TECHNOLOGIES WEB

Informatique de gestion - bloc 2

Haute-École de Namur-Liège-Luxembourg

Atelier 3A : fonctions, tableaux, objets

Objectifs

- Faire le point sur diverses manières de définir et d'utiliser des fonctions en Javascript
- Apprendre à utiliser des tableaux en Javascript
- Voir comment définir, modifier et utiliser des objets (sans méthodes) en Javascript

Exercice 1 : définitions de fonctions

Vous avez déjà défini et utilisé des fonctions dans les exercices précédents. Le but de cet atelier est de faire le point sur divers éléments concernant les définitions de fonctions en Javascript.

Étape 1 : un cadre de travail

Créez un fichier HTML complet contenant le code suivant.

```
<!doctype HTML>
<html>
  <head>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Tests sur les fonctions en Javascript</h1>
    <script>
      function salue (nom) {
        document.write("<p>Bonjour, " + nom + " !</p>");
      }
      let utilisateur = prompt("Entrez votre nom : ");
      salue(utilisateur);
    </script>
  </body>
</html>
```

Notez dès à présent qu'insérer du code directement dans la partie <body> n'est pas du tout recommandé… mais cela nous facilitera le travail par la suite (juste dans le cadre de cet exercice).

Lisez le code et tentez de déterminer ce qui va se produire quand la page sera chargée. Vérifiez votre prédiction en chargeant la page dans Firefox.

Réécrivez la ligne `document.write` en utilisant un gabarit de chaîne de caractères.

```
document.write(`<p>Bonjour,  ${nom}  !</p>`);
```

Étape 2 : le retour du hoisting

Que produira la code donné ci-dessus si on déplace la ligne `salue(utilisateur);` (et uniquement elle) en tête de script, à savoir :

```
  <script>
    salue(utilisateur);
    function salue (nom) {
      document.write("<p>Bonjour, " + nom + " !</p>");
    }
    let utilisateur = prompt("Entrez votre nom : ");
  </script>
```

```
Une erreur apparait :
Uncaught ReferenceError: can't access lexical declaration 'utilisateur' before initialization
```

Là aussi, tentez de déterminer le résultat puis, ensuite seulement, vérifiez votre réponse sur Firefox.

Note. « Le code n'est pas exécuté » et « ça ne fait rien » ne sont pas des réponses complètes… vous devez être capables de préciser la nature exacte de l'erreur (si vous êtes bloqués, examinez le contenu de la fenêtre d'erreurs !)

Examinez à nouveau l'effet du code si, cette fois-ci, on déplace les deux dernières lignes (le `let` et l'appel à salue) en tête de script, à savoir :

```
<script>
  let utilisateur = prompt("Entrez votre nom : ");
  salue(utilisateur);
  function salue (nom) {
    document.write("<p>Bonjour, " + nom + " !</p>");
  }
</script>
```

Dans ce cas-ci, tout se déroule correctement… ce qui signifie que Javascript permet d'utiliser une fonction (l'appel à salue se trouve en 2e ligne) avant sa définition (en 3e ligne du script). Techniquement, on dit que « les définitions de fonctions sont hissées/hoistées. »

Ce vocabulaire ne devrait pas vous poser de problème : c'est le même que celui utilisé dans le cadre des variables.

Étape 3 : expressions fonctionnelles

Revenez à la version originale du code (où le script commence par la définition de fonction).

Les lignes

```
  function salue (nom) {
    document.write("<p>Bonjour, " + nom + " !</p>");
  }
```

définissent une fonction qui ajoute un paragraphe au document HTML et donnent à cette fonction le nom `salue`.

Cette action peut également se faire sous la forme d'une affectation :

```
  let salue = function (nom) {
    document.write("<p>Bonjour, " + nom + " !</p>");
  };
```

Remplacez la définition de fonction par cette seconde version (notez le point-virgule en fin d'affectation) et lancez la page sous Firefox pour vous assurer que tout fonctionne toujours.

Notez qu'il s'agit bel et bien d'une affectation ici ! Du point de vue de sa structure, le code est similaire à `let salue = 47;` si ce n'est qu'au lieu de donner une valeur numérique à la variable `salue`, on lui affecte une valeur fonctionnelle.

Étape 4 : Et le hoisting maintenant ?

Testez si le principe du hoisting fonctionne également avec cette autre méthode pour définir des fonctions, à savoir si la définition de la fonction `salue` est bien hoistée (et donc utilisable avant qu'elle ne soit faite).

Quelle modification devez-vous apporter au code pour effectuer ce test ?

Quel est le résultat de ce test ?

```
let utilisateur = prompt("Entrez votre nom : ");
salue(utilisateur);
```

```
Une erreur : Uncaught ReferenceError: can't access lexical declaration 'salue' before initialization
```

Assurez-vous que vous comprenez pourquoi (jetez un coup d'œil dans la fenêtre d'erreurs entre autres).

```
Car ici, c'est une variable de type fonction. Hors une variable de type let ne connais pas d'hosting. A noter que même un type var il y a uniquement la déclaration qui est hoisté
```

Étape 5 : À l'ancienne…

Lorsqu'on définit une fonction en utilisant une affectation, on n'est pas obligé d'utiliser `let`. Le mot-clef `const` est également une option (et même une option très pertinente vu qu'on ne modifiera sans doute pas la valeur associée à salue).

