


# Le langage Javascript (compléments)

HENALLUX

Technologies web

IG2 – 2015-2016

# Au programme...



Manipuler, au fil de quelques exemples concrets, quelques-unes des particularités les plus avancées du langage Javascript.

Entre autres :

- Flexibilité des définitions de fonctions (surcharge)
- Fonctions en tant qu'objets de premier ordre
- (Exceptions)
- Fonctions internes / fonctions imbriquées
- Closures

# Flexibilité des fonctions

## Restrictions sur la définition de fonction

- Si on redéfinit une fonction, seule la dernière définition est prise en compte : pas de surcharge via définitions multiples (comme en Java).

```
function f (entier) { ... }  
function f (chaîne) { ... }
```

*Seule la seconde définition comptera.*

- Pas de distinction de type sur les arguments.

# Flexibilité des fonctions

## Restrictions sur la définition de fonction

- **Pour simuler la surcharge** : définir tous les cas en une seule fois et différencier à l'intérieur de la fonction.

```
function f (x) {  
  if (x est entier)  
    { ... }  
  if (x est une chaîne)  
    { ... }  
}
```

# Flexibilité des fonctions

Quelques pistes pour des définitions de fonctions flexibles :

- JS permet d'appeler une fonction avec n'importe quel nombre d'arguments (quoi que dise la définition).  
*Exemple : 3 args dans la définition ; 2 args dans l'appel → ok*
- Utilisation de `||` pour les paramètres par défaut  
`anneeInscription = anneeInscription || 1;`  
*Mais attention aux cas des valeurs acceptables "falsy".*
- Utilisation du tableau des arguments (appelé `arguments`)
- Utilisation de `typeof/instanceof` pour déterminer le type des arguments.

# Exercices (voir aussi slide suivant)

Via la console, définissez et testez les fonctions suivantes...

- `coteEnPourcents(cote [,max])`  
donne la cote en pourcents (par défaut max = 20)
- `prixTVAC(prix [,tva])`  
donne le prix TVA comprise (par défaut tva = 21%)
- `min(x1,x2,x3,...,xn)`  
donne le minimum parmi toutes les valeurs données
- modifiez la fonction min pour qu'elle accepte également un tableau de valeurs : `min(valeurs)`
- `somme(x1,x2,x3,...,xn)` ou `somme(f,x1,x2,x3,...,xn)`  
calcule la somme des valeurs ou des résultats  $f(x_1), \dots, f(x_n)$

# Exercices (valeurs à tester)


```
coteEnPourcents(14)  
coteEnPourcents(15,30)
```

```
prixTVAC(100)  
prixTVAC(200,10)  
prixTVAC(150,0)
```

```
min(2,3,1)  
min(7,2,3,4)  
min(15,20,21,42,53)  
min([3,5,4])
```

```
somme(1,2,3)  
function carre(x) { return x*x; }  
somme(carre,1,2,3)  
somme(function (x) { return 2*x; }, 1, 2, 3, 4)
```

# Programmation fonctionnelle

- 
- Les fonctions comme objets de premier ordre  
(= objets qu'on peut ...
    - placer dans une variable,
    - utiliser comme arguments d'une fonction,
    - renvoyer comme résultat d'une fonction)

Ensuite : (Exceptions)



# Programmation fonctionnelle

## Fonctions en tant qu'objets de premier ordre / première classe

- On peut placer une valeur fonctionnelle dans une variable.  
`var carre = function (x) { return x*x; }`
- On peut passer une fonction comme argument d'une fonction.  
`tabNombres.forEach(affichecarre)`
- Une fonction peut renvoyer une fonction comme résultat.  
`function actionAfficher(msg) {  
 return function () { alert(msg); }  
}  
bt.onclick = actionAfficher("Vous avez cliqué ?");`

# Note sur les collections

*Attention...*

La plupart des fonctions prédéfinies (`getElementsByTagName`) et des groupes de valeurs prédéfinis (`arguments`) sont présentés sous la forme de collections et pas de tableaux simples.

