HTML ET JAVASCRIPT - RAPPELS

HTML, Javascript et CSS sont trois langages complémentaires utiles au développement web et dont l'évolution est suivie et encadrée, en théorie par les organismes suivants :

- ECMA – European association for standardizing information and communication systems - http://www.ecma-international.org/ – pour Javascript
- w3c -World Wide Web Consortium (W3C) –https://www.w3.org/ – pour HTML et pour la partie UI events commune avec Javascript
- whatwg – https://whatwg.org/ idem W3C pour ce qui est du domaine mais perspective, personnes et organisation différentes.

Le processus d'évolution des normes au W3C est en théorie consensuel et, en théorie, prévu pour passer par le cycle d'états suivants :

Idée → Editor Draft → Working Draft → Candidate Recommendation → Recommendation (Standard)

Les implémentations dans les navigateurs sont censées commencer au moment de "candidate recommendation". Cependant, en pratique, pour le DOM3, par exemple, les implémentations ont commencé avec le Working Draft qui a fait office de candidate et standard de fait.

Quelques repères de dates

1992	NCSA Mosaic, naissance d'un des premiers navigateurs graphiques, par NCSA, National Center for Computing, Univ Illinois at Urbana Champaign.
1994	Netscape Communicator détrône Mosaic
1995	Brendan Eich, est recruté par Netscape pour écrire un prototype de langage complémentaire de Java qui permettrait de rendre dynamiques les pages web.
1996	ECMA débute EcmaScript comme une spécification implémentée par Javascript (Netscape), ActionScript (Macromedia), Jscript (Microsoft).
1998	Ecmascript 2
1999	Ecmascript 3
	Tentatives de création d'Ecmascript 4 ralenties par MS dont la priorité est .net
2005	Jesse James Garrett écrit un article dans lequel il introduit le terme Ajax.
2008	Suite des évolutions Ecmascript 3 → Ecmascript 5.
2011	Ecmascript 5.1 ;
2015	ECMAScript 2015 - ECMA-262, edition 6
2016	ECMAScript 2016 - ECMA-262, edition 7
2017	ECMA-262, edition 8 [source ecma-international, 26-09-2017]

ASPECTS PRATIQUES UTILES AU DÉMARRAGE

OÙ ÉCRIRE LES SCRIPTS ? L'ÉLÉMENT <SCRIPT> ET SES ATTRIBUTS

Les scripts Javascript sont écrits dans les éléments <script> contenus dans un document HTML. Cette balise peut directement contenir du code source ou/et spécifier un fichier externe qui contient le code source via l'attribut src.

Exemple : `<script src="utils.js">var prompt_fr= « Bienvenue » ;</script>`

Les scripts sont interprétés en deux étapes. La première étape consiste à charger le document et les scripts qu'il contient de manière :

- *synchrone* (mode par défaut) : les scripts sont chargés et interprétés dans l'ordre de lecture du fichier html.

- *asynchrone* (attribut async) : le navigateur choisi les moments de chargement des scripts pour éviter les blocages tout en optimisant le chargement. C'est l'attribut async de l'élément <script>, qui permet de passer dans ce mode. Par exemple : `<script async src="utils.js"></script>`
- *reportée* (attribut defer) : le chargement du script est reporté au moment où le DOM est entièrement chargé et prêt à être manipulé.

Dans la deuxième étape, les scripts de gestion d'évènements sont traités de manière asynchrone selon les évènements issus des actions des utilisateurs et de leur propagation/réception aux éléments du DOM par le navigateur. Ce fonctionnement est décrit plus amplement dans la suite de ce document.

PROTÉGER LES ESPACES DE NOMS

Afin de préserver l'espace global des noms (variables, fonctions, objets), une première technique consiste à utiliser les objets et à définir toutes vos fonctions et variables comme des propriétés et des méthodes des objets.

Une deuxième approche consiste à utiliser une fonction et à définir toutes vos fonctions et variables avec une portée de fonction au sein de cette fonction ou de ses fonctions imbriquées.
Exemple:

```
function nommagePerso(){
    var i; //portée locale à nommagePerso pas de problème de collision de noms
dans l'espace global a priori…
    function print(){var w=5};//print local
}
nommagePerso(); // n'ajoute qu'un seul nom de fonction dans l'espace global
```

Cette deuxième approche, comme la première utilise néanmoins au moins un nom de fonction ou d'objet potentiellement en collision avec d'autres.

Pour remédier à ce problème éventuel, une troisième approche, qui évite toute collision avec les autres espaces de noms, consiste à protéger toutes vos fonctions et variables en les déclarant au sein d'une fonction anonyme auto-invoquée – c'est à dire une fonction anonyme dont la définition et l'invocation se font en une seule expression. Cette approche est très fréquemment utilisée en Javascript.

