**JAVASCRIPT**

**Histoire du langage JavaScript**

JavaScript a été créé en 1995, pour le compte de la société Netscape (éditeur d’un des premiers navigateurs web, dont Mozilla Firefox est l’héritier), puis standardisé sous le nom d’ECMAScript (souvent réduit à ses initiales “ES”).

Utilisable sur de nombreuses plate-formes, JavaScript est le seul langage de programmation inclus en standard dans les navigateurs web. On dit donc que c’est “le langage du web”.

Malgré ses nombreuses évolutions, JavaScript est plutôt un langage de script qu’un langage de programmation, à proprement parler. En effet, il a été conçu comme une solution simple pour ajouter des interactions aux pages HTML du web, et non pour créer des logiciels à haut niveau de sûreté. Une des conséquences : c’est un langage faiblement typé. Nous allons voir ça plus bas.

**Exécution de code JavaScript**

Il y a deux façons d’exécuter nos instructions JavaScript :

1. De manière interactive : via une console
2. De manière programmative : en rédigeant un code source

Dans le deuxième cas, on va écrire une liste d’instructions dans un fichier (appelé “code source”), pour que notre machine puisse exécuter ce fichier en une seule fois.

Pour tester JavaScript en mode interactif :

(Ancienne methode)

* Ouvrez une fenêtre de navigation privée (mode incognito) depuis Google Chrome,
* Ouvrez la console JavaScript en utilisant le raccourci clavier Cmd+Alt+J (sur Mac) ou Ctrl+Shift+J (sur PC/Window)
* Tapez 1+1 et validez. => La console devrait afficher le résultat de cette opération. Ensuite tapez alert('bonjour'); et observez ce qui se passe.

(Nouvelle méthode)

Ou alors plus simplement vous ouvrez un balise script dans votre html et vous mettez votre code javascript dedans quand vous sauvegarde, vous ouvrez votre inspecteur et vous cliquez sur console et vous aurez le résultat.

## Manipulation de valeurs

### Types de valeurs

En langue Française, il existe plusieurs types de mots : les noms, les verbes, les adjectifs, les prénoms, etc…

De la même façon, en maths, on peut manipuler plusieurs types de valeurs : nombres entiers, décimaux, rationnels (fractions), complexes, etc…

En JavaScript, c’est pareil. Il est possible d’exprimer et de manipuler des valeurs de différents types.

#### Types simples

* booléen (*boolean*): true, false
* nombre (*integer/float number*): 999, 0.12, -9.99
* chaîne de caractères (*string*): 'coucou'
* aucune valeur: null

#### Types avancés

* non défini: undefined
* objet (*object*): { prop: 'valeur' }
* tableau (*array*): [ 1, 2, 3 ]
* fonction (*function*): function(){ /\* ... \*/ }

**Exercice 1 Quel Type ?**

### Variables et opérateur d’affectation

#### Variables et valeurs littérales

Dans l’exercice ci-dessus, nous avons manipulé des valeurs de manière **littérale**. C’est à dire qu’elles étaient explicitement affichées dans le code.

Dans un véritable programme, les valeurs dépendent souvent d’une saisie de l’utilisateur, ou d’autre chose. Du coup, elles sont rarement exprimées de manière littérale. On utilise pour ça une représentation symbolique : **les variables**.

En maths, on représente habituellement une variable sous forme d’une lettre minuscule. Par exemple “*soit x=4*” veut dire qu’on définit une variable appelée x représente actuellement la valeur 4. (Un nombre entier, en l’occurrence)

En JavaScript, une variable représentée par une suite de lettres (minuscules et/ou majuscules) pouvant contenir aussi des chiffres et le symbole *underscore* (\_).

Généralement, on emploie une notation appelée **camel case** pour nommer les variables en JavaScript. Cette notation consiste à coller plusieurs mots, en mettant en majuscule seulement la première lettre de chaque mot, sauf celle du premier mot.

Exemple : nombreSaisiParUtilisateur est un bon nom pour une variable JavaScript.

*Camel* veut dire “chameau”, en Français. Avez-vous compris pourquoi ?

#### Opérateur d’affectation

Comme en maths, l’opérateur d’affectation = permet d’affecter une valeur (à droite) à une variable (à gauche).

Par exemple, si on tape monNombre = 1 dans une console JavaScript, puis qu’on y tape monNombre, la console nous répondra 1 car c’est la valeur actuelle de la variable monNombre.