Avant la version ES6, il n'y avait que le mot-clef `var` pour écrire une affectation.

Dans ce cas-ci, remplacer `let` par `var` (c'est-à-dire utiliser le code ci-dessous) change-t-il quelque chose ?

```
<script>
  var salue = function (nom) {
    document.write("<p>Bonjour, " + nom + " !</p>");
  }
  let utilisateur = prompt("Entrez votre nom : ");
  salue(utilisateur);
</script>
```

```
Aucun problème dans ce code
```

Pouvez-vous déterminer ce que produira le code suivant ? Fonctionnera-t-il ou engendrera-t-il une erreur (dans ce cas, laquelle) ?

```
<script>
  let utilisateur = prompt("Entrez votre nom : ");
  salue(utilisateur);
  var salue = function (nom) {
    document.write("<p>Bonjour, " + nom + " !</p>");
  }
</script>
```

Assurez-vous de bien comprendre le message d'erreur et d'être capable de répondre à la question suivante : `salue` n'est pas une fonction… d'accord, mais c'est quoi alors ?

```
Avec une variable déclaré avec VAR, elle va être hissé en haut de la fonction. Mais c'est uniquement sa déclaration qui l'est. Pas sa déclaration. Ici, l'appel à la variable se fait dans la TDZ. Donc à cet endroit là, la variable est déclarée mais non initialisée, elle vaut donc undefined !
```

Étape 6 : définir une fonction comme un objet

Il existe encore une autre syntaxe permettant de définir une fonction, ressemblant à la création d'un objet (on verra plus tard que c'est bien de ça qu'il s'agit).

Cette syntaxe utilise le constructeur `Function` pour créer la fonction. Ce constructeur reçoit comme arguments (1) les différents paramètres de la fonction à créer et (2) le code interne de la fonction en question, les deux sous la forme de chaînes de caractères.

Dans le cas de l'exemple utilisé ci-dessus, cela donnerait le code suivant.

```
<script>
  const salue = new Function ("nom",
    'document.write("<p>Bonjour, " + nom + " !</p>");');
  let utilisateur = prompt("Entrez votre nom : ");
  salue(utilisateur);
</script>
```

Notez qu'on a utilisé des apostrophes pour encadrer le code de la fonction (car celui-ci contenait déjà des guillemets).

Testez le fonctionnement de ce code sous Firefox.

Note. Cette méthode de définition des fonctions est déconseillée car le code interne de la fonction (fourni sous forme de chaîne de caractères) doit être évalué lors de chaque appel, ce qui se révèle très peu efficace.

Note (2). Si la fonction à définir comporte plusieurs paramètres, on les cite les uns après les autres. Par exemple :

```
const produit = new Function ("nb1", "nb2", "return nb1 * nb2;");
```

Étape 7 : en guise d'exercice récapitulatif

Directement dans la console cette fois-ci, définissez une fonction `citeNFois` qui reçoit une chaîne de caractères et un nombre et affiche cette chaîne dans la console autant de fois que le nombre donné, en numérotant les répétitions.

Par exemple, un appel `citeNFois("Schtroumpf", 4)` devrait produire l'affichage suivant :

```
1) Schtroumpf
2) Schtroumpf
3) Schtroumpf
4) Schtroumpf
```

- Dans un premier temps, utilisez la syntaxe standard pour définir la fonction. Pensez à utiliser des gabarits de chaîne si c'est pertinent. Histoire de revoir la gestion des types de valeurs, arrangez-vous aussi pour que la fonction affiche un message d'erreur si le second argument reçu n'est pas un nombre positif (c'est-à-dire dans le cas où ce n'est pas un nombre, et dans le cas où c'est un nombre négatif).

- Testez votre implémentation puis modifiez le code pour utiliser la syntaxe sous forme d'affectation.
- Finalement, modifiez votre code pour utiliser la syntaxe « orienté objet ».

Une fois l'exercice terminé, vous pouvez comparer votre solution à celle qui est présentée sur le fichier A3AE1_7.html.

Étape 8 : Et concrètement ?

D'un point de vue pratique, quelle syntaxe utiliser pour définir des fonctions ? Dans la grande majorité des cas, la réponse sera… la première. Il faut cependant être capable de lire et de comprendre les deux autres syntaxes, car elles sont parfois rencontrées dans du code existant.

Exercices 2 : utilisation des fonctions

Quand on définit une fonction en Javascript, on ne précise naturellement pas le type des arguments. Ainsi, si on définit une fonction telle que

```
function triple (x) { return 3 * x; }
```

rien n'empêche de l'appeler avec, par exemple, une chaîne de caractères : Javascript n'effectue aucune vérification à ce sujet.

Étape 1 : retour aux conversions

D'ailleurs, pouvez-vous déterminer les résultats des appels suivants ? (Comme d'habitude, vérifiez vos prévisions dans la console.)

```
triple("2"); 6
triple("deux"); NaN
triple(""); 0
triple(true); 3
```

Étape 2 : une histoire de nombres

Si Javascript ne vérifie pas que les arguments de l'appel correspondent, en type, aux paramètres de la définition… il ne vérifie pas non plus qu'il y en a le bon nombre !