Exemple:

```
(function (){
    var i;
    function print(){};
}()); // aucun nouveau nom dans l'espace global
```

CONVENTIONS DE NOMMAGE DES *CLASSES* D'OBJETS

Dans les langages fortement typés comme Java ou C++, il est d'usage de donner une majuscule à la première lettre des noms de classes. Même s'il n'y a pas vraiment de classe en Javascript, cette convention se retrouve en donnant habituellement aux *constructeurs* une majuscule comme première lettre de sorte à différencier ces fonctions au rôle particulier des autres fonctions.

JAVASCRIPT ET HTML - ACCÈS, CRÉATION, MODIFICATION DU DOM ET DU BOM

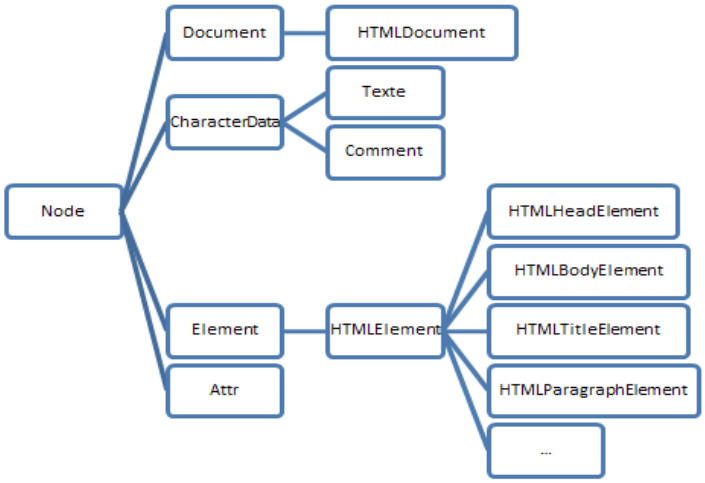
BOM = Browser Object Model. Ce modèle décrit principalement `window, location, navigator, screen, history`

DOM = Document Object Model. Ce modèle décrit un document sous la forme d'un arbre de *nœuds*. Ces nœuds sont de types différents et les types les plus fréquemment utilisés sont: éléments html, textes, attributs, fragments et commentaires.

MÉTHODES JAVASCRIPT DE CRÉATION DES PRINCIPAUX NŒUDS:

`createElement(elementTag), createTextNode(text), setAttribute(name, value), createFragment().`

```
//Exemple de création dynamique de noeuds
   var elt = document.createElement('div') ;
   var msg = document.createTextNode('whatever text you want') ;
   var attribute = document.createAttribute('at');
   attribute.value = '0' ;
   elt.setAttribute(attribute)
   var fragment = document.createDocumentFragment() ;
   elt.appendChild(msg) ;
   fragment.appendChild(elt) ;
```



Hiérarchie partielle des principales classes de nœuds qui composent le DOM

Méthodes de modification du DOM :

`nodeParent.appendChild(nodeChild)`, `node.insertBefore(newNode, refNode)`, `nodeParent.cloneNode(isRecursiveCloning?)`, `nodeParent.replaceChild(newNode, oldNode)`, `nodeParent.removeChild (newNode, refNode)`.

Propriétés génériques des nœuds :

`nodeType`	1 pour `Element`, 9 pour `Document`, 3 pour `Text`, 8 pour `Comment`, etc.
`nodeName`	tag des nœuds de type `Element`
`nodeValue`	le contenu textuel des nœuds de type `Text` ou `Comment`,
`textContent`	pour les nœuds de type `CharacterData`, `textContent` retourne le texte correspondant et équivaut à `nodeValue`. Pour les autres nœuds, `textContent` retourne la concaténation des attributs `textContent` de ses enfants.
`innerHTML`	retourne toute la sous-arborescence d'un nœud en html. Cela permet entre autres de modifier une partie du DOM, en injectant directement du code html.
`childNodes, parentNode, firstChild, lastChild, nextSibling,previousSibling`	Méthodes utiles au parcours des noeuds du DOM pour tous les types de noeuds.
`children, firstElementChild,lastElementChild, nextElementSibling, previousElementSibling, childElementCount`	Méthodes utiles au parcours des noeuds du DOM concernant seulement les nœuds de la classe Element et de ses sous-classes.

Modification des attributs des nœuds du DOM

Il existe des attributs Javascript pour définir les attributs standards des nœuds du DOM. Les noms de ces attributs sont inspirés des noms des attributs en utilisant des minuscules et en remplaçant les – par *camélisation*. Par exemple:

```
   var f = document.forms[0];
   f.method = "POST";
```

De manière plus générale, on utilise les méthodes `setAttribute(attribute, value)` et `getAttribute(attribute)` en prenant pour `attribute` la chaîne de caractère du nom de l'attribut en majuscule.