Chaque usage de l’opérateur d’affectation sur une variable **changera sa valeur actuelle**, quitte à remplacer la valeur qu’elle portait précédemment.

Par exemple, si on tape monNombre = 1, monNombre = 2 puis monNombre, la console JavaScript nous répondra 2 car c’est la dernière valeur qui a été affectée à la variable monNombre.

La valeur affectée (à droite du =) peut être une valeur **littérale**, ou celle d’une **autre variable**.

Par exemple, si on tape autreNombre = monNombre, puis autreNombre, la console JavaScript nous répondra 2 car nous avons affecté la valeur de la variable monNombre (2, tel qu’affecté dans l’exemple précédent) à notre nouvelle variable autreNombre.

À noter que l’opérateur d’affectation se comporte différemment selon que la valeur affectée est de **type simple** (nombre, chaîne de caractères…) ou **avancé** (objet, tableau, fonction…):

* L’affectation d’une valeur de type simple **dupliquera** cette valeur dans la variable affectée ;
* Alors qu’en affectant une valeur de type avancé, notre variable affectée sera en fait une **référence** à cette valeur.

Exemple d’affectation de valeur de type simple :

maValeurSimple = 1;

autreValeurSimple = maValeurSimple;

// à ce stade, nos deux variables valent 1

maValeurSimple = 2;

// => autreValeurSimple vaut toujours 1, alors que maValeurSimple vaut 2

Exemple d’affectation de valeur de type avancé :

monTableau = [ 1, 2, 3 ];

autreVariable = monTableau;

// à ce stade, nos deux variables valent [ 1, 2, 3 ]

monTableau = [ 4, 5, 6 ];

// cette instruction a affecté un autre tableau à monTableau

// => autreValeurSimple vaut toujours [ 1, 2, 3 ], car il référence toujours le tableau qui lui avait été affecté

Exemple d’affectation de valeur de type avancé, **avec modification** :

monTableau = [ 9, 8, 7 ];

autreVariable = monTableau;

// à ce stade, nos deux variables valent [ 9, 8, 7 ]

monTableau.sort();

// cette instruction a rangé notre tableau dans l'ordre croissant

// => monTableau vaut désormais [ 7, 8, 9 ]

// => ... et autreVariable vaut aussi [ 7, 8, 9 ], car la tableau qu'il référencé a été modifié.

#### Création de variable

Dans nos exemples, les variables ont été créées automatiquement par la console JavaScript lors de leur première affectation de valeur.

Mais nous allons désormais créer nos variables de manière explicite, à l’aide du **mot-clé let ou var**.

En effet, l’usage de var permet de s’assurer qu’on n’a pas déjà créé une variable du même nom, let a le même effet mais a l’intérieur de son bloc seulement (portée locale). Il est possible **d’affecter une valeur** à notre variable lors de sa création avec var.

Il restera évidemment possible d’affecter d’autres valeurs à notre variable, à l’aide de l’opérateur d’affectation, mais sans utiliser var ensuite.

Exemple :

// on va crééer une variable de type chaîne de caractères :

let maVariable = 'mon texte';

// => maVariable vaut 'mon texte'

maVariable = `mon nouveau texte`;

// => maVariable vaut désormais 'mon nouveau texte'

Vous remarquerez que :

* J’ai ajouté des **espaces** autour de l’opérateur d’affectation pour améliorer la lisibilité de mon code ;
* Et j’ai terminé mes instructions par un **point-virgule** (;).

Afin d’améliorer la lisibilité de votre code, et d’éviter tout ambiguïté pouvant occasionner des erreurs, je compte sur vous pour appliquer ces règles.

**Exercice 2 Création de variables**

**Les conditions**

## **Comparaisons de valeurs et conditions**

Un programme est sensé pouvoir prendre des décisions de manière autonome, sur la base de conditions.

Pour exprimer ces conditions, on s’appuie généralement sur des comparaisons entre valeurs. Nous allons commencer par la comparaison la plus simple : **l’égalité de valeurs**.

### Comparaison d’égalité : == et ===

En JavaScript il existe deux opérateurs de comparaison d’égalité :

* == vérifie l’égalité de deux valeurs de manière laxiste ;
* === vérifie l’égalité de deux valeurs de manière stricte.