Testez les appels suivants.

```
triple();  NaN
triple(10,20,30); 30
```

Étape 3 : trop ou trop peu

Lorsqu'on appelle une fonction avec trop d'arguments, les arguments « en trop » sont tout simplement ignorés. Vous avez pu le constater avec le second exemple ci-dessus.

Mais comment réagit Javascript quand il y a trop peu d'arguments ?

Pour le déterminer, définissez une fonction `affiche(x,y,z)` qui affiche (par exemple dans la console) les valeurs de chacune des trois variables. Par exemple, `affiche(1,2,3)` devrait afficher

```
x = 1, y = 2, z = 3
```

(ici encore, un gabarit de chaînes peut vous faciliter la tâche !)

Testez ensuite les appels suivants.

```
affiche(true,"oui"); true oui undefined
affiche(-1);-1 undefined undefined
affiche(); undefined undefined undefined
```

Quelle valeur Javascript donne-t-il aux paramètres qui ne correspondent à aucun argument ? Unedfined

Est-ce cohérent avec le résultat du premier appel de l'étape 2 ? Pour répondre, vous devrez peut-être consulter les règles de conversion vues dans l'atelier précédent.
/!\ Je ne comprends pas cette question /!\

Étape 4 : avec ou sans argument

Considérez la fonction définie ci-dessous.

```
function salue (nom) {
  let msg = "Salut, ";
  if (nom)
    msg += nom;
  else
    msg += "inconnu(e)";
  msg += " !";
  alert(msg);
}
```

Pouvez-vous déterminer l'objectif de cette fonction et le décrire en quelques mots ?
Cette fonction affiche un message en concaténant le nom ou pas selon que la conversion en boolean du nom donne vrai ou faux
Dans votre description, avez-vous supposé que `nom` était une chaîne de caractères ou avez-vous tenu compte du fait qu'il pouvait en fait s'agir de n'importe quel type de valeur ?

Si `nom` est bien une chaîne de caractères, pour quelle(s) valeur(s) la fonction affichera-t-elle « Salut, inconnu(e) ! » ? Testez votre réponse.
Pour tout, sauf chaine vide
De manière générale (c'est-à-dire sans se restreindre aux chaînes de caractères), pour quelle(s) valeur(s) de `nom` la fonction affichera-t-elle « Salut, inconnu(e) ! » ?
Une chaine vide, un 0, ou ne pas donner d'argument (alors considéré comme undefined)
Quel effet produire l'appel `salue()` sans argument ? Assurez-vous de comprendre pourquoi. Comme expliqué juste avant, la variable nom est considéré comme undefined, et donc vaut faux

Étape 5

En vous inspirant des réflexions de l'étape précédente, définissez une fonction `afficheDurée(heure1,heure2)` qui affichera (dans la console ou dans une fenêtre popup) « Le cours durera de <heure1>h à <heure2>h. »

Par exemple, l'appel `afficheDurée(10,13)` devra afficher

```
Le cours durera de 10h à 13h.
```

Arrangez-vous pour que, si on ne fournit pas l'heure de fin, celle-ci soit calculée comme étant l'heure de début plus 2 heures. Ainsi, l'appel `afficheDurée(8)` devra afficher

```
Le cours durera de 8h à 10h.
```

Note. Vous pouvez supposer que les arguments fournis sont bien des entiers valides (inutile donc de traiter les cas d'erreurs).

```
function afficherDuree(heure1, heure2){
    heure2 = heure2 || heure1 + 2;
    console.log(`Le cours durera de ${heure1}h à ${heure2}h`);
}
```

Étape 6 : en guise de conclusion…

Le fait que Javascript soit un langage non typé a plusieurs conséquences. On a déjà abordé dans l'atelier précédent les conséquences au niveau des déclarations de variables et de l'évolution de leur contenu. Dans cet atelier, on a vu que cette liberté s'étend également aux paramètres des fonctions, tant au niveau de leur type que de leur nombre.

La liberté offerte par Javascript a toutefois un prix. Dans des langages moins permissifs, comme le C et le Java, certains types d'erreurs sont rapidement détectés, dès la compilation ou l'analyse du code. Il n'en va pas de même en Javascript, où certains codes erronés seront acceptés… Il n'y a donc pas de secret : il faut donc prendre le temps de réfléchir et de construire son code soigneusement !

Exercices 3 : les tableaux en Javascript

Les valeurs primitives telles que les nombres, les chaînes de caractères et les booléens ne suffisent pas pour écrire des scripts plus complexes. On a déjà traité un type de valeurs ci-dessus : les fonctions. Dans la suite de cet atelier, on va en aborder deux autres : les tableaux et les objets.

Étape 1 : manipuler des tableaux

Les tableaux de Javascript se distinguent des tableaux utilisés en C ou en Java. Tout d'abord, comme Javascript est un langage non typé, il ne possède pas de déclaration permettant de préciser le type (commun) de tous les éléments d'un tableau.

Dans les déclarations suivantes (respectivement du C et du Java),

```
int temperaturesSemaine [7];
String [] nomsÉtudiants = new String [30];
```

on indique que les tableaux en question vont contenir des entiers (pour le premier) et des chaînes de caractères (pour le second). Il s'agit de tableaux *homogènes*, dont tous les éléments ont le même type.

En Javascript, les tableaux peuvent être *hétérogènes*, c'est-à-dire contenir des éléments de types différents (notez que c'est une possibilité, pas une obligation : dans beaucoup de cas, les tableaux qu'on utilise seront homogènes).