Modification dynamique des styles des éléments

Trois principales stratégies sont utilisées: modification des styles inline, modification des styles css, modification de la classe des éléments.
La modification des styles inline revient à modifier les styles d'un élément en utilisant l'attribut style de l'élément, cela revient à faire une modification du style inline qui est prioritaire sur les autres définitions de style hormis celles qui sont étiquetées `!important`. Les noms des attributs de style Javascript sont inspirés des noms des attributs css en utilisant la même règle que ci-dessus: minuscules *camélisées*. Exemple:

```
e.style.color = "blue";
e.style.fontSize = "24pt"; // pour l'attribut css font-size
```

La modification de la (des) classe(s) associées à un élément permet de changer simplement un ensemble de styles associés à cette (ces) classe(s). Si `className` est l'attribut Javascript permettant de modifier l'attribut HTML définissant la classe d'un élément, il est préférable de modifier la liste des classes associées à l'élément en utilisant les méthodes `add`, `remove`, `toggle`, et `contains` disponibles en Javascript à partir d'HTML5.

Recherche d'élements dans le DOM

Pour sélectionner les éléments du DOM les méthodes d'accès Javascript suivantes retournent des collections qui se comportent comme des tableaux en lecture seule et qui sont *dynamiques* ou pas. Les collections *dynamiques* sont automatiquement mises à jour lorsque le DOM est modifié après la création de la collection, les autres ne le sont pas.

`elt.getElementById(idelt)`	retourne l'élément dont l'id est `idelt`	
`elt.getElementsByTagName(htmlName)`	retourne les éléments html de *type* `htmlName`	dynamique
`elt.getElementsByName(ename)`	retourne les éléments html ayant pour attibut name `eName`	dynamique
`elt.getElementsByClassName(name)`	retourne les éléments de la classe `name`	
`elt.querySelector(selecteur)`	retourne le premier élément correspondant à `selecteur`.	
`elt.querySelectorAll(selecteur)`	retourne tous les éléments du dom correspondant à `selecteur`	
`document.images`	retourne toutes les images du document	dynamique
`document.forms`	retourne tous les éléments de type form	dynamique
`document.links`	retourne tous les liens ayant un attribut href	dynamique

Dans les méthodes `querySelector`, le paramètre `selecteur` s'écrit comme un sélecteur css dont la syntaxe est résumée dans la section CSS.

Variables, contexte, portée, chaîne de portée et this

Les variables sont typées dynamiquement et peuvent prendre les types `undefined`, `boolean`, `number`, `string` pour les valeurs primitives, `object` pour les références et `function` pour les fonctions que l'on peut considérer comme un genre d'objets à part.
NB : l'opérateur `typeof` retourne le type d'une variable et retourne `object` pour `null`.

Le contexte d'exécution d'un programme désigne les structures invisibles du programmeur qui permettent de déterminer les valeurs ou les expressions associées à chaque symbole (c'est-à-dire principalement variable ou fonction).

La déclaration des variables peut se faire en utilisant l'un des mots clés optionnels suivant :

- var : portée de fonction
- let : portée de bloc
- const : « variable » non modifiable

Déclaration, Portée, chaîne de portée, hissage de déclaration (*declaration hoisting*)

La portée d'une variable qui définit la valeur ou l'expression associée à cette variable, est déterminée par la position de la déclaration de cette variable dans le code source ainsi que du mot-clé utilisé (ou pas) dans cette déclaration.

Par exemple :

```
function f (a, b){              // a, b ont la portée de la fonction f
    var c = 3 ;                 // c a également la portée de f
    d = 15 ;                    // d a une portée globale
    for (let i=0; i < 10; i++){  // I a une portée de bloc
          console.log(i); //0,1…9
    }
    console.log(i);             //undefined car i a une portée de bloc
}
```

La *portée globale* est la portée des symboles qui sont déclarés

- dans le contexte global (c'est à dire hors de toute fonction) en utilisant le mot-clé `var`, ou sans l'utiliser,
- dans une fonction en n'utilisant aucun des mots-clé `var, let ou const.`

Les variables et fonctions globales sont partagées par tout le code, on peut aussi les considérer comme propriétés et variables de l'objet `window`.

La *portée de fonction* est la portée des symboles introduits comme *paramètres formels* des fonctions ou déclarés avec `var` à l'intérieur d'une fonction.

La *portée de bloc* correspond aux symboles déclarés en utilisant le mot-clé let.

On parle de *chaîne de portée* lorsque des définitions de fonctions sont imbriquées. Une chaîne de portée est alors automatiquement créée. Cette chaîne de portée permet aux symboles déclarés dans une fonction d'être accessibles aux fonctions imbriquées. A l'inverse, les symboles définis dans une fonction imbriquée *ne sont pas* accessibles aux fonctions englobantes.

