

# Le langage JavaScript

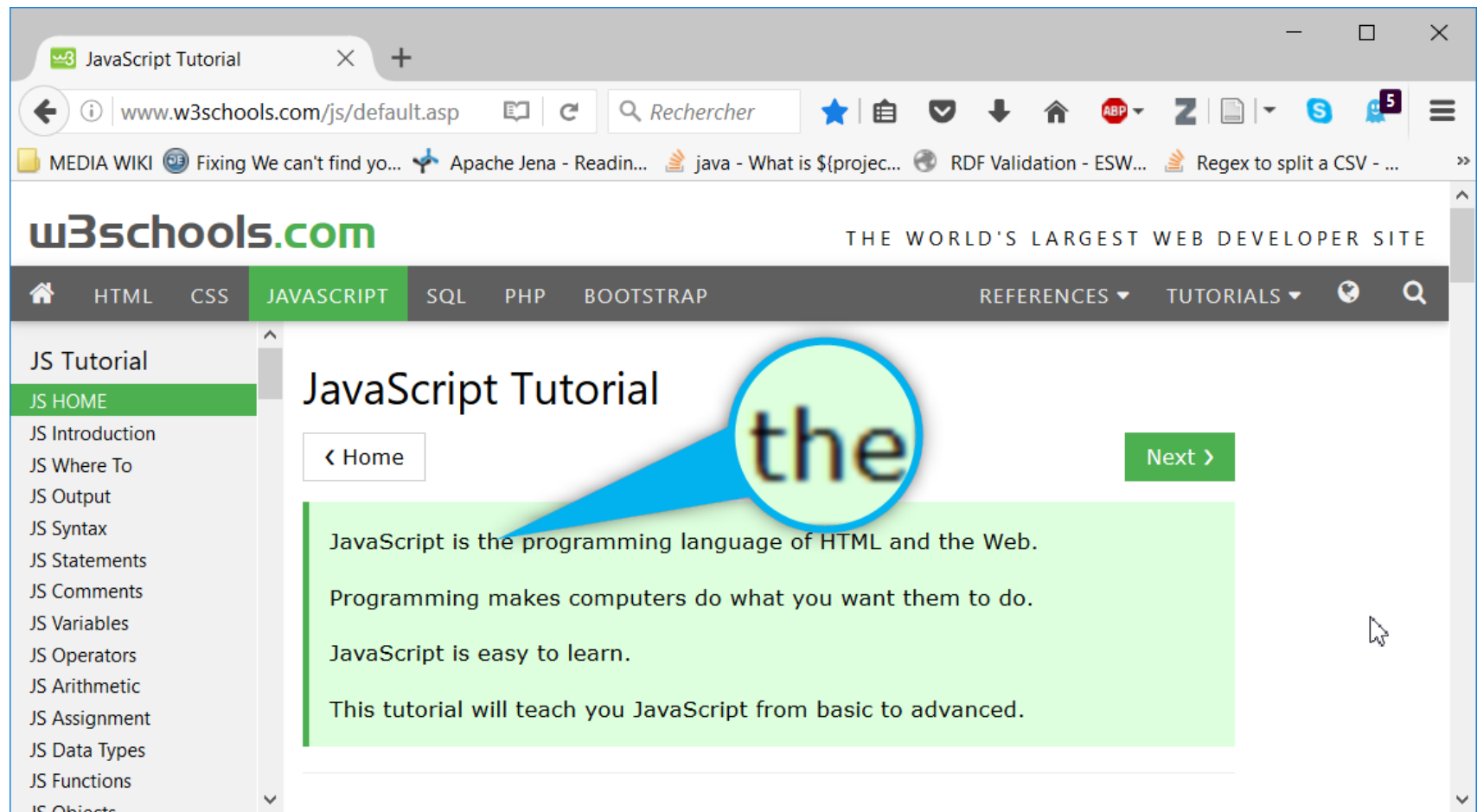


*Attention.... ce cours est  
en chantier et en cours  
de modification*

*dernière mise à jour : 24/10/2016*

- JavaScript un langage incontournable du développement Web

<http://www.w3schools.com/js/default.asp>



# JavaScript

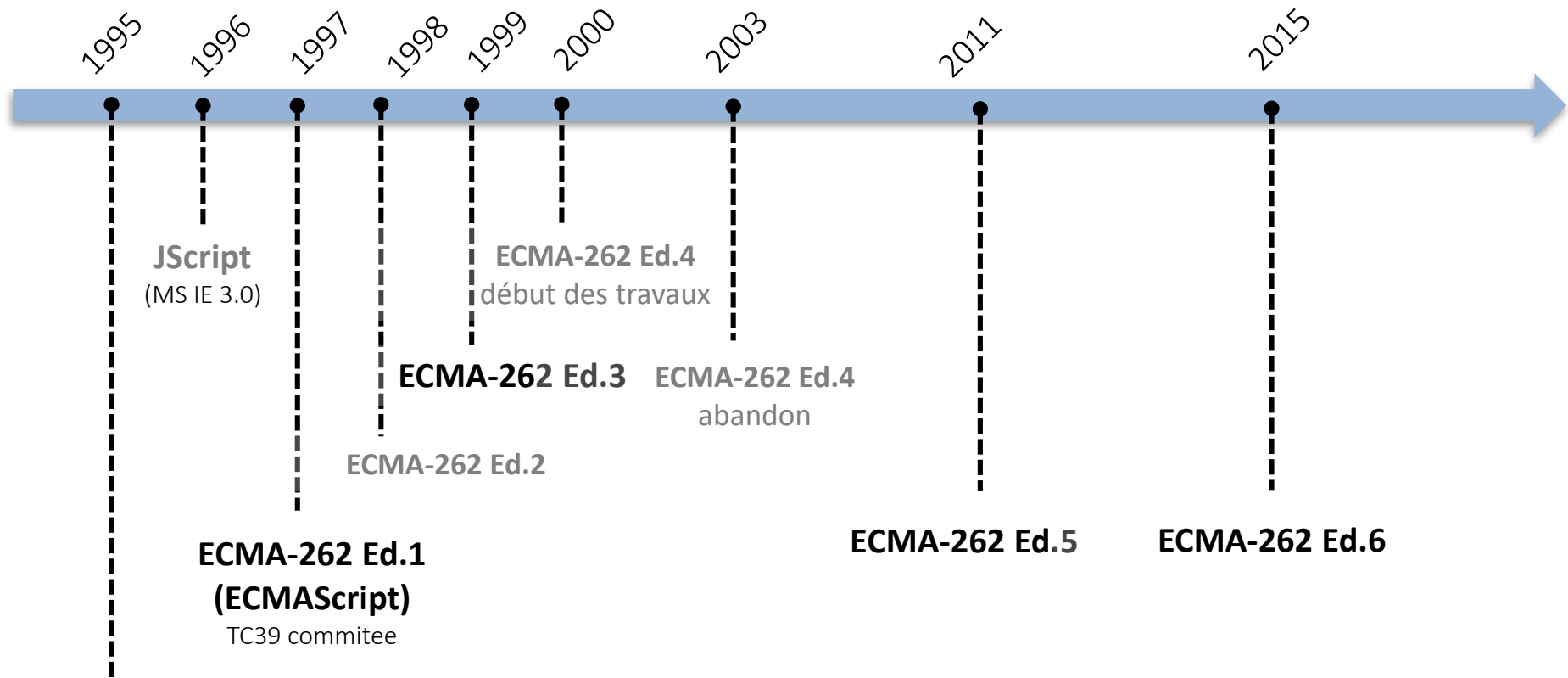
JAVA *is to*  
JAVASCRIPT  
*as* HAM *is to*  
HAMSTER



ILLUSTRATION BY SEGUE TECHNOLOGIES

<http://www.seguetech.com/blog/2013/02/15/java-vs-javascript>

- Créé par Netscape (nom original: LiveScript – sept.1995)
- Syntaxe proche de Java (sur certains points)
- Langage interprété :
  - Pas de compilation vers du ByteCode  
→ débogage à l'exécution !
- Langage orienté objet
  - Pas de notion de classe : prototypes



(Mai) Brendan Eich → Mocha  
(Sep.) Mocha → LiveScript  
(Dec.) LiveScript → **JavaScript**  
(Netscape Navigator 2.0)

- Attention JavaScript n'est pas un langage sans défauts et il peut être parfois délicat à utiliser

*"La plupart des langages contiennent des bons et des mauvais éléments. ... Le JavaScript est un langage particulièrement bien loti en ce qui concerne les mauvais éléments... Mais le JavaScript contient heureusement un certain nombre d'éléments exceptionnellement bons..."*

Douglas Crockford  
*JavaScript, les bons éléments*  
Ed. Pearson France, 2013



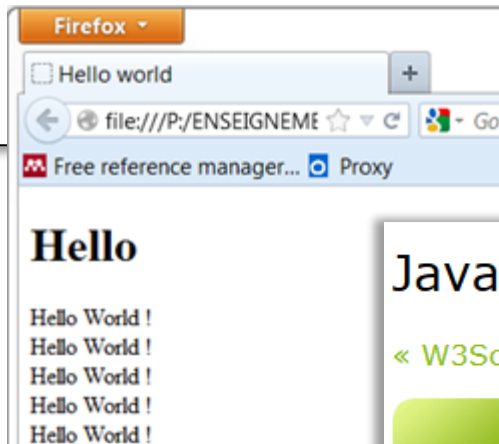
JavaScript: The World's Most Misunderstood Programming Language  
<http://javascript.crockford.com/javascript.html>

# JavaScript

- Code JavaScript peut être inclus dans des pages HTML

HelloWorld.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>
      <script>
        for (i=0; i < 5; i++) {
          document.write("Hello World !<br>");
        }
      </script>
    </p>
  </body>
</html>
```



HelloWorld.php

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>
      <?php>
        for (i=0; i < 5; i++) {
          echo "Hello World !<br>";
        }
      <?>
    </p>
  </body>
</html>
```

Et alors ? En quoi JavaScript est-il si différent d'autres langages comme PHP, JSP, ASP ?

## JavaScript Tutorial

« W3Schools Home



JavaScript is the scripting language of the Web.

All modern HTML pages are using JavaScript.

This tutorial will teach you JavaScript from basic to advanced.

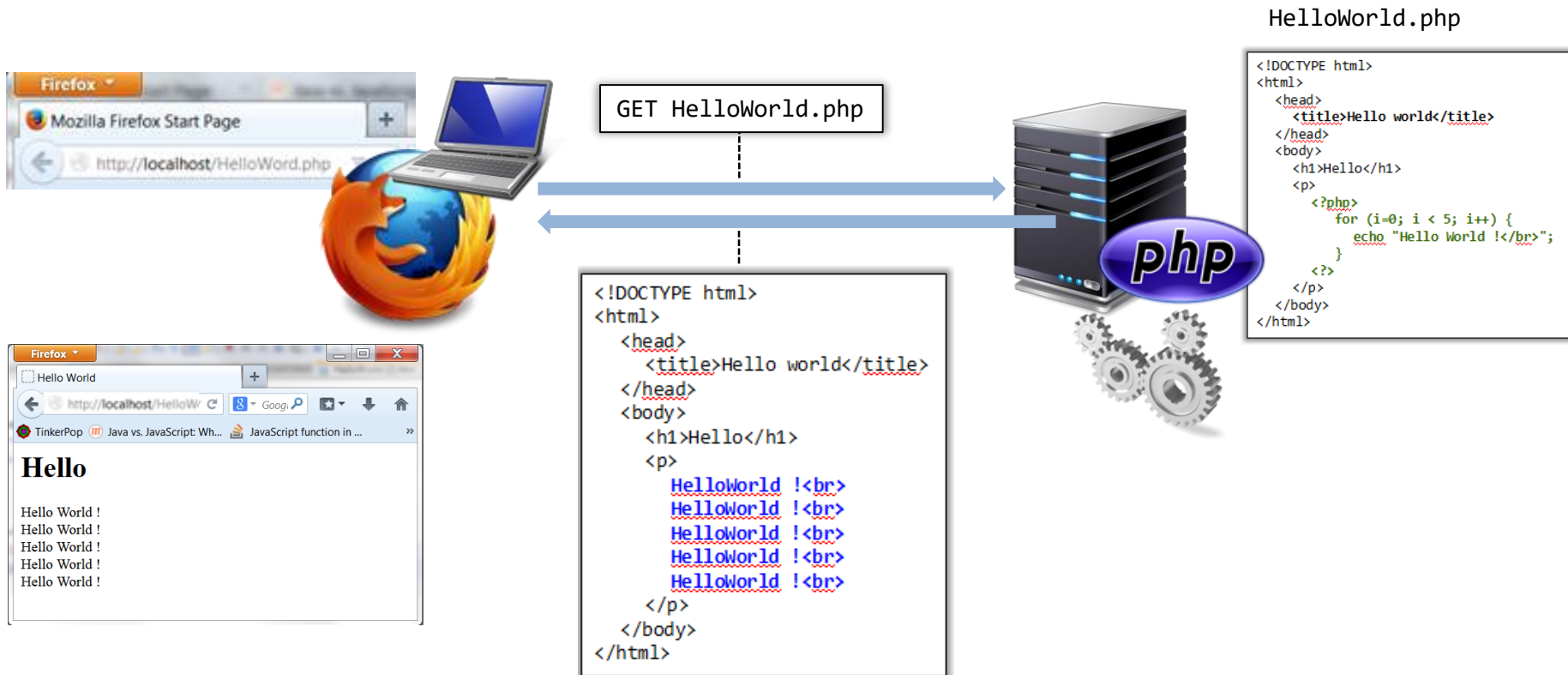
[http://www.w3schools.com/js/js\\_intro.asp](http://www.w3schools.com/js/js_intro.asp)



# JavaScript

## Introduction

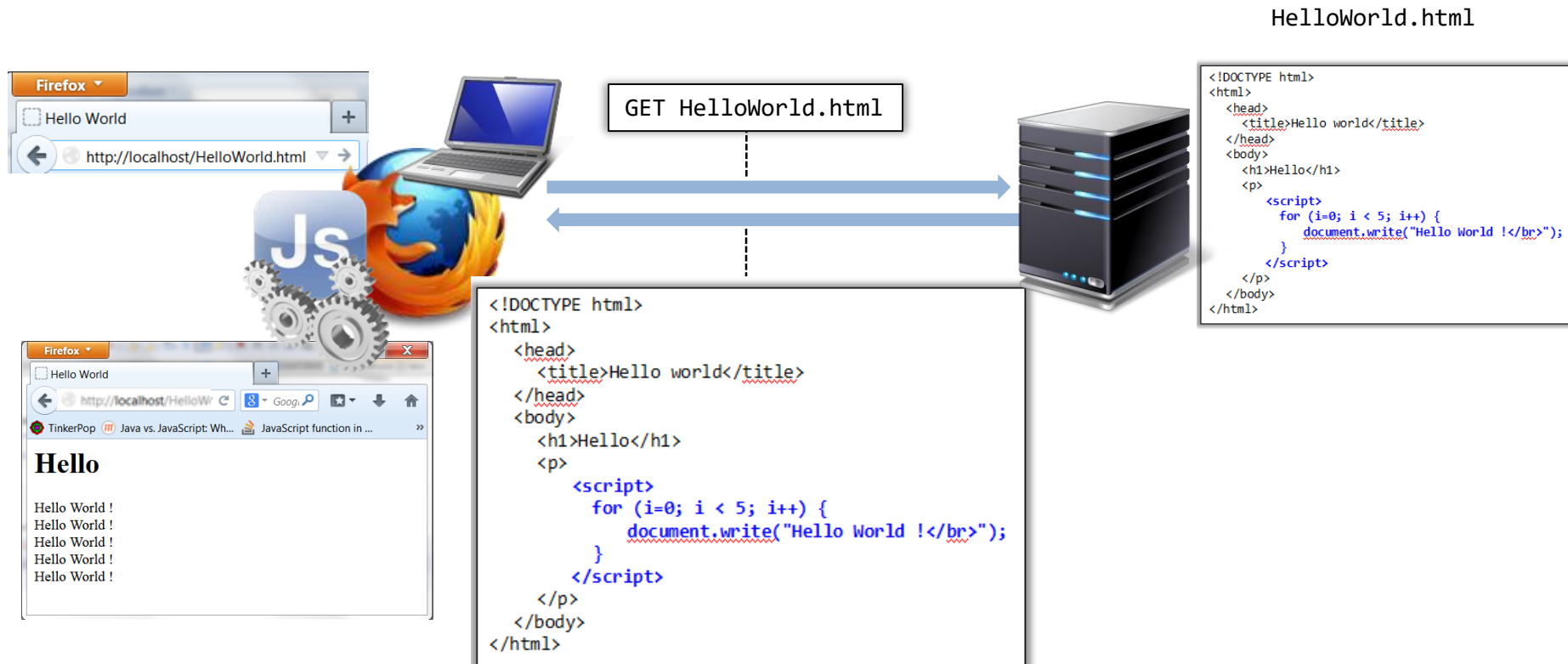
- code PHP, ASP, JSP s'exécute côté serveur



# JavaScript

## Introduction

- code JavaScript s'exécute côté client (dans le navigateur)