Considérez par exemple les initialisations suivantes (que vous pouvez entrer via la console).

```
let notes = ["do", "ré", "mi", "fa", "sol"];
let valeursFalsy = [0, false, "", undefined, null];
let matriceI3 = [ [1, 0, 0], [0, 1, 0], [0, 0, 1] ];
```

Pour accéder au contenu d'un tableau (le lire ou le modifier), on utilise une syntaxe identique à celles du C et de Java.

```
notes[3];                   // affiche « fa »
typeof(valeursFalsy[2]);    // affiche « string »
matriceI3[1][1] = 2;        // remplace le 1 central par un 2
```

Entrez une par une les lignes ci-dessus en Javascript et observez-en les résultats. Observez également comment la console affiche la valeur d'un tableau en entrant les trois lignes suivantes.

```
notes;  Array(5) [ "do", "ré", "mi", "fa", "sol" ]
valeursFalsy;  Array(5) [ 0, false, "", undefined, null ]
matriceI3;  Array(3) [ (3) […], (3) […], (3) […] ]
```

En vous inspirant des bouts de code ci-dessus, rédigez une instruction d'initialisation en Javascript pour les trois tableaux suivants.

1. le tableau `diviseurs` qui reprend, dans l'ordre décroissant, les six diviseurs du nombre 12 ; `const diviseurs = [1,2,3,4,6,12];`
2. le tableau `jours` qui reprend, du lundi au dimanche, les noms des jours de la semaine (en utilisant chaque fois leurs trois premières lettres comme abréviation) ; `const semaine = ["lun","mar","mer","jeu","ven","sam","dim"];`
3. le tableau `matièresWebParAnnée` dont chacun des éléments est lui-même un tableau reprenant les matières abordées au fil des 3 années d'études (HTML et CSS pour la 1re, Javascript et le DOM pour la 2e, Ajax et PHP pour la 3e). `const matiereWebParAnnee = [ ["HTML", "CSS"], ["Javascript", "Dom"], ["Ajax", "PHP"]];`

Étape 2 : parcourir un tableau

Javascript propose plusieurs méthodes pour parcourir un tableau ; on en abordera trois dans cet atelier et une quatrième plus tard.

La méthode « standard » valable également en C et en Java reste applicable. Il s'agit d'utiliser une variable indice (qu'on peut appeler `i` si elle n'est utilisée que localement et que le code est simple) initialisée à 0 et évoluant jusqu'à atteindre la longueur du tableau (moins 1). On peut obtenir la longueur d'un tableau `tab` en écrivant `tab.length` (comme en Java).

```
for (let i = 0 ; i < tab.length ; i++) …tab[i]…
```

Écrivez une fonction `afficheTableau` qui reçoit comme argument un tableau et affiche son contenu. La fonction produira une ligne par élément du tableau au format suivant :

```
Contenu de la case <numéro> : <contenu>
```

Testez votre fonction en l'appliquant aux trois tableaux de l'étape précédente.

Avez-vous pensé à utiliser un gabarit de chaîne de votre implémentation ?

Étape 3 : des tableaux dynamiques !

Une autre différence par rapport aux tableaux du C et de Java : les tableaux Javascript sont dynamiques. Pratiquement, cela signifie que leur taille peut être modifiée.

Entrez les lignes suivantes une par une…

```
let notes = ["do", "ré", "mi", "fa", "sol"];
notes.length;                // Que 5 notes ? On en a oublié deux…
notes[5] = "la";
notes.length;
notes[6] = "si";
notes.length;
notes;
```

Note. Javascript permet également de modifier directement la valeur de `tab.length`, mais c'est un procédé moins « propre » qui, pour être bien compris, nécessite d'étudier des particularités de Javascript qui ne sont pas abordées dans ce cours.

En C ou en Java, lorsqu'on désire ajouter de nouveaux éléments à la fin d'un tableau déjà remplis (c'est-à-dire accroître la taille du tableau), il faut tout d'abord créer un tableau

plus grand, puis y recopier les éléments existants, puis finalement y écrire les valeurs à ajouter. Javascript, quant à lui, permet d'ajouter directement les nouveaux éléments à la fin du tableau existant (dont la taille sera ajustée dynamiquement).

Étape 4 : tableaux et piles/files

Piles et files… deux concepts fondamentaux en informatique vus dans le cours d'Organisation et Exploitation des Données.

Comme on peut facilement ajouter des éléments à un tableau, il est relativement facile d'en utiliser un pour stocker les éléments d'une pile (en considérant que la fin du tableau est le « sommet » de la pile).

Partez du fichier ci-dessous et ajoutez du code dans la balise script afin de répondre aux consignes suivantes.

```
<!doctype HTML>
<html>
  <head>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Log de travail</h1>
    <script>
    </script>
  </body>
</html>
```

Imaginez un fonctionnaire qui doit traiter une série de dossiers. On lui amène de temps en temps de nouveaux dossiers à traiter, qui sont déposés au sommet de sa pile de travail. De temps en temps, il finit de traiter un dossier ; dans ce cas-là, il le classe, prend un nouveau dossier au sommet de sa pile de travail et commence à travailler sur celui-ci.

