Support Initiation JS

partie 1 (ex01)

Préambule et définitions

Déf de la programmation :

C'est l'écriture d'un programme,

qui est une suite d'instructions plutôt simples, données à la machine pour qu'elle exécute 1 ou plusieurs tâches complexes voire très complexes comme des applications (web, mobile), moteur 3D, traitement de sons, d'images ou de vidéos, etc.
Ces instructions sont écrites dans un langage à la syntaxe rigoureuse qu'on appelle langage de programmation.

J Script est un langage de programmation ; universel (tout type d'application) comme les autres. Sa particularité ? Il s'exécute dans (est interprété par) un navigateur (chrome, firefox, etc) contrairement à ceux qui s'exécutent directement sous Windows, Mac Os ou Linux ; ce qui impose l'absence de certaines fonctionnalités habituelles pour raison de sécurité. Mais cela n'enlève pas son caractère fondamentalement universel. A contrario Css et Html sont puissants mais ne sont pas universels ; ils ne décrivent que le visuel avec quelques capacités d'interactivité pour Css.

Noter qu'il existe une version "serveur" de JS qui est Node JS (voir ateliers Silex) qui s'exécute directement sur la machine.

Conclusion :
On va écrire du Javascript avec notre éditeur et on testera dans notre navigateur...

légendes pour la suite :

T:	=> Théorie
P:	=> Pratique
Mot en italique	=> mot ou expression typique de JS
texte en bleu ; :	=> code JS

Les variables

P:
Download et install. de https://goo.gl/EPTNJ5 soit https://github.com/flashline/Atelier-JS-basic
Ouvrir ./ex01/index.html
<script src="index.js"></script> suffit à exécuter notre programme JS

Ouvrir ./ex01/index.js

T:
Un programme manipule essentiellement des données ou valeurs que le programmeur stocke dans des **variables.**
Les variables peuvent être comparées à des contenants (des tiroirs, boites, bouteilles par ex.) auxquels ont donne un **nom** et dont le contenu peut **varié.**
On peut dans certains cas plutôt se les représenter comme une adresse -à laquelle on donne aussi un nom- et qui permet l'accès à une donnée dans ce cas plusieurs variables peuvent pointer vers une valeur identique. (Nous verrons cela dans le § sur les objets).

Contrairement à un tiroir ou une bouteille, on ne transfert pas les contenus. Quand on copie le contenu d'une var x vers une var y, la var x garde son contenu intacte.

P:
Création de variables
var prenomEleve ;
Détail :
var => mot clé imposé par JS
; => termine chaque instruction.
prenomEleve => c'est le nom choisi par le programmeur
certains caractères sont interdits (espace, +, -, etc).
usages :
On choisit (contrairement au maths) des noms "parlants".
Les signes a à z , A à Z et 0 à 9 sont recommandés.
Commencer par une minuscule.
Les majuscules remplacent les espaces (écriture en 'chameau').

T:
Il y a plusieurs types de valeur donc de variable.
les basiques JS sont :

- Les chaînes de caractères alpha-numérique (soit du texte)
  ex : *106 av. Ph Auguste75011* est une chaîne
- Les nombres (entier, flottants)

P:
assignation d'une valeur alpha-numérique (type JS String)
prenomEleve = "Jean-Michel" ;
le signe égal => ne signifie pas 'égale' au sens mathématique mais 'mettre l'expression à droite dans la variable à gauche'.
Les valeurs alpha/num sont tjrs entre " " (sinon dans ce cas, JS comprendrait : contenu de la var *Jean* moins celui de *Michel* !!)

assignation d'une valeur numérique (type JS Number)
var ageEleve = 26 ;
Noter qu'on peut créer et assigner dans une même instruction.
Les nombres sont sans " ".
On choisit le type nombre (type JS *Number)* quand on sait qu'on aura besoin de faire des calculs ou des tris numériques. Ex : 26<100 ; mais "26" > "100" en tri lexicographique.

Petite question ?

Si je crée :
    var codePostal ;
dois-je choisir
    codePostal=18270 ;
ou
    codePostal="18270" ;

Solution : codePostal="18270" ; car
  1/ codePostal = 03120 ; donnerait 3120 (ou pire ; faites le test)
  2/ on ne fait généralement pas de calcul avec les codes postaux.

pour tester :
document.write("Mon prénom est "+prenomEleve) ;
document.write(" et j'ai "+ageEleve+" ans.
Mon code postal est "+codePostal) ;
on reverra plus loin les notions expliquant document.write() .
Pour l'instant on se dit juste que ça permet de tester notre programme dans la page web.
Les '+' ne font pas ici d'addition.