# JavaScript

- La manière dont JavaScript fonctionne\*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
  </head>
  <body>
    <h1>Hello</h1>
    <p>
      <script>
        for (i=0; i < 5; i++) {
          document.write("Hello World !<br>");
        }
      </script>
    </p>
  </body>
</html>
```

## 1 Ecriture

Vous créez vos pages HTML et votre code JavaScript que vous mettez dans un ou plusieurs fichiers

\*d'après "Head First HTML 5 programming"  
Eric Freeman, Elisabeth Robson  
Ed. O'Reilly, 2011

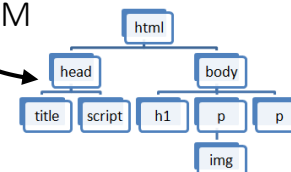


## 2 Chargement

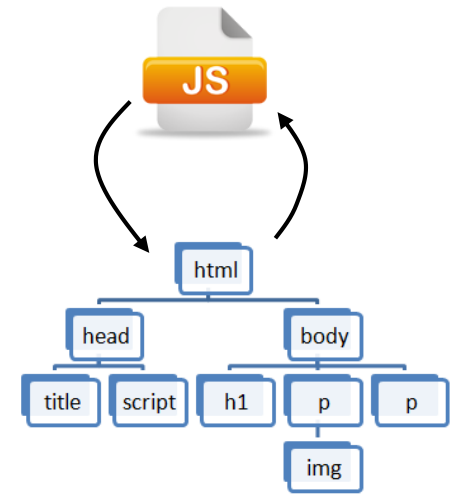
Le navigateur retrouve et charge la page en analysant (parsing) son contenu de haut en bas.

Lorsque le navigateur rencontre du code JavaScript il l'analyse, vérifie sa correction puis l'exécute

Le navigateur construit un modèle interne de la page HTML : le DOM



# Introduction



## 3 Exécution

Le navigateur affiche la page. JavaScript continue à s'exécuter, en utilisant le DOM pour examiner la page, la modifier, recevoir des événements depuis celle-ci, ou pour demander au navigateur d'aller chercher d'autres données sur le serveur web



*Et alors ? Quel intérêt de charger le client ?*

« W3Schools Home

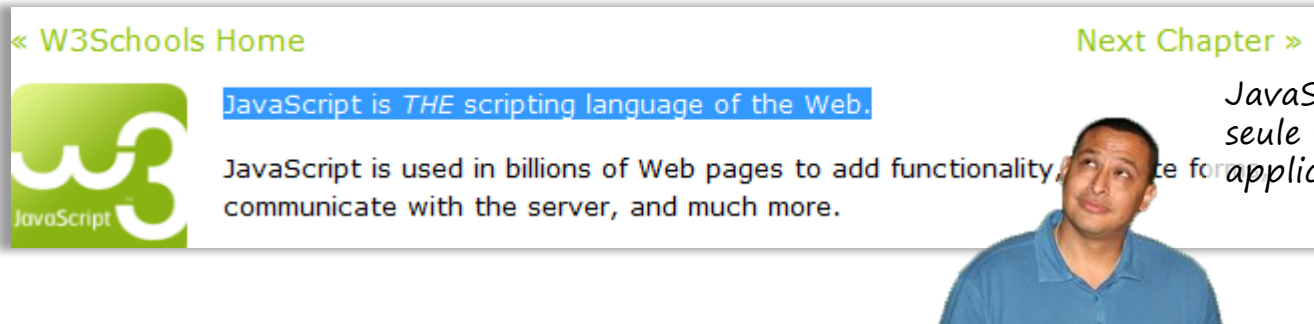
Next Chapter »



JavaScript is *THE* scripting language of the Web.

JavaScript is used in billions of Web pages to add functionality, validate forms, communicate with the server, and much more.

- Vérification des données saisies.
    - N'envoyer au serveur que des données correctes
  - Réduire les traitements sur le serveur.
  - Réduire les coûts de communication.
  - Pouvoir modifier la présentation (animations etc...) sans avoir recours au serveur.
- ⇒ améliorer l'interactivité, diminuer les temps de réponse.
- Mais attention :
    - Respecter la confidentialité.
    - Risque d'analyse du code transmis
      - Exemple : algorithme de vérification de la validité d'un numéro de carte bleue.



*JavaScript serait donc la seule technologie applicative côté client ?*

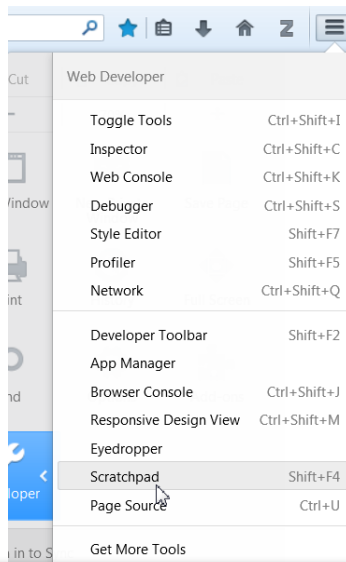
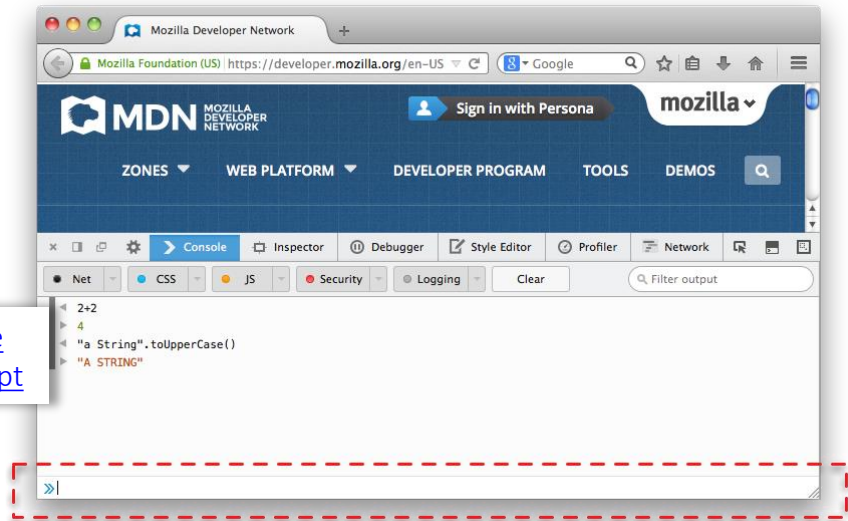
- Non, d'autres technologies existent
  - Adobe Flash
  - applets Java, JavaFX
- Mais avec le développement d'AJAX et l'introduction de HTML5 (et de ses nombreuses API JavaScript) JavaScript est revenu au 1<sup>er</sup> plan
- Possibilité d'utiliser aussi JavaScript côté serveur :
  - Internet Information Server (Microsoft)
  - Intra Builder (Inprise)
  - Node.js
  - Angular.js ...



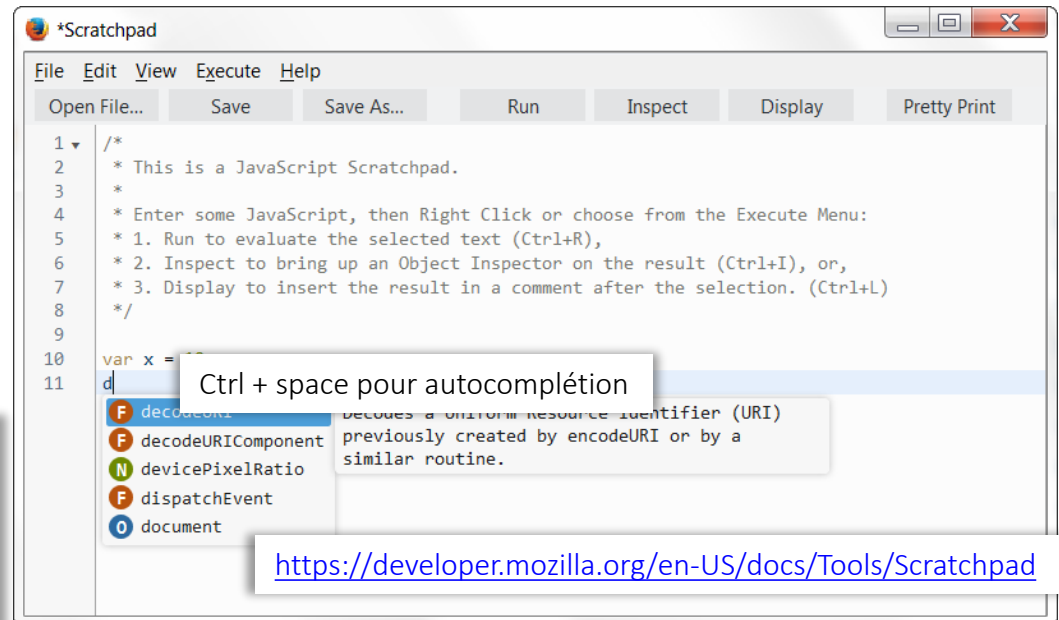
# Firefox – Outillage JavaScript

Console JavaScript: ligne de commandes pour entrer des instructions JavaScript.

[https://developer.mozilla.org/en-US/docs/Tools/Web\\_Console](https://developer.mozilla.org/en-US/docs/Tools/Web_Console)  
[https://developer.mozilla.org/fr/docs/Outils/Console\\_JavaScript](https://developer.mozilla.org/fr/docs/Outils/Console_JavaScript)



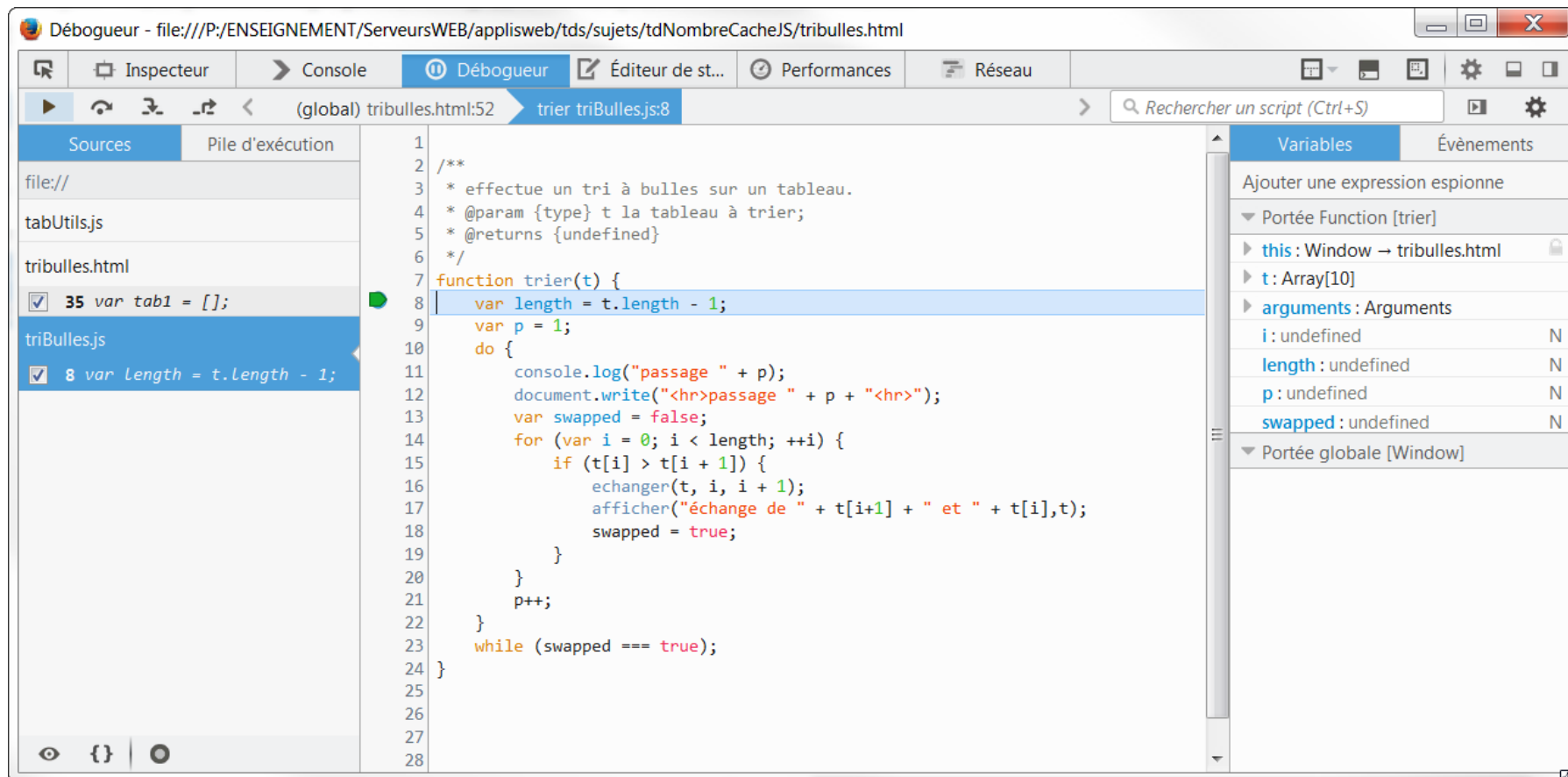
**Scratchpad:** fournit un environnement pour expérimenter avec du code JavaScript. Possibilité d'écrire, d'exécuter et d'examiner les résultats de code interagissant avec la page web.



<https://developer.mozilla.org/en-US/docs/Tools/Scratchpad>

# Firefox – Outillage JavaScript

**Débogueur:** permet d'avancer pas à pas dans du code JavaScript et de l'examiner ou de le modifier, afin de retrouver et de corriger les bugs.



<https://developer.mozilla.org/en-US/docs/Tools/Debugger>  
<https://developer.mozilla.org/fr/docs/Outils/Débogueur>

# balise script

- `<script> </script>`
  - délimite code JavaScript
  - peut être insérée n'importe où dans la page HTML



commentaires HTML : masque code pour navigateurs ne supportant pas JavaScript (inutile avec dernières version de navigateurs)

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      document.write("<title>Hello world 1</title>");
    </script>
  </head>
  <body>
    <h1>Hello</h1>
    <p>
      <script>
        for (i=0; i < 5; i++) {
          document.write("Hello World !<br>");
        }
      </script>
    </p>
    <h1>Bye Bye</h1>
    <p>
      <script>
        <!--
          for (i=0; i < 5; i++) {
            document.write("Bye Bye World !<br>");
          }
          // -->
        </script>
        <noscript>
          Java script pas activé
        </noscript>
      </p>
    </body>
  </html>
```

alternative si JavaScript désactivé

Hello

Bye Bye

Java script pas activé

# balise script

- **type**

- ex : `<script type="text/javascript"> ... </script>`
- utilisé par certains "vieux codes".
- inutile avec les navigateurs modernes et HTML5 : JavaScript est le langage par défaut.

fichier texte contenant code JavaScript

- ***src="url fichier externe"***

- pour utiliser code JavaScript externalisé dans un fichier distinct de la page HTML.
- fonctionnement équivalent à une intégration directe du code JavaScript à cet endroit.
- ex: `<script src="./mesScripts/monScript.js"></script>`

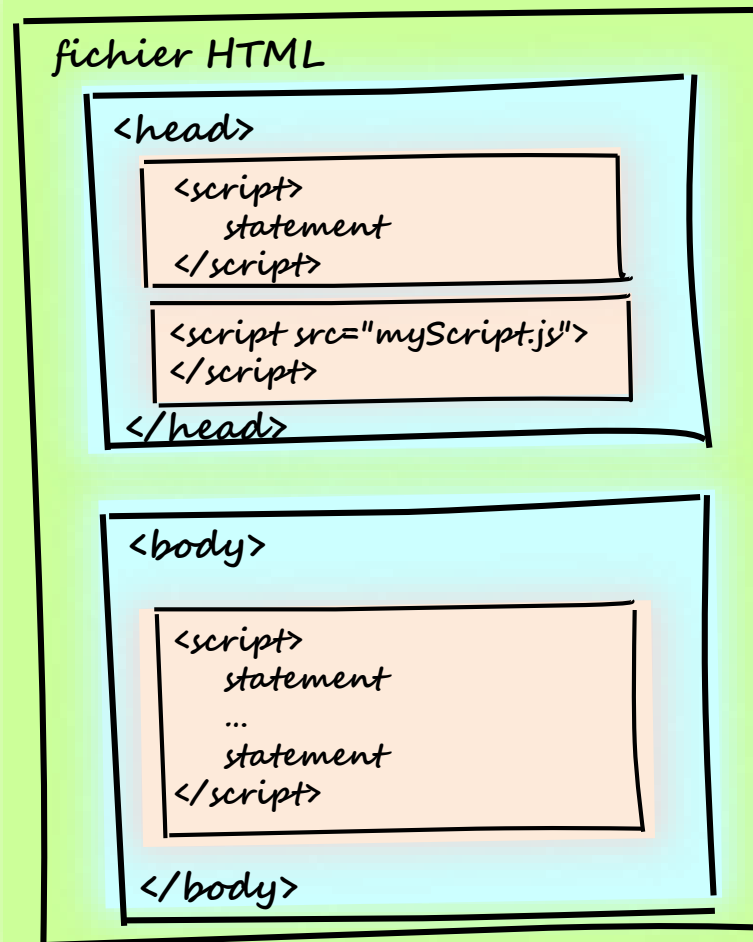


il ne faut pas ajouter d'instructions entre les balises `<script>` et `</script>`

`<script src="./mesScripts/monScript.js" />`

# balise script

- peut être insérée n'importe où dans la page HTML



\* figure d'après "Head First HTML 5 programming"  
Eric Freeman, Elisabeth Robson- Ed. O'Reilly, 2011

Placer les scripts dans l'en tête pour qu'ils soient exécutés avant le chargement de la page

Les scripts placés dans le body sont exécutés au fur et à mesure que celui-ci est chargé

Le plus souvent les scripts sont placés dans l'entête de la page.

Il peut parfois y avoir de légers avantages à les placer à la fin du corps de la page (juste avant </body>) : des images en provenance de différents sites peuvent être téléchargées simultanément mais une fois que le navigateur a rencontré une balise script ce n'est plus possible → les scripts bloquent les téléchargements parallèles



# Variables et typage

- déclaration par **var nomDeVariable**
  - langage non typé → type d'une variable défini qu'au moment de l'exécution
  - pas obligation d'initialiser une variable à sa déclaration → une variable non initialisée est **undefined**
  - possibilité de changer le type d'une variable à l'exécution
  - possibilité d'initialiser une variable à sa déclaration à l'aide d'expressions littérales

```
//Déclaration de a, de b et de c.  
var a;  
var b,c;  
  
//Affectation de a avec un entier.  
a = 1;  
// Affectation de a avec une chaîne  
a = "une chaîne de caractères";  
  
//forme littérale pour une entier  
var prix = 150;  
  
//forme littérale pour un réel.  
var tva= 19.6;  
  
//forme littérale pour un tableau  
//d'entiers  
var tab1 = [ 1, 2, 3, 4 ];  
  
//forme littérale pour un objet  
var obj = {  
    "attribut1":"valeur1",  
    "attribut2":"valeur2"  
};
```

# Types primitifs

- correspondent à des données stockées directement dans la pile d'exécution
- trois types primitifs
  - **boolean**
    - deux valeurs littérales : **true** (vrai) et **false** (faux).
  - **number** :  $10^{-308} > \text{nombre} < 10^{308}$ 
    - Les nombres entiers
    - les nombres décimaux en virgule flottante
    - valeurs particulières:
      - Positive Infinity ou +Infinity (valeur infinie positive)
      - Negative Infinity ou -Infinity (valeur infinie négative)
      - Nan (Not a Number)
  - **string**
    - chaînes de caractères
  - correspondent à des pseudo-objets, ils possèdent des propriétés et méthodes

```
var nombre=114;  
var chaine = nombre.toString();  
var nbreChiffres = chaine.length;
```

# Variable de type objet

- variable correspondent à des références vers des objets stockés dans le tas (heap).
- possibilité d'utiliser plusieurs variables pointant sur la même adresse (même instance)
- opérateur **typeof** permet de déterminer le type d'une variable.

```
var ref1 = new Object();  
var ref2 = ref1;  
// ref1 et ref2 pointent sur la même adresse  
// elles référencent le même objet
```

```
var v0;  
document.write("type de v0 " + typeof v0 + "<br/>");  
var v1 = 125;  
document.write("type de v1 " + typeof v1 + "<br/>");  
var v2 = "toto";  
document.write("type de v2 " + typeof v2 + "<br/>");  
var v3 = ["val1", "val2", "val3" ];  
document.write("type de v3 " + typeof v3 + "<br/>");  
var v4 = null;  
document.write("type de v4 " + typeof v4 + "<br/>");
```

```
type de v0 undefined  
type de v1 number  
type de v2 string  
type de v3 object  
type de v4 object
```

# Types spéciaux

- aux types primitifs s'ajoutent 2 types spéciaux
  - **Undefined** :
    - possède une unique valeur : **undefined**.
    - une variable déclarée dont le contenu n'a jamais été initialisé.
  - **Null** :
    - possède une unique valeur : **null**.
    - spécifie qu'une variable a bien été initialisée mais qu'elle ne pointe sur aucun objet.

# Opérateurs et instructions de contrôle

- JavaScript propose les opérateurs et instructions standards de C++/Java
- opérateurs
  - arithmétiques  
+, -, \*, /, %, ++, --, ...
  - comparaisons  
==, !=, <, >, <=, >=
  - logiques  
&&, ||, !
- instructions
  - conditionnelles  
if, if-else, switch
  - itératives  
for, while, do

```
html>
<!-- Dave Reed  js03.html  2/01/04 -->

<head>
  <title>Folding Puzzle</title>
</head>

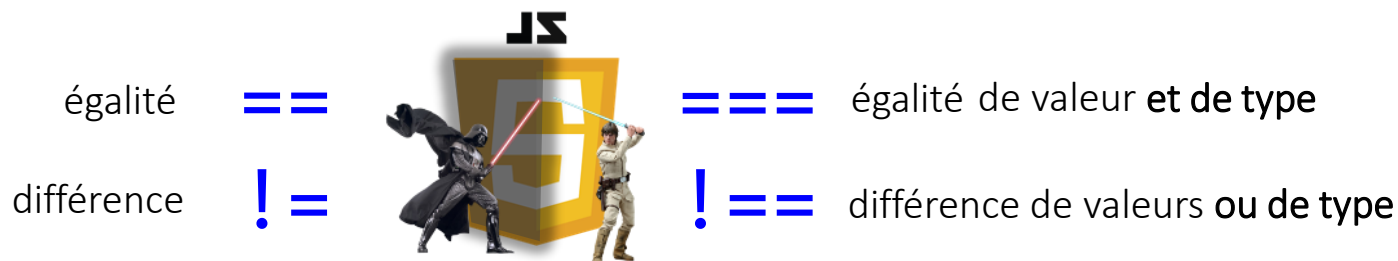
<body>
  <script type="text/javascript">
    var distanceToSun = 149597870e3;
    var thickness = .002;

    var foldCount = 0;
    while (thickness < distanceToSun) {
      thickness *= 2;
      foldCount++;
    }
    document.write("Number of folds = " +
                  foldCount);

  </script>
</body>
</html>
```

# Opérateurs d'égalité

- JavaScript possède deux jeux d'opérateurs d'égalité



mais avec conversion implicite des valeurs si les opérandes sont de type différent !!!

```
var x = 3;  
x == 3   → true  
x == '3' → true
```

```
var x = 3;  
x === 3   → true  
x === '3' → false
```

```
' ' == '0'      → false  
0 == ' '       → true  
0 == '0'       → true  
false == 'false' → false  
false == '0'    → true  
false == undefined → false  
false == null   → false  
null == undefined → true  
' \t\r\n' == 0   → true
```

- Règles de conversion compliquées et difficiles à mémoriser

*utiliser === et !==  
toujours tu dois !*



exemple tiré de *JavaScript, les bons éléments*  
Douglas Crockford, Ed. Pearson France, 2013

# Fonctions


- Définition de fonctions similaire à C++/Java sauf :
  - pas de type de retour pour la fonction
  - pas de types pour les paramètres
  - passage des paramètres par valeur uniquement (les paramètres sont une copie des arguments)

```
<html>
<head>
  <title>Nombre Premier</title>
  <meta charset="utf-8" />
  <script>
    function isPrime(n)
    // Assumes: n > 0
    // Returns: true if n is prime, else false
    {
      if (n < 2) {
        return false;
      }
      else if (n == 2) {
        return true;
      }
      else {
        for (var i = 2; i <= Math.sqrt(n); i++) {
          if (n % i == 0) {
            return false;
          }
        }
        return true;
      }
    }
  </script>
</head>
<body>
  <h1>Exemple de fonction</h1>
  <script>
    var nb = prompt("Entrez un nombre : ", "");
    nb = parseInt(nb);
    document.write("le nombre " + nb +
      " est premier : " + isPrime(nb));
  </script>
</body>
</html>
```

les définitions de fonctions placées dans l'en-tête du document HTML <head></head> sont chargées en premier. une fonction est ainsi définie avant son utilisation dans le body

# Fonctions

- possibilité d'affecter une fonction à une ou plusieurs variables
  - affectation à partir d'une fonction existante
  - à l'aide d'une fonction anonyme à la création de la variable



```
Hello World !  
Hello World !  
Hello World !  
Bye Bye World !  
Bye Bye World !  
Bye Bye World !
```

```
<html>  
  <head>  
    <title>Expression régulières</title>  
    <meta charset="UTF-8">  
    <script>  
      function helloWorld(nb) {  
        for (var i=0; i < nb; i++) {  
          document.write("Hello World !<br\\>");  
        }  
      }  
      var var1 = helloWorld;  
      var fonctionAnonyme1 = function(message,nb) {  
        for (var i=0; i < nb; i++) {  
          document.write(message + " World !<br\\>");  
        }  
      };  
    </script>  
  </head>  
  <body>  
    <h1>Affectation d'une fonction à une variable</h1>  
    <p>  
      <script>  
        var1(3); // équivalent à helloWorld(3)  
        fonctionAnonyme1("Bye Bye",3);  
      </script>  
    </p>  
  </body>  
</html>
```



# Fonctions

- une fonction peut être passée en paramètre d'une autre fonction

11,12,13,14,15  
110,120,130,140,150



```
<html>
  <head>
    <title>Expression régulières</title>
    <meta charset="UTF-8">
    <script>
      function ajoute10(a) {
        return a + 10;
      }
      function multipliePar10(a) {
        return a * 10;
      }
      function traiterTableau(tab,operation) {
        for (var i = 0; i < tab.length; i++) {
          tab[i] = operation(tab[i]);
        }
      }
    </script>
  </head>
  <body>
    <h1>Affectation d'une fonction à une variable</h1>
    <p>
      <script>
        var tab = [1,2,3,4,5];
        traiterTableau(tab,ajoute10);
        document.write(tab + "<br>");
        traiterTableau(tab,multipliePar10);
        document.write(tab + "<br>");
      </script>
    </p>
  </body>
</html>
```

# Fonctions

- JavaScript n'utilise pas les signatures pour identifier les fonctions
  - pas de surcharge comme en Java
  - si deux fonctions ont le même nom , l'interpréteur JavaScript utilise celle définie en dernier


Hello World !  
Hello World !  
Hello World !  
  
Hello World !  
Hello World !  
Hello World !

```
<html>
<head>
  <title>Expression régulières</title>
  <meta charset="UTF-8">
  <script>
    function helloWorld(nb,message) {
      for (var i=0; i < nb; i++) {
        document.write(message + " World !<br\>");
      }
    }
    function helloWorld(nb) {
      for (var i=0; i < nb; i++) {
        document.write("Hello World !<br\>");
      }
    }
  </script>
</head>
<body>
  <h1>Surcharge de fonction</h1>
  <p>
    <script>
      helloWorld(3,"Salut");
      helloWorld(3);
    </script>
  </p>
</body>
</html>
```



# Fonctions

- possibilité de contourner le problème de l'absence de surcharge
  - liste des arguments d'une fonction accessible via la variable implicite **arguments**



Salut World !  
Salut World !  
Salut World !  
Hello World !  
Hello World !  
Hello World !

```
<html>
  <head>
    <title>Expression régulières</title>
    <meta charset="UTF-8">
    <script>
      function helloWorld() {
        var nb = 1;
        var message = "Hello";

        if (arguments.length == 2) {
          nb = arguments[0];
          message = arguments[1];
        }
        if (arguments.length == 1) {
          nb = arguments[0];
        }

        for (var i=0; i < nb; i++) {
          document.write(message + " World !<br\>");
        }
      }
    </script>
  </head>
  <body>
    <h1>Surcharge de fonction</h1>
    <p>
      <script>
        helloWorld(3,"Salut");
        helloWorld(3);
      </script>
    </p>
  </body>
</html>
```

# Fonctions

- Faut-il spécifier une valeur de retour pour toutes les fonctions ?
  - toutes les fonctions retournent une valeur
    - **undefined** si pas d'instruction **return** dans la fonction
  - attention aux erreurs si votre code n'est pas capable de gérer ce type de valeur
  - c'est une bonne idée de spécifier une valeur de retour (par ex. **false**)
- Est-ce qu'il y a des restrictions sur les arguments (paramètres) que l'on peut passer à une fonction ?
  - Non, on peut passer n'importe quel objet, variable ou valeur à une fonction
  - on peut passer plus de paramètres que ce que la fonction attend → ils sont ignorés
  - on peut passer moins de paramètres que ce que la fonction attend → les paramètres manquant seront automatiquement initialisés à **undefined**

# Portée (*scope*) des variables

- Scope: ensemble des variables\* auxquelles on a accès dans un contexte d'exécution donné.
- Lorsque l'on rentre dans un fonction, le contexte d'exécution, et par conséquence le scope, changent.