Une **égalité laxiste** consiste à dire que deux valeurs sont vues comme équivalentes, mais pas exactement égales.

Exemple : 1 (nombre) et "1" (chaîne de caractères) représentent tous les deux le chiffre 1, mais sont de types différents => ils sont égaux seulement selon l’opérateur d’égalité laxiste.

L’**égalité stricte** vérifie en plus que le type des deux valeurs comparées est le même.

Quelques exemples :

1 == '1'; // => true

1 === `1`; // => false (car types différents)

0 == false; // => true

0 === false; // => false

null == undefined; // => true

null === undefined; // => false

La comparaison entre **valeurs de types avancés** fonctionne différemment. Ce ne sont pas les valeurs à proprement parler qui sont comparées, mais la référence vers cette valeur.

Explication :

[1,2] == [1,2]; // => false, car deux tableaux ont été créés

let monTab = [1,2];

monTab == monTab; // => true, car la variable monTab référence un seul et même tableau

monTab == [1,2]; // => false, car ce sont deux références de tableaux différents

let monTab2 = monTab;

monTab == monTab2; // => true, car monTab2 fait référence au même tableau que monTab

**ATTENTION !**

Notez que les opérateurs de comparaison d’égalité == et === ne doivent pas être confondus avec l’opérateur d’affectation =.

Vous verrez plus tard que se tromper d’opérateur peut causer des erreurs silencieuses qui peuvent mettre des heures à être détectées et corrigées dans votre code !

Donc soyez attentifs à bien les différencier quand vous écrivez votre code.

### Opérateurs d’inégalité

Quand on sait que deux valeurs ne sont pas égales, notre programme peut s’adapter à différents cas de figure.

Nous avons vu que les opérateurs === et == permettaient d’évaluer l’égalité (stricte ou laxiste), en retournant une valeur true quand c’était le cas. Le langage JavaScript fournit aussi leurs **opérateurs contraires** : !== et != (strict, et laxiste, respectivement). Ceux-ci retournent une valeur true quand les valeurs comparées ne sont pas égales.

Exemples :

1 == 1; // => true

1 === 1; // => true

1 == '1'; // => true

1 === '1'; // => false

1 != 1; // => false

1 !== 1; // => false

1 != '1'; // => false

1 !== '1'; // => true

Le langage JavaScript fournit aussi les opérateurs de comparaison suivants:

* Strictement inférieur: <,
* Strictement supérieur: >,
* Inférieur ou égal: <=,
* Supérieur ou égal: >=.

Exemples :

1 < 1; // => false

1 > 1; // => false

1 <= 1; // => true

1 >= 1; // => true

1 < 2; // => true

1 > 2; // => false

1 <= 2; // => true

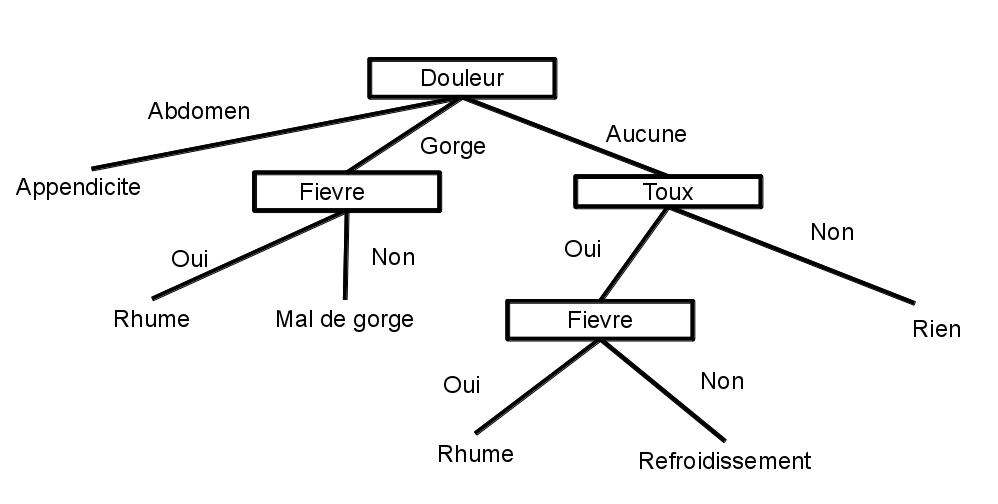
1 >= 2; // => false

### Conditions

Dans un programme (codé en langage JavaScript ou pas), les conditions sont une des instructions les plus incontournables.

