentes parties de l'application limitant ainsi la factorisation du code.
- Peut provoquer des effets de bord du fait de valeurs inattendues prises par ces variables au cours de l'exécution du programme.

2.2. *Les variables doivent être déclarées*

Vous aurez ainsi une meilleure maîtrise des portées de vos variables et éviterez des effets de bord désastreux.

Les variables locales à une fonction seront déclarées avec les mots clé **var** ou **let**, var pour une portée de niveau fonction, **let** de portée du bloc courant.

Recourez au mot clé **const** lorsqu'une seule assignation est autorisée.

L'assignation se fait alors lors de la déclaration et la valeur sera donc constante.

Contrairement au mot clé var qui autorise le hissing (hoisting), à savoir l'utilisation d'une variable avant sa déclaration, la déclaration des variables avec let et const doit se faire avant leur usage.

Dans tous les cas, les variables doivent être initialisés avant d'être sollicités au sein d'opérations de calcul ou de lectures.

Javascript autorise malheureusement comme C# la possibilité de déclarer deux fois une variable de même nom dans la même portée. Voir exemple page suivante.

```
(function (){  
var a = 3;  
if (a < 4){  
    var a = 2;  
    console.log(a);  
}  
console.log(a);  
})();
```

Premier cas avec var : une seule variable a.

Deuxième cas avec let : deux variables de scopes différents.

```
(function (){  
let a = 3;  
if (a < 4){  
    let a = 2;  
    console.log(a);  
}  
console.log(a);  
})();
```

2.3. Terminer les instructions

Spécifier systématiquement le caractère ; pour terminer vos instructions. Javascript peut de lui-même ajouter un ; et terminer une instruction si celle-ci est correcte.

Ainsi, l'exemple suivant, la valeur retournée par la fonction est non définie.

```
> function mafonction(a,b)
{
  return
  a+b
}
console.log(mafonction(1,2));
undefined
```

Le résultat recherché est celui-ci :

```
function mafonction(a,b)
{
  return a+b;
}
console.log(mafonction(1,2));
3
```

L'utilisation systématique du séparateur d'instruction est aussi nécessaire pour les opérations de minification du code.

Cette opération permet de contracter le code JavaScript en modifiant les noms des termes et en supprimant les caractères inutiles.

Le fichier comportant le code « minifié » est d'un poids plus léger mais est très difficile à lire : il s'agit d'une version de production et non de développement.

2.4. Utiliser le mode de codage strict

Utilisez la directive **use strict** pour passer en mode de codage strict.

Ce mode nécessite de déclarer explicitement les variables.

Le mode strict permet aussi de convertir en erreurs des fautes qui resteraient muettes du fait du caractère permissif de javascript. Par exemple deux arguments de fonction qui portent le même nom provoquent la levée d'une exception.

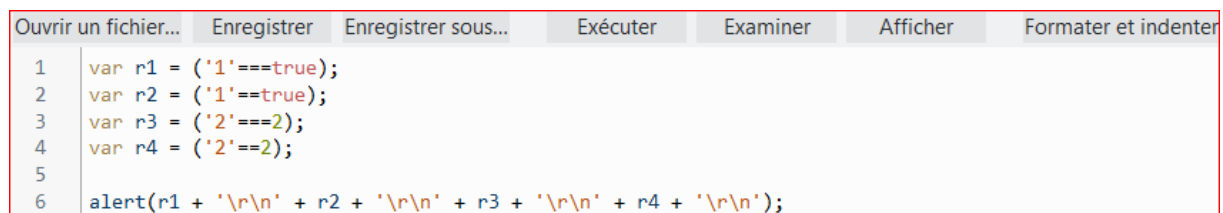
2.5. Utiliser l'égalité stricte

Préférez toujours l'opérateur `===` à l'opérateur `==` pour comparer deux éléments et déterminer s'ils sont égaux.

L'opérateur `===` permet de comparer à la fois la valeur et le type.

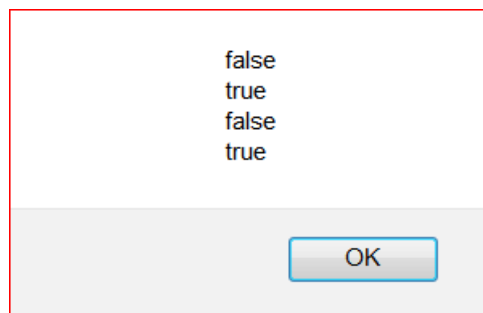
Avec l'opérateur `==`, javascript réalise une coercition ou, selon le terme anglais plus usité coercion. Le principe est que lorsque deux opérandes d'un opérateur sont de types différents, alors une conversion implicite de l'un des opérandes dans l'autre type sera réalisée afin de pouvoir exécuter l'opération.