\* En JavaScript objets et fonctions sont aussi des variables

# Variables scope (portée)

- une variable déclarée à l'intérieur d'une fonction est une variable **locale** à cette fonction
  - Créée au début de l'exécution de la fonction, détruite à la fin de son exécution
  - Ne peut être accédée qu'à l'intérieur de la fonction
  - Des variables avec le même nom peuvent être utilisées dans des fonctions différentes.

```
// le code ici ne peut accéder à x

function hello1 () {
  var x = "World";
  document.write("Hello1 " + x + "<br>");
}
```

```
// le code ici ne peut accéder à x
console.log("Hello " + x);
```

```
hello1();
```

```
hello2();
```

```
function hello2 () {
  var x = 3;
  document.write("Hello2 " + x + "<br>");
}
```

si présente cette instruction provoquerait une erreur d'exécution et le reste du script ne serait pas exécuté

Hello1 World

c'est la variable **x** locale à **hello1** qui est utilisée

Hello2 3

c'est la variable **x** locale à **hello2** qui est utilisée

# Variables scope (portée)

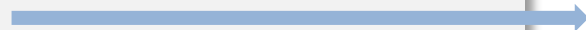
## Variables globales

- une variable déclarée à l'extérieur d'une fonction est une variable **globale**
  - Tous les scripts et fonctions JavaScript d'une page web peuvent y accéder.
  - Quand une variable locale a le même nom qu'une variable globale, la déclaration locale masque la déclaration globale

```
function hello1 () {  
    document.write("Hello1 " + x + "<br>");  
}
```

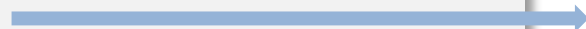
```
// x est une variable globale elle  
// est accessible partout  
var x = "World";
```

```
hello1();
```



Hello1 World c'est la variable x globale qui est utilisée

```
hello2();
```



Hello2 3 c'est la variable x globale  
qui est modifiée et utilisée

```
function hello2 () {  
    x = 3;  
    document.write("Hello2 " + x + "<br>");  
}
```

# Variables scope (portée)

## Variables globales automatiques

- si une valeur est affectée à une variable non déclarée, celle-ci devient automatiquement une **variable globale**

```
function hello1 () {  
  x = 3;  
  document.write("Hello1 " + x + "<br>")  
}  
  
hello1();  
  
document.write("Global " + x + "<br>");  
  
hello2();  
  
document.write("Global " + x + "<br>");  
  
function hello2 () {  
  x = x + 1;  
  document.write("Hello2 " + x + "<br>")  
  x = 'World';  
}
```

x n'a pas été déclarée, une variable globale est automatiquement créée

Hello1 3

Global 3

Hello2 4

Global World



# Portées des variables

- Variables peuvent être **globales** ou **locales**.
- Variable globale** : accessible à n'importe quel endroit du script.
- Variable locale** : n'est utilisable que dans la fonction où elle est déclarée

```
<html>
<head>
  <title>variable locale/globale</title>
  <meta charset="UTF-8">
  <script>
    function foo1() {
      document.write("dans foo1 x = " + x + "<br>");
      x = 14;
      y = 0;
      document.write("dans foo1 y = " + y + "<br>");
      y++;
      var z = "une chaîne";
      document.write("dans foo1 z = " + z + "<br>");
    }

    function foo2() {
      document.write("dans foo2 x = " + x + "<br>");
      document.write("dans foo2 y = " + y + "<br>");
      document.write("dans foo2 z = " + z + "<br>");
    }
  </script>
</head>
<body>
  <h1>Variable locale/variable globale</h1>
  <script>
    var x = 'hello world';
    try {
      foo1();
      foo2();
      document.write("terminé");
    } catch (ex) {
      document.write("pb : " + ex);
    }
  </script>
</body>
</html>
```

variable globale

variable locale

accès à une variable non déclarée  
ERREUR à l'exécution

variable globale

dans foo1 x = hello world  
dans foo1 y = 0  
dans foo1 z = une chaîne  
dans foo2 x = 14  
dans foo2 y = 1  
pb : ReferenceError: z is not defined

# Déclaration et affectation

- Lecture d'une variable non déclarée → une erreur
- Lecture d'une variable déclarée mais non affectée, → *undefined*

toujours déclarer les variables en utilisant **var**

```
<body>
  <h1>Variable locale/variable globale</h1>
  <script>
    try {
      var x = 'hello world';
      foo1();
      var z;
      foo2();
      document.write("terminé");
    } catch (ex) {
      document.write("pb : " + ex);
    }
  </script>
</body>
</html>
```

dans foo1 x = hello world  
dans foo1 y = 0  
dans foo1 z = une chaîne  
dans foo2 x = 14  
dans foo2 y = 1  
dans foo2 z = undefined  
terminé

```
<html>
  <head>
    <title>variable locale/globale</title>
    <meta charset="UTF-8">
    <script>
      function foo1() {
        document.write("dans foo1 x = " + x + "<br>");
        x = 14;
        y = 0
        document.write("dans foo1 y = " + y + "<br>");
        y++;
        var z = "une chaîne";
        document.write("dans foo1 z = " + z + "<br>");
      }

      function foo2() {
        document.write("dans foo2 x = " + x + "<br>");
        document.write("dans foo2 y = " + y + "<br>");
        document.write("dans foo2 z = " + z + "<br>");
      }
    </script>
  </head>
  <body>
```

<h1>Variable locale/variable globale</h1>

```
<script>
  try {
    var x = 'hello world';
    foo1();
    foo2();
    document.write("terminé");
  } catch (ex) {
    document.write("pb : " + ex);
  }
```

dans foo1 x = hello world  
dans foo1 y = 0  
dans foo1 z = une chaîne  
dans foo2 x = 14  
dans foo2 y = 1  
pb : ReferenceError: z is not defined

# Remontée des variables (hoisting)

```
document.write(x);
```

x n'a pas été déclarée, erreur d'exécution  
**ReferenceError: x is not defined.**

```
var x;  
document.write(x);
```

x déclarée, mais pas initialisée  
**undefined.**

```
var y = 1;  
document.write(y);
```

y déclarée et initialisée  
**1**

```
document.write(x);
```

**undefined.**

```
document.write(y);
```

**undefined.**

```
var x;  
var y = 1;
```

???



```
var x, y;  
  
document.write(x);  
  
document.write(y);  
  
y = 1;
```

- En JavaScript des variables peuvent être déclarées après avoir été utilisées ⇔ des variables peuvent être utilisées avant d'avoir été déclarées.
- Le comportement par défaut de JavaScript consiste à faire remonter les déclarations (*hoist* = *hisser*) de variables au début de la portée (scope) courante (le script ou la fonction courante).



Seules les déclarations sont remontées, pas les initialisations.

# Remontée des variables (hoisting)

```
function hello1 () {  
    document.write("Hello1 " + x + "<br>");  
}
```

```
var x = "World";
```

```
hello1();
```

Hello1 World

```
hello2();
```

Hello2 3

```
function hello2 () {  
    x = 3;  
    document.write("Hello2 " + x + "<br>");  
}
```

```
function hello1 () {  
    document.write("Hello1 " + x + "<br>");  
}
```

```
hello1();
```

Hello1 undefined

```
hello2();
```

Hello2 3

```
var x = "World";
```

```
function hello2 () {  
    x = 3;  
    document.write("Hello2 " + x + "<br>");  
}
```



```
function hello1 () {  
    document.write("Hello1 " + x + "<br>");  
}
```

```
var x;
```

```
hello1();
```

```
hello2();
```

```
x = "World";
```

```
function hello2 () {  
    x = 3;  
    document.write("Hello2 " + x + "<br>");  
}
```

# Remontée des variables (hoisting)

inspiré de "Comment le hoisting fonctionne en JavaScript et pourquoi" Fabien Huet, sept. 2014  
<http://blog.wax-o.com/2014/09/comment-le-hoisting-fonctionne-en-javascript-et-pourquoi/>

```
var x = 1;

(function() {
  document.write(x + "<br>");
})();

document.write(x + "<br>");
```

1

1

```
var x = 1;

(function() {
  document.write(x + "<br>");
  var x = 10;
})();

document.write(x + "<br>");
```

undefined

1

↕

```
var x;
x = 1;

(function() {
  var x;
  document.write(x + "<br>");
  x = 10;
})();

document.write(x + "<br>");
```

- variables globales automatiques
- remontée des variables



- **toujours** déclarer toutes les variables en début de portée.



*avec ES5 utiliser mode strict tu peux*

# Mode strict ("use strict")

- mode strict
  - introduit avec JavaScript 1.8.5 (ECMAScript version 5).
  - facilite l'écriture de code JavaScript plus sûr.
  - change "la mauvaise syntaxe" précédemment tolérée en de véritables erreurs
    - ex : plus la possibilité d'utiliser des variables sans les avoir déclarées

## JavaScript "normal"

```
var maVariable = true;

...
while (maVariable) {
    if (...) {
        maVariable = false;
    }
}
...

```

 faute de frappe  
→ crée une nouvelle variable globale  
→ boucle infinie

## JavaScript mode strict

```
"use strict";

var maVariable = true;

...
while (maVariable) {
    if (...) {
        maVariable = false;
    }
}
...

```

expression qui doit être au début du code JavaScript ou au début d'une fonction. Ignorée par versions antérieures de JavaScript

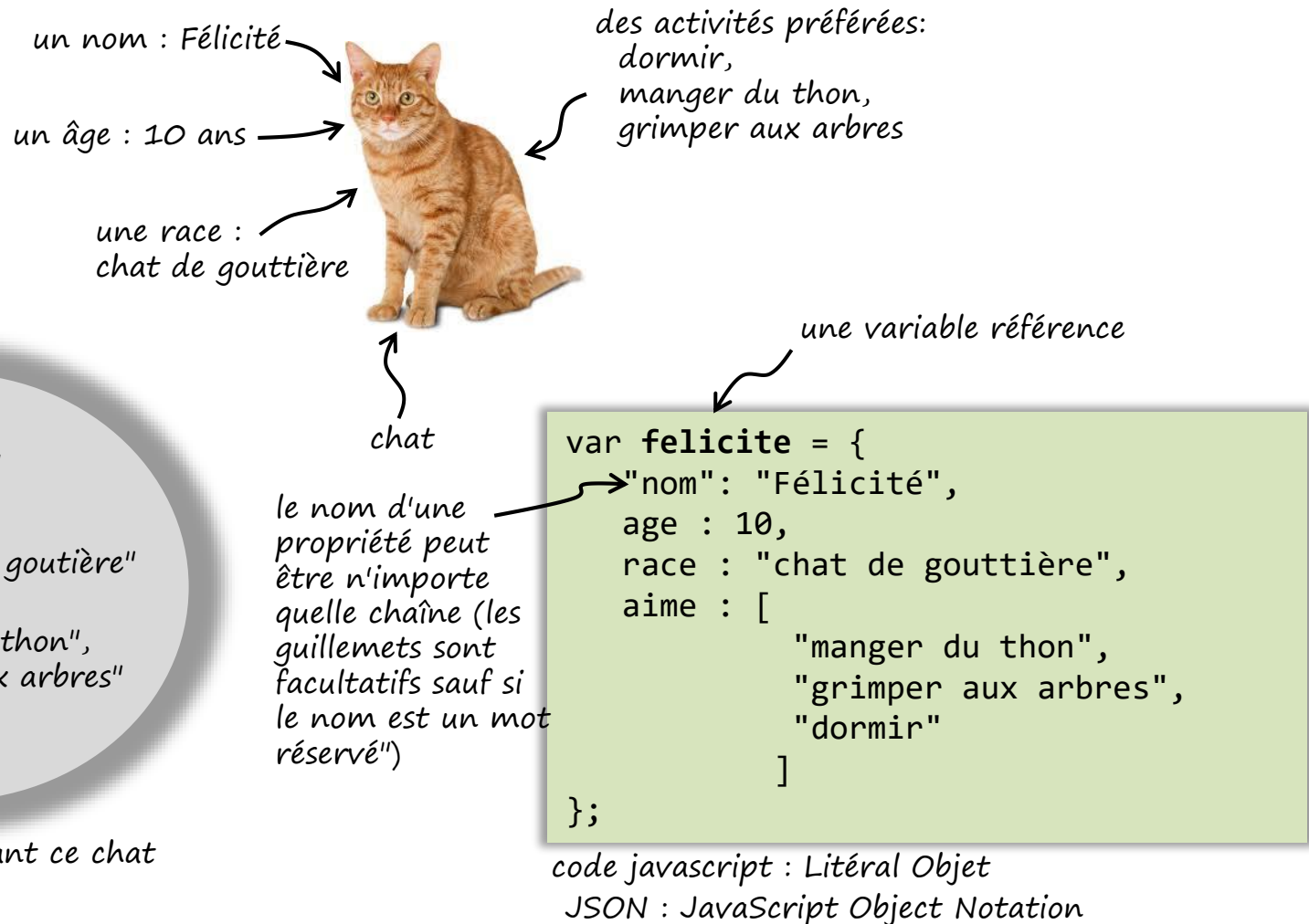
provoque une erreur  
**ReferenceError:**  
**maVariable is not defined.**

pour plus de détails sur le mode strict  
[http://www.w3schools.com/js/js\\_strict.asp](http://www.w3schools.com/js/js_strict.asp)

# JavaScript et programmation orientée objet

- JavaScript permet l'utilisation d'objets et dispose d'objets natifs mais n'est pas à proprement parler un langage orienté objet
  - ne fournit pas d'éléments de langage pour supporter ce paradigme (par exemple pas de notion explicite de classe comme en Java, C++) mais émule certains de ses principes
- Deux types d'objets
  - Objets natifs
    - types prédéfinis : Array, String, Date ...
  - Objets personnalisés
    - types définis par l'application

- objet javascript = un ensemble de propriétés

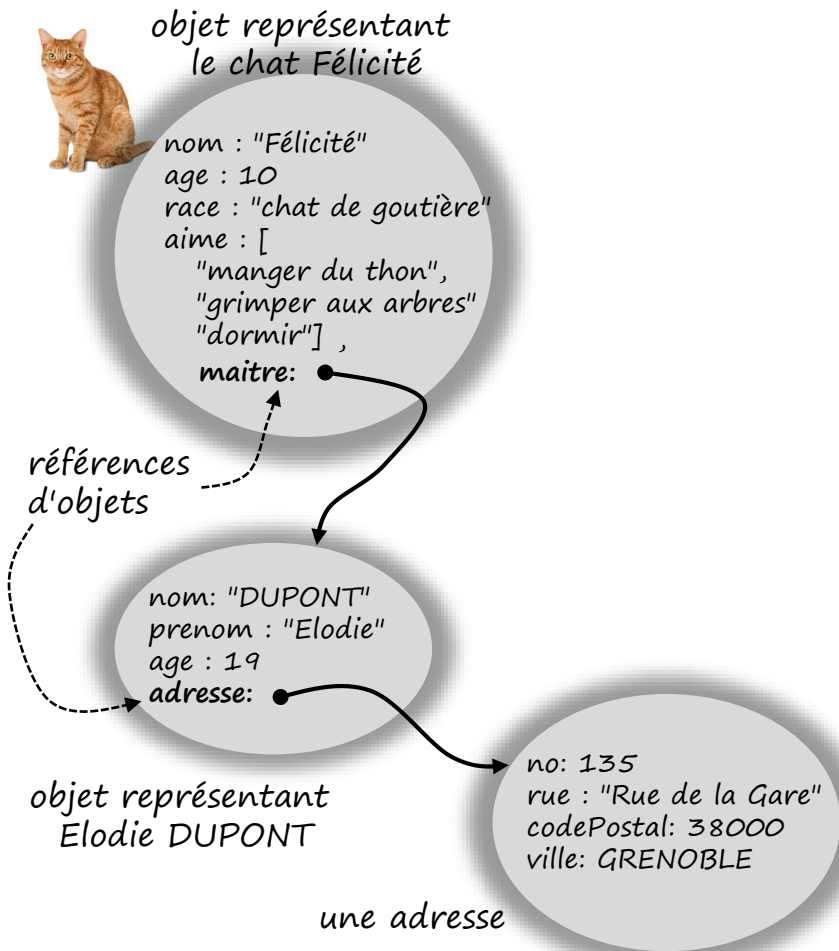




# Objets JavaScript

## définir un objet

- objet javascript = un ensemble de propriétés (couples nom : valeur)
- Valeur d'une propriété peut être définie par n'importe quelle expression, et même depuis un autre objet littéral

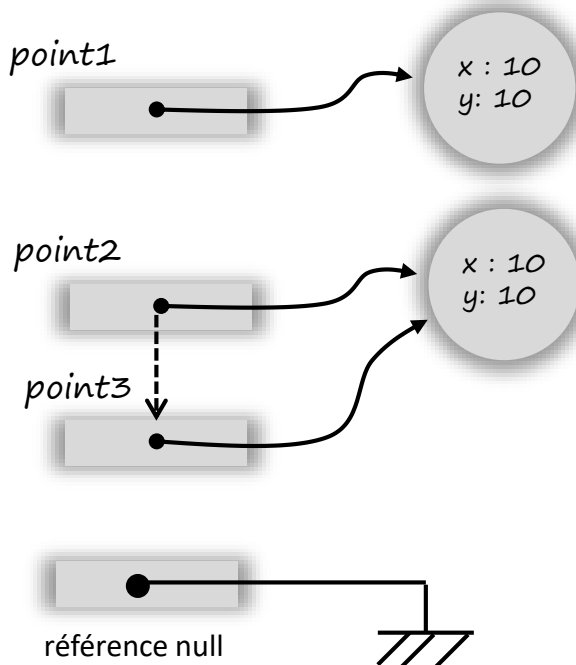


```
var elodie = {  
  nom: "DUPONT",  
  prenom: "Elodie"  
  age : 19,  
  adresse: {  
    no: 135,  
    rue: "Rue de la Gare",  
    codePostal: 38000,  
    ville: "Grenoble"  
  }  
};
```

```
var felicite = {  
  nom: "Félicité",  
  age : 10,  
  race : "chat de gouttière",  
  aime : [  
    "manger du thon",  
    "grimper aux arbres"  
    "dormir"  
  ],  
  maitre: elodie  
};
```

# Objets JavaScript

- Pour désigner des objets on utilise des variables d'un type particulier : les **références**
- Une référence contient l'**adresse** d'un objet
  - pointeur vers la structure de données correspondant aux propriétés (attributs) de l'objet
- Affecter une référence à une autre référence consiste à recopier les pointeurs
- Une référence peut posséder la valeur **null**
  - aucun objet n'est accessible par cette référence



```
var point1 = {  
  x = 10,  
  y = 10  
};
```

```
var point2 = {  
  x = 10,  
  y = 10  
};
```

```
var point3 = point2;
```

*une référence n'est pas un objet, c'est un nom pour accéder à un objet*

# Objets JavaScript

## ce que l'on peut faire avec un objet

- accéder aux propriétés

*ref.propriété*

une alternative à la notation pointée  
*ref["propriété"]*

```
if (felicite.age > 2) {  
    alert("MAAOUU");  
} else {  
    alert("miaou");  
}
```

```
if (felicite["age"] > 2) {  
    ...  
}
```

- énumérer toutes les propriétés

```
var prop;  
for (prop in felicite) {  
  
    alert("ce chat a " + prop + " comme propriété");  
  
    if (prop === "nom") {  
        alert("Ce chat s'appelle " + felicite[prop]);  
    }  
}
```

boucle for-in

la variable **prop**  
contient le nom de  
la propriété



c'est une chaîne  
de caractères

utilisation de la notation  
tableau pour accéder à la  
valeur de la propriété

# Objets JavaScript

## ce que l'on peut faire avec un objet (suite)

- changer la valeur d'une propriété

```
felicite.race = "mélange Chartreux + X";  
felicite.age++;  
felicite.aime.push("faire ses griffes sur le canapé");
```

- ajouter/supprimer des propriétés

```
felicite.poids = 3.5;           à partir de ce point l'objet félicité a une propriété poids  
...  
delete felicite.poids;         à partir de ce point félicité.poids est undefined
```

- passer l'objet comme paramètre d'une fonction

```
function miauler(unChat) {  
    if (unChat.age > 2) {  
        alert("MAAOUU");  
    } else {  
        alert("miaou");  
    }  
}  
  
miauler(felicite);
```

la référence de l'objet est passée en paramètre comme n'importe quelle autre variable

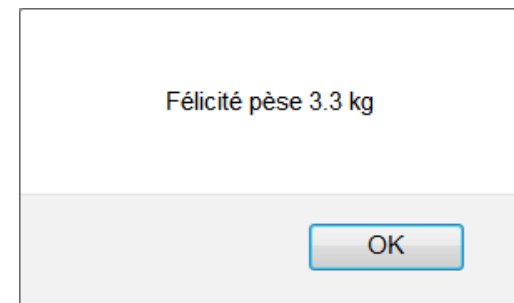
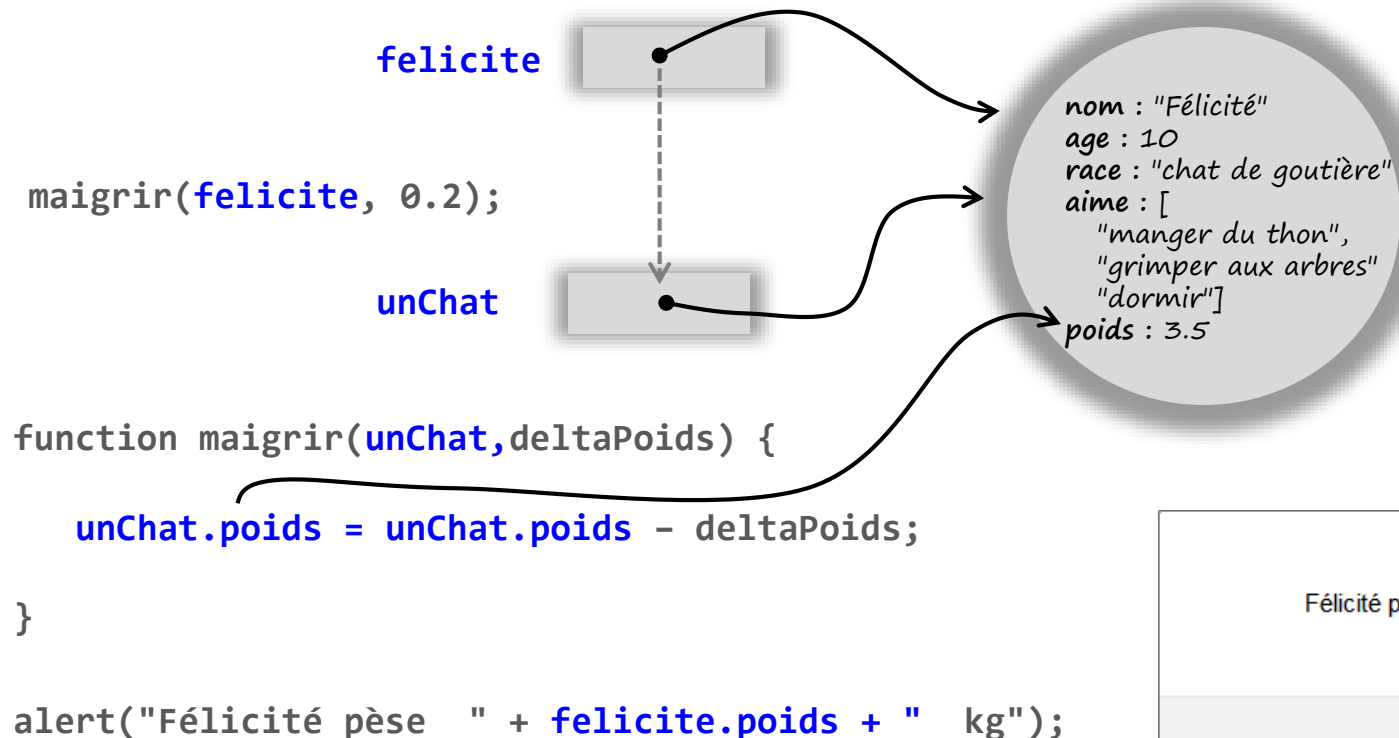
accès aux propriétés de l'objet

appel de la fonction avec la référence de l'objet sur lequel l'appliquer.