Les fonctions comme "`forEach`" ne s'appliquent donc pas...

À moins de réaliser tout d'abord une transformation :

```
tab = Array.prototype.slice.call(collection, 0);
```

# Exercices

- Complétez pour afficher (console.log) la longueur de chaque string :

```
["un", "deux", "trois", "quatre", "cinq"].forEach(...)
```

- Complétez pour obtenir le tableau des éléments pairs :

```
[1,2,3,4,5,6,7,8,9,10].filter(...)
```

# Exercices

- Considérez la fonction suivante.

```
function reduce (f, val, tab)
{
  var res = val;
  for (var i = 0 ; i < tab.length ; i++)
    res = f(res,tab[i]);
  return res;
}
```

Que calculent...

- `reduce(function (x,y) { return x+y; }, 0, [1,2,3,4])` ?
- `reduce(function (x,y) { return x*y; }, 1, [1,2,3,4])` ?
- `reduce(function (x,y) { return x+y; }, "", ["un","deux","trois"])` ?
- Comment utiliser `reduce` pour calculer le maximum d'un tableau de nombres ?

# Exceptions

- 
- (pour information, hors matière)

Ensuite : **Fonctions internes/imbriquées**

# Exceptions (pour info)

- Le traitement des exceptions est semblable à celui de Java :

```
try { instructions } catch (ident) { instructions }  
[ finally { instructions } ]
```

```
try { pas_def(3); }  
catch (erreur) {  
    msg = "Erreur !\n";  
    msg += erreur.message + "\n";  
    alert(msg);  
}
```

Affichage

```
Erreur !  
pas_def is not defined
```

- Lancer une exception : **throw** *expr*  
*Note. On peut utiliser n'importe quel type de valeur.*

# EXCEPTIONS

- Voir aussi
- <https://developer.mozilla.org/fr/docs/JavaScript/Reference/Instructions/try...catch>
- <https://developer.mozilla.org/fr/docs/JavaScript/Reference/Instructions/throw>

# Fonctions internes/imbriquées

- 
- Des fonctions définies à l'intérieur d'autres fonctions...

Ensuite : **Closures**

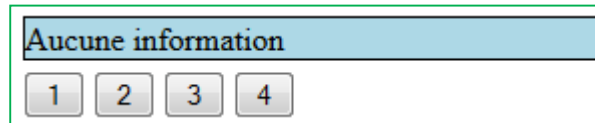


# Fonctions internes/imbriquées

Javascript permet de définir des fonctions "locales" à l'intérieur d'autres fonctions.

Ces fonctions internes ont accès aux variables locales de la fonction englobante.

Exemple :



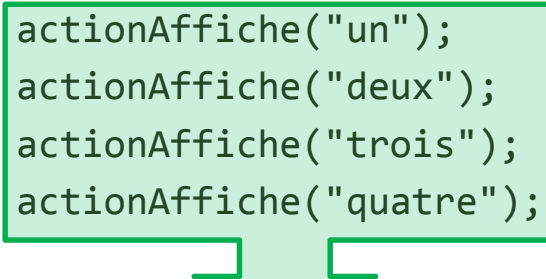
*faire en sorte qu'un clic sur un bouton affiche "un", "deux", "trois" ou "quatre" dans la barre d'information.*

# Fonctions internes/imbriquées

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <style>
      #cadre {
        background-color: lightblue;
        border: 1px solid black;
        margin-bottom: 5px;
      }
    </style>
  </head>
  <body>
    <div id="cadre">Aucune information</div>
    <button id="but1">1</button>
    <button id="but2">2</button>
    <button id="but3">3</button>
    <button id="but4">4</button>
  </body>
</html>
```

# Fonctions internes/imbriquées