Le script à ajouter simulera ce fonctionnement. Il s'agira d'une boucle qui demandera une entrée à l'utilisateur (via `prompt`) et ce jusqu'à ce que l'utilisateur clique sur « Cancel », ce qui terminera la boucle (dans ce cas-là, la fonction `prompt` renvoie la valeur `null`).

Après chaque entrée de l'utilisateur, le script ajoutera un nouveau paragraphe dans le document (via `document.write`) en suivant les règles suivantes (les parties entre <chevrons> sont à remplacer par des valeurs).

- Si l'entrée de l'utilisateur n'est pas vide (il a tapé au moins un caractère), son entrée sera considérée comme un nouveau nom de dossier et ajoutée au sommet d'une pile de valeurs conservées en mémoire. On affichera
  Nouveau dossier : <entrée> - actuellement : <nombre> dossier(s) dans la pile
- Si, par contre, l'utilisateur s'est contenté de cliquer sur « Ok » (ou d'appuyer sur « Enter »), cela signifie qu'il a clôturé un dossier et est prêt à travailler sur un autre. On lira la valeur située au sommet de la pile (on l'enlèvera de la pile) et on affichera
  Dossier traité : <entrée> - reste : <nombre> dossier(s) dans la pile

- Si l'utilisateur n'a pas entré d'informations mais que la pile est vide, on affichera *Plus de dossier dans la pile !*

Par exemple, si l'utilisateur rentre (dans l'ordre) « recrutement », « salaires », (rien), « stock », (rien), (rien), (rien), (cancel), le script devrait afficher les lignes suivantes.

```
Nouveau dossier : recrutement – actuellement : 1 dossier dans la pile
Nouveau dossier : salaires – actuellement : 2 dossiers dans la pile
Dossier traité : salaires – reste : 1 dossier dans la pile
Nouveau dossier : stock – actuellement : 2 dossiers dans la pile
Dossier traité : stock – actuellement : 1 dossier dans la pile
Dossier traité : recrutement – actuellement : 1 dossier dans la pile
Plus de dossier dans la pile !
```

Quelques consignes supplémentaires…

• Avant de pouvoir placer une valeur dans un tableau (par exemple via `tab[0] = val`), il faut indiquer qu'il s'agit d'un tableau, par exemple via `let tab = [];`

• **Pensez clean code !!!**

Étape 5 : push et pop !

Les tableaux sont des objets qui possèdent un bon nombre de méthodes prédéfinies. Parmi celles-ci se trouvent les deux méthodes suivantes, qui permettent d'utiliser un tableau comme une pile.

- Si `tab` est un tableau, `tab.push(val)` va ajouter la valeur `val` en fin de tableau. Cette méthode renvoie également la nouvelle longueur du tableau.
- Si `tab` est un tableau, `tab.pop()` renvoie le dernier élément du tableau et réduit sa longueur ; si le tableau est vide, cette méthode renvoie `undefined`.

Réécrivez le code de l'étape précédente en utilisant ces deux méthodes.

Une fois l'exercice terminé, vous pouvez comparer votre code avec celui qui est disponible dans le fichier A3AE3_4.html.

Exercice 4 : deux boucles de plus !

Vous maîtrisez les boucles `for`, les boucles `while` et les boucles `do…while`. Javascript propose deux autres types de boucles utilisables, entre autres, avec les tableaux.

Étape 1 : for-in

Plutôt que d'utiliser une variable `i` allant de 0 à la taille du tableau (moins 1), on peut se servir de la syntaxe suivante.

```
for (let i in tab) …tab[i]…
```

Dans un premier temps, on peut interpréter `let i in tab` comme étant « pour `i` qui prend tour à tour les valeurs des indices valables du tableau `tab` ». Même si la syntaxe semble plus libre, elle impose que `i` passe en revue les cellules du tableau dans l'ordre.

Testez cette syntaxe sur les tableaux définis précédemment en observant l'ordre dans lequel les cellules sont traitées, par exemple avec le code

```
for (let i in tab) { console.log(tab[i]); }
```

Étape 2 : for-of

Dans de nombreux cas, les tableaux sont utilisés pour garder en mémoire une liste de choses (des noms de notes, de noms de jour…) et l'indice exact auquel chacune de ces choses est associé importe peu.

Plutôt que d'utiliser une variable `i` qui prend comme valeur tour à tour les différents indices des cellules, on peut se servir de la syntaxe suivante, où `val` prend comme valeur tour à tour le contenu des différentes cellules.

```
for (let val of tab) …val…
```

Comme `val` est une valeur du tableau, on peut l'utiliser directement ; l'écriture `tab[val]` n'a, a priori, pas de sens (sauf si le tableau contient des valeurs qui font référence à ses propres indices) !

Ici encore, testez cette syntaxe sur les tableaux définis précédemment.

Étape 3 : applications

Utilisez à chaque fois la boucle la plus appropriée pour définir chacune des fonctions demandées ci-dessous. Testez ensuite ces fonctions sur divers tableaux.

1. La fonction `somme` reçoit un tableau de nombres (on peut supposer que c'est une précondition, qu'il est donc inutile de le vérifier ou de traiter des cas d'erreurs) et renvoie la somme de tous ces nombres.
2. La fonction `premierPair` reçoit un tableau de nombres et renvoie le premier de ces nombres qui est pair.
3. La fonction `dernierPair` reçoit un tableau de nombres et renvoie le dernier de ces nombres qui est pair.