# Objets JavaScript

## objet paramètre d'une fonction

- comme en Java le passage de paramètres de fonctions est un passage par valeurs
  - la fonction dispose d'une copie de la variable et ne peut modifier cette dernière
  - si la variable est une référence, les propriétés de l'objet référencé peuvent elles être modifiées.



# Objets JavaScript

Les objets peuvent  
aussi avoir un  
comportement

- Les objets ne sont pas qu'un regroupement de valeurs les propriétés peuvent être aussi des fonctions.

```
var felicite = {  
  nom: "Félicité",  
  age : 10,  
  race : "chat de gouttière",  
  aime : [  
    "manger du thon",  
    "grimper aux arbres",  
    "dormir"  
  ],  
  poids: 3.5,  
  miauler: function () {  
    alert("Miaou ! Miaou !");  
  }  
};
```

*une fonction anonyme est affectée à une propriété de l'objet*

*ce type de fonction est généralement appelé **méthode** de l'objet*

invocation d'une méthode

```
felicite.miauler();
```

*envoi du **message** miauler à l'objet référencé par felicite*

# Objets JavaScript

le mot clé  
this

- comment accéder aux attributs d'un objet dans une méthode

```
var felicite = {  
  nom: "Félicité",  
  age : 10,  
  race : "chat de goutière",  
  aime : [  
    "manger du thon",  
    "grimper aux arbres",  
    "dormir"  
  ],  
  poids: 3.5,  
  miauler: function () {  
    alert("Miaou ! Miaou !");  
  },  
  maigrir :function (deltaPoids) {  
    this.poids = this.poids - deltaPoids;  
  }  
};
```

```
function maigrir(unChat,deltaPoids) {  
  unChat.poids = unChat.poids - deltaPoids;  
}
```

```
maigrir(felicite, 0.2);
```

*transformer cette fonction  
en une méthode de l'objet.*

*pour désigner l'une des  
propriétés de l'objet, le mot clé  
this doit être utilisé*

*appel de la méthode*

**felicite.maigrir(0.2);**

# Objets JavaScript

## Créer plusieurs objets du même type ?



```
var felicite = {  
  nom: "Félicité",  
  age : 10,  
  race : "chat de gouttière"  
  aime : [  
    "manger du thon",  
    "grimper aux arbres"  
    "dormir"  
  ],  
  poids: 3.5,  
  miauler: function () {  
    alert("Miaou ! Miaou !");  
  },  
  maigrir : function(deltaPoids) {  
    this.poids -= deltaPoids;  
  }  
};
```

Comment ne  
pas répéter  
de code ?



```
var felix = {  
  nom: "Félix",  
  age : 6,  
  race : "siamois"  
  aime : [  
    "se lécher",  
    "manger des croquettes",  
    "dormir"  
  ],  
  poids: 3.,  
  miauler: function () {  
    alert("Miaou ! Miaou !");  
  },  
  maigrir : function(deltaPoids) {  
    this.poids -= deltaPoids;  
  }  
};
```

```
felix.miauler = felicite.miauler;  
felix.maigrir = felicite.maigrir;
```



*ces solutions sont des  
sources potentielles d'erreur*



# Objets JavaScript

## Créer plusieurs objets du même type ?



- définir un constructeur

*une fonction comme une autre  
par convention débute par Majuscule*

*les paramètres définissent les  
valeurs des propriétés que l'on  
souhaite que l'objet ait.*

```
var felicite = {  
  nom: "Félicité",  
  age : 10,  
  race : "chat de gouttière"  
  aime : [  
    "manger du thon",  
    "grimper aux arbres"  
    "dormir"  
  ],  
  poids: 3.5,  
  miauler: function () {  
    alert("Miaou ! Miaou !");  
  },  
  maigrir : function(deltaPoids) {  
    this.poids -= deltaPoids;  
  }  
};
```



**function Chat(nom,age,race,poids) {**

**this.nom = nom ;**

**this.age = age ;**

**this.race = race ;**

**this.poids = poids ;**

*initialisation des  
propriétés de l'objet  
avec les valeurs des  
paramètres*

**this.miauler = function () {**

**alert("Miaou ! Miaou !");**

**};**

**this.maigrir = function(deltaPoids) {**

**this.poids -= deltaPoids;**

**};**

**}**

*définition des  
méthodes*

*les objets référencés par felicite  
et felix sont des instances de  
Chat.*

*un constructeur est invoqué par l'opérateur new*

```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
var felix = new Chat("Felix",6,"siamois",3);
```

# Objets JavaScript

En JavaScript les fonctions sont des objets !

```
function Chat(nom,age,race,poids) {  
  this.nom = nom ;  
  this.age = age ;  
  this.race = race ;  
  this.poids = poids ;  
  this.miauler = function () {  
    console.log(this.nom + "-> Miaou ! Miaou !");  
  };  
  this.maigrir = function(deltaPoids) {  
    this.poids -= deltaPoids;  
  };  
}
```

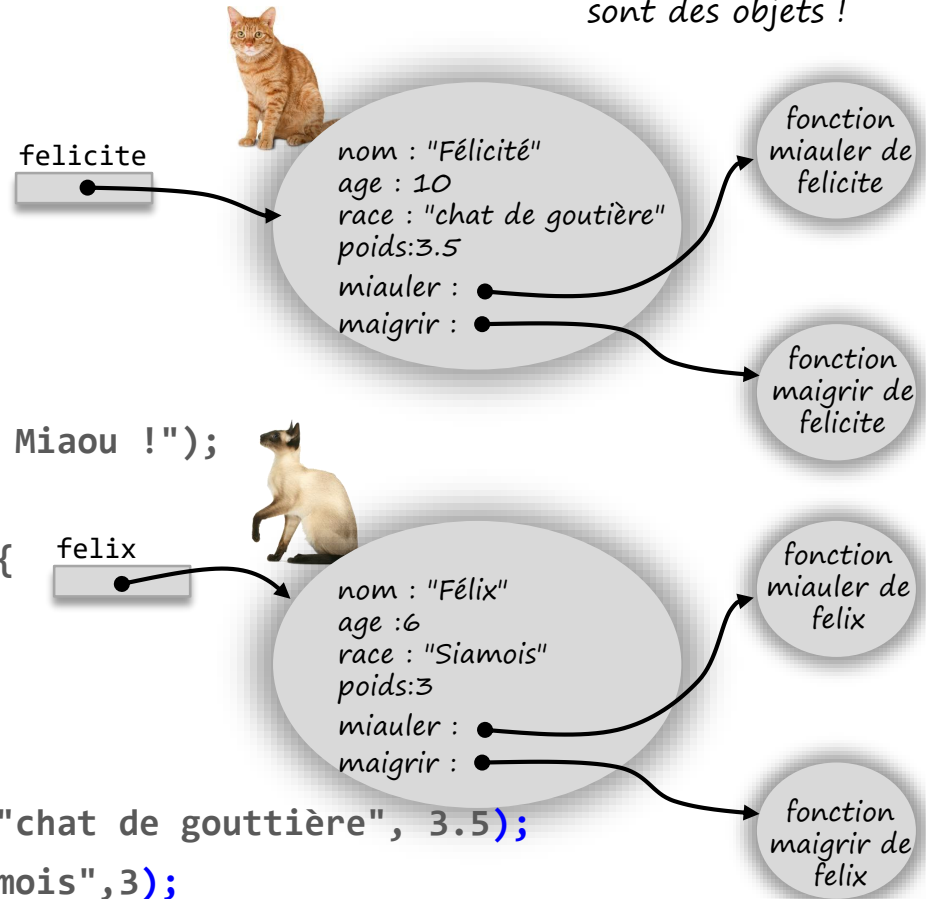
```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
var felix = new Chat("Felix",6,"siamois",3);
```

```
felicite.miauler(); ➡ Félicité -> Miaou ! Miaou !
```

```
felix.miauler(); ➡ Felix -> Miaou ! Miaou !
```

```
console.log("felicite.miauler === felix.miauler --> " +  
  (felicite.miauler === felix.miauler)); ➡ false
```



*felicite.miauler et felix.miauler référencent deux objets fonctions différents (même si ils font la même chose)*

# Objets JavaScript



- Utiliser prototype du constructeur

```
function Chat(nom,age,race,poids) {  
  this.nom = nom ;  
  this.age = age ;  
  this.race = race ;  
  this.poids = poids ;  
}
```

```
Chat.prototype.miauler = function () {  
  console.log(this.nom + "-> Miaou ! Miaou !");  
};  
Chat.prototype.magrir = function(deltaPoids) {  
  this.poids -= deltaPoids;  
};
```

```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
var felix = new Chat("Felix",6,"siamois",3);
```

```
felicite.miauler(); ➡ Felicité -> Miaou ! Miaou !
```

```
felix.miauler(); ➡ Felix -> Miaou ! Miaou !
```

```
console.log("felicite.miauler === felix.miauler --> " +  
  (felicite.miauler === felix.miauler)); ➡ true
```

- *prototype :*
  - liste de propriétés attachée à un constructeur
  - initialement vide
  - une propriété rajoutée sur le prototype du constructeur devient disponible sur toutes les instances de ce constructeur (*fallback*)

# Objets JavaScript

```
function Chat(nom,age,race,poids) {  
  this.nom = nom ;  
  this.age = age ;  
  this.race = race ;  
  this.poids = poids ;  
}
```

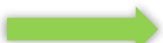
```
Chat.prototype.famille = "Felidés";
```

```
Chat.prototype.miauler = function () {  
  console.log(this.nom + "-> Miaou ! Miaou !");  
};
```

```
Chat.prototype.magrir = function(deltaPoids) {  
  this.poids -= deltaPoids;  
};
```

```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
var felix = new Chat("Felix",6,"siamois",3);
```

```
felicite.miauler();  Félicité -> Miaou ! Miaou !
```

```
console.log(felicite.famille);  Félidés
```

- A propos des prototypes

*les propriétés rajoutées sur le prototype d'un constructeur peuvent être de n'importe quel type*

# Objets JavaScript

```
function Chat(nom,age,race,poids) {  
  this.nom = nom ;  
  this.age = age ;  
  this.race = race ;  
  this.poids = poids ;  
}
```

```
Chat.prototype.miauler = function () {  
  console.log(this.nom + "-> Miaou ! Miaou !");  
};
```

```
Chat.prototype.maigrir = function(deltaPoids) {  
  this.poids -= deltaPoids;  
};
```

```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
var felix = new Chat("Felix",6,"siamois",3);
```

```
felix.miauler = function () {  
  console.log(this.nom + "-> Meaow ! Meaow !");  
};
```

*redéfinition de la  
méthode miauler*

```
felicite.miauler(); ➡ Félicité -> Miaou ! Miaou !
```

```
felix.miauler(); ➡ Felix -> Meaow ! Meaow !
```

- A propos des prototypes

*Une propriété du prototype peut  
être redéfinie sur une instance.*

*Dans ce cas la redéfinition ne  
concerne que l'instance*

# Objets JavaScript

```
function Chat(nom,age,race,poids) {  
  this.nom = nom ;  
  this.age = age ;  
  this.race = race ;  
  this.poids = poids ;  
}
```

```
Chat.prototype.miauler = function () {  
  console.log(this.nom + "-> Miaou ! Miaou !");  
};
```

```
Chat.prototype.maigrir = function(deltaPoids) {  
  this.poids -= deltaPoids;  
};
```

```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
felicite.miauler(); ➡ Félicité -> Miaou ! Miaou !
```

```
Chat.prototype.miauler = function () {  
  console.log(this.nom + "-> Meaow ! Meaow !");  
};
```

*modification de la méthode  
miauler du prototype du  
constructeur*

```
felicite.miauler(); ➡ Félicité -> Meaow ! Meaow !
```

```
var felix = new Chat("Felix",6,"siamois",3);
```

```
felix.miauler(); ➡ Felix -> Meaow ! Meaow !
```

- A propos des prototypes

*Une modification du prototype est immédiate pour les instances déjà existantes. Le fallback se fait à l'exécution (runtime) au moment de l'accès à la propriété.*

# Objets JavaScript

```
function Chat(nom,age,race,poids) {  
  this.nom = nom ;  
  this.age = age ;  
  this.race = race ;  
  this.poids = poids ;  
}
```

```
Chat.prototype.miauler = function () {  
  console.log(this.nom + "-> Miaou ! Miaou !");  
};
```

```
Chat.prototype.maigrir = function(deltaPoids) {  
  this.poids -= deltaPoids;  
};
```

```
var felicite = new Chat("Félicité",10,"chat de gouttière", 3.5);
```

```
felicite.miauler();  Félicité -> Miaou ! Miaou !
```

```
felicite.ronronner();  Erreur La méthode n'est pas définie
```

```
Chat.prototype.ronronner = function () {  
  console.log(this.nom + "-> Rrr... Rrr...");  
};
```

*ajout de la méthode  
ronronner au prototype du  
constructeur*

```
felicite.ronronner();  Félicité -> R... R...
```

- A propos des prototypes

*De la même manière qu'une  
modification, un ajout est  
immédiatement actif et s'applique à  
toutes les instances déjà créées*

# Objets JavaScript

```
function Point(x,y) {  
  this.x = x ;  
  this.y = y;  
}  
Point.prototype.afficher = function() {  
  document.write("x:" + this.x + " y:" + this.y + "<br>");  
}
```

```
var p1 = new Point(10,10);  
var p2 = new Point(p1.x + 10, p1.y + 10);
```

```
p1.afficher();  
p2.afficher();
```

```
x:10 y:10  
x:20 y:20
```

- pas d'encapsulation des données(on ne peut protéger ni les données, ni les méthodes)

- on peut même ajouter des attributs et méthodes

```
p1.x = "une chaîne";  
p1.afficher();  
p1.afficher = function() {  
  document.write("Hello Point");  
}  
p1.afficher();
```

```
x:une chaîne y:10  
Hello Point
```

```
p2.nom = "le point p2";  
p2.distanceOrigine = function() {  
  return Math.sqrt(this.x * this.x + this.y * this.y);  
}  
document.write("distance " + p2.distanceOrigine());
```

```
distance 28.284271247461902
```

- possibilité de simuler héritage avec prototypes (mais dépasse le cadre de ce cours)
  - <http://blog.xebia.fr/2013/06/10/javascript-retour-aux-bases-constructeur-prototype-et-heritage/>
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details\\_of\\_the\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model)



# "Classes" de base

Classe	Description
Array	représente un tableau sous forme d'objet
Boolean	représente le type primitif booléen sous forme d'objet
Date	représente une date sous forme d'objet
Error	classe d'erreur générique
EvalError	classe d'erreurs survenant lors de l'interprétation de code JavaScript par <code>eval</code>
Function	représente une fonction sous forme d'objet
Number	représente un nombre (entier, réel) sous forme d'objet
Object	classe de base de toutes les classes
RangeError	classe d'erreurs survenant lors de l'utilisation de nombres dépassant les bornes autorisées (MIN_VALUE, MAX_VALUE)
ReferenceError	classe d'erreurs survenant lors de l'utilisation d'une référence incorrecte
RegExp	représente une expression régulière
String	représente une chaîne de caractères
SyntaxError	classe relative aux erreurs de syntaxe
TypeError	classe relative aux erreurs de typage
URIError	classe relative aux erreurs d'utilisation des méthodes de traitement d'URI de Globals

# Chaînes de caractères

- type objet prédéfini **String**  
`var chaine = "essai" ;`  
`var chaine = 'essai' ;`  
`var chaine = new String('essai') ;`
- Nombre de caractères  
`chaine.length ;`
- + Concatenation  
`chaine = chaine + " une autre chaine";`
- Nombreuses méthodes prédéfinies

# Chaînes de caractères

- méthodes de manipulation de chaînes
  - `charAt(i)` retourne le ième caractère de la chaîne
  - `indexOf(ch,i)` index de la première occurrence de `ch` à partir de `i` (optionnel)
  - `lastIndexOf(ch, i)` index de la dernière occurrence de `ch` à partir de `i` (optionnel)
  - `split(ch)` transforme la chaîne en tableau
  - `match(exp)` recherche les sous-chaînes correspondant à l'expression régulière `exp`
  - `search(exp)` index de la première correspondance entre la chaîne et `exp`
  - `replace(exp, ch)` remplace les occurrences de `exp` par `ch`
  - `substr(d,l)` sous-chaîne de longueur `l` commençant en `d`
  - `substring(d,f)` sous-chaîne entre `d` et `f`
  - `toLowerCase()` convertit la chaîne en minuscules
  - `toUpperCase()` convertit la chaîne en majuscules

# Tableaux

- séquence d'éléments accessibles via un index
  - JavaScript étant faiblement typé, les éléments d'un même tableau peuvent être de types différents
- création d'un tableau

```
items = new Array(10);           // alloue espace pour 10 éléments
items = new Array();             // si pas de taille, ajustement dynamique
items = [];
```

`items = [0,1,2,3,4,'a','b','c','d'];` // définit la taille et les valeurs

- `[]` (comme en C++/Java) pour accéder aux éléments

```
for (i = 0; i < 10; i++) {
    items[i] = i;                // range la valeur de i à chaque index
}
```

- la propriété **length** donne le nombre d'éléments du tableau

```
for (i = 0; i < items.length; i++) {
    document.write(items[i] + "<br>"); // affiche les éléments
}
```

# Tableaux

- nombreuses méthodes de manipulation de tableaux
  - `concat(tab)` concatène le tableau à un autre
  - `join(sep)` transforme le tableau en chaîne de caractères
  - `pop()` dépile et retourne le dernier élément du tableau
  - `push(val, ..)` ajoute des éléments en fin de tableau
  - `shift()` supprime et retourne le premier élément du tableau
  - `unshift(val, ..)` ajoute des éléments en tête du tableau
  - `sort()` trie le tableau
  - `reverse()` inverse l'ordre des éléments du tableau
  - `splice(deb, nb, val...)` insère, supprime ou remplace éléments du tableau
  - `toString()` transforme le tableau en chaîne (séparateur =',')

# Tableaux

- exemples  
[tableaux1.html](#)

```
tab[0] = Premier élément  
tab[1] = undefined  
tab[2] = undefined  
tab[3] = Dernier élément
```

```
tab[0] = un  
tab[1] = DEUX  
tab[2] = TROIS  
tab[3] = TROIS ET DEMI  
tab[4] = quatre
```

```
tabRes[0] = deux  
tabRes[1] = trois
```

```
tab[0] = un  
tab[1] = TROIS ET DEMI  
tab[2] = quatre
```

```
tabRes[0] = DEUX  
tabRes[1] = TROIS
```

```
un;TROIS ET DEMI;quatre
```

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Tableaux JS</title>  
    <meta charset="UTF-8">  
    <script>  
      function afficherTableau(tabName,t) {  
        for (var i = 0; i < t.length; i++) {  
          document.write(tabName+ "[" + i + "] = " + t[i] + "<br/>");  
        }  
        document.write("<br/>");  
      }  
    </script>  
  </head>  
  <body>  
    <h1>Tableaux</h1>  
    <script>  
      var tab = [];  
      tab[0] = "Premier élément";  
      tab[3] = "Dernier élément";  
      afficherTableau("tab",tab);  
  
      tab = ["un", "deux", "trois", "quatre"];  
      var tabRes = tab.splice(1, 2, "DEUX", "TROIS",  
                             "TROIS ET DEMI");  
  
      afficherTableau("tab",tab);  
      afficherTableau("tabRes",tabRes);  
  
      tabRes = tab.splice(1, 2);  
      afficherTableau("tab",tab);  
      afficherTableau("tabRes",tabRes);  
  
      document.write(tab.join(";"));  
    </script>  
  </body>  
</html>
```

# Tableaux associatifs

tableaux avec des noms comme indexes (supporté dans de nombreux langages: PHP...)

- permet d'associer clés/valeurs

tabAssoc1  
cle1 -->valeur1  
clé2 -->valeur2  
clé3 -->valeur3

tabAssoc2  
cle1 -->valeur1  
clé2 -->valeur2  
clé3 -->valeur3

tabAssoc2 après modification  
cle1 -->valeur1  
clé2 -->valeur2bis  
clé3 -->valeur3

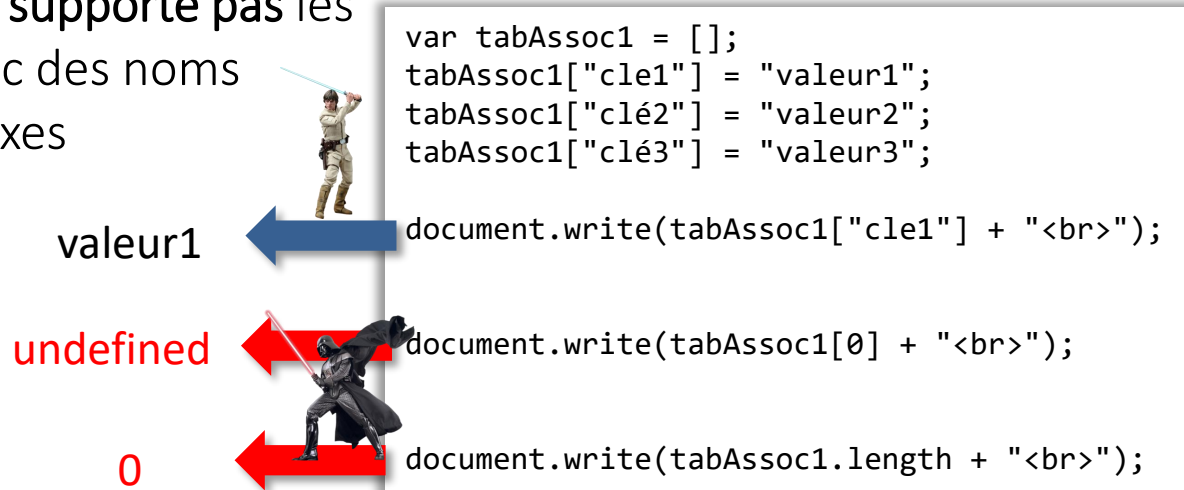