De plus, lorsque le même symbole est déclaré dans plus d'une fonction, la chaîne de portée permet de déterminer l'ordre dans lequel la liaison entre le symbole et son expression ou sa valeur est résolu: la résolution se fait du contexte lexical le plus *intérieur* jusqu'au contexte le plus *extérieur* – et potentiellement global.

On parle également de chaîne de portée dans le cas de l'héritage de prototype, comme on le verra dans la section sur les objets. La chaîne de portée dont on parle lors de l'héritage participe de la même idée que la chaîne de portée liée à l'imbrication de fonction et garantit ainsi qu'un objet enfant ait accès aux attributs et méthodes de ces parents, cependant, dans le cas de l'héritage, la chaîne de portée passe par une chaîne de prototypes dont la mise en œuvre est un peu plus opaque que dans le cas plus simple des fonctions imbriquées.

Exemple avec fonctions imbriquées

```
function f (a, b){          // a, b ont la portée de f
    function g(x){
          var a = 5;
          return a*x + b; //le a de la portée de g
    }
    return g(a); // mais ici le a de la portée de f.
}
console.log(f(1,2)); // 7
```

Augmentation explicite de la chaîne de portée

Il est possible artificiellement d'augmenter la chaîne de portée avec l'instruction `with`. C'est ce qu'utilisent les navigateurs pour augmenter la chaîne de portée au document et au formulaire pour rendre leurs éléments accessibles aux callbacks susceptibles de les gérer ensuite.

Le cas de This

En Javascript, `this` désigne le contexte d'appel et l'association entre le contexte et `this` est manipulable par toutes sortes de moyens. Ainsi, alors que dans un langage comme Java, on peut être à peu près certain que dans une méthode d'une classe A, `this` désigne une instance de la classe A, ou éventuellement d'une classe héritant de A, en Javascript, au contraire, on ne sait jamais vraiment à l'avance ce qu'il y aura dans `this`. Dans un constructeur, on peut penser que `this` représentera bien l'objet en cours de construction, mais dans les méthodes rien de moins sûr. D'où l'usage, parfois répandu dans certains programmes, d'une variable privée locale à un objet initialisée dans le constructeur qui servira à garder une référence sur le `this` de la création de l'objet.

«USURPATION »DE THIS

L'utilisation de `apply`, et de `call` permettent d'appeler une fonction sur n'importe quel objet et d'associer ainsi ladite fonction au contexte de l'objet appelant.
L'utilisation de `addEventListener` permet d'associer une callback à un objet, la fonction utilisée comme callback s'exécute alors dans le contexte de l'objet appelant `addEventListener`.
D'une manière plus ancienne, l'utilisation de la syntaxe HTML permet également d'associer une callback à un élément du DOM. Comme dans le cas précédent, dans la fonction utilisée comme callback, `this` correspondra alors dynamiquement à l'objet appelant.
A partir de Ecmascript 5, il est également possible d'utiliser la méthode `bind` pour associer une méthode à un objet.

HISSAGE DE DÉCLARATIONS (*DECLARATION HOISTING*)

Rappel terminologique: on parle de (1) *déclaration* pour ce qui est de nommer une fonction ou variable, (2) *d'affectation* pour l'association entre une variable/fonction et une expression ou valeur et (3) de *définition* lorsque *déclaration* et *affectation* sont faites par la même instruction.
Exemple:

```
var f;                              // déclaration
f = function (x,y){return x*y ;};// affectation
var f = function (x,y){return x*y ;};   // définition
```

Le mécanisme de *declaration hoisting* consiste à remonter automatiquement toutes les déclarations en début de script. Dans le cas des variables et des *définitions de fonction par expression*, seules les *déclarations* sont remontées. L'affectation et l'évaluation des variables et des fonctions définies par expression se fait ensuite et dans l'ordre lexical du haut vers le bas. [...]

FONCTIONS

1. Définition

Les définitions de fonctions Javascript peuvent se faire par des expressions (en anglais function definition expression) ou par des instructions (en anglais function declaration statement).
Exemple:

```
var f = function(x,y){return x*y ;}     //function definition expression - expression
function produit(x,y){return x*y ;};    //function declaration statement – instruction
```

2. Arguments

Toute fonction est associée à un tableau dynamique d'arguments accessible par la variable `arguments`, qui peut être considérée comme un tableau un peu spécial. Les arguments nommés sont optionnels. Le nommage est surtout utile à la lecture et la compréhension du code.