```
window.onload = init;  
function init () {  
    document.getElementById("but1").onclick = actionAffiche("un");  
    document.getElementById("but2").onclick = actionAffiche("deux");  
    document.getElementById("but3").onclick = actionAffiche("trois");  
    document.getElementById("but4").onclick = actionAffiche("quatre");  
}
```



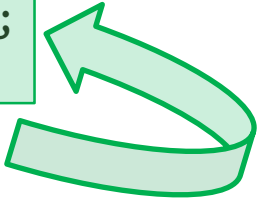
Ces éléments doivent être des fonctions sans arguments dont le code correspond à l'action à effectuer.

Donc, des " `function () { ...afficher msg dans le cadre... }` "

Donc, `actionAffiche` doit renvoyer une telle fonction !

# Fonctions internes/imbriquées

```
function actionAffiche (msg) {  
    var cadre = document.getElementById("cadre");  
    function affiche () {  
        cadre.innerHTML = msg;  
    }  
    return affiche;  
}
```



*Note : la fonction affiche fait appel à certaines variables locales à actionAffiche, à savoir msg et cadre !*

## Exercice

Modifiez le code pour que, lors d'un clic sur un des boutons, la couleur de fond du cadre change également : lightblue pour 1, lightgreen pour 2, yellow pour 3 et orange pour 4.

*Attention : solution sur la page suivante !*

# Exercice (solution)

```
function actionAffiche (msg, col) {
    var cadre = document.getElementById("cadre");
    function affiche () {
        cadre.innerHTML = msg;
        cadre.style.backgroundColor = col;
    }
    return affiche;
}

window.onload = init;
function init () {
    document.getElementById("un",
        "lightblue");
    document.getElementById("deux",
        "lightgreen");
    document.getElementById("trois",
        "yellow");
    document.getElementById("quatre",
        "orange");
}
```

# Introduction aux closures

- 
- Via 2 exercices

Ensuite : **les closures**

# Exercice introductif 1

*Conseil : créez le code demandé sur Notepad++ puis copiez/collez-le dans la console pour l'exécuter.*

- Écrivez une boucle (sur la variable `i` allant de 0 à 10) qui va afficher la valeur de `i` dans la console (via `console.log`).
- Modifiez la boucle précédente pour retarder les affichages (via `setTimeout`) : un affichage toutes les secondes.

(comparez votre solution au code du slide suivant)

# Exercice introductif 1 (solution)

- Première solution (qui ne fonctionne pas) :

```
for (var i = 0 ; i <= 10 ; i++)  
    setTimeout(function () { console.log(i); }, i * 1000);
```

- Seconde solution (qui fonctionne... mais pourquoi ?)

```
function actionAffiche(i) {  
    return function () { console.log(i); };  
}
```

```
for (var i = 0 ; i <= 10 ; i++)  
    setTimeout(actionAffiche(i), i * 1000);
```



# Exercice introductif 2

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <script>
      for (var i = 1 ; i <= 4 ; i++)
        document.write("<button id='but" + i + "'>XXX</button>");
    </script>
    <button id="bGo">go</button>
  </body>
</html>
```



En cas de clic sur "go", le texte des quatre premiers boutons doit devenir "Afficher 1", "Afficher 2", "Afficher 3" et "Afficher 4". Modifiez le code en conséquence !



# Exercice introductif 2 (solution)

```
function go() {  
  for (var i = 1 ; i <= 4 ; i++)  
    document.getElementById("but" + i).innerHTML = "Afficher " + i;  
}
```

```
<button id="bGo" onclick="go();">go</button>
```

Modifiez le code pour que, lors du clic sur "go", on associe également une action à chacun des quatre premiers boutons.


L'action de ces boutons sera d'afficher le chiffre correspondant (en utilisant la fonction alert). Par exemple, un clic sur le bouton "1" affichera "1"; un clic sur le bouton "2" affichera "2" ; et ainsi de suite.

# Exercice introductif 2 (solution ?)

```
function go() {  
  for (var i = 1 ; i <= 4 ; i++)  
  {  
    var but = document.getElementById("but" + i);  
    but.innerHTML = "Afficher " + i;  
    but.onclick = function () { alert(i); };  
  }  
}
```

Que se passe-t-il lors d'un clic sur un bouton ?