C’est grâce aux conditions que votre programme peut prendre des **décisions** et donc d’effectuer des actions de manière autonome (ou automatique), en fonction des données qui lui sont fournies.

On peut représenter ces décisions sous forme d’un arbre:



Cet exemple d’arbre illustre un diagnostic médical, en fonction des symptômes.

Cet arbre pourrait être traduit en pseudo-code de la manière suivante :

* Si *douleur à abdomen*, alors **appendicite**
* Sinon, si *douleur à la gorge* et :
  + Si *fièvre*, alors **rhume**

## Sinon, mal de gorge

* Sinon, si…

Puis être implémenté en langage JavaScript de la manière suivante :

// supposons que les variables douleur et fievre soient fournies

let diag;

if (douleur === 'abdomen') {

diag = 'appendicite';

} else if (douleur === 'gorge') {

if (fievre === true) {

diag = 'rhume';

} else {

diag = 'mal de gorge';

}

}

En JavaScript, les conditions s’expriment à l’aide des mots-clés if et else. On les emploie de la manière suivante :

if (expression) {

// si expression == true, alors les instructions entre ces accolades vont s'exécuter

} else {

// sinon, ce sont les instructions entre ces accolades-là qui vont s'exécuter

}

Le mot-clé else (facultatif) permet d’exécuter une séquence d’instructions seulement si expression n’est pas vraie.

Vous pouvez utiliser une comparaison en guise d’expression.

Exemple :

let monNombre = 1, resultat;

if (monNombre === 1) {

resultat = 'monNombre vaut 1';

} else {

resultat = 'monNombre ne vaut pas 1';

}

resultat;

// => cette liste d'instructions va afficher 'monNombre vaut 1'

A retenir : dans un bloc if-else, soit les instructions entre la première paire d’accolades sera exécuté, soit celles de la deuxième paire d’accolades.

Par ailleurs, observez bien la manière d’agencer les accolades et les espaces.

### Condition avec multiples alternatives

Dans le cas où vous aimeriez définir plus de deux comportements alternatifs, vous pouvez employer ajouter des cas else if() entre votre bloc if{} et votre bloc else{}.

Exemple :

if (monNombre === 1) {

resultat = 'monNombre vaut 1';

} else if (monNombre > 1) {

resultat = 'monNombre est supérieur à 1';

} else {

resultat = 'monNombre n\'est ni égal à 1, ni supérieur à 1';

}

resultat;

Comme pour les blocs if-else, seules les instructions d’**une** paire d’accolades seront exécutées.

Par contre, si vous écrivez plusieurs blocs if-else à la suite les uns des autres, ceux-ci seront complètement indépendants.

### Instructions pour interagir avec l’utilisateur : prompt et alert

Maintenant qu’on a vu comment faire en sorte qu’un programme prenne des décisions, il faut qu’on soit capable d’interagir avec lui.

A ce stade, nous allons employer deux instructions pour cela:

* alert permet d’afficher un message à l’utilisateur;
* et prompt permet de lui demander de saisir une chaîne de caractères.

Par exemple, voici comment **afficher** Bonjour !:

alert('Bonjour !');

Comme vous le voyez, il faut fournir le message à afficher entre parenthèses. Et, comme il s’agit ici d’une chaîne de caractères littérale, il ne faut pas oublier de mettre le texte entre apostrophes.

Il est aussi possible d’afficher la valeur d’une variable:

let monMessage = 'Hello !';

alert(monMessage); // usage d'une variable => pas d'apostrophes

Maintenant, voici comment **inviter l’utilisateur à saisir** une chaîne de caractères, puis l’afficher:

let sonPrenom = prompt('Quel est ton prénom ?');

alert('Bonjour, ' + sonPrenom + ' ! :-)');

Êtes-vous capable d’interpréter ce que signifie le code à l’intérieur des parenthèses du alert ? On appelle ça “la concaténation”. On verra ça plus en détail dans les cours à venir.

Exercice 4(1, 2, 3) Le chat bot

**Combinaison d’expressions conditionnelles && et ||**

Dans l’exemple précédent, on a vu que chaque alternative dépendait du résultat d’une seule expression de comparaison de valeurs.

if (maPremiereValeur === maDeuxiemeValeur) { /\* ... \*/

} else if (maPremiereValeur > maDeuxiemeValeur) { /\* ... \*/ }

Dans certains cas, une alternative est définie par la combinaison de plusieurs expressions.