--- tester index.html ---

```
Autres manipulations avec les valeurs et variables
document.write("<br/><br/>--- Concaténation avec signe +");
var nomEleve = "Delettre";
// concaténation
var nomComplet = prenomEleve+" "+nomEleve;
// le <br> ou <br/> c'est du html et signifie passage à la ligne.
document.write("<br>Nom complet "+nomComplet) ;

//variables Number et calcul
var ageEleve1 = 26 ;
var ageEleve2 = 56 ;
var ageEleve3 = 12 ;
var ageEleve4 = 32 ;
var ageEleve5 = 18 ;
var ageMoyen=(ageEleve1+ageEleve2+ageEleve3+ageEleve4+ageEleve5)/5;
// les opérateurs de calcul basiques sont + - / *
document.write("<br>Moyenne des ages : "+ageMoyen+" ans ") ;

// expression typique avec les 4 opérateurs
document.write("<br/><br/>--- Calcul avec les 4 opérateurs");
var ht=200; var acompte=80; var tva=20;
var aPayer=ht+(ht*tva/100)-acompte ;
// La priorité des * et / sur les – et +, évite certaines parenthèses et permet d'écrire.
aPayer=ht+ht*tva/100-acompte ;
// si on met ht en facteur on doit alors mettre des parenthèses
//aPayer=ht*(tva/100+1)-acompte  ;
document.write("<br>Net à payer "+aPayer+" €" ) ;
```

--- tester index.html ---

Les tableaux soit les vars de type JS Array ou variables indicées

T:
Ils sont utilisés entre autres dans les cas suivants :
un grand nombre de valeurs (généralement de même catégorie) ou
un nombre indéterminé de valeurs à stocker...
le principe est de regrouper la liste des valeurs dans une même variable
et de donner à chaque valeur une position de 0 à N appelée indice.

P:
création array vide

```
var eleves = [ ] ;
// on va charger notre tableau
eleves[0] = "jeanmi";
eleves[1] = "martine";
eleves[2] = "camel";
eleves[4] = "robert";
eleves[3] = "nicolas";

document.write("<br><br>--- Les tableaux (type js Array) :<br>");
```

faire copier/coller de : for (var i=0;i<eleves.length;i++)

```
for (var i=0;i<eleves.length;i++) document.write(eleves[i]+"<br/>") ;
```

On expliquera plus loin cette structure de boucle 'for'
ici c'est juste pour permettre le test ; juste savoir que la var i va varier de 0 à 4 et que le code après le for (...) sera exécuté 5 fois pour les 5 élèves

variante pour charger un Array en ajoutant la donnée en fin de tableau.

```
eleves.push("Joëlle");
eleves.push("René");
eleves.push("Ginette");
eleves.push("Fernande");
```

Quand JS exécute la création du tableau à vide avec *eleves=[ ]*, il associe automatiquement à *eleves* un ensemble de fonctions et variables typiques d'un Array.
On verra plus tard ces notions quand on abordera les Function et Object JS natifs
pour l'instant admettons par exemple que :
.push() est un moyen d'ajouter un élément en fin de tableau et
.length est une var ou propriété qui contient le nombre d'éléments

```
document.write("<br>Variante :<br>");
```

copier/coller ligne du dessus entièrement :

```
for (var i=0;i<eleves.length;i++) document.write(eleves[i]+"<br/>");
// vous verrez 9 lignes, les 5 élèves + les 4 ajoutés par push.

document.write("Nombre d'élèves : "+eleves.length);
```

--- tester index.html ---

Utilisation des fonctions et propriétés natives

T: appel de fonctions **fournies par js** (api JS)

Une fonction JS exécute une tache précise ; retourne parfois un résultat.
elle a un nom (similaire à un nom de variable mais choisi par JS) suivi d'une liste de paramètres d'aucun à n ; séparés par une virgule)

P:

```
alert("bonjour "+eleves[3]);
document.write("<br/><br/>--- Les fonctions (type JS Function) ");
// En JS, toutes les fonctions et propriétés sont rattachées à une variable, un objet.
document.write("<br>je suis la fonction write de l'objet document");
// L'objet lui-même est rattaché à un objet de niveau supérieur sauf l'objet window qui est
// le niveau le plus élevé.
// Même les fonctions comme alert() ou confirm() devraient s'écrire :
window.alert("je suis la fonction alert de l'objet window (le plus élevé dans la hiérarchie) ");
window.document.write("<br>je suis la fonction write() de document qui lui-même
appartient à l'objet window");
// A noter que c'est vrai aussi pour nos variables
document.write("<br/>window.ageEleve : "+window.ageEleve);
// les niveaux peuvent être assez nombreux ; ex :
window.document.body.style.backgroundColor="red" ;
```

signifie : l'élement *body* de *document* de *window* aura le *style* (css background-color) changé en rouge.