```
var tabAssoc1 = [];  
tabAssoc1["cle1"] = "valeur1";  
tabAssoc1["clé2"] = "valeur2";  
tabAssoc1["clé3"] = "valeur3";  
  
document.write("tabAssoc1<br>");  
for (var cle in tabAssoc1) {  
    document.write(cle + " -->" + tabAssoc1[cle] + "<br>");  
}  
  
var tabAssoc2 = {  
    "cle1" : "valeur1",  
    "clé2" : "valeur2"  
};  
tabAssoc2["clé3"] = "valeur3";  
  
document.write("tabAssoc2<br>");  
for (var cle in tabAssoc2) {  
    document.write(cle + " -->" + tabAssoc2[cle] + "<br>");  
}  
  
tabAssoc2["clé2"] = "valeur2bis";  
document.write("tabAssoc2 après modification<br>");  
for (var cle in tabAssoc2) {  
    document.write(cle + " -->" + tabAssoc2[cle] + "<br>");  
}
```

# Tableaux associatifs

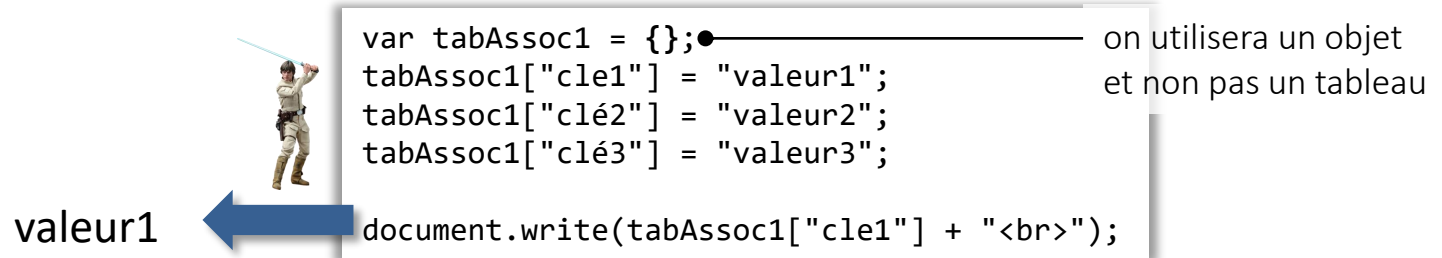
tableaux avec des noms comme indexes (supporté dans de nombreux langages: PHP...)

- JavaScript **ne supporte pas** les tableaux avec des noms comme indexes



```
var tabAssoc1 = [];  
tabAssoc1["cle1"] = "valeur1";  
tabAssoc1["clé2"] = "valeur2";  
tabAssoc1["clé3"] = "valeur3";  
  
document.write(tabAssoc1["cle1"] + "<br>");  
  
document.write(tabAssoc1[0] + "<br>");  
  
document.write(tabAssoc1.length + "<br>");
```

- En JavaScript, les **objets** utilisent des noms comme indexes
- Les tableaux sont des cas particuliers d'objets avec des indexes entiers



```
var tabAssoc1 = {};  
tabAssoc1["cle1"] = "valeur1";  
tabAssoc1["clé2"] = "valeur2";  
tabAssoc1["clé3"] = "valeur3";  
  
document.write(tabAssoc1["cle1"] + "<br>");
```

on utilisera un objet et non pas un tableau

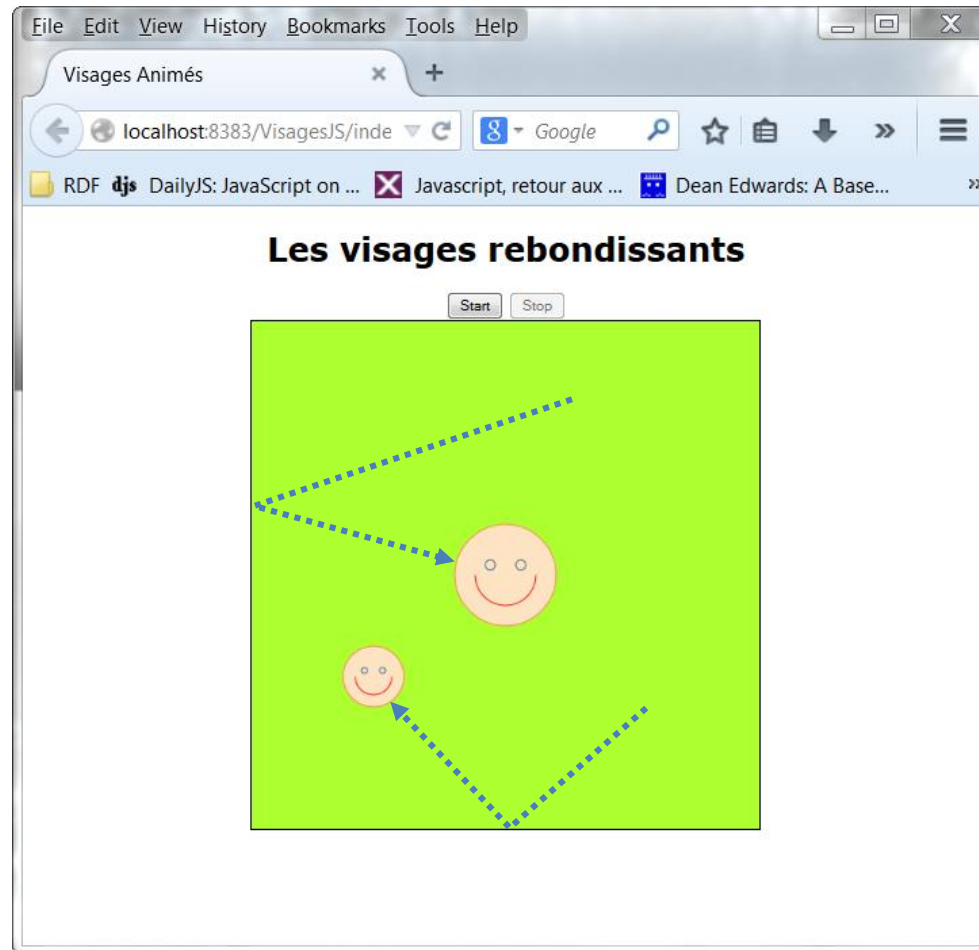


<http://andrewdupont.net/2006/05/18/javascript-associative-arrays-considered-harmful/>



# Objets: exemple

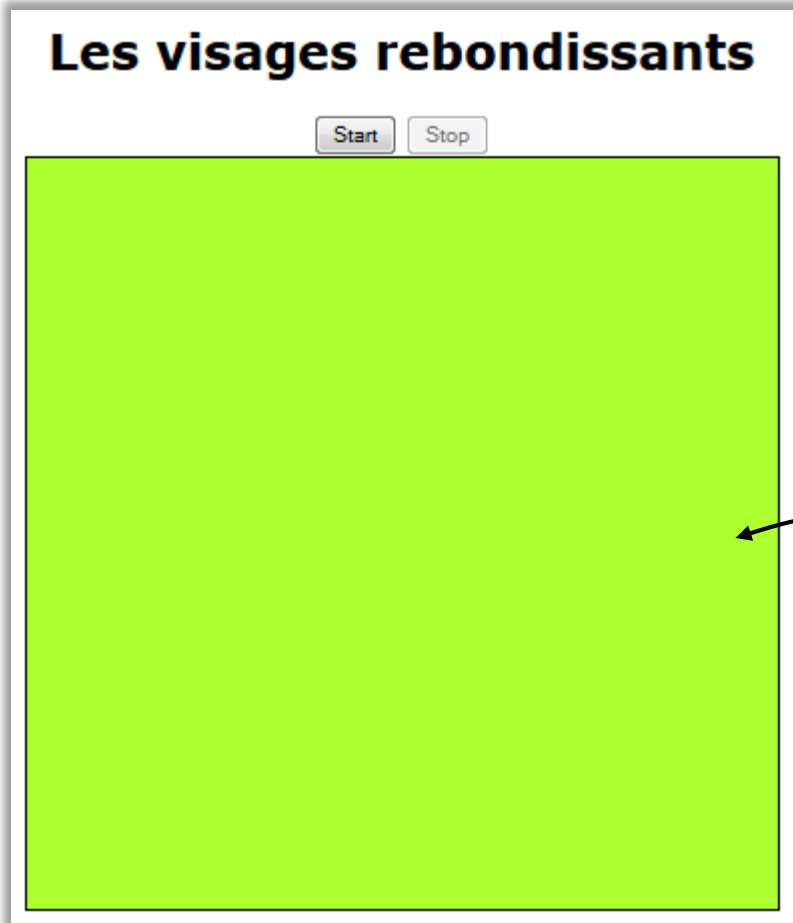
## Les Visages Rebondissants



<http://lig-membres.imag.fr/genoud/teaching/applisweb/tds/sujets/VisagesJS/>

# Objets: exemple

## Les Visages Rebondissants



```
<body onload="init()">
  <div id="wrapper">
    <h1>Les visages rebondissants</h1>
    <!-- le boutons pour lancer et interrompre l'animation -->
    <div>
      <button id="startBtn">Start</button>
      <button id="stopBtn" disabled>Stop</button>
    </div>
    <!-- le canvas pour dessiner -->
    <canvas id="myCanvas" width="500" height ="500">
      la zone de dessin
    </canvas>
  </div>
</body>
```

**canvas** nouvel élément HTML5 qui définit une zone dessin dans laquelle il est possible de dessiner grâce à l'API Canvas (API javascript de bas niveau qui permet de dessiner dans un élément canvas)

[http://www.w3schools.com/tags/ref\\_canvas.asp](http://www.w3schools.com/tags/ref_canvas.asp)

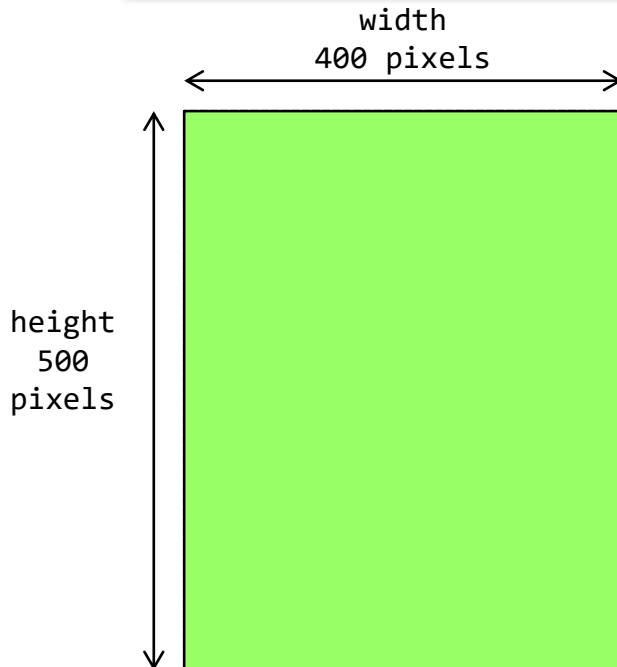
<http://www.alsacreations.com/tuto/lire/1484-introduction.html>

<http://openclassrooms.com/courses/la-balise-canvas-avec-javascript>

# Objets: exemple

## Les Visages Rebondissants

```
<canvas id="myCanvas" width="400" height="500">  
  la zone de dessin, si ce message s'affiche c'est que  
  votre navigateur ne supporte pas la balise canvas de HTML5...  
  il serait temps de changer de version !  
</canvas>
```



par défaut le canvas est une zone blanche.  
Pour le matérialiser ajoutons lui une bordure  
et une couleur de fond

```
canvas {  
  border:1px solid;  
  background-color: greenyellow;  
}
```

ensuite tout se passe du côté JavaScript

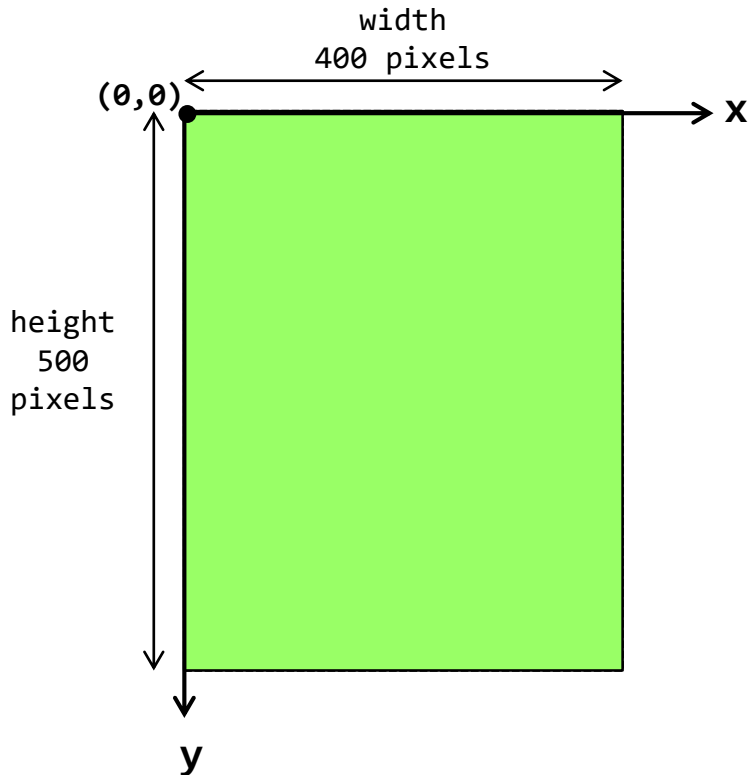
*recupère l'élément correspondant au canvas*

```
var canvas = document.getElementById("myCanvas");  
var ctxt = canvas.getContext("2d");
```

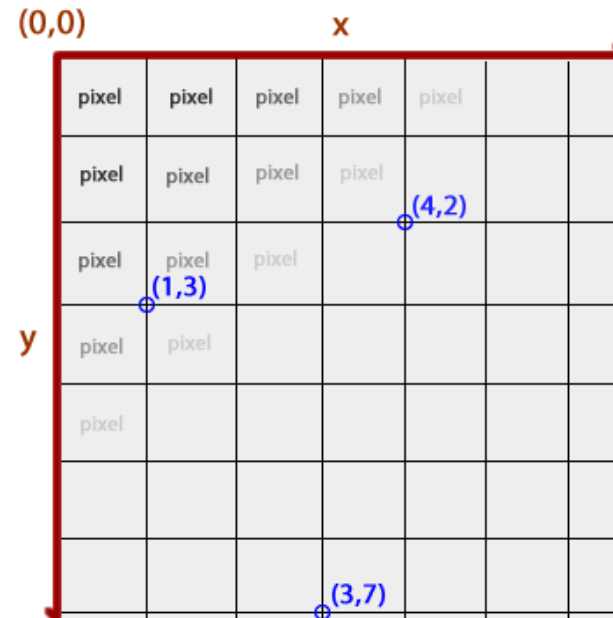
*recupère le contexte de dessin. Ce sont les méthodes de cet objet que l'on utilisera pour dessiner sur le canvas*

## Objets: exemple

- Au canvas est associé un système de coordonnées 2D:
  - l'origine(0,0) est située en haut à gauche
  - L'axe horizontal (x) est défini par la première coordonnée
  - L'axe vertical (y) est défini par la seconde coordonnée



- Les valeurs des coordonnées correspondent à la grille entourant les pixels, et non pas aux pixels eux-mêmes

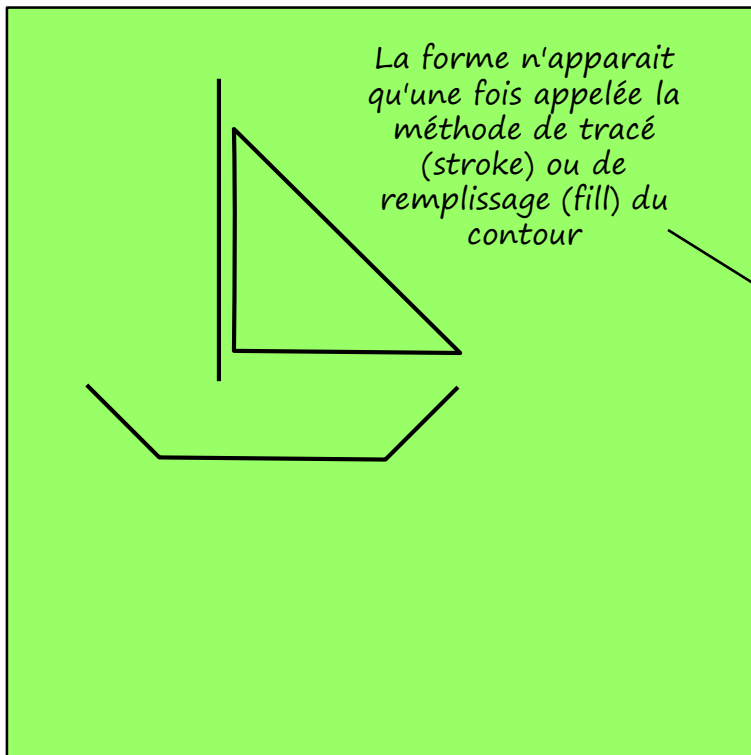


<http://www.alsacreations.com/tuto/lire/1484-introduction.html>

# Objets: exemple

## Les Visages Rebondissants

- dessiner des formes simples dans un canvas (tracés et chemins)
- étapes: initialisation, point de départ puis point d'arrivée, clôture, affichage du contour et/ou du remplissage



```
var c = document.getElementById('myCanvas');
var ctx = c.getContext("2d");

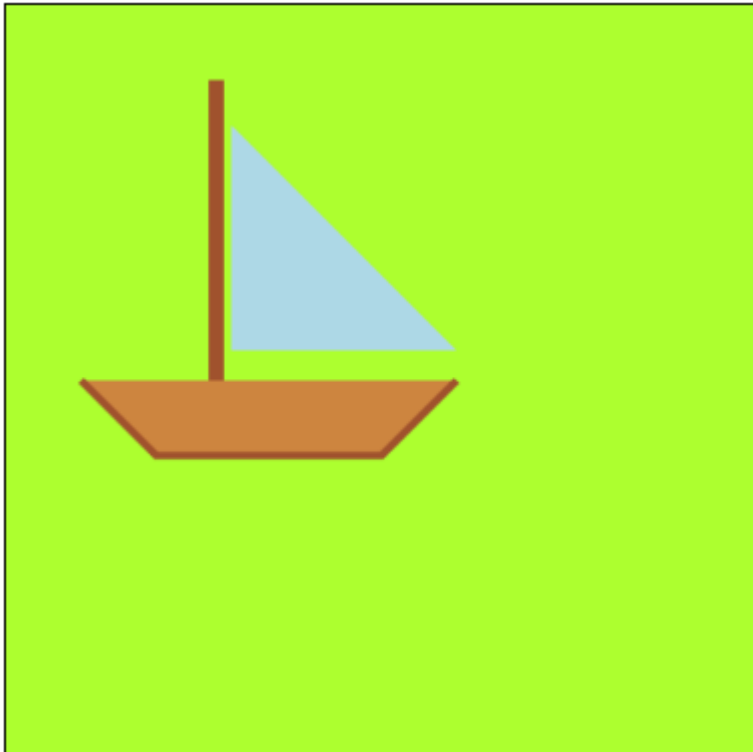
// Voile du bateau
ctx.beginPath();           // Début du chemin
ctx.moveTo(150,80);        // Le tracé part du point 150,80
ctx.lineTo(300,230);       // Un segment est ajouté vers 300,230
ctx.lineTo(150,230);       // Un segment est ajouté vers 150,230
ctx.closePath();           // Fermeture du chemin
ctx.stroke();               // dessine le contour

// Coque du bateau
ctx.beginPath();           // Début d'un autre chemin
ctx.moveTo(50,250);
ctx.lineTo(100,300);
ctx.lineTo(250,300);
ctx.lineTo(300,250);
ctx.stroke();               // dessine le contour

// Mât
ctx.beginPath();
ctx.moveTo(140,50);
ctx.lineTo(140,250);
ctx.stroke();
```

<http://www.alsacreations.com/tuto/lire/1484-introduction.html>

- possibilité de modifier les styles des couleurs de contour et de remplissage dont dépendent `fill()` et `stroke()` en agissant sur les propriétés `fillStyle` et `strokeStyle` du contexte de dessin.



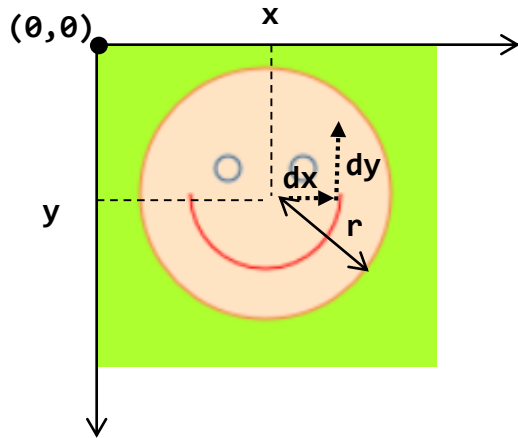
```
// Voile du bateau
ctx.beginPath();           // Début du chemin
ctx.moveTo(150,80);        // Le tracé part du point 150,80
ctx.lineTo(300,230);       // Un segment est ajouté vers 300,230
ctx.lineTo(150,230);       // Un segment est ajouté vers 150,230
ctx.closePath();           // Fermeture du chemin
ctx.fillStyle = "lightblue"; // la couleur de remplissage
ctx.fill();                // Remplissage du dernier chemin tracé

// Coque du bateau
ctx.beginPath();           // Début d'un autre chemin
ctx.moveTo(50,250);
ctx.lineTo(100,300);
ctx.lineTo(250,300);
ctx.lineTo(300,250);
ctx.fillStyle = "peru";
ctx.strokeStyle = "sienna"; // Définition de la couleur de contour
ctx.lineWidth = 5;          // Définition de la largeur de ligne
ctx.fill();                 // Application du remplissage
ctx.stroke();               // Application du contour

// Mât
ctx.beginPath();
ctx.moveTo(140,50);
ctx.lineTo(140,250);
ctx.lineWidth = 10;
ctx.stroke();
```

<http://www.alsacreations.com/tuto/lire/1484-introduction.html>

## Objets: exemple

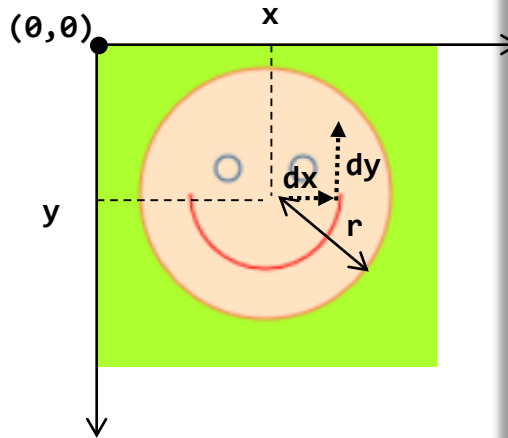


*Constructeur*

- Un visage est un objet
- Attributs :
  - $x, y$  : coordonnées de son centre
  - $r$  : rayon
  - $dx$  : déplacement élémentaire horizontal
  - $dy$  : déplacement élémentaire vertical
  - **canvas**: la référence de l'objet canvas dans lequel il se déplace