Par exemple, imaginons un programme qui propose comment s’habiller en fonction du temps qu’il fait, en suivant les critères suivants :

* S’il fait beau et chaud, suggérer de porter un short,
* S’il pleut ET qu’il fait chaud, suggérer de prendre un parapluie,
* S’il pleut ET qu’il fait froid, suggérer de prendre un manteau à capuche.

Sachant qu’il est possible d’imbriquer des conditions, on pourrait l’implémenter de la manière suivante :

if (temps === 'beau') {

suggestion = 'short';

} else if (temps === 'pluie') {

if (temperature >= 20) {

suggestion = 'parapluie';

} else if (temperature < 20) {

Suggestion = 'manteau à capuche';

}

}

Mais il est aussi possible d’exprimer nos critères de manière plus linéaire, en combinant les conditions à l’aide de l’opérateur && :

if (temps === 'beau') {

suggestion = 'short';

} else if (temps === 'pluie' && temperature >= 20) {

suggestion = 'parapluie';

} else if (temps === 'pluie' && temperature < 20) {

Suggestion = 'manteau à capuche';

}

Dans cette implémentation, nous avons combiné les critères de temps et de temperature dans une même expression conditionnelle.

Lorsqu’une alternative if/else est définie par plusieurs expressions liées par l’opérateur && (appelé **et**), il faut que *toutes* ces expressions soient vraies afin que les instructions associées à cette alternative soient exécutées.

Il existe aussi un opérateur || (appelé **ou**) qui permet de définir des alternatives qui seront exécutées si *au moins une* des expressions est vraie.

Exemple :

if (rdvPrevu === true || envieDallerAuxToilettes === true) {

message = 'excusez-moi, je vais devoir vous laisser';

} else {

message = 'nous pouvons en discuter tout de suite, si vous voulez !';

}

… ce qui pourrait aussi s’écrire ainsi, sous sa forme développée :

if (rdvPrevu === true) {

message = 'excusez-moi, je vais devoir vous laisser';

} else if (envieDallerAuxToilettes === true) {

message = 'excusez-moi, je vais devoir vous laisser';

} else {

message = 'nous pouvons en discuter tout de suite, si vous voulez !';

}

Dans ce cas, nous voyons que la combinaison d’expression conditionnelle avec || permet de réduire la redondance de notre code, en ne définissant qu’une seule fois une même liste d’instructions correspondante à deux cas alternatifs.

**Indentation et autres conventions à respecter**

Dans le cadre de ce cours, nous allons respecter un extrait des conventions de codage et mauvaises manières d’écrire son code

Extrait des règles qui seront à respecter :

* mettre les chaînes de caractères entre apostrophes,)
* usage et placement des accolades (*braces*, en anglais),
* et indentation de 2 espaces.

**ALGORITHMES**

Une recette de cuisine peut être réduite à un algorithme, si on peut réduire sa spécification aux éléments constitutifs :

* des entrées (les ingrédients, le matériel utilisé) ;
* des instructions élémentaires simples, dont l’exécution amène au résultat voulu ;
* un résultat : le plat préparé.

**Boucle For**

**Qu’est-ce qu’une boucle ?**

Une boucle permet de répeter plusieurs fois une séquence d’instuctions.

Pour afficher les nombres de 1 à 3 dans la console JavaScript, on pourrait utiliser les instructions suivantes :

console.log(1);

console.log(2);

console.log(3);

Et ça fonctionne très bien !

En revanche, le code deviendrait très fastidieux à écrire (et à lire) dans le cas où on voudrait afficher les nombres de 1 à 10000 !

Pour ce genre de répétition, le mot-clé for permet de définir une seule fois les instructions qui doivent êtres répétés, puis de spécifier combien de fois on souhaite qu’elles soient répétées.

Pour afficher les nombres de 1 à 10000, il suffit donc d’écrire le code suivant :

for ( let monNombre = 1; monNombre <= 10000; monNombre++ ) {

console.log( monNombre );

}

On pourrait traduire ce code de la manière suivante :

Pour chaque valeur de monNombre, croissant de 1 à 10000 (compris), afficher la valeur de monNombre dans la console