# Closures

- 
- Pourquoi certaines solutions aux exercices introductifs ne fonctionnent-elles pas ?

# Persistance et closure

- Principe du **garbage collector** : si on a besoin de libérer de la mémoire,
  - on repère toutes les données qui ne sont plus accessibles
  - on libère l'espace qui leur est consacré

- Exemple :

```
function somme (x) {  
  var s = 0;  
  for (var i = 1 ; i <= x ; i++)  
    s += i;  
  return s;  
}  
somme(7);
```

Lors d'un appel, on réserve de la place pour les variables locales s et i.

L'espace réservé pour s et i peut être libéré une fois l'appel terminé.

# Persistance et closure

- Autre exemple :

```
function decodeHHMM (hhmm) {  
  var mm = hhmm % 100;  
  var hh = (hhmm - mm) / 100;  
  var obj = { heures : hh, minutes : mm };  
  return obj;  
}
```

Lors d'un appel, on réserve de la place pour les variables locales mm, hh et obj.

```
debutCours = decodeHHMM(1310);  
var msg = "Le cours debute a " + debutCours.heures;  
msg += ":" + debutCours.minutes + ".";  
console.log(msg);
```

Après l'appel, on ne peut pas libérer l'espace occupé par obj car debutCours s'y réfère encore !

- Ici, on ne peut pas libérer l'espace réservé pour "obj" car on peut encore y accéder même après la fin de l'exécution !

# Persistance et closure

- Encore un autre exemple :

```
function actionAfficheMaj (msg) {  
    var msgMaj = msg.toUpperCase();  
    function affiche () { alert(msgMaj); }  
    return affiche;  
}
```

Idem avec la syntaxe alternative :

```
return function () { alert(msgMaj); };
```

```
bDisBonjour.onclick = actionAfficheMaj("bonjour");  
bDisAurevoir.onclick = actionAfficheMaj("au revoir");  
bDisHello.onclick = actionAfficheMaj("hello");
```

- Ici, on ne peut pas libérer l'espace réservé pour "msgMaj" car on peut encore y accéder (via un appel à la fonction affiche) même après la fin de l'exécution de actionAfficheMaj !

# Persistance et closure

- Encore un autre exemple :

```
function actionAfficheMaj (msg) {  
    var msgMaj = msg.toUpperCase();  
    function affiche () { alert(msgMaj); }  
    return affiche;  
}
```

- La fonction actionAfficheMaj ne retourne pas seulement comme résultat une fonction (la fonction affiche).
- Elle renvoie la fonction + un [contexte](#).
  - Le contexte sera utilisé pour exécuter la fonction.
  - C'est dans ce contexte qu'on retient la valeur de la variable msgMaj qui est utilisée dans la fonction affiche.




# Persistance et closure

- Dans certains cas, le contexte d'une fonction (ou une partie de celui-ci) **persiste** même **après la fin de son exécution**.
- Ce qui persiste, c'est tout ce qu'on peut encore utiliser / à quoi on peut encore faire référence par la suite !
- Un cas particulier (comme l'exemple précédent) :
  - une variable locale (ex : msgMaj) doit persister
  - parce qu'une fonction locale (ex : affiche) y fait référence
  - et cette fonction locale est exportée (via un « return »).
- C'est ce qu'on appelle une **closure** (clôture/fermeture).

# Persistance et closure

- Exemple de closure

```
function ajoute (x) {  
  function avec (y) { return x + y; } Fonction locale exportée  
  return avec;  
}
```



```
var plus2 = ajoute(2);  
alert("5 + 2 = " + plus2(5));  
alert("7 + 2 = " + plus2(7));  
var plus4 = ajoute(4);  
alert("5 + 4 = " + plus4(5));  
alert("7 + 4 = " + plus4(7));
```

Dans "plus2", on doit retenir  
que x vaut 2 !

Dans "plus4", on doit retenir  
que x vaut 4 !