```
function Visage(canvas, x, y, r, dx, dy) {  
    this.canvas = canvas;  
    this.x = x;  
    this.y = y;  
    this.r = r;  
    this.dx = dx;  
    this.dy = dy;  
}
```

## Objets: exemple



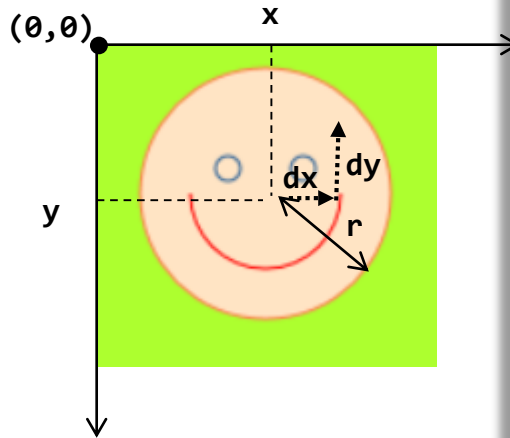
Constructeur

- Un visage est un objet
- Méthodes:
  - **dessiner** :  
affiche le visage  
dans le canvas

```
Visage.prototype.dessiner = function() {  
    var ctx = this.canvas.getContext("2d");  
    var yYeux = this.y - this.r * 0.20;  
    var dxYeux = this.r * 0.3;  
    // le cercle délimitant le Visage  
    ctx.beginPath();  
    ctx.arc(this.x, this.y, this.r, 0, Math.PI * 2, true);  
    ctx.strokeStyle = "coral";  
    ctx.fillStyle = "bisque";  
    ctx.fill();  
    ctx.stroke();  
  
    // la bouche  
    ctx.beginPath();  
    ctx.arc(this.x, this.y, this.r * 0.6, 0, Math.PI, false);  
    ctx.strokeStyle = "red";  
    ctx.stroke();  
  
    // les yeux  
    ctx.beginPath();  
    ctx.strokeStyle = "#369";  
    ctx.fillStyle = "#c00";  
    ctx.arc(this.x + dxYeux, yYeux, this.r * 0.1, 0, Math.PI * 2, false);  
    ctx.stroke();  
    ctx.beginPath();  
    ctx.arc(this.x - dxYeux, yYeux, this.r * 0.1, 0, Math.PI * 2, false);  
    ctx.stroke();  
};
```



## Objets: exemple

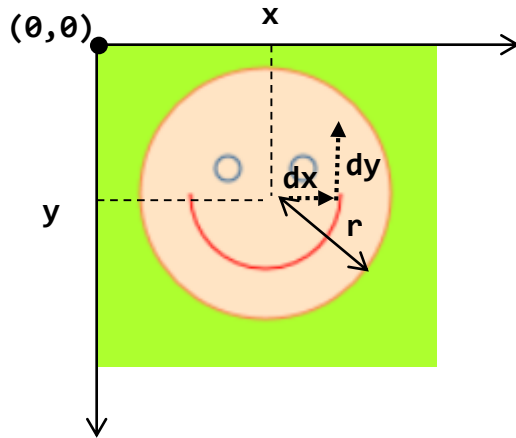


```
Visage.prototype.deplacer = function() {  
  this.x += this.dx;  
  this.y += this.dy;  
  if (this.x < this.r || this.x > (this.canvas.width - this.r)) {  
    // le visage touche le bord gauche ou le bord droit  
    // au prochain déplacement il devra changer de direction  
    horizontale  
    this.dx = -this.dx;  
  }  
  if (this.y < this.r || this.y > (this.canvas.height - this.r)) {  
    // le visage touche le bord haut ou le bord bas  
    // au prochain déplacement il devra changer de direction verticale  
    this.dy = -this.dy;  
  }  
};
```

- Un visage est un objet
- Méthodes:
  - **deplacer** : fait effectuer au visage un déplacement élémentaire et quand il atteint l'un des bords d'une canvas, il rebondit (change sa direction de déplacement)

# Objets: exemple

## Les Visages Rebondissants



Visage.js

- Au chargement de la page :
  - création de deux visages et affichage de ceux-ci

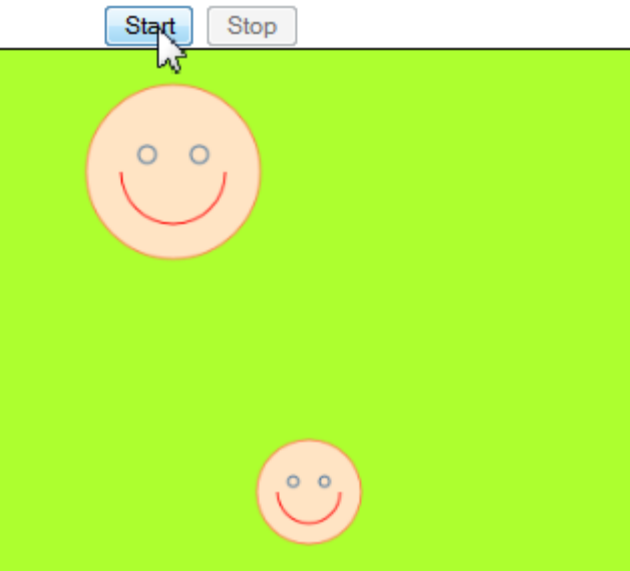
```
function init() {  
  
    var canvas = document.getElementById("myCanvas");  
    var ctx = canvas.getContext("2d");  
  
    // cree les deux visages et les affiche  
    var visage1 = new Visage(canvas, 250, 250, 50, 4, 2);  
    var visage2 = new Visage(canvas, 120, 350, 30, -3, -2);  
    visage1.dessiner();  
    visage2.dessiner();  
  
}
```

```
<html>  
  <head>  
    ...  
    <script src="scripts/Visage.js"></script>  
  </head>  
  <body onload="init()">  
    <div id="wrapper">  
      ...  
    </div>  
  </body>
```

*appelle la fonction init une fois que le document a été chargé*

# Objets: exemple

## Les Visages Rebondissants



- lorsque l'utilisateur clique sur le bouton **Start** lancer la boucle d'animation

associe un traitement à un événement **click** sur l'élément HTML **startBtn** (le bouton Start) défini ici par une fonction anonyme

crée un timer qui effectuera une action (définie par une fonction) à intervalles de temps réguliers

```
function init() {                                     Visage.js

    var timerId = 0;
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");

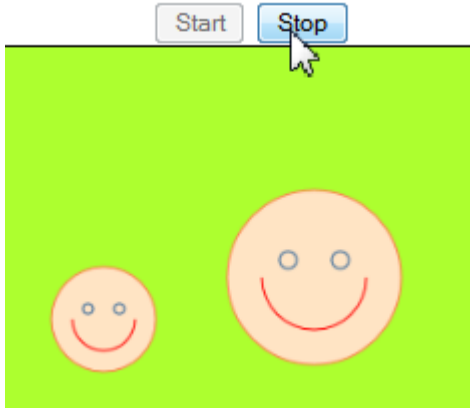
    // cree les deux visages et les affiche
    var visage1 = new Visage(canvas, 250, 250, 50, 4, 2);
    var visage2 = new Visage(canvas, 120, 350, 30, -3, -2);
    visage1.dessiner();
    visage2.dessiner();

    document.getElementById("startBtn").onclick = function() {
        document.getElementById("stopBtn").disabled = false;
        document.getElementById("startBtn").disabled = true;
        timerId = setInterval(function() {
            ctx.clearRect(0, 0, canvas.width, canvas.height);
            visage1.deplacer();
            visage1.dessiner();
            visage2.deplacer();
            visage2.dessiner();
        }, 16);
    };
}
```

# Objets: exemple

## Les Visages Rebondissants

- lorsque l'utilisateur clique sur le bouton **Stop** l'animation est interrompue



associe un traitement à un événement **click** sur l'élément HTML **stopBtn** (le bouton Start) défini ici par une fonction anonyme

interruption du timer

```
function init() {                                     Visage.js

    var timerId = 0;
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");

    // cree les deux visages et les affiche
    var visage1 = new Visage(canvas, 250, 250, 50, 4, 2);
    var visage2 = new Visage(canvas, 120, 350, 30, -3, -2);
    visage1.dessiner();
    visage2.dessiner();

    document.getElementById("startBtn").onclick = function() {
        ...
    }

    • document.getElementById("stopBtn").onclick = function() {
        document.getElementById("stopBtn").disabled = true;
        document.getElementById("startBtn").disabled = false;
        • clearInterval(timerId);
    };
}
```

# Expressions régulières

- notation compacte et puissante qui décrit de manière concise un ensemble de chaînes de caractères
- en JavaScript support avec syntaxe empruntée à Perl
  - déclaration `var reg = /exp/options ;`
    - options
      - `i` pour ignorer les majuscules,
      - `g` pour rechercher toutes les correspondances.
  - opérateur `[]` définition de classe de caractères
    - `[aeiou]` # Classe des voyelles
    - `[0-9]` # Classe des chiffres  
#(équivalent à `[0123456789]` et `\d`)
    - `[0-9a-zA-Z]` # Chiffre ou lettre
    - `[\~\@,;\^_]` # Classe de caractères spéciaux

# Expressions régulières

- syntaxe en JavaScript (suite)
  - opérateur `[^...]` négation de classe ou de caractères
    - `[^0-9]` # Caractères hors chiffres (équivalent à `[^\d]` ou `\D`)
  - Caractères spéciaux ou classes de caractères
    - `\n` # Nouvelle ligne
    - `\r` # Retour chariot
    - `\t` # Tabulation
    - `\d` # Caractère numérique (équivalent à `[0-9]`)
    - `\D` # Caractère non numérique (équivalent à `[^0-9]`)
    - `\w` # Caractère de mot (alphanumérique) (équivalent à `[0-9a-zA-Z]`)
    - `\W` # Caractère de non mot (équivalent à `[^\w]`)
    - `\s` # Espace (équivalent à `[\t\n\r\f]`)
    - `\S` # Non espace (équivalent à `[^\s]`)

# Expressions régulières

- syntaxe en JavaScript (suite)
  - Méta-caractères
    - ^ Début de ligne
    - \$ Fin de ligne
    - \ Supprime l'interprétation des méta-caractères
    - . Tous les caractères sauf \n
    - | Alternative. Exemple : (0|1|2|3|4|5|6|7|8|9)
    - () Regroupement. Exemple : Frs|(f|francs)
    - [] Spécification d'une classe de caractères. Exemple : [a-A]

# Expressions régulières

- syntaxe en JavaScript (suite)

- Quantifieurs

**c{n}**    n instances du caractère c

**c{n,}** au moins **n** instances du caractère **c**

**c{n,m}** au moins n et au plus m instances du caractère c

**C+** une ou plusieurs instances du caractère `c` équivaut à `c{1,}`

**c\*** zéro ou plusieurs instances du caractère **c** équivaut à **c{0,}**

**c?** zéro ou une instance du caractère **c** équivaut à  $c\{0,1\}$

- Examples

Mot [a-zA-Z]+

Mot de 2 lettres [a-zA-Z]{2}

Nombre entier [0-9]<sub>+</sub>

Nombre réel  $([0-9]+\backslash.[0-9]^*) \mid (\backslash.[0-9]+)$

Variable C/C++ `[_a-zA-Z][_a-zA-Z0-9]*`



# Expressions régulières

- Expression régulière = objet.
- Méthodes :
  - **reg.test(chaine)** renvoie **true** si la chaîne contient l'expression régulière.
  - **reg.exec(chaine)** recherche l'expression régulière dans la chaîne et renvoie la chaîne trouvée.
- Exemple

```
var chaine = "hello world" ;  
var reg = /\w+/ ;  
var reg1 = /\w+/g ;  
var res = reg.test(chaine) ; // res == true  
var mot ;  
mot = reg.exec(chaine) ; // mot == hello  
mot = reg.exec(chaine) ; // mot == hello  
mot = reg1.exec(chaine) ; // mot == hello  
mot = reg1.exec(chaine) ; // mot == world  
mot = reg1.exec(chaine) ; // mot == null  
mot = reg1.exec(chaine) ; // mot == hello
```

# Expressions régulières

- Utilisation dans des méthodes de la classe **String** :
  - `replace`, `search`, `match`
- Exemple :

```
var ch  = "00 12 34".replace(/\d{2}/, "xx") ;  
// ch  == "xx 12 34"
```

```
var res = "00 12 34".match(/\d{2}/g) ;  
// res == [00,12,34]
```

- voir "La gestion des événements en JavaScript" de Juilien Royer  
<http://www.alsacreations.com/article/lire/578-La-gestion-des-evenements-en-JavaScript.html>

- Les interactions de l'utilisateur avec la page peuvent provoquer l'exécution de code JavaScript
  - clic sur un bouton ou sur un lien
  - survol de la souris au dessus d'un élément HTML
  - modification d'une zone de saisie
  - appui sur une touche de clavier
  - ...
- Cette interaction est gérée via des **événements**
  - Événement = tout changement d'état du navigateur
- Production d'un événement
  - Déclenchée par l'utilisateur ou par le code JavaScript

## Principaux événements javascript

nom	événement	s'applique à
abort	interruption de chargement	image
blur	perte du focus	champs de saisie, window
focus	attribution du focus	champs de saisie, window
click	clic sur un objet	button, checkbox, radio, reset, submit, lien
change	changement de champ de saisie	fileupload, password, select, text, textarea
dblClick	double clic lien	image, bouton
keyDown	touche enfoncée	document, image, lien, text
keyPress	l'utilisateur a appuyé sur une touche	document, image, lien, text
keyUp	touche relâchée	document, image, lien, text
load	chargement	image, window
mouseDown	un bouton de la souris est enfoncé	document, lien, image, button
mouseUp	le bouton de la souris est relâché	document, lien, image, button
mousemove	déplacement de la souris	document, lien, image, button
mouseout	la souris vient de quitter une zone	lien, image, layer
mouseover	la souris entre sur une zone	lien, image, layer
reset	annulation des données saisies	formulaire
submit	soumission du formulaire	formulaire
select	sélection d'un champ de texte	champs de saisie

- JavaScript : langage événementiel
    - flux d'exécution du code, déterminé principalement par les interactions avec l'environnement (activation d'un lien, mouvement de la souris, chargement du contenu du document, ...)
    - le développeur a un contrôle limité sur celui-ci
    - Gestion des événements conforme aux spécifications du W3C DOM Level 2 Events Specification  
<http://www.w3.org/TR/DOM-Level-2-Events/> [nov. 2000]\*
      - adopté à partir de version 9 de IE (Internet Explorer)
      - IE < 8, interface spécifique avec un sous-ensemble des fonctionnalités DOM niveau 2 (plus supportée depuis IE-11)
      - utilisation encore très courante de gestionnaires d'événements DOM-0, standard de facto hérité de Netscape
- ...ing draft pour DOM level 3 [sept. 2014] <http://www.w3.org/TR/DOM-Level-3-Events/>



- principes généraux



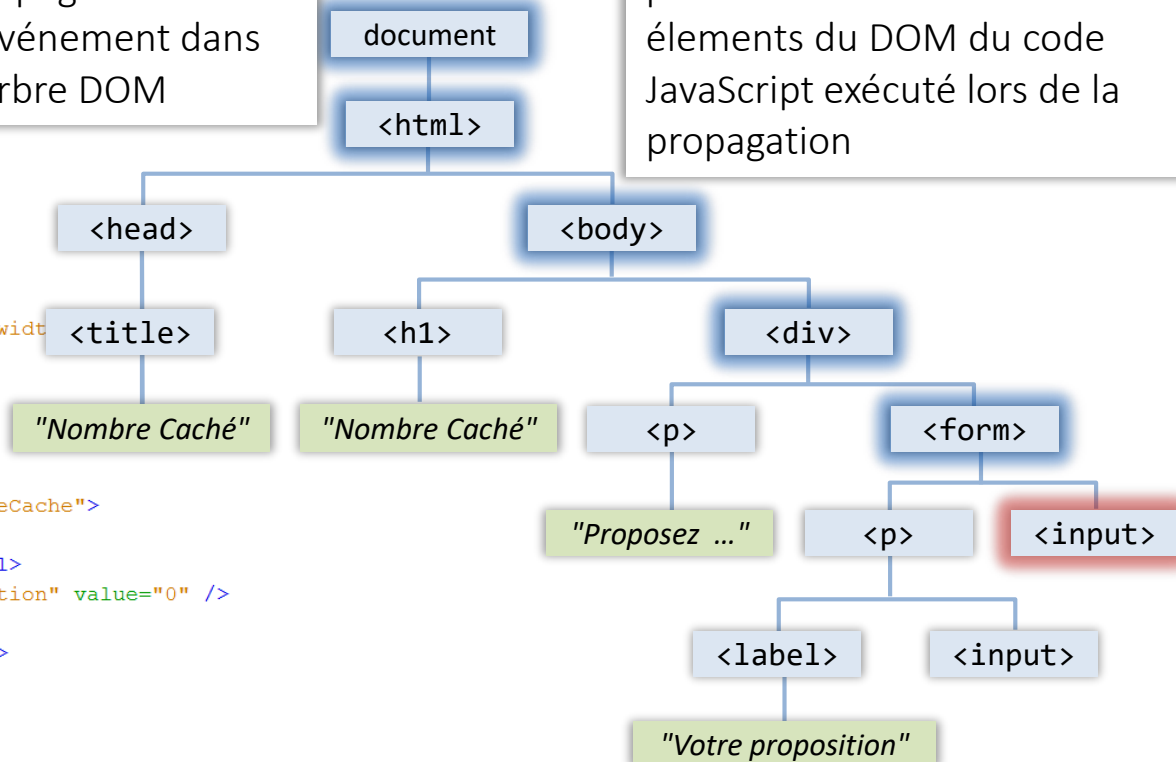
```
<html>
<head>
  <title>Nombre Caché</title>
  <meta charset="windows-1252">
  <meta name="viewport" content="width=device-width">
</head>
<body>
  <h1>Nombre caché</h1>
  <div>
    <p>Proposez un nombre entre 0 et 100</p>
    <form name="formulaireJeu" action="nombreCache">
      <p>
        <label>Votre proposition :</label>
        <input type="text" name="proposition" value="0" />
      </p>
      <input type="submit" value="Jouer" />
    </form>
  </div>
</body>
</html>
```

event

① Création d'un objet Event décrivant l'événement

② Propagation de l'événement dans l'arbre DOM

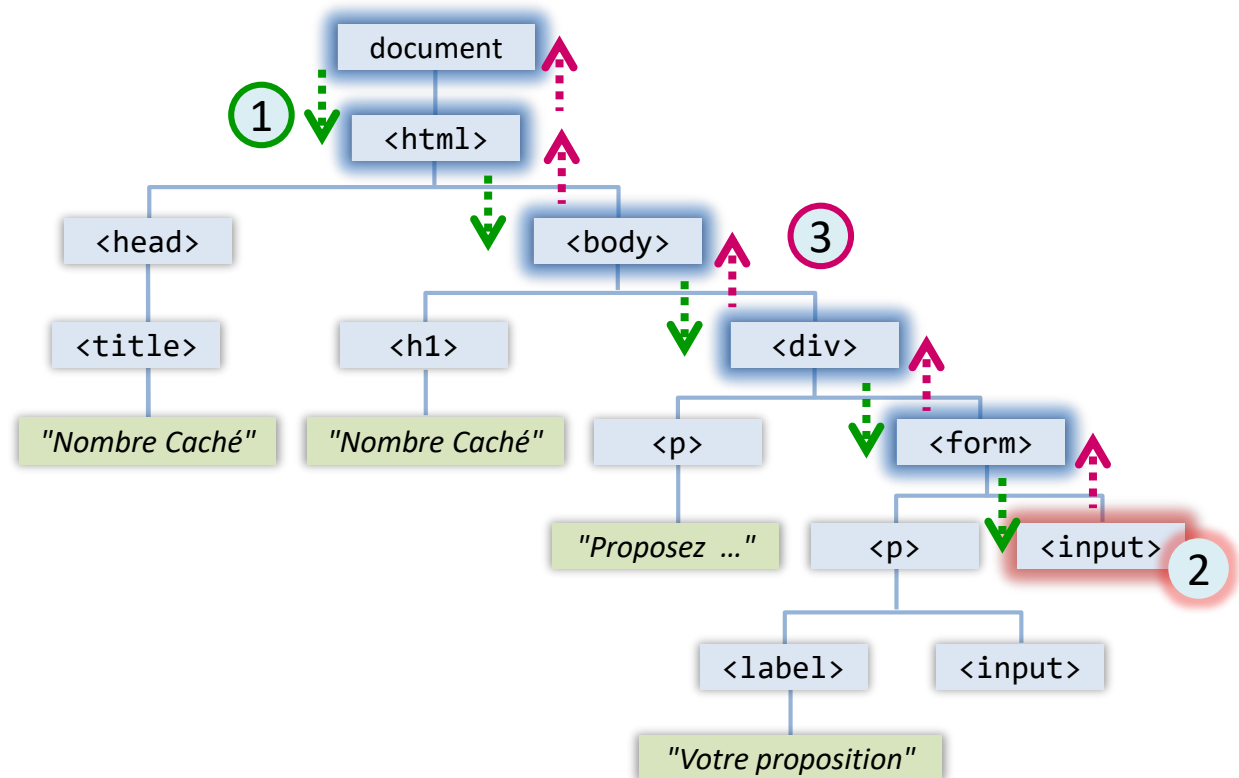
possibilité d'associer aux éléments du DOM du code JavaScript exécuté lors de la propagation



- attributs
  - données spécifiques au type d'événement (ex touche pressée lors d'un événement `keydown`)
  - données communes à tous les types d'événements
    - **target** : noeud de l'arbre DOM cible de l'événement (par exemple l'élément le plus profond de l'arbre au-dessus duquel se trouve la souris pour un événement de souris)
    - **type** : le type de l'événement ("`load`", "`focus`", "`click`" ...)
    - **currentTarget** : noeud sur lequel l'événement se trouve actuellement lors de la propagation dans l'arbre DOM
- méthodes
  - **stopPropagation()** : permet d'arrêter la propagation de l'événement dans l'arbre DOM
  - **preventDefault()** : Pour les types d'événements qui l'autorisent, permet d'annuler l'action implicite correspondante (exemple envoi d'un formulaire après un `submit`)



- Les événements se propagent selon un flux bien précis qui se décompose en 3 phases
  - 1 phase 1 (capture) : l'événement se propage du noeud document (inclus) au noeud cible (target) exclu
  - 2 phase 2 (cible) : l'événement atteint le noeud cible
  - 3 phase 3 (bouillonnement) : événement se propage du noeud cible au noeud document inclus



- **gestionnaire d'événements** : fonction JavaScript attachée à un noeud et qui automatiquement appelée lorsque le noeud est atteint lors de la propagation de l'événement
- modèle d'événements DOM niveau 2 définit deux méthodes qui permettent d'attacher ou de détacher un gestionnaire d'événement d'un nœud
  - **addEventListener(type, listener, useCapture)**
    - **type** (string) le type d'événement
    - **listener** (function) la fonction gestionnaire d'événements
    - **useCapture** (boolean) si **true** le gestionnaire sera appelé lors de la phase de capture (1), si **false** le gestionnaire sera appelé lors de la phase cible (2) ou de bouillonnement (3)
  - **removeEventListener(type, listener, useCapture)**

- gestionnaire d'événements : fonction

- Standard de facto hérité de Netscape, encore très utilisé
- Ne supporte que la phase de bouillonnement et la cible
- Pour ajouter un gestionnaire d'événement à un élément, il suffit de définir une fonction comme propriété de l'objet JavaScript correspondant
  - Ne permet pas d'ajouter plusieurs gestionnaires d'événements de même type à un nœud de l'arbre DOM

# Associer une action à un événement

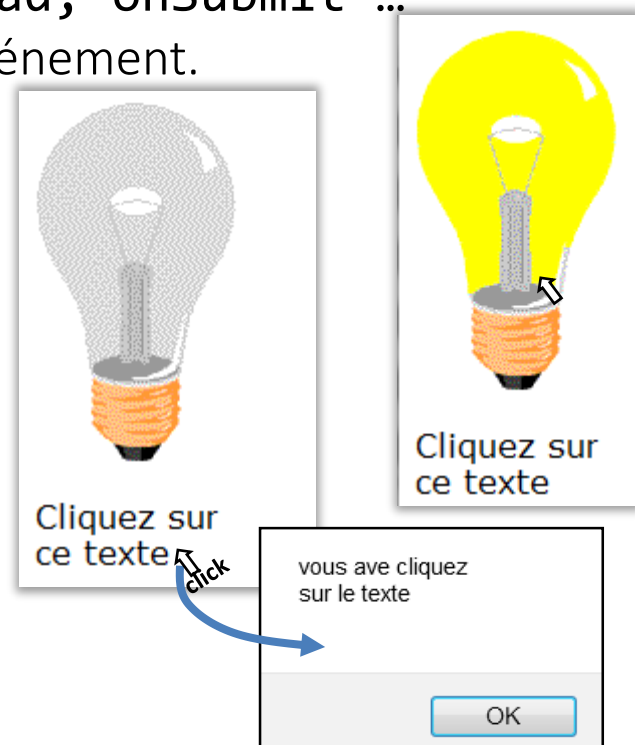
- possibilité d'associer un code JavaScript à un type d'événement sur un élément HTML de la page

`<nomElément attributi="propriétéi" événementj = "actionj">`

**événement<sub>j</sub>** : nom de l'événement précédé de **on** : `onBlur`, `onChange`, `onClick`, `onFocus`, `onLoad`, `onSubmit` ...

**action<sub>j</sub>** : code JavaScript exécuté lors de l'événement.

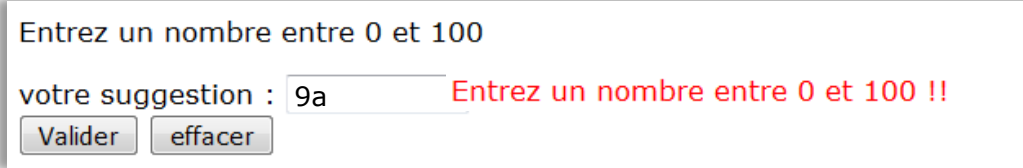
```
<html>
  <head>
    <title>Evénements</title>
    <meta charset=UTF-8">
  </head>
  <body>
    <h1>Evénements</h1>
    