Exercice 5 : premiers objets

En orienté objet « classique » (comme en Java), avant de pouvoir construire un objet, il faut définir une classe. Puis, dans un second temps, l'objet est construit comme une instanciation de la classe.

Javascript suit une autre approche et permet de construire des objets directement en précisant quels attributs ils ont, quelles sont les valeurs de ces attributs et quelles sont ses méthodes.

Si on ignore les méthodes (ce qui sera le cas dans un premier temps, c'est-à-dire dans cet atelier), un objet n'est rien d'autre qu'une collection d'attributs qui ont une valeur. Cela revient en fait à la notion de « structure » du langage C.

Étape 1 : syntaxe orienté objet

Dans cet exercice, vous manipulerez des objets représentant divers langages de programmation abordés au cours du cursus en IG. Dans un premier temps, on retiendra le nom du langage, l'année où il est enseigné pour la première fois et s'il s'agit d'un langage orienté objet ou pas. On utilisera les noms suivants pour les attributs : `nom` (chaîne de caractères), `année` (1, 2 ou 3), `oo` (booléen indiquant si le langage est orienté objet).

Javascript offre plusieurs méthodes pour créer un objet. Une d'elles consiste à créer un objet « vide » (sans aucun attribut) puis à ajouter les attributs un par un. Entrez les lignes suivantes une par une dans la console pour définir l'objet correspondant au langage C.

```
let c = {};            // c est un objet vide
c;                     // affiche la valeur de c : objet vide
c.nom = "C";           // on lui ajoute un attribut « nom ».
c;                     // affiche la valeur de c
c.année = 1;
c;
c.oo = true;           // oui, c'est une erreur : C n'est pas oo !
c;
c.oo = false;          // on peut modifier la valeur d'un attribut
c;
```

Si vous avez effectué votre travail correctement, quand vous demandez à Firefox d'afficher la valeur finale de `c`, il devrait répondre quelque chose comme ceci :

```
Object { nom: "C", année: 1, oo: false }
```

Positionnez votre curseur sur le mot « Object » (qui se souligne) et cliquez. Une nouvelle portion de fenêtre s'ouvre, indiquant le contenu de l'objet en question. Vous y retrouvez les trois attributs définies (`nom`, `année` et `oo`) ainsi qu'une propriété appelée `__proto__` qui est lié au fonctionnement de l'orienté objet en Javascript et que vous pouvez ignorer pour le moment.

Étape 2 : syntaxe des tableaux associatifs

La syntaxe utilisée ci-dessus (à part à la toute première ligne) est tout à fait similaire à ce qui se fait en Java : si `obj` est un objet et `attr` un de ses attributs, on peut utiliser `obj.attr` pour accéder (en lecture ou en écriture) à cet attribut.

La syntaxe `obj.attr` n'est cependant pas la seule disponible. Javascript permet également d'accéder à l'attribut en question en écrivant `obj["attr"]`, comme si l'objet était un tableau et qu'on allait voir ce qui se trouve dans la « cellule » portant le nom `attr`. (On parle de tableau associatif dans ce cas-ci.)

Entrez les lignes suivantes une à une dans la console pour définir un second objet.

```
let java = {};
java;
java["nom"] = "Java";
java;
java["année"] = 1;
java;
java["oo"] = true;
java;
```

Cette syntaxe peut sembler étrange et faire double-emploi avec la précédente mais elle permet d'accéder à la valeur d'une propriété dont le nom ne serait pas connu à l'avance.

Imaginez par exemple un objet cotes reprenant les cotes d'un étudiant à travers plusieurs sessions.

```
cotes.mathIG1juin2025 = 7;
cotes.mathIG1août2025 = 8;
cotes.mathIG2janvier2026 = 3;
cotes.mathIG1juin2026 = 10;
cotes.mathIG2août2026 = 11;
cotes.javaIG1juin2025 = 12;
…
```

Pour obtenir la cote correspondant à un cours donné, à une session donnée et à une année donnée, on pourrait utiliser la syntaxe suivante.

```
cotes[nomCours + "IG" + annéeCours + moisSession + annéeSession]
```

Ainsi, si on a

```
nomCours = "math";
annéeCours = 1;
moisSession = "juin";
annéeSession = 2026;
```

alors l'attribut recherché sera « mathIG1juin2026 ». Une telle liberté est seulement possible avec la notation « tableau associatif », où le nom de l'attribut prend la forme d'une chaîne de caractères.

Étape 3 : une troisième syntaxe

Définissez un nouvel objet appelé `javascript`. Définissez deux de ses attributs en utilisant la syntaxe « orienté objet » et le troisième en utilisant la syntaxe « tableau associatif ».

Outre ces deux syntaxes, Javascript permet également de définir l'objet tout entier en une seule instruction. Voici ce que cela donnerait pour le langage C#.

```
let csharp = { nom : "C#", année : 2, oo : true };
```

Dans cette syntaxe (littéral pour objets), observez que

- le tout est entouré d'accolades ;
- chaque attribut est donné sous le format nomAttribut : valeurAttribut ;
- les attributs sont séparés par des virgules.

Notez que l'écriture

```
let obj = {};
```

permettant de définir un objet vide est un cas particulier de cette syntaxe.