Tous les arguments de fonction sont passés en valeur. Si l'argument est une référence à un objet ou une fonction, la référence est copiée, ce qui rend possible la modification durable des objets à l'intérieur d'une fonction mais **pas l'affectation**. Par exemple :

```
function setValue(obj){
    obj.value = "nouvelle valeur" ;
    obj = new Object() ;
    obj.value = "une autre valeur" ;
```

```
}
console.log(obj.value) ; // affiche "nouvelle valeur"
```

3. Fonctions imbriquées

En Javascript, il est fréquent d'imbriquer des définitions de fonctions. Par exemple:

```
function distanceEuclidienne (a, b, c, d){ // entre les 2 points 2D de
coordonnées resp. (a,b) et (c,d)
    function square(x){return x*x;}
    return Math.sqrt(square(c-a)+(d-b));
}
```

Certains cas particuliers de fonctions imbriquées demandent de faire attention à la portée des variables qu'elles concernent et sont détaillées dans la section sur les *fermetures*, le *currying* et les *callback*.

4. Invocation indirecte des fonctions

Les fonctions Javascript peuvent être invoquées de plusieurs manières différentes: (1) appel simple, combiné potentiellement à une définition, (2) invocation de méthode sur un objet, (3) invocation de constructeur, (4) invocation indirecte grâce à `call` et `apply` et (5) invocation dans une expression évaluée avec `eval`.

(a) L'appel simple a été évoqué précédemment. Il peut être conjugué avec la définition de la fonction comme dans l'exemple suivant:

Exemple: `(function (a){console.log(a);})("bonjour");`

(b) `eval(code)`

Evalue le code Javascript à partir d'une chaîne de caractère passée en argument. Retourne la valeur du code évalué. Emet une exception de type SyntaxError si le code est invalide syntaxiquement.

(c) `apply et call`

Ces méthodes permettent d'appliquer une fonction à un objet particulier et elles diffèrent principalement dans la manière de préciser la liste des paramètres à passer à la fonction. Apply utilisera un tableau ou un "presque tableau". Call précise les paramètres un à un.

Exemple:

```
Math.max.apply(Math, [1,2,4,5,7]) -> 7
Math.max.call(Math, 1,2,4,5,7) -> 7
```

`aFunction.apply(objet, argument_list){}` applique la fonction `aFunction` à l'objet `objet` en passant comme arguments les arguments `arg1, arg2, argn`.

Cependant, on peut également le faire sur l'objet `null` comme par exemple ici:

```
var f = function(x,y){return x*y ;}
console.log(f.apply(null, [6,7]));          // 42
```

Avec cette invocation, le premier argument (`null`) est l'objet sur lequel la fonction est appelée, et le deuxième argument (`[6,7]`) est tableau des arguments de `f`.

(d) `bind`

A partir d'EcmaScript 5, il est possible d'utiliser `bind` pour attacher une méthode à un objet en y ajoutant potentiellement quelques arguments. La méthode `bind` s'invoque sur une fonction et prend en paramètre en premier un objet auquel associer la fonction et d'autres paramètres optionnels potentiellement utiles. Ainsi, si `bind` est créée en premier lieu pour créer des correspondances nouvelles, `bind` permet également de paramétrer des fonctions ou dit autrement d'utiliser des fonctions avec des listes partielles d'arguments.

Par exemple:

```
function f(x, z){return x + this.y + z;}
var o = {y:20};
var g = f.bind(o,3);
var t = g(2); // x=2, y=20 et z=3 => t vaut 25
```

FERMETURE – CALLBACK - CURRYING

Une fermeture est une fonction imbriquée qui capture le contexte lexical de ses variables. Le mécanisme de fermeture permet ainsi à plusieurs fonctions internes de partager au moment de

l'invocation un même contexte lexical, généralement celui d'une ou plusieurs fonctions englobantes.

Les fermetures peuvent être utilisées dans de nombreux contextes notamment dans la création des méthodes d’objets. Elles peuvent également être utilisées lors de la gestion des évènements comme dans les deux exemples ci-dessous.

Certaines fermetures se distinguent des autres fermetures : ce sont les fermetures qui sont utilisées comme retour de fonction. En effet dans ce cas, la fermeture emporte alors avec elle les liaisons vers les variables de la (des) fonction(s) englobante(s) et les conserve même après que la ou les fonctions englobantes aient terminé.

```
// Exemple de fermetures qui partagent une variable
(function(){
      var sharedMessage="";
      var previousEvent=null;
//cette fonction crée des fonctions de rappel (callback)
      function ge(){
            return function(e){
                  if (previousEvent == null || previousEvent != e){
                        previousEvent = e;
                        sharedMessage = "";
                        nbTraitements = 0;
                  }
                  var bubbling = (e.eventPhase == 3) ? "bubbling" : (e.eventPhase
==2 ? "at target" : "capture");
                  sharedMessage += " target (" + e.target.id + "), this (" +
this.id+") "+bubbling+"\n";
                  if (this == document.body)
                        console.log("\n\n"+e.timeStamp+"\t"+sharedMessage);
                  }
      }


      function handleMouseOver(){
            var elts = document.querySelectorAll("div, p");
            var nbCallbacks = elts.length;
            for (var i = 0;i <nbCallbacks;i++){
                  tmp = elts[i];
                  if (i%2==0) tmp.addEventListener("mouseover", ge(), false);
                  else tmp.addEventListener("mouseover", ge(), true);
            }
      }

      document.body.addEventListener("mouseover", ge(), false);
      window.addEventListener("load", handleMouseOver, false);
})();
```

Remarque : dans cette version de fermetures utilisées comme fonctions de rappel, ces fonction de rappel partagent les variables sharedMessage et previousEvent qui perdurent après l’appel à la fonction anonyme qui les a créées. Les variables sont donc mises à jour de manière asynchrone lors du traitement des évènements par les callback.