Avec l'opérateur `===`, javascript ne réalise pas de conversion. Deux opérandes de types différents seront donc toujours considérées comme différentes.



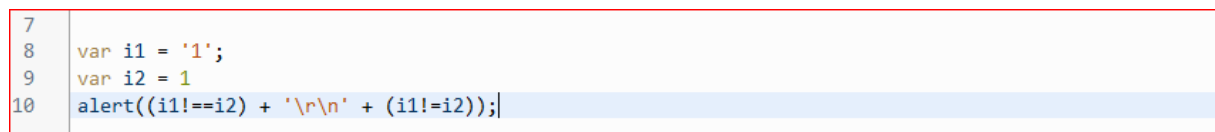
```
Ouvrir un fichier...  Enregistrer  Enregistrer sous...  Exécuter  Examiner  Afficher  Formater et indenter
1  var r1 = ('1'===true);
2  var r2 = ('1'==true);
3  var r3 = ('2'===2);
4  var r4 = ('2'==2);
5
6  alert(r1 + '\r\n' + r2 + '\r\n' + r3 + '\r\n' + r4 + '\r\n');
```

Résultat



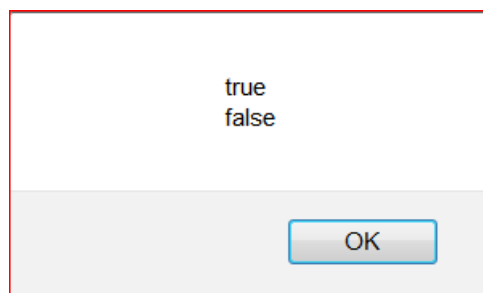
```
false
true
false
true
OK
```

Le pendant de cet opérateur est l'opérateur `!==` en lieu et place de `!=`.



```
7
8  var i1 = '1';
9  var i2 = 1
10 alert((i1!==i2) + '\r\n' + (i1!=i2));
```

Résultat



```
true
false
OK
```

2.6. Spécifier la base lors de l'utilisation de `parseInt`

Même si dans les nouvelles implémentations de JavaScript la base 10 est retenue par défaut, d'autres interprètent les chaînes numériques qui commencent par 0 pour une valeur exprimée dans la base octale.

Il est donc préférable de toujours fournir la base dans laquelle doit être convertie la chaîne.

Si la conversion ne peut aboutir, le résultat obtenu sera NaN (Not a Number) qui peut être testée avec la fonction `isNaN()`.

Vous ne devez pas utiliser la valeur NaN dans une opération arithmétique : son utilisation provoquera toujours comme résultat de l'opération NaN.

Ouvrir un fichier...	Enregistrer	Enregistrer sous...	Exécuter	Examiner	Afficher	Formater et indenter
1	<code>var texte = '0256';</code>					
2	<code>var entier1 = parseInt(texte,10); // 256</code>					
3	<code>var entier2 = parseInt(texte,8); // 170</code>					
4	<code>var entier3 = parseInt(texte,2); // 0</code>					
5	<code>texte = '0xAB1'</code>					
6	<code>var entier4 = parseInt(texte,16); // 2737</code>					
7	<code>var entier5 = parseInt(texte,10); // 0</code>					

2.7. Comparaisons avec *rvalue* à gauche

Une expression en programmation représente une lvalue (locator value et left value de l'opérateur d'assignation) ou une rvalue (right value de l'opérateur d'assignation).

Une lvalue fait référence à un objet qui persiste au-delà d'une expression unique. Une lvalue est définie par un type, un nom et une adresse.

Vous pouvez considérer une lvalue comme un objet qui porte un nom. Toutes les variables, y compris les variables non modifiables (`const`), sont des lvalues.

Une rvalue est une valeur temporaire qui n'est pas persistante au-delà de l'expression qui l'utilise.

Lorsque vous faites une affectation, l'opérande de gauche doit être une lvalue.

Dans l'exemple qui suit, la première expression est donc valide, la seconde non.