Étape 4 : utiliser les objets

Définissez une fonction `afficheLangage` qui reçoit comme argument un objet et affiche ses informations (dans la console ou dans une fenêtre pop-up) au format suivant :

```
C (vu en IG1) : pas orienté objet
Java (vu en IG1) : orienté objet
```

Pour accéder aux attributs de l'objet utilisé comme argument (que vous appellerez `langage`), vous pouvez utiliser la syntaxe « orienté objet » ou la syntaxe « tableau associatif », au choix.

Avez-vous utilisé un gabarit de chaîne ? Avez-vous utilisé une expression conditionnelle (c'est-à-dire `test ? val1 : val 2`) ?

Étape 5 : pas de classe = aucune garantie sur les attributs présents !

Créez l'objet suivant.

```
let scala = { nom : "python", oo : true };
```

Que va produire l'appel `afficheLangage(scala)` selon vous ?

Vérifiez votre réponse en utilisant Firefox.

Pour vérifier si un objet `obj` possède un attribut `attr`, on peut utiliser l'expression suivante, qui possède une valeur booléenne (vrai si l'attribut existe dans l'objet, faux sinon).

```
"attr" in obj
```

Pour tester votre compréhension, prévoyez le résultat des expressions suivantes puis vérifiez votre réponse dans la console.

```
"année" in c;
"année" in scala;
```

En utilisant la condition `in`, redéfinissez la fonction `afficheLangage` pour que, dans le cas des langages ne possédant pas d'attribut « année », l'affichage produise quelque chose de similaire à la ligne suivante.

```
Scala (pas vu en IG) : orienté objet
```

Testez votre nouveau code puis répondez à la question suivante.

Que produirait l'appel `afficheLangage(basic)` avec la définition suivante ?

```
let basic = { nom : "basic", année : null, oo : false };
```

Modifiez la définition de la fonction pour que, dans les cas où l'année n'est pas un nombre, elle affiche également « pas vu en IG ». L'ordre des conditions a-t-il de l'importance ?

Vous pourrez trouver un exemple de solution sur la page A3AE5_5.html.

Étape 6 : des objets dynamiques

Entrez les définitions suivantes (via un copier/coller par exemple).

```
let php = { nom : "php", année : 3, oo : false, inutile : "oui" };
let prolog = { nom : "Prolog", oo : false };
```

Remarquez que quelques erreurs se sont glissées dans ces définitions. Heureusement, les objets Javascript sont complètement dynamiques : on peut non seulement leur ajouter des attributs et modifier la valeur de leurs attributs mais également supprimer certains attributs !

Voici la syntaxe générale pour chacune de ces opérations :

```
obj.attr = valeur;          // ajouter un attribut
obj.attr = valeur;          // ou modifier sa valeur
delete obj.attr;            // supprimer un attribut
```

Corrigez les objets définis ci-dessus en exécutant des instructions qui vont :

- ajouter l'information que Prolog est enseigné en 2e année ;
- enlever la propriété « inutile » de PHP ;
- faire en sorte que PHP soit correctement décrit comme un langage orienté objet.

Vérifiez le résultat en affichant les nouvelles valeurs des objets.

Étape 7 : tableau d'objets

Créez un tableau `langages` contenant tous les objets « langages » définis jusqu'ici.

```
let langages = [c, java, javascript, basic, scala, basic,
```

```
  php, prolog, csharp];
```

Définissez une fonction `listeNoms` qui prend comme argument un tableau (on va supposer qu'il s'agit d'un tableau de langages) et qui affiche uniquement leurs noms (par exemple, un nom par ligne, dans la console).

For ? For-in ? For-of ? À vous de choisir l'option la plus pertinente !

Étape 8 : et on mélange tout !

Considérez la définition suivante, relatives aux 3 anneaux donnés aux seigneurs elfes (voir le Seigneur des Anneaux et autres écrits de Tolkien).

```
let anneauxElfes = [
  { nom : "Narya", porteur : "Cirdan" },
  { nom : "Nenya", porteur : "Galadriel" },
  { nom : "Vilya", porteur : "Gil-Galad" }
];
```

Quel type d'objet est `anneauxElfes` ?

Que produira l'appel `listeNoms(anneauxElfes)` ?

Exercice 6 : (presque) tout est objet

Le but de cet exercice est de montrer que les tableaux et les fonctions sont des cas particuliers d'objets.

Étape 1 : un tableau et une fonction

Considérez la définition de tableau suivant.

```
let cotes = [5, 12, 9, 17, 4, 13, 15, 8, 11, 16];
```

Définissez une fonction `compteSupérieur` qui reçoit deux arguments, un tableau de nombres et un nombre, et qui renvoie le nombre de valeurs du tableau qui sont plus grandes ou égales au nombre donné. Ainsi, un appel `compteSupérieur(cotes,13)` devrait renvoyer 4 car le tableau `cotes` contient 4 valeurs plus grandes ou égales à 13.

Étape 2 : dissection d'un objet

Définissez la fonction suivante, qui permet d'afficher les noms de propriétés d'un objet donné.

```
function afficheProp (o) {
  for (let nom in o) console.log(nom);
}
```

Vérifiez son comportement en l'appliquant à un objet. Par exemple :

```
afficheProp( { année : 2040, nom : "Spock", vulcain : true } );
```

Étape 3 : tableaux = cas particulier des objets

Demandez à Firefox d'évaluer cotes. Il devrait afficher ceci :

```
Array [ 5, 12, 9, 17, 4, 13, 15, 8, 11, 16 ]
```

Cliquez sur `Array` et observez les informations affichées. (Une fois encore, ignorez `__proto__`).