La technique parfois appelée *currying* peut être vue comme une manière simple de générer des familles de fonctions paramétrées. Supposons par exemple qu'une fonction f prenne comme

paramètres (x, y, z) et qu'un usage de cette fonction avec seulement x et y soit courante pour quelques cas seulement de valeurs de z différentes. Une manière d'écrire simplement une famille de fonctions paramétrée par seulement z consiste à créer une fonction utilisant f mais ayant seulement z comme paramètre. La fonction g suivante fait ce travail.

```
// Une fonction à "paramétrer"
   function f(x,y,z){
       return x+y+z;
   }

// la fonction qui génère des fonctions f "paramètrées" pour z
   function g(z){
       return function(x, y){
           f(x,y,z);
       }
   }
   var f0 = g(0); // création d'une fonction "f" pour z = 0
   var p = f0(3, 4); // p = 7
```

TABLEAUX – ARRAYS

En Javascript, les tableaux associatifs sont des objets, les tableaux décrits ici sont les tableaux à indices entiers.

CRÉATION

```
   var a = [];
   var b = [3, 5, 7, 11];
   var c = 10;
   var tc = [ c+1, c+3];
```

```
var d = new Array();
var e = new Array(10); //1 seul argument c'est la taille 10
var f = new Array(1,3,5,7);//plus d'un argument ce sont les elements
```

TAILLE ET PARCOURS ET TABLEAUX INCOMPLETS

La taille d'un tableau est un attribut modifiable. La modification de la taille du tableau permet de créer des tableaux à trous éléments (en affectant une taille plus grande que la taille réelle du tableau) ou de supprimer des éléments (en affectant une taille plus petite que la taille réelle du tableau)

```
   var a = [3, 5, 7, 11];
   a.length = 3;                // a devient [3, 5, 7] -> suppression de 11

   var b = [3, 5, , 11];
   console.log(b.length);       //4
   console.log(b[2]);           // undefined
```

PARCOURS

Itération dans l'ordre des indices avec `for`

```
   var b = [3, 5, 7, 11];
   var len = b.length;
   for(var i=0;i<len;i++){
       console.log(b[i]);
   }
```

Itération simplifiée avec `in`, mais, attention, ordre du parcours non garanti et pas de parcours des éléments undefined

```
   var b = [3, 5, , 11];
   for(var index in b){
       console.log(b[index]);
   }
```

I

Itération avec `forEach` à partir d'Ecmascript 5

```
   var b = [3, 5, 7, 11];
   var len = b.length;
   b.forEach(function(x){console.log(x);});
```

TABLEAUX MULTIDIMENSIONNELS

```
   //declaration
   var z = new Array(100); //un tableau de 100 lignes
   var len = b.length;
   for(var i=0;i<len;i++){
       z[i] = new Array(20); //chaque ligne a 20 colonnes
   }

   //initialisation
```

```
for(var i=0;i<len;i++){
    for(var j=0;j<z[i].length;j++){
          z[i][j] = i * j; //
}
```

QUELQUES MÉTHODES UTILES SUR LES TABLEAUX

`join(sep)` retourne la chaîne de caractère obtenues par la concaténation des éléments du tableau convertis en chaînes avec `sep` entre deux éléments successifs.

`reverse()` modifie le tableau en inversant l'ordre des éléments dans le tableau.

`sort(f)` modifie le tableau pour les faire apparaître du plus petit au plus grand dans l'ordre défini par f. Les éléments `undefined` sont mis en fin de tableau. Attention, s'il n'y a pas de fonction de tri, c'est l'ordre alphabétique qui est utilisé avec conversion de type éphémère (10 arrive avant 9 car "10" < "9" dans l'ordre alphabétique). Exemple:

```
var a = [9,10, 1];
a.sort();
console.log(a.join(","));//1,10,9
```

`concat()` retourne un nouveau tableau créé par la concaténation du tableau appelant et du (ou des) tableau(x) de l'argument.

`slice(i,j)` retourne un nouveau tableau créé par extraction des éléments de `i` inclus à `j` exclus du tableau appelant. Invoqué avec un seul argument `t`, slice extrait de 0 à `t` exclu, invoqué avec des arguments négatifs `-k` (`k>0`) les convertit en `n-k` où n représente la taille du tableau.