--- tester index.html ---

T:

Comme vu précédemment avec les *Array*, dès qu'on créé une variable, elle possède un ensemble de fonctions natives JS selon sont type.

P: On va voir par ex :

La fonction String.substr() qui extrait une sous chaîne d'une chaîne en partant de la position donnée par le 1er paramètre. La longueur de la sous-chaîne est donnée par le 2ème paramètre.

```
document.write("<br/><br>--- Fonction String.substr()  ");
var chaine="abcdefghij" ;
var sousChaine=chaine.substr(3,2);  //
document.write("<br>"+sousChaine); //  'de'
```

la propriété length de String qui contient le nombre de caractères.

```
document.write("<br>longueur de chaine = "+chaine.length);
```

la fonction Array.pop() qui est le contraire de Array.push(), qui retire et renvoie le dernier élément d'un tableau.

```
document.write("<br/><br/>--- Fonction Array.pop() ");
document.write("<br><br>"+eleves.pop()); // Fernande
document.write("<br><br>"+eleves.pop()); // Ginette
```

la fonction parseInt (window,parseInt) qui supprime la partie décimale d'un nombre.

```
document.write("<br>Moyenne des ages "+parseInt(ageMoyen)+" ans ") ;
// A noter que les fonctions sont des variables comme les autres (de type JS Function)
var affich= alert;
affich("J'ai mis alert() dans la variable 'affich'. Je  l'exécute par affich() ");
```

--- tester index.html ---

Créer ses propres fonctions

T:
Une fonction programmeur est un bout de code avec un nombre d'instructions plus ou moins important. Comme les natives elles peuvent avoir des paramètres et retourner un résultat ou pas.
Exemple de petite fonction très pratique.
Dans index.html on a . "header" est l'identifiant unique de cette div.
Pour récupérer une balise html (div,button,etc) dans un *Element* JS, la syntaxe de la fonction est un peu longue : *document.getElementById(id_de_l'élément)*.
On va donc créer une fonction qui aura par exemple comme nom un simple $.
Cette fonction recevra 1 seul argument (paramètre, variable) en entrée : l'id de l'élément html.
P:

```
// Nous allons déclarer (créer) notre fonction avec les 2 syntaxes possibles.
// 1ère syntaxe
function $ (id) {
        return document.getElementById(id);
}
```

T:
//détail :
function est le mot-clé imposé par JS
$ est le nom qui nous permettra d'appeler (d'exécuter) la fonction par $("header")
Entre les *()* on a 0 ou plusieurs arguments séparés par des virgules, pour passer des valeurs à la fonction. Dans notre exemple :
l'appel par $("header") mettra auto. la valeur "header" dans l'argument *id* de la fonction.
A noter qu'il y a pas besoin d'initialiser les arguments par *var*
Le corps de la fonction est entre des *{ }*
Si la fonction doit retourner un résultat le mot clé *return* suivi d'une expression est nécessaire.

```
// 2ème syntaxe
  // var $ = function (id) { return document.getElementById(id) } ;
  // ou idem :
  // var $ = document.getElementById ;
  // pour tester quelque soit la syntaxe :
  $("header").innerHTML="Je mets ce que je veux dans le sous-titre";
  $("header").style.color="red";
  $("header").innerHTML+="<br>avec la couleur qui me plaît.";
  // 2ème exemple avec 2ème syntaxe:
  affich = function (msg) {document.write("<br>"+msg)};
  affich ("--- Exemple de ma fonction affich()");
  affich ("Hello, dear taxes payers");
```

Note : La 1ère syntaxe qui crée la fonction dès que le code est chargé, n'est pas identique à la 2ème qui elle, crée la fonction qu'au moment de l'exécution de la ligne. Autrement dit, seule la 1ère syntaxe peut être écrite à n'importe quel endroit du programme et être appelée à n'importe quel autre endroit.

```
// Dans la partie II on fera d'autres fonctions plus longues.
```

--- tester index.html ---

Les boucles et l'exécution conditionnelle

T:
le *for* => répétition de n fois, un bloc d'instructions
le *while* => exécution d'un même bloc tant que la condition est vraie
le *if* => exécution d'un bloc ou d'un autre selon qu'une condition est vraie ou pas