# Closure et partage de scopes

- Que se passe-t-il si deux closures entraînent la persistance du même scope ? On ne garde qu'une seule copie du scope !

Exemple théorique :

```
function capsule(nom) {  
  function set(nvNom) { nom = nvNom.toUpperCase(); }  
  function dit() { alert ("Je suis " + nom); }  
  return {change : set, affiche : dit};  
}
```

```
var chien = capsule("Fido");  
chien.affiche();  
chien.change("Boule");  
chien.affiche();
```

chien.affiche et  
chien.change partagent le  
même contexte !

# Closure et partage de scopes

- Ce partage de scope peut être pratique...

Exemple :

- 4 cases
- Si on clique sur une case (pour la sélectionner) puis sur un bouton, changer la couleur de la case (selon le bouton).



1<sup>re</sup> solution (sans closure) : utiliser une variable globale pour stocker la case "active".

2<sup>e</sup> solution : stocker la case "active" comme variable locale dans un scope qui sera partagé par toutes les fonctions.

# Closure et partage de scopes

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <style>
      td { height: 12px; width: 30px; text-align: center;
          border: 2px solid lightgray; }
      td.focus { border: 2px solid black; }
    </style>
  </head>
  <body>
    <table><td>1</td><td>2</td><td>3</td><td>4</td></table>
    <button id="bBleu">Bleu</button>
    <button id="bVert">Vert</button>
    <button id="bRouge">Rouge</button>
  </body>
</html>
```



# Closure et partage de scopes

```
window.onload = function () {  
    var celluleFocus = null;
```

celluleFocus : lien vers la cellule active,  
partagé par toutes les fonctions internes

```
function clic () {  
    if (celluleFocus) celluleFocus.classList.toggle("focus");  
    celluleFocus = this;  
    celluleFocus.classList.toggle("focus"); }  
}
```

```
function fondBleu () {  
    if (celluleFocus) celluleFocus.style.backgroundColor = "blue"; }  
function fondVert () {  
    if (celluleFocus) celluleFocus.style.backgroundColor = "green"; }  
function fondRouge () {  
    if (celluleFocus) celluleFocus.style.backgroundColor = "red"; }  
}
```

```
var cellules = document.getElementsByTagName("td");  
for (var i = 0 ; i < cellules.length ; i++)  
    cellules[i].onclick = clic;  
document.getElementById("bBleu").onclick = fondBleu;  
document.getElementById("bVert").onclick = fondVert;  
document.getElementById("bRouge").onclick = fondRouge;  
}
```

# Closure et partage de scopes

- Autre version (1 seule fonction pour toutes les couleurs)

```
function actFond (col) {  
  return function () {  
    if (celluleFocus)  
      celluleFocus.style.backgroundColor = col; };  
}
```

...

```
document.getElementById("bBleu").onclick = actFond("blue");  
document.getElementById("bVert").onclick = actFond("green");  
document.getElementById("bRouge").onclick = actFond("red");
```

# Closure et partage de scopes

Et parfois... c'est un désavantage... (cfr exercices introductifs)

```
function go() {  
  for (var i = 1 ; i <= 4 ; i++)  
  {  
    var but = document.getElementById("but" + i);  
    but.innerHTML = "Afficher " + i;  
    but.onclick = function () { alert(i); };  
  }  
}
```

Toutes ces fonctions partagent le même scope (et la même variable i).

Le scope évolue au cours de la boucle ; au final, i = 5.  
C'est dans ce scope final que les fonctions sont exécutées.

Donc, tous les boutons affichent "5" !



# Closure et partage de scopes

Une solution : faire en sorte que le nombre affiché ne soit pas une variable partagée par toutes les fonctions.