`splice(i, elts)` modifie le tableau pour y insérer et/ou en enlever des éléments. i est l'index à partir duquel les éléments sont ajoutés, elts les éléments à ajouter. Attention, si elts est omis les éléments à partir de i sont enlevés du tableau appelant et splice retourne les éléments enlevés. Attention à l'usage de cette méthode qui peut se montrer contre-intuitive au premier abord.
Exemple:

```
var a = [9,10, 1];
var b = a.splice(1);
console.log(a.join(","));
console.log(b.join(","));
```

`push(), pop()` modifie le tableau comme si c'était une pile. Permet d'ajouter (`push`) ou de retirer(`pop`) des éléments en fin de tableau. Permet de modifier un tableau en `first-in, last-out`.

`shift(),unshift()` modifie le tableau comme `push` et `pop` sauf que `shift` et `unshift` agissent sur des éléments en fin de tableau

`toString()` retourne la liste des éléments du tableau convertis en chaîne de caractère et séparés par des virgules.

Méthodes apparues à partir de EcmaScript 5: `map(), filter(), every(), some(), reduce(), reduceRight(),indexOf(),lastIndexOf()`

PROGRAMMATION ÉVÈNEMENTIELLE

Comme on l'a écrit précédemment, les interactions avec les utilisateurs sont gérées de manière asynchrone. Les scripts qui permettent de le faire sont gérés par des fonctions de rappel aussi appelées callback, gestionnaires d'évènements, ou fonctions auxiliaires.

PROPAGATION DES ÉVÈNEMENTS

Lorsqu'un utilisateur interagit avec les éléments d'une page web, des évènements sont émis en fonction des interactions. Ces évènements sont ensuite diffusés par le navigateur aux éléments du DOM susceptibles de les traiter. Cette diffusion comporte trois phases : la phase de capture qui consiste à diffuser les évènements de la racine du DOM vers la cible, (2) la sélection de la cible (élément le plus profond dans le DOM qui est concerné par l'évènement) et (3) la remontée de l'évènement de la cible vers la racine.

La phase 1 est appelée *capture*, la phase 2, *at target* et la phase 3, *bubbling*.

LIER ÉVÈNEMENTS - ÉLÉMENTS DU DOM - CALLBACK

Pour associer une ou plusieurs callback à chaque élément du DOM trois manières différentes sont disponibles:

- MODE HTML - écriture dans l'attribut HTML de l'élément -
  - exemple: `<div id="bouton" onclick="redraw()">`
- DOM 0 affectation aux méthodes de l'élément
  - exemple: `getElementById("bouton").click = redraw;`
- DOM 2 argument de méthodes génériques de gestion d'évènement
  - exemple: `getElementById("bouton").addEventListener("click",redraw, false);`

Cette dernière méthode qui utilise, `addEventListener(typeEvent, callbackName, useCapture)`, permet à elle seule d'associer plusieurs callbacks à un même élément. Le paramètre `useCapture` permet d'expliciter la phase durant laquelle l'évènement est traité par la callback passée en paramètre. Avec `true` comme valeur de ce paramètre, la callback est appelée dans la phase de capture, sinon elle est appelé dans la phase de bubbling.

Enfin, les callback associées par cette méthode peuvent être retirées dynamiquement en utilisant la fonction `removeEventListener(type, listener, useCapture), où l'argument` listener référence la callback utilisée par addEventListener (ce qui rend néanmoins l'exercice potentiellement impossible lorsque la callback est une fonction anonyme).

Selon les navigateurs, les variables auxquelles les callbacks ont accès peuvent différer. Selon le mode d'attachement de la callback à l'élément, aussi.

- mode html: dans certains navigateurs, la callback voit sa chaîne de portée augmentée automatiquement avec le formulaire englobant si la callback est associée à un élément de formulaire. Dans ces conditions, la callback a un accès direct à tous les attributs de l'élémént du formulaire ainsi qu'aux autres champs du formulaire recevant l'évènement.

  NB: Quelques limites: (1) rien n'assure automatiquement que les éléments de la chaîne de portée soient bien déjà chargés lorsqu'ils sont utilisés, (2) tous les navigateurs ne gèrent pas l'augmentation de chaîne de portée de la même manière, et (3) attention aux confusions/conflits de noms qui peuvent résulter de cette augmentation.

- DOM 0: la callback est considérée comme une méthode de l'élément qui reçoit l'évènement. Elle s'éxécute donc dans la portée de l'élément. Le recours à `this` dans la méthode est possible et référence alors l'élément qui a reçu l'évènement.