P:
syntaxe du *for* :

```
affich ("<br>--- le bloc FOR");
var max=5;
affich ("affiche les "+max+" 1ers élèves :");
for (var i=0;i<max;i=i+1) {
        affich ( eleves[i]+" a le n° "+(i+1));
}
```

T:
Détail

- for => mot clé JS
- *var* => si la variable existe déjà le *var* n'est pas nécessaire
- i=0; => i démarrera avec la valeur de l'expression suivant le '='
- les {} => encadre le bloc d'instructions
- i=i+1 => à chaque fin du bloc i sera incrémenté d'un pas donné. (1 dans cet ex.)
- i<max => si cette condition devient fausse, alors JS sort de la boucle.
  les principaux opérateurs de comparaison sont <,>,==,!=,<=,>=

P:
Autre ex. pour afficher les multiples de 3, de 6 à 30 :

```
affich ( "Les multiples de 3, de 6 à 30 avec FOR ");
for (var i=6;i<31;i=i+3) {
        affich ( i);
}
```

Idem avec *while*

```
var i=6;
affich ( "<br>--- Les multiples de 3, de 6 à 30 avec WHILE ");
while (i<31) {
        affich (i);
        i+=3; // i=i+3 ;
}
```

T:
Détail du while:

- while : le mot clé
- entre les ( ) : Une condition qui tant que vraie, le bloc entre { } est exécuté.
- i++; : Généralement une instruction à l'intérieur du bloc doit modifier le résultat du test conditionnel. C'est facultatif en théorie mais ça donne une boucle sans fin !! Une autre instruction avant le *while* doit généralement assigner une valeur à la variable testée (var i=6;)

--- tester index.html ---

// Syntaxe du *if ... else*
P:
```
affich ( "<br>--- Les branchements conditionnels avec if...else ");
var i=10;
if (i%5==0) {
        affich ( ""+i+" est un multible de 5.");
}
else  {
        affich ( ""+i+" N'est PAS un multible de 5.");
}
```
T:
Note : % est un opérateur qui renvoi le reste de la division : i/5
Détail du if :
- if et else : sont les mots clé
- entre les ( ) : condition qui est vraie ou fausse
  - si vraie on exécute le bloc entre { } suivant le *if*
  - si fausse on exécute le bloc entre { } suivant le *else*
- Note : *if* n'est pas obligé d'être couplé à un *else ;* mais c'est assez fréquent.

P:
// on ajoute un while pour rendre l'exemple (un peu) plus intéressant.
```
i=10;
while (i<30) {
        if (i%5==0) {
                affich ( ""+i+" est un multible de 5.");
        }
        else  {
                affich ( ""+i+" N'est PAS un multible de 5.");
        }
        i++; // i=i+1
}
```

Note : Il n'est pas obligatoire qu'un *if...else* soit couplé avec un *while* c'est même assez rare.

--- tester index.html ---

Les vars de type Object (objet en fr).

T:

Déf simple :

Un objet est une variable qui regroupe en sont sein :

de 0 à plusieurs variables (propriétés, attributs)

de 0 à plusieurs fonctions

de 0 à plusieurs autres objets.

un objet peut aussi être vide

De même qu'un tableau regroupait plusieurs éléments en leur donnant un indice différent, un objet regroupe plusieurs éléments (membres) en leur donnant un nom unique dans l'objet.

P:

```
// Syntaxe
affich ( "<br>--- Les vars de type Object ");
var eleve={}; // C'est tout ! l'objet est crée à vide avec tout plein de membres natifs.

// Je remplis mon objet avec la syntaxe pointée : <nom de l'objet>.<nom la variable>
eleve.prenom="Jean-Michel";
eleve.nom="Delettre";
eleve.age=45;
eleve.email="info@pixaline.net";
// on écrit pour tester :
affich (eleve.prenom+" "+eleve.nom+" est agé de "+eleve.age+" ans."+"<br/>"+
                                    "Son adresse mail est "+eleve.email+".<br/><br/>");
//
// Nous allons créer une fonction que nous ajouterons à chaque objet 'eleve'
var affichCetEleve = function() {
        affich ("<br>"+this.prenom+" "+this.nom+" est agé de "+this.age+" ans."+"<br/>"+
                                    "Son adresse mail est
"+this.email+".<br/>------------------------------<br/>");
};
eleve.affich=affichCetEleve;
```

T:

A l'exécution, le mot clé *this* aura comme valeur l'objet précis à partir duquel on appellera la fonction affich() ; ex : eleve.affich() ;,