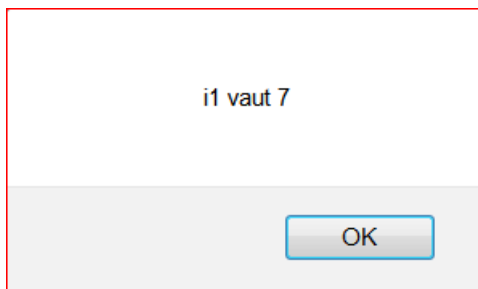
Ouvrir un fichier...	Enregistrer	Enregistrer sous...	Exécuter	Examiner	Afficher	Formater et indenter
1	<code>var i1;</code>					
2	<code>i1 = 7; // OK</code>					
3	<code>7 = i1; // Exception: invalid assignment left-hand side</code>					

Lorsqu'une expression représente une comparaison entre une lvalue et une rvalue, il est préférable de mettre la rvalue à gauche. Pourquoi ? Pour éviter une erreur de codage du fait de la confusion entre opérateur d'affectation `=` et opérateur d'égalité `==`.

Ainsi, lors de l'exécution du code suivant :

Ouvrir un fichier...	Enregistrer	Enregistrer sous...	Exécuter	Examiner	Afficher	Formater et indenter
1	<code>var i1;</code>					
2	<code>i1=6;</code>					
3	<code>if (i1=7)</code>					
4	<code>{</code>					
5	<code> alert("i1 vaut 7")</code>					
6	<code>}</code>					

Nous obtenons la boîte de dialogue suivante ! :



Ce problème aurait pu être évité en plaçant la rvalue à gauche. Nous n'aurions pas pu alors confondre les opérateurs = et ==.

```
Ouvrir un fichier...  Enregistrer  Enregistrer sous...  Exécuter  Examiner  Afficher  Formater et indenter
1  var i1;
2  i1=6;
3  if (7==i1)
4  {
5      alert("i1 vaut 7")
6  }
```

Nous n'avons alors plus de boîte de dialogue affichée car la condition n'est pas remplie. Cette approche ne vaut évidemment que pour les expressions sollicitant une rvalue.

2.8. Mise en cache des résultats d'une méthode

Exemple ici de la technique de Memoization, à ne pas confondre avec la memorization bien que des similitudes existent.

Il s'agit d'une technique d'optimisation qui consiste à stocker en mémoire, mettre en cache, le résultat de l'exécution d'une expression ou d'une fonction lorsqu'elle celle-ci demande des temps de traitement importants.

Vous devez tout d'abord prendre garde à ne pas invoquer à plusieurs reprises des méthodes exigeantes comme celles qui effectuent une recherche dans l'arbre du Document Object Model (DOM).

Programmation de gestionnaires d'événements (implémentation vide...).

Plutôt que de parcourir à chaque fois l'arbre du DOM pour extraire la référence d'un élément :

```
function programmerElement(){
    document.getElementById("#identifiant").onclick(function (){});
    document.getElementById("#identifiant").blur(function () { });
    document.getElementById("#identifiant").style.backgroundColor='#ff0000';
    //...
}
```

Stocker la référence de cet élément dans une variable et faites référence à cette dernière.

```
function programmerElement(){
    var elem = document.getElementById("#identifiant");
    elem.onclick(function (){});
    elem.blur(function () { });
    elem.style.backgroundColor='#ff0000';
}
```

Mise en cache du résultat de l'exécution d'une fonction

Si l'attribut cache de la méthode n'est pas défini, alors la fonction est exécutée et son résultat stocké dans l'attribut cache de la fonction.

Si l'attribut cache existe, alors le résultat stocké est retourné sans devoir exécuter de nouveau les opérations internes à la fonction.

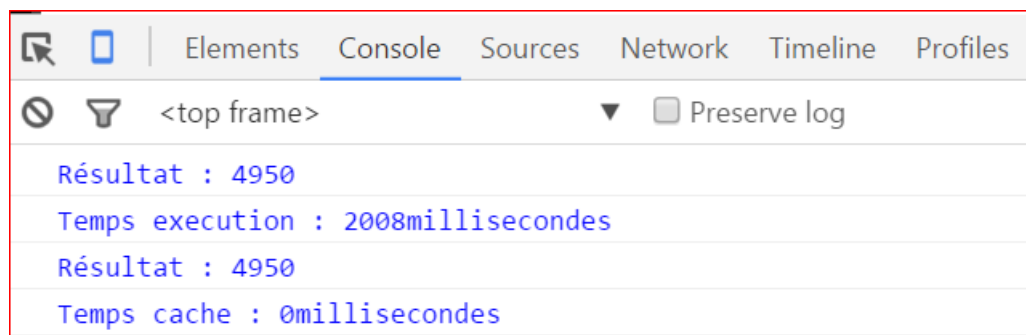
Pour simuler un temps d'exécution long, programmation d'une fonction pause.

```
function executionLourde() {  
    if (undefined !== executionLourde.cache)  
        return executionLourde.cache;  
    var result=0;  
    // Ici figurera le code dont l'exécution nécessite  
    // la consommation de ressources importantes  
  
    for (var i = 0; i < 100; i++) {  
        result += i;  
    }  
    pause(2000)  
    executionLourde.cache = result;  
    return result;  
}  
  
function pause(millisecondes) {  
    var currentTime = new Date().getTime() + millisecondes;  
  
    while (currentTime >= new Date().getTime()) {  
    }  
}
```