Demandez à Firefox d'évaluer chacune des expressions suivantes.

```
cotes.length
cotes["length"]
cotes[1]
cotes["1"]
"1" in cotes
"length" in cotes
```

Étape 4 : tableaux = cas particulier des objets (suite)

Exécutez la fonction `afficheProp` sur l'objet/tableau `cotes`. L'une des propriétés affichées par Firefox n'est pas reprise. Laquelle ?

Il s'agit d'une propriété dite « non énumérable ». Elle existe bel et bien en tant que propriété de l'objet mais elle n'est pas prise en compte lorsqu'on parcourt le contenu de l'objet (comme avec une boucle for-in).

En tant qu'objet, le tableau `cotes` supporte l'ajout de propriétés. Ajoutez-lui une propriété `max` associée à la valeur 20 par exemple, puis exécutez à nouveau la fonction `afficheProp`.

Étape 5 : fonctions = cas particulier des objets

Demandez à Firefox d'afficher la valeur de la fonction `compteSupérieur` définie plus haut. Dans sa réponse, cliquez sur le nom de la fonction et observez les informations affichées.

Une fonction est un objet un peu plus compliqué qu'un tableau ; certaines de ses propriétés sont plus difficiles à interpréter. On se concentrera ici principalement sur les propriétés `name` et `length`, qui indiquent respectivement le nom interne de la fonction et le nombre d'arguments utilisés dans la définition.

```
compteSupérieur.length
compteSupérieur.name
compteSupérieur["length"]
```

Exécutez une instruction pour modifier la valeur de la propriété `length` et la mettre à 7. Ensuite, affichez la valeur de cette propriété. Que remarquez-vous ?

Pour les fonctions, la propriété `length` est immuable (en anglais : immutable, ou readonly). Cela signifie qu'elle ne peut pas être modifiée directement.

(Pour information) La propriété `arguments` reprend les arguments fournis à la fonction quand elle est appelée. `Caller` contient une référence vers l'objet qui est ciblé par le mot réservé `this` dans le code de la fonction, ce qui est particulièrement utile si la fonction en question est en fait une méthode / propriété d'objet. Les propriétés `prototype` et `__proto__` sont relatives à l'héritage prototypal (voir plus tard).

Étape 6 : fonctions = cas particulier des objets (suite)

Déterminez les noms internes des fonctions définies ci-dessous en utilisant la console ?

```
function foisTrois (x) { return x * 3; }
let foisQuatre = function quadruple (x) { return x * 4; }
let puissance2 = function (x) { return x * x; }
```

Ajoutez une propriété `résultat` à chacune des trois fonctions définies ci-dessus, de sorte que celle-ci soit associée au string "triple" pour `foisTrois`, "quadruple" pour `foisQuatre` et "carré" pour `puissance2`.

Définissez ensuite une fonction `beauRésultat` qui reçoit deux arguments : une fonction `f` et une valeur numérique `x` et qui affiche le résultat f(x) en indiquant de quoi il s'agit. Par exemple, les appels

```
beauRésultat(foisQuatre,7);
```

```
beauRésultat(puissance2,-3);
beauRésultat(foisTrois,-1.5);
```

devront afficher les messages suivants.

```
Le quadruple de 7 est 28.
Le carré de -3 est 9.
Le triple de -1.5 est -4.5.
```

Étape 7

Et les chaînes de caractères ? les nombres, et les booléens ?

Ce sont des données primitives mais, à côté de chacun des trois grands types de valeurs primitives (nombres, chaînes, booléens), il existe un type d'objet correspondant (tout comme en Java d'ailleurs).

Javascript passe discrètement des valeurs primitives à leur équivalent « objet » lorsque c'est nécessaire.

Entrez par exemple les lignes suivantes.

```
let nom = "Géralt de Rivia";
nom.length;
nom.repeat(4);

let deuxTiers = 2/3;
deuxTiers.toFixed(4);
```

Remarquez que, lorsque ces valeurs primitives ont été utilisées comme des objets (en appelant des méthodes), Javascript les a convertis en objets sans rien dire. Nous reviendrons plus tard sur ces objets qui correspondent aux valeurs primitives (on les appelle « boxing objects » ou « wrappers » et le fait que Javascript traduise automatiquement les valeurs simples en objets s'appelle « l'auto-boxing »).

Exercice 7 : Question d'examen 2 [janvier 2016]

1. On considère le code ci-dessous. Qu'affichera la console ? (Vous pouvez tout écrire sur une seule ligne).

```
// Définition 1
function f(x) { console.log(x+7); }
// Appel 1
f(2);
// Définition 2
function f(x) { console.log(x+3); }
// Appel 2
f(3);
```

2. Si, dans la question précédente, on écrit les deux définitions de fonctions sous la forme

```
var f = function (x) { ... };
```

qu'affichera la console ?

3. Les réponses aux deux questions précédentes sont différentes. C'est dû à un principe qui, en Javascript, s'applique un peu différemment aux définitions de fonctions et aux déclarations/initialisations de variables. Quel est le nom de ce principe ?