Attendre l'exemple à venir page suivante !...

```
// On a donc un objet avec 4 propriétés et 1 fonction.
```

P:

```
eleve.affich() ;
```

--- tester index.html ou attendre page suivante ---

On va utiliser le tableau 'eleves' pour stocker nos objets 'eleve'
Note :
le fait de choisir un nom au pluriel pour un *Array* contenant des *Object* de même nom mais au singulier, n'est qu'un usage mais il est assez utilisé par les programmeurs.
P:
```
eleves=[];
eleves.push(eleve);
eleve={};
```
T:
Attention la ligne ci-dessus est essentielle.
Elle dit à JS met dans 'eleve' un **nouvel *Object*** et ne touche
plus à l'objet que j'y avais mis précédemment.
En d'autres termes eleves[0] n'est pas affecté.
Si par contre je faisais directement eleve.prenom="Marie"; eleve.nom="Lombard"; etc
alors les propriétés de *eleves[0]* (jean-michel etc) seraient écrasées.
P:
```
eleve.prenom="Marie";
eleve.nom="Lombard";
eleve.age="18";
eleve.email="marie.lombard@laposte.net";
eleve.affich=affichCetEleve;
eleves.push(eleve);
eleve={};
eleve.prenom="Eric";
eleve.nom="Enneke";
eleve.age="32";
eleve.email="e.neke@gmail.com";
eleve.affich=affichCetEleve;;
eleves.push(eleve);
```
Pour tester :
```
affich("Boucle de tous mes objets 'eleve' mis dans 'eleves' ");
for (var i=0;i<eleves.length;i++) {
        eleves[i].affich();
}
```
T:
on parcourt le tableau (Array) 'eleves'
dont chaque élément est un objet (Object) 'eleve'
avec la fonction affich() qui affiche les propriétés de l'objet.
Noter
que tous les objets pointent exactement vers la seule et même fonction.
C'est en ce sens qu'au début du § *variable* je parlais de variables qui pointaient vers une même donnée au lieu d'être vues comme de simples contenants distincts.
La fonction *affich* agit différemment selon l'objet en cours qui se retrouve dans *this* à l'éxécution.

--- tester index.html ---

T:
...Et même très théorique juste pour donner une idée d'ateliers futures d'initiation à la Programmation Orientée Objet (POO).

Les avantage à avoir une structure de type Object sont entres autres :

1. Toutes les propriétés et fonctions sont bien regroupées et donc isolées dans chaque objet. L'objet **encapsule** les données (Les objets de catégorie différente ont chacun leurs propriétés ; les objets de même nature (comme les 'eleve' de notre exemple) ont juste, des contenus différents.
2. Si une application a besoin de plusieurs catégories d'objet ; exemple des '*eleve*', '*professeur*', '*proviseur*', etc ; ces différentes catégories auront vraisemblablement des variables communes comme prenom et nom et d'autres spécifiques ; par exemple, '*appreciation*', '*niveau*', appartiendront à '*eleve*' mais pas à '*professeur'.* Par contre tous ces objets pourront avoir une fonction de même nom comme *affich* avec un code et donc un comportement complètement différemment selon la catégorie. (en POO orthodoxe cela s'appelle le **polymorphisme**)
3. Un objet peut avoir besoin des mêmes propriétés et fonctionnalités qu'un objet déjà défini ; ceci en plus de celles qui lui seront propres. Dans ce cas JS offre un mécanisme permettant de récupérer les membres de l'objet existant qui deviennent ainsi membres du nouvel objet. Ce mécanisme s'appelle l'**héritage**.
4. Certains objets peuvent être utilisés à l'identique dans plusieurs applications complètement différentes. On parle de **réusabilité**.
5. Un programmeur peut avoir besoin de plusieurs ensembles de données. Ces ensembles doivent dans certains cas être utilisés simultanément et pourtant être rangés dans des variables distinctes. Croyez-moi sur parole : sans le type Object on mettrait ses ensembles dans des tableaux à plusieurs dimensions ou on bidouillerait en donnant un préfixe différent à chaque tableau... Ce serait l'enfer et pas du tout lisible !
6. En POO les objets peuvent être vus comme autant de programmes indépendants, agissant et interagissant simultanément. Leurs membres sont persistants (contrairement aux variables des fonctions).
   Une application POO n'est donc qu'un ensemble d'objets plus petits qu'elle. Son évolution et sa correction en sont d'autant simplifiées.

FIN

Dans le prochain atelier on écrira une vraie application simple (pas POO;-) mais concrète.
On commencera par voir comment récupérer les actions de l'utilisateur ...