    <p onclick="alert('vous ave cliquez\nsur le texte')">
      Cliquez sur<br/>
      ce texte
    </p>
  </body>
</html>
```



# Expressions régulières

- exemple : validation d'un formulaire

```
<html>
<head>
  <title>Expression régulières</title>
  <meta charset="UTF-8">
  <script>
    function validFormulaire() {
      var valeurSaisie = document.forms["form1"]["val"].value;
      var reg = /^[0-9]|[1-9][0-9]|100$/;
      if (!! reg.test(valeurSaisie.toString())){
        document.getElementById("messageErreur").innerHTML = "Entrez un nombre entre 0 et 100 !!";
        return false;
      }
      document.getElementById("messageErreur").innerHTML = "";
      return true;
    }
  </script>
</head>
<body>
  <h1>Exemple de validation avec JavaScript</h1>
  <p>Entrez un nombre entre 0 et 100</p>
  <form name="form1" action="formulaireOK.jsp" onsubmit="return validFormulaire();">
    votre suggestion : <input type="text" name="val" value="" size="10" />
    <span id="messageErreur" class="erreur"></span><br/>
    <input type="submit" value="Valider" name="valid" /> <input type="reset" value="effacer" />
  </form>
</body>
</html>
```



# Formulaire HTML5

- HTML 5 offre support direct pour expressions régulières

```
<html>
  <head>
    <title>Expression régulières</title>
    <meta charset="UTF-8">
  </head>
  <body>
    <h1>Exemple de validation avec HTML5</h1>
    <p>Entrez un nombre entre 0 et 100</p>
    <form name="form1" action="formulaireOK.jsp">
      <label>votre suggestion :</label>
      <input type="text" name="val" pattern="/[0-9]|[1-9][0-9]|100/"
        placeholder="nombre entre 0 et 100"/><br/>
      <input type="submit" value="Valider" name="valid" /> <input type="reset" value="effacer" />
    </form>
  </body>
</html>
```




Attention au support selon les navigateurs

Entrez un nombre entre 0 et 100

votre suggestion :

Entrez un nombre entre 0 et 100

votre suggestion :

 Veuillez respecter le format requis.

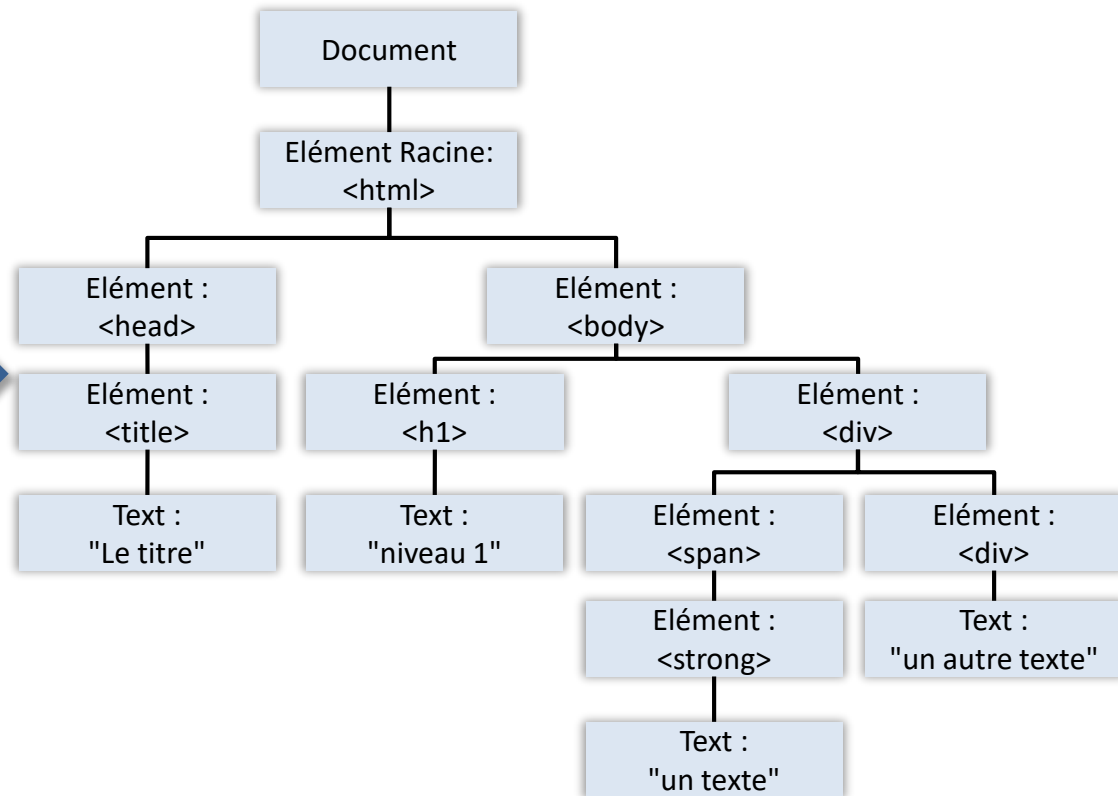
pour en savoir plus

<http://www.coreservlets.com/html5-tutorial/input-types.html>

# arbre DOM

- Quand une page est chargée le navigateur crée un arbre d'objets correspondant aux éléments de la page : l'arbre **DOM** (Document Object Model)
  - représentation objet normalisée (W3C) des documents HTML/XML dont le contenu est arborescent, permet d'accéder à une page Web, de manipuler son contenu, sa structure et ses styles

```
<html>
  <head>
    <title>Le titre</title>
  </head>
  <body>
    <h1>niveau 1</h1>
    <div id="unedivision">
      <span>
        <strong>un texte</strong>
      </span>
      <div id="autresdivision">
        un autre texte
      </div>
    </div>
  </body>
</html>
```





# Le type Node

- représente un nœud de l'arbre DOM
  - définit propriétés permettant d'accéder aux informations relative au nœud

**nodeName** : nom de la balise du nœud

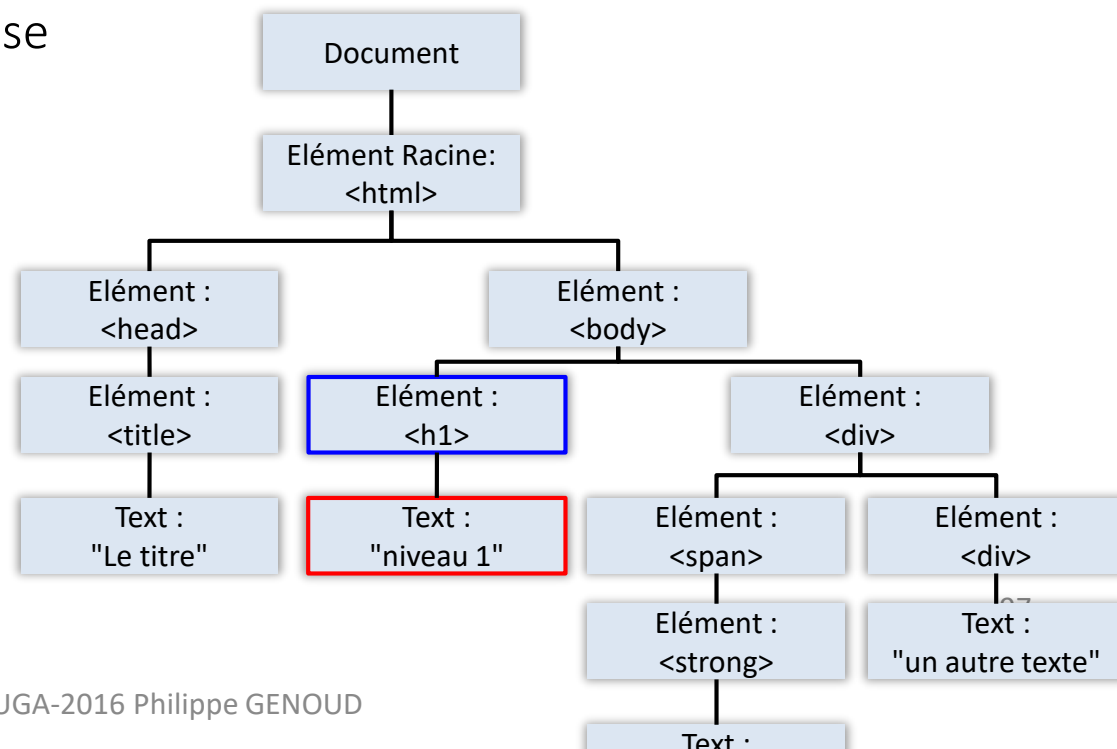
**attributes** : attributs du nœud

**nodeType** : type du nœud

ATTRIBUTE\_NODE, COMMENT\_NODE, DOCUMENT\_NODE, ELEMENT\_NODE, TEXT\_NODE

**nodeValue** : valeur de la balise

```
<html>
  <head>
    <title>Le titre</title>
  </head>
  <body>
    <h1> niveau 1</h1>
    <div id="unedivision">
      <span>
        <strong>un texte</strong>
      </span>
      <div id="autredivision">
        un autre texte
      </div>
    </div>
  </body>
</html>
```

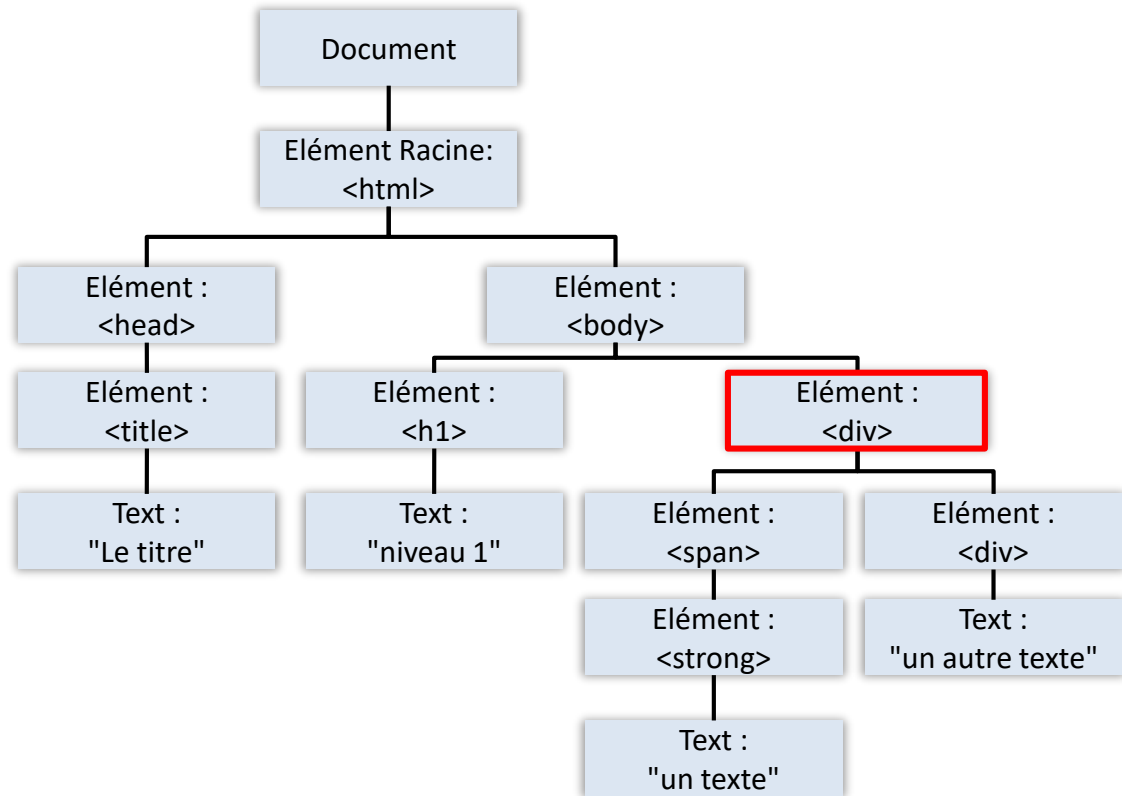


# Accès aux éléments

- accès direct à un nœud identifié par un `id`

```
var elt1 = document.getElementById("unedivision");
```

```
<html>
  <head>
    <title>Le titre</title>
  </head>
  <body>
    <h1> niveau 1</h1>
    <div id="unedivision" >
      <span>
        <strong>un texte</strong>
      </span>
      <div id="autredivision">
        un autre texte
      </div>
    </div>
  </body>
</html>
```



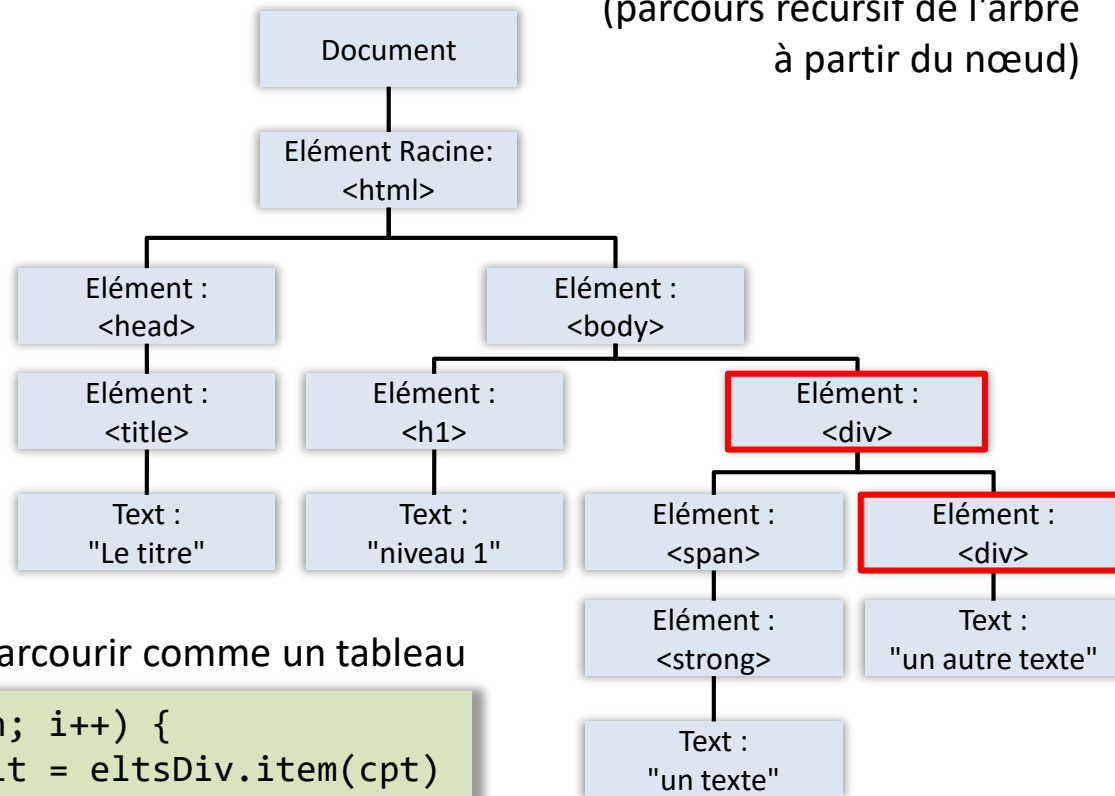
# Accès aux éléments

- accès à une liste de nœuds par nom de balise

```
var eltsDiv = document.getElementsByTagName("div");
```

peut s'appliquer à l'objet document, ou à un objet Nœud  
(parcours récursif de l'arbre  
à partir du nœud)

```
<html>
  <head>
    <title>Le titre</title>
  </head>
  <body>
    <h1> niveau 1</h1>
    <div id="unedivision" >
      <span>
        <strong>un texte</strong>
      </span>
      <div id="autredivision">
        un autre texte
      </div>
    </div>
  </body>
</html>
```



eltsDiv est de type NodeList, peut se parcourir comme un tableau

```
for (var i = 0, i < eltsDiv.length; i++) {
  var elt = eltsDiv[i]; // <=> elt = eltsDiv.item(cpt)
  ...
}
```

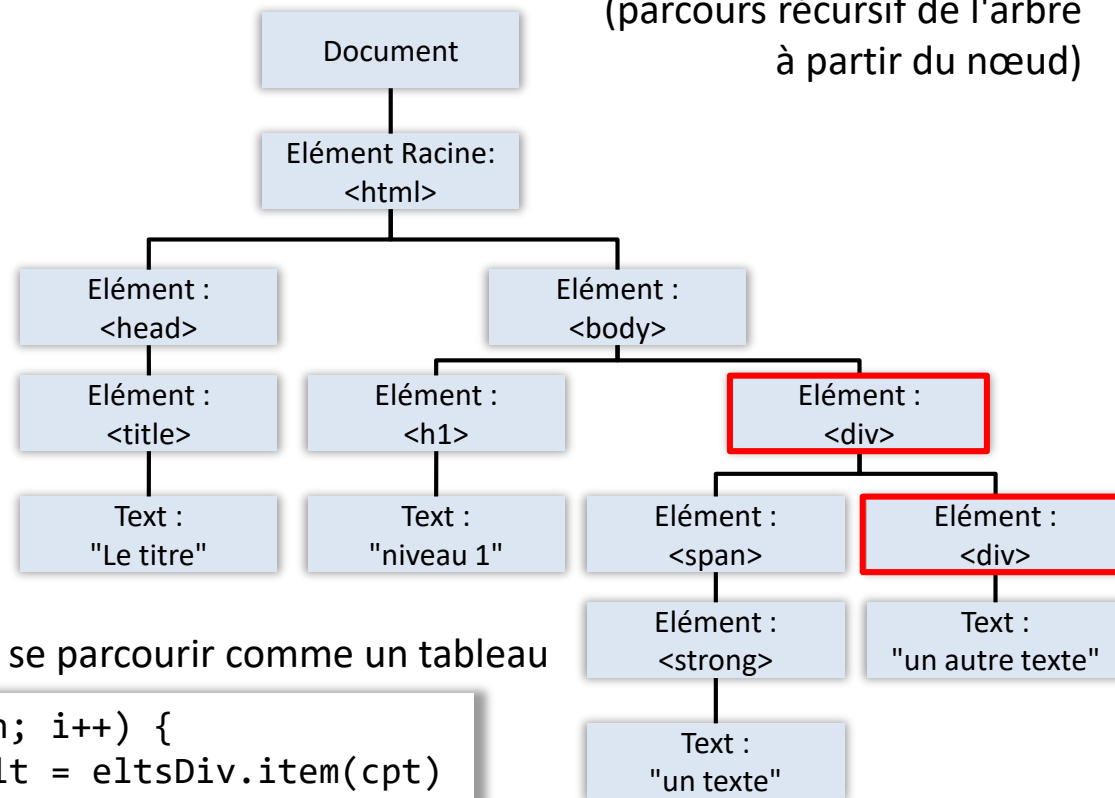
# Accès aux éléments

- accès à une liste de nœuds par nom de balise

```
var eltsDiv = document.getElementsByTagName("div");
```

peut s'appliquer à l'objet document, ou à un objet Nœud  
(parcours récursif de l'arbre à partir du nœud)

```
<html>
  <head>
    <title>Le titre</title>
  </head>
  <body>
    <h1> niveau 1</h1>
    <div id="unedivision" >
      <span>
        <strong>un texte</strong>
      </span>
      <div id="autredivision">
        un autre texte
      </div>
    </div>
  </body>
</html>
```



eltsDiv est de type NodeList, peut se parcourir comme un tableau