Expression des résultats

```
function testCache(){  
    var tempsDebut = new Date().getTime();  
    console.debug("Résultat : " + executionLourde());  
    var tempsFin1 = new Date().getTime();  
    console.debug("Temps execution : " + (tempsFin1 - tempsDebut) + "millisecondes");  
    console.debug("Résultat : " + executionLourde());  
    var tempsFin2 = new Date().getTime();  
    console.debug("Temps cache : " + (tempsFin2 - tempsFin1) + "millisecondes");  
}
```

Il n'y a pas photo !



A noter : Cet exemple peut être décliné pour des fonctions prenant en compte des paramètres. Il vous faudra alors calculer un code de hachage à partir de la valeur des paramètres et tenir compte de la valeur de ce code pour savoir s'il est nécessaire de prendre le résultat en cache ou s'il doit être recalculé.

2.9. Spécifier la base lors de l'utilisation de `parseInt`

Voici quelques exemples d'analyse de chaîne suivis de la conversion dans un type numérique intégral.

Il faut fournir la base dans laquelle convertir la chaîne afin d'éviter des problèmes d'interprétation des navigateurs.

Si la base fournie vaut `undefined` ou `0` (ou si elle n'est pas utilisée comme paramètre), le moteur JavaScript procédera comme suit :

- Si l'argument string commence avec "0x" ou "0X", la base considérée sera la base 16 (hexadécimale) et le reste de la chaîne sera analysé et converti en conséquence.
- Si l'argument string commence avec "0", la base considérée sera la base 8 (octale) ou la base 10 (décimale). La base exacte qui sera choisie dépendra de l'implémentation. ECMAScript 5 et ultérieur définit que la base 10 doit être utilisée. Cependant, cela n'est pas supporté par tous les navigateurs. C'est pour cette raison qu'il faut toujours spécifier une base lorsqu'on utilise `parseInt()`.
- Si l'argument string commence avec une autre valeur, la base considérée sera la base 10.

Le résultat de la conversion est toujours 15.

```
function conversionChaineIntegral() {  
    console.debug("0xF" + " : " + parseInt("0xF", 16));  
    console.debug(" F" + " : " + parseInt(" F", 16));  
    console.debug("17" + " : " + parseInt("17", 8));  
    console.debug("015" + " : " + parseInt("015", 10));  
    console.debug("1111" + " : " + parseInt("1111", 2));  
    console.debug("12" + " : " + parseInt("12", 13));  
}
```

0xF : 15

F : 15

17 : 15

015 : 15

1111 : 15

12 : 15

A noter : il est souvent préférable de fournir une valeur en argument plutôt que de laisser la valeur par défaut.

Vous effectuerez la même chose lors de la transformation d'un intégral en chaîne `integral.toString(base)`.

2.10. Externaliser les scripts

Il convient d'externaliser les scripts et de les charger après le document HTML afin que l'utilisateur puisse voir très rapidement le contenu de la page.

Exception faite des scripts qui ont une incidence sur la mise en forme des éléments qui figurent dans la page et qui, au même titre que les feuilles de styles, doivent être chargés dans la section d'entête de la page.

2.11. Consulter régulièrement la documentation...

Consulter régulièrement la documentation (pour ma part je suis plutôt fan de MDN) afin de faire évoluer son code en fonction de l'évolution.

Ainsi, nous apprenons que le cycle `for ... each` doit être abandonné au profit de `for ... of`

Retrouver la liste actualisée à cette adresse :

https://developer.mozilla.org/fr/docs/JavaScript/Reference/Annexes/Fonctionnalit%C3%A9s_d%C3%A9pr%C3%A9ci%C3%A9es