TYPOLOGIE DES ÉVÈNEMENTS

La typologie de référence des évènements de base a été spécifiée en plusieurs versions dont les principales sont:

- DOM-Level-2 (2000) http://www.w3.org/TR/DOM-Level-2-Events/events.html#Events-flow
- DOM-Level 3 (2016) ui events http://www.w3.org/TR/DOM-Level-3-Events/)

Cette typologie comporte actuellement:

User Interface Event Types	**Focus Event Types**	**Mouse Event Types**	**Wheel Event Types**
`load event`	`blur event`	`click event`	`wheel event`
`unload event`	`DOMFocusIn event`	`dblclick event`	
	`DOMFocusOut event`	`mousedown event`	**Keyboard Event Types**
	`focus event`		

`abort event`
`error event`
`select event`
`resize event`
`scroll event`

`focusin event`
`focusout event`

Drag-and-Drop Events
`dragstart, drag, dragend` - élément déplacé

`dragenter, dragleave, dragover` - élément cible

`mouseenter event`
`mouseleave event`
`mousemove event`
`mouseover event`
`mouseout event`
`mouseup event`

`keydown event`
`keypress event`
`keyup event`

A ces typologies de base s'ajoutent les spécifications/implémentations des évènements tactiles permettant de gérer des surfaces tactiles multi-points. Dans ce domaine, deux versions principales :

- Touch Events (2013) https://www.w3.org/TR/touch-events/
- Touch Events 2 (2017) https://w3c.github.io/touch-events/

L'ensemble des principaux attributs et méthodes communs à tout évènement est rappelé dans le tableau ci-dessous

ATTRIBUTS

`event.bubbles`	`true/false`
`event.currentTarget`	élément du chemin de propagation auquel l'évènement se trouve à gérer
`event.defaultPrevented`	`true/false`
`event.eventPhase`	renvoie un entier identifiant : 1. → `capture`, 2 → `at Target`, 3 → `bubbling`
`event.target`	`target` *de l'évènement* ie élément le plus profond des éléments du chemin de propagation de l'évènement
`event.timeStamp`	
`event.type`	
`event.which`	Renvoie la valeur Unicode de la touche dans un évènement clavier, qu'elle soit ou non un caractère.

CANVAS

Le canvas est un élément HTML prévu pour le dessin en Javascript dont le principe de dessin repose sur

- un contexte graphique qui représente les attributs graphiques du dessin en cours
- des objets graphiques représentés par des chemins, des formes prédéfinies ou des images
- les coordonnées du stylo qui représentent l'origine du repère pour le dessin en cours
- des fonctions de dessin en fil de fer ou en remplissage

Les dessins du canvas sont donc le résultat de suites d'instructions qui composent la création des objets graphiques avec des modifications des attributs du contexte graphique, des déplacements du stylo, et finalement des instructions de dessin.

CRÉATION DES FORMES LIBRES

beginPath() [...] closePath() ; lineTo() ; quadraticCurveTo(p1.x, p1.y,x,y) ; bezierCurveTo(p1.x, p1.y, p2.x, p2.y, end.x, end.y) ;

Exemple :

```
g2d.beginPath();
g2d.moveTo(100, 200);
g2d.lineTo(100, 400);
g2d.lineTo(200, 400);
g2d.bezierCurveTo(50, 100, 180, 10, 20, 10);
g2d.closePath() ;
```

Cet exemple crée deux lignes droites puis une ligne courbe de Bezier cubique (2 points de contrôles et deux extrémités), mais ne les dessine pas.

FONCTIONS DE DESSIN

```
stroke(), fill(), clearRect(), fillRect(), strokeRect()
```

MODIFICATION DES ATTRIBUTS GRAPHIQUES

```
fillStyle ;
strokeStyle ;shadowOffsetX ;shadowOffsetY ;shadowBlur ;shadowColor
```

Exemple :

```
g2d.beginPath();
g2d.moveTo(40, 0);
g2d.lineTo(0, -60);
g2d.lineTo(-40, 0);
g2d.lineTo(40, 0);
g2d.closePath();
g2d.lineWidth = 10;
g2d.shadowOffsetX = 5;
g2d.shadowOffsetY = 5;
g2d.shadowBlur = 10;
g2d.shadowColor = 'rgba(0, 0, 0, 0.3)';
g2d.endPath() ;
```

CRÉATION DE DÉGRADÉS POUR LE REMPLISSAGE OU LE STYLE DE TRAIT

```
var gradient = gc.createLinearGradient(0, 0, 100, 0);
gradient.addColorStop("0", "magenta");
gradient.addColorStop("0.5", "blue");
gradient.addColorStop("1.0", "red");
gc.fillStyle = gradient;
```

INSERTION D'IMAGES

```
zimg = new Image();
zimg.src = 'cannelle.jpg';
zimg.onload = function(){
      console.log("the img is "+zimg);
      gc.drawImage(zimg,300, 250);
}
```