```
for (var i = 0, i < eltsDiv.length; i++) {
  var elt = eltsDiv[i]; // <=> elt = eltsDiv.item(cpt)
  ...
}
```

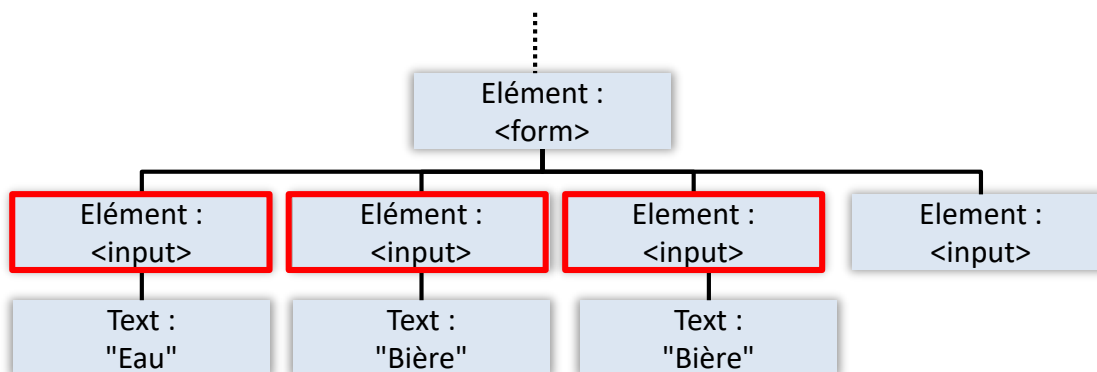
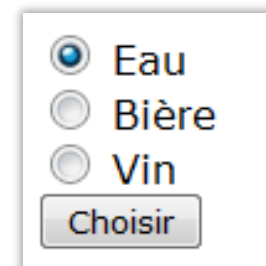
# Accès aux éléments

- accès à une liste de nœuds par valeur de l'attribut name

```
var radioButtons = document.getElementsByName("boisson");
```

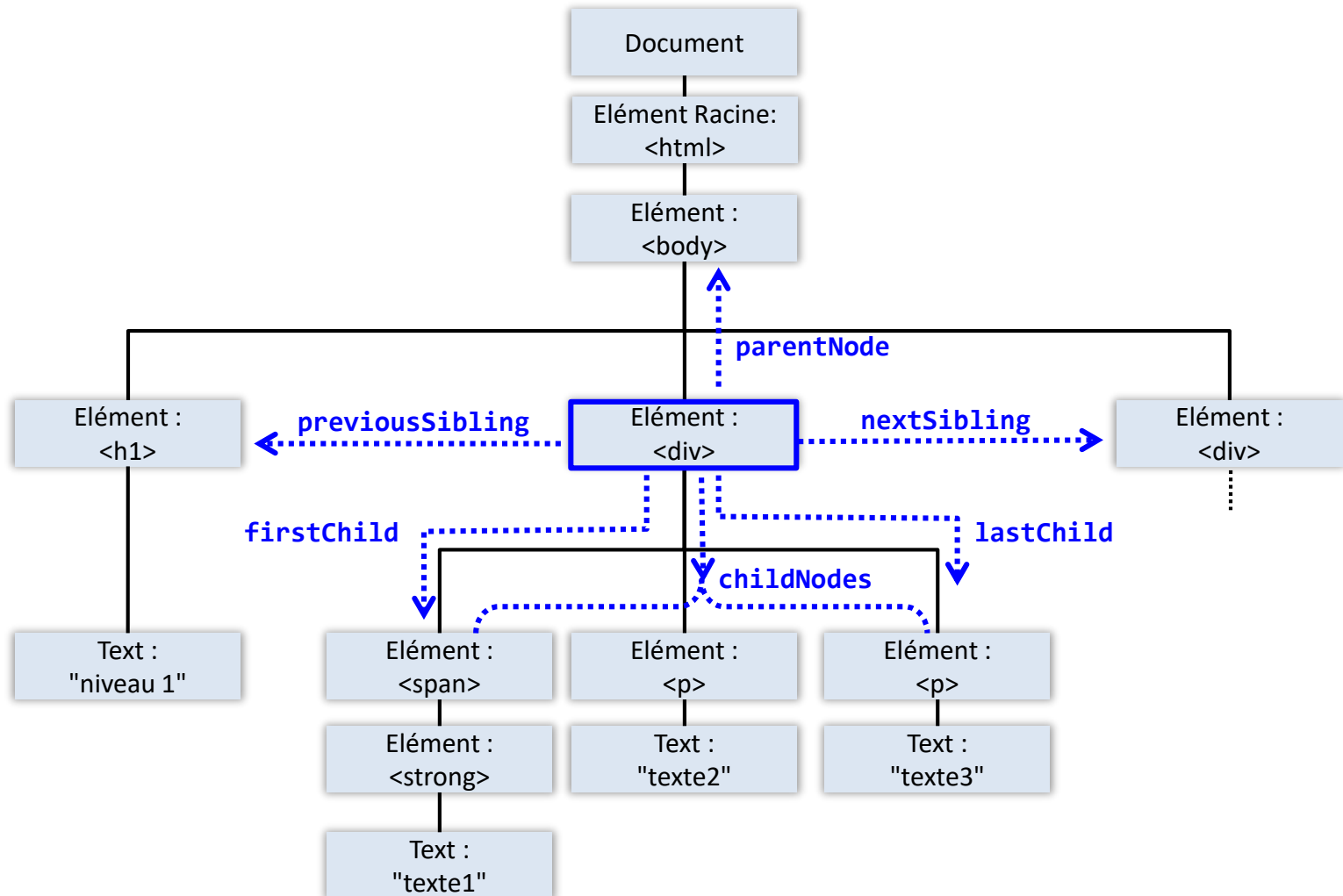
- bien adapté à manipulation de case d'options (radio boutons)

```
<form method="post" id="monFormulaire" action="choisirBoisson">  
  <input type="radio" name="boisson" value="eau" checked> Eau<br/>  
  <input type="radio" name="boisson" value="biere"> Bière<br/>  
  <input type="radio" name="boisson" value="vin"> Vin <br/>  
  <input type="submit" value="Choisir"/>  
</form>
```



# Accès aux éléments

- propriétés de la classe **Node** pour parcours de l'arbre DOM



# Manipulation des Nœuds

- création de nœuds : méthodes de l'objet `document`  
`var eltDiv = document.createElement("div");`  
`var eltTxt = document.createTextNode("un texte");`  
...
- rattacher un nœud créé à un nœud existant : méthodes `appendChild` ou `insertBefore` de `Node`
- retirer un nœud de l'arbre : methode `removeChild` de `Node`

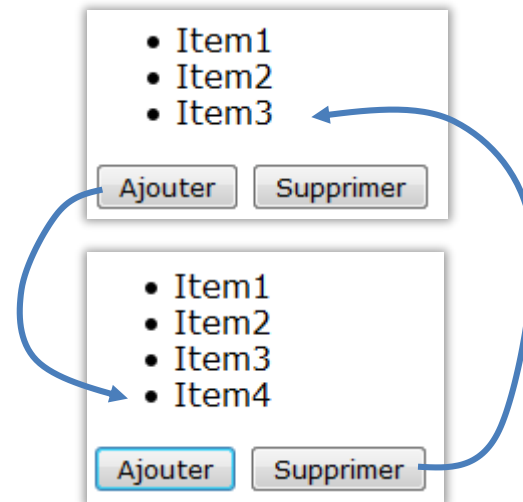


avec les navigateurs Firefox ou Chrome la propriété `childNodes` renvoie l'ensemble des espaces (espaces, retours à la ligne) compris entre les balises HTML sous forme de nœuds `TEXT_NODE`.

Attention de prendre en compte ces nœuds texte lors des traitements.

# Manipulation des Nœuds

```
<html>
  <head>
    <title>Modification DOM</title>
    <meta charset="UTF-8">
    <script>
      // ajoute un item en fin de la liste
      function ajouterItem() {
        var liste1 = document.getElementById("liste1");
        var newLi = document.createElement("li");
        newLi.appendChild(document.createTextNode("Item" + (liste1.childNodes.length + 1)));
        liste1.appendChild(newLi);
      }
      // enleve le dernier item
      function supprimerItem() {
        var liste1 = document.getElementById("liste1");
        liste1.removeChild(liste1.lastChild);
      }
    </script>
  </head>
  <body>
    <h1>Modification du DOM</h1>
    <ul id="liste1">
      <li>Item1</li>
      <li>Item2</li>
      <li>Item3</li>
    </ul>
    <input type="button" value="Ajouter" onclick="ajouterItem();" />
    <input type="button" value="Supprimer" onclick="supprimerItem();" />
  </body>
</html>
```



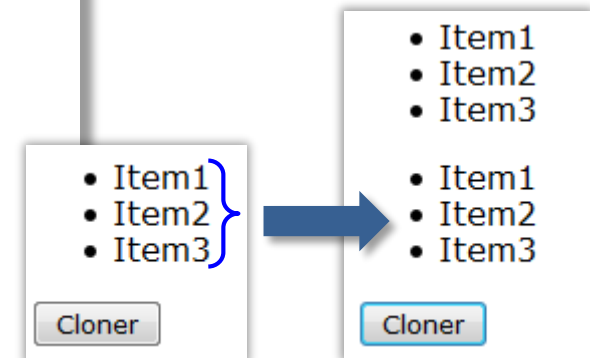


# Manipulation des Nœuds

- méthode `cloneNode` de `Node` permet de créer un nouveau nœud à partir d'un nœud existant.

```
<html>
  <head>
    <title>Modification DOM</title>
    <meta charset="UTF-8">
    <script>
      // clone la liste et l'ajoute au document
      function cloneList() {
        var liste1 = document.getElementById("liste1");
        newList = liste1.cloneNode(true);
        document.getElementById("div1").appendChild(newList);
      }
    </script>
  </head>
  <body>
    <h1>Modification du DOM</h1>
    <div id="div1">
      <ul id="liste1">
        <li>Item1</li>
        <li>Item2</li>
        <li>Item3</li>
      </ul>
    </div>
    <input type="button" value="Cloner" onclick="cloneList();" />
  </body>
</html>
```

Tout le sous-arbre issu du nœud doit être cloné

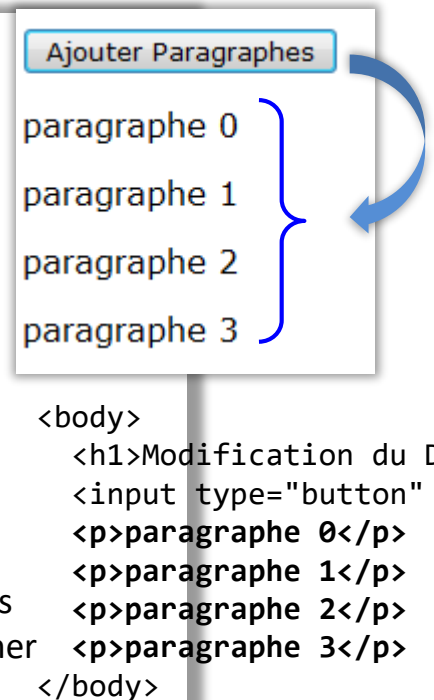


# Fragments d'arbre

- **DocumentFragment** permet de travailler sur une portion d'arbre
  - implémente toutes méthodes de Node
  - utile lorsqu'une nouvelle portion de l'arbre DOM doit être créée et ajoutée (évite de rafraîchir la page à chaque ajout)

```
<html>
<head>
  <title>Modification DOM</title>
  <meta charset="UTF-8">
  <script>
    // ajoute un item en fin de la liste
    function ajouterParagraphes(nb) {
      var frag = document.createDocumentFragment();
      for (var i=0; i < nb; i++) {
        var pElt = document.createElement("p");
        pElt.appendChild(document.createTextNode("paragraphe " + i));
        frag.appendChild(pElt);
      }
      document.getElementsByTagName("body")[0].appendChild(frag);
    }
  </script>
</head>
<body>
  <h1>Modification du DOM</h1>
  <input type="button" value="Ajouter Paragraphe" onclick="ajouterParagraphes(4);" />
</body>
</html>
```

seuls les nœuds enfants du fragment sont insérés  
le fragment est seulement utilisé comme container



# Manipulation des attributs

- méthodes la classe **Node** pour la manipulation des attributs
  - getAttribute(nomAttr)** récupère un attribut à partir de son nom
  - setAttribute(nomAttr, val)** crée ou remplace un attribut existant
  - removeAttribute(nomAttr)** supprime un attribut
  - hasAttribute(nomAttr)** détermine un attribut est présent ou non
- la propriété **attributes** permet d'avoir la liste des attributs d'un nœud

— cette liste est un objet **NamedNodeMap** (tableau associatif)

```
<script>
function afficherAttributs() {
  var frag = document.createDocumentFragment();
  var attrs = document.getElementsByTagName("input")[0].attributes;
  for (var i = 0; i < attrs.length; i++) {
    var txt = "attribut:" + attrs[i].name +
      " = " + attrs[i].value + "<br>";
    var pElt = document.createElement("p");
    pElt.appendChild(document.createTextNode(txt));
    frag.appendChild(pElt);
  }
  document.getElementsByTagName("body")[0].appendChild(frag);
}
</script>
</head>
<body>
  <h1>Attributss</h1>
  <input type="button" value="Attributss" onclick="afficherAttributs();" />
</body>
```

Attributss

attribut:type = button<br>

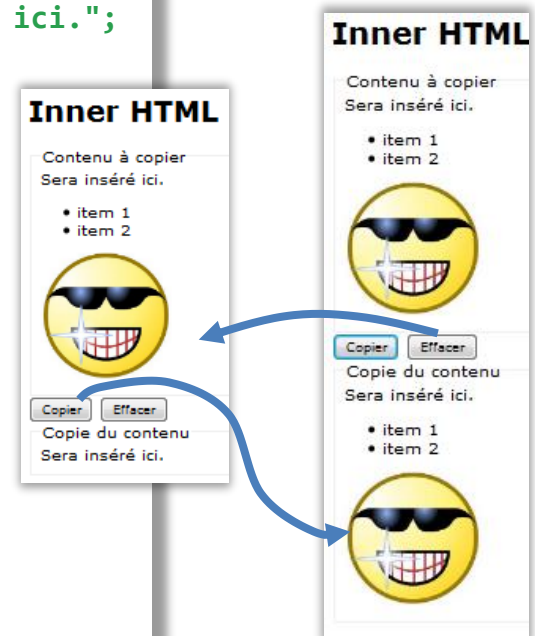
attribut:onclick = afficherAttributs();<br>

attribut:value = Attributss<br>

# Attribut **innerHTML**

- Permet de remplacer complètement le contenu d'un élément par celui spécifié par une chaîne de caractères

```
<html>
  <head>
    <script>
      function recopierSource() {
        var sourceTxt = document.getElementById("source").innerHTML;
        document.getElementById("destination").innerHTML = sourceTxt;
      }
      function effacerCopie() {
        document.getElementById("destination").innerHTML = "Sera inséré ici.";
      }
    </script>
  </head>
  <body>
    <h1>Inner HTML</h1>
    <fieldset><legend>Contenu à copier</legend>
      <div id="source">
        <ul>
          <li>item 1</li>
          <li>item 2</li>
        </ul>
        
      </div>
    </fieldset>
    <input type="button" value="Copier" onclick="recopierSource();" />
    <input type="button" value="Effacer" onclick="effacerCopie();" />
    <fieldset><legend>Copie du contenu</legend>
      <div id="destination">Sera inséré ici.</div>
    </fieldset>
  </body>
</html>
```



# Modification des styles CSS

```
<html>
  <head>
    <title>Modification Styles</title>
    <meta charset="UTF-8">
    <style>
      * {
        font-family: verdana;
      }
    </style>
    <script>
      function modifCouleurParagraphe(newcolor) {
        var p1 = document.getElementById("p1");
        p1.style.color=newcolor; // style associé à l'élément
      }
    </script>
  </head>
  <body>
    <h1>Modification du Style</h1>
    <p id="p1">
      Un paragraphe avec style "style1".
    </p>
    <p>
      Un autre paragraphe sans style
    </p>
    <p>
      changer le style du 1er paragraphe<br/>
      <input type="button" value="Rouge" onclick="modifCouleurParagraphe('red')"/>
      <input type="button" value="Vert" onclick="modifCouleurParagraphe('green')"/>
    </p>
  </body>
</html>
```

- modification du style css d'un nœud par sa propriété **style**

## Modification du Style

Un paragraphe avec style "style1".

Un autre paragraphe sans style

changer le style du 1er paragraphe

Rouge

Vert

# Modification des styles CSS

```
<html>
<head>
  <title>Modification des styles CSS</title>
  <meta charset="UTF-8">
  <style>
    .style1 {
      color:green;
      font-size: 1.5em;
    }
    .style2 {
      color:red;
      font-size: 1.5em;
    }
  </style>
  <script>
    function modifCouleurParagraphe(newClasse) {
      var paragraphes = document.getElementsByTagName("p");
      for (var i = 0; i < paragraphes.length; i++) {
        if (paragraphes[i].className.match(/style1|style2/)) {
          paragraphes[i].className=newClasse;
        }
      }
    }
  </script>
</head>
```

- modification du style css d'un nœud en changeant sa classe
- propriété `className`

```
<body>
  <h1>Modification du Style</h1>
  <p class="style1">
    Un paragraphe avec style "style1".
  </p>
  <p class="style1">
    Un autre paragraphe avec style "style1".
  </p>
  <p>
    changer le style<br/>
    <input type="button" value="Rouge" onclick="modifCouleurParagraphe('style2')"/>
    <input type="button" value="Vert" onclick="modifCouleurParagraphe('style1')"/>
  </p>
</body>
</html>
```

## Modification du Style

Un paragraphe avec style "style1".

Un autre paragraphe avec style "style1".

changer le style

# Modification des styles CSS

- modification d'une définition de style css

```
<html>
<head>
  <title>Modification du style</title>
  <meta charset="UTF-8">
  <style>
    .style1 {
      color:green;
      font-size: 1.5em;
    }
  </style>
  <script>
    function modifStyleGlobal(newcolor) {
      crossRules = (document.styleSheets[0].cssRules)?
        document.styleSheets[0].cssRules
        :document.styleSheets[0].rules;
      for (var i = 0; i < crossRules.length; i++) {
        if (crossRules[i].selectorText == ".style1") {
          crossRules[i].style.color=newcolor;
          return;
        }
      }
    }
  </script>
</head>
```

```
<body>
  <h1>Modification du Style</h1>
  <p class="style1">
    Un paragraphe avec style "style1".
  </p>
  <p class="style1">
    Un autre paragraphe avec style "style1".
  </p>
  <p>
    changer le style<br/>
    <input type="button" value="Rouge" onclick="modifStyleGlobal('red')"/>
    <input type="button" value="Vert" onclick="modifStyleGlobal('green')"/>
  </p>
</body>
</html>
```

## Modification du Style

Un paragraphe avec style "style1".

Un autre paragraphe avec style "style1".


changer le style

Rouge Vert

# Exceptions

- Mécanisme d'exception
  - syntaxe similaire à Java : **try**, **catch**, **finally**

```
try {  
    document.write("avant fonction inconnue<br>");  
    fonctionInconnue(); // cette fonction n'est pas définie  
    document.write("après fonction inconnue<br>");  
} catch (error) {  
    document.write("une exception a été levée<br>");  
    document.write("son nom : " + error.name + "<br>");  
    document.write("son type : " + typeof error + "<br>");  
    document.write("son message : " + error.message + "<br>");  
} finally {  
    document.write("dans finally<br>");  
}
```



avant fonction inconnue  
une exception a été levée  
son nom :ReferenceError  
son type :object  
son message :fonctionInconnue is not defined  
dans finally



# Exceptions

- possibilité d'intercepter les exceptions de manière globale

définition d'un observateur d'événements

```
<html>
  <head>
    <title>Gestion des exceptions</title>
    <meta charset="UTF-8">
    <script>
      function gestionErreurs(message,fichier,ligne) {
        var txt = "erreur dans " + fichier + "\nà ligne "
          + ligne + "\n" + message ;
        alert(txt);
      }
      onerror = gestionErreurs;
    </script>
  </head>
  <body>
    <h1>Exception</h1>
    <p>
      Exemple de page contenant une erreur JavaScript<br/>
      <script>
        fonctionInconnue(); // cette fonction n'est pas définie
      </script>
      Ce texte sera quand même affiché<br/>
      <script>
        fonctionInconnue(); // cette fonction n'est pas définie
      </script>
      Et celui-ci aussi<br/>
    </p>
  </body>
</html>
```

## Exception

Exemple de page contenant une erreur JavaScript

erreur dans http://localhost:8084/ExemplesCoursJavaScript  
/exceptions\_1.html  
à ligne 32  
ReferenceError: fonctionInconnue is not defined

OK

# Exceptions

- Lancement d'exceptions
  - possibilité de lancer des exceptions **throw**
  - possibilité de définir ses propres classes d'exception (qui peuvent être de n'importe quelle classe)

## Lancer une Exception

Exemple de page contenant une erreur JavaScript

erreur MonException  
essai

OK

```
<html>
<head>
  <title>Gestion des exceptions</title>
  <meta charset="UTF-8">
  <script>
    function MonException(message) {
      this.name = "MonException";
      this.message = message;
    }
    function fonctionBoguer() {
      throw new MonException("essai");
    }
  </script>
</head>
<body>
  <h1>Lancer une Exception</h1>
  <p>
    Exemple de page contenant une erreur JavaScript<br/>
    <script>
      try {
        fonctionBoguer();
        document.write("Ce texte ne sera pas affiché<br>");
      } catch (ex) {
        var txt = "erreur " + ex.name + "\n"
          + ex.message ;
        alert(txt);
      }
    </script>
    Ce texte sera quand même affiché<br/>
  </p>
</body>
</html>
```

# Apply et Call

- en Javascript les fonctions sont des objets de "classe" **Function**
  - méthodes **call** et **apply** permettent d'exécuter une fonction pour un objet donné sans affecter la fonction à l'objet.

```
<html>
<head>
</head>
<body>
  <h1>Utilisation de this</h1>
  <p>
    <script>
      function test(nb,msg) {
        for (var i = 0; i < nb; i++){
          document.write(msg + " : " + this.attribut + "<br>");
        }
        document.write("<br>");
      }
      var obj1 = new Object();
      obj1.attribut = "attribut obj1";
      test.call(obj1,3,"Call Obj1 ");    // obj1.methode = test;
                                         // obj1.methode(3,"Attribut de Obj1 ");
      var obj2 = new Object();
      obj2.attribut = "attribut obj2";
      test.call(obj2,2,"Call Obj2 ");
      test.apply(obj1,[2,"Apply Obj1"]);
      test.apply(obj2,[2,"Apply Obj2"]);
    </script>
    Ce texte sera quand même affiché<br/>
  </p>
</body>
</html>
```

Call Obj1 : attribut obj1  
Call Obj1 : attribut obj1  
Call Obj1 : attribut obj1

Call Obj2 : attribut obj2  
Call Obj2 : attribut obj2

Apply Obj1 : attribut obj1  
Apply Obj1 : attribut obj1

Apply Obj2 : attribut obj2  
Apply Obj2 : attribut obj2

Ce texte sera quand même affiché

call : liste de paramètres

apply: tableau de paramètres