```
function go() {  
    function actAffiche (k) {  
        return function () { alert(k); }  
    };  
    for (var i = 1 ; i <= 4 ; i++)  
    {  
        var but = document.getElementById("but" + i);  
        but.innerHTML = "Afficher " + i;  
        but.onclick = function () { alert(i); } actAffiche(i);  
    }  
}
```

# Exercices (1/2)

But : fonction `fadeOut(elem)` qui fait disparaître un élément HTML.

- [DOM] faire décroître `elem.style.opacity` (1 = visible, 0 = transparent) de 0.1 cinq fois par secondes.
- Définir une **fonction interne action ()** qui va
  - tester si l'opacité de l'élément est  $> 0.1$
  - (si c'est vrai) réduire l'opacité de 0.1 et se rappeler 1/5 de seconde plus tard
  - (si c'est faux) mettre l'opacité à 0 [en cas d'erreurs d'arrondis]
- Dans `fadeOut`
  - initialiser `elem.style.opacity` à 1 ;
  - appeler `action`.
- [Test] ouvrir une page web, clic droit sur un élément pour obtenir son identificateur, le placer dans une variable via la console puis exécuter `fadeOut` sur cette variable.

# Exercices (2/2)

But : en évitant de répéter du code, faire en sorte que les boutons + et – permettent de modifier les valeurs correspondantes.

Force : 10	<input data-bbox="1551 375 1605 415" type="button" value="+"/>	<input data-bbox="1624 375 1678 415" type="button" value="-"/>
Dextérité : 10	<input data-bbox="1551 432 1605 472" type="button" value="+"/>	<input data-bbox="1624 432 1678 472" type="button" value="-"/>
Constitution : 10	<input data-bbox="1551 489 1605 529" type="button" value="+"/>	<input data-bbox="1624 489 1678 529" type="button" value="-"/>
Intelligence : 10	<input data-bbox="1551 546 1605 586" type="button" value="+"/>	<input data-bbox="1624 546 1678 586" type="button" value="-"/>
Sagesse : 10	<input data-bbox="1551 604 1605 644" type="button" value="+"/>	<input data-bbox="1624 604 1678 644" type="button" value="-"/>
Charisme : 10	<input data-bbox="1551 661 1605 701" type="button" value="+"/>	<input data-bbox="1624 661 1678 701" type="button" value="-"/>

- Partir du code HTML du slide suivant.
- Définir une fonction `prepareTrait(span, bPlus, bMoins)` qui reçoit les éléments HTML concernés par un trait et leur associe des événements (la valeur du trait sera conservée dans une variable locale partagée par les fonctions internes).
- Dans `window.onload`, écrire une boucle qui va préparer les éléments pour chacun des 6 traits.

# Exercices (2/2)

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <table>
      <script>
        var traits = ["Force", "Dextérité", "Constitution", "Intelligence",
"Sadess", "Charisme"];
        for (var i = 0 ; i < traits.length ; i++)
        {
          document.write("<tr><td>" + traits[i] + " : <span id='s" + i +
"'>10</span></td>");
          document.write("<td><button id='bPlus" + i + "'>+</button></td>");
          document.write("<td><button id='bMoins" + i + "'>-
</button></td></tr>");
        }
      </script>
    </body>
  </html>
```

# Solution (1)

```
function fadeOut (elem) {
  elem.style.opacity = 1;
  function action () {
    if (elem.style.opacity >= 0.1) {
      elem.style.opacity -= 0.1;
      setTimeout(action, 200);
    } else {
      elem.style.opacity = 0;
    }
  }
  action();
}
```

Après l'exécution de fadeOut, son contexte reste persistant car, via setTimeout, on conserve une référence à la fonction action, qui elle-même référence elem.

# Solution (2)

```

<script>
function prepareTrait (span, bPlus, bMoins)
{
    var valeur = 10;
    function maj () { span.innerHTML = valeur; };
    function plus () { valeur++; maj(); }
    function moins () { valeur--; maj(); }
    bPlus.onclick = plus;
    bMoins.onclick = moins;
}

window.onload = function () {
    for (var i = 0 ; i < traits.length ; i++)
        prepareTrait(document.getElementById("s" + i),
            document.getElementById("bPlus" + i),
            document.getElementById("bMoins" + i));
}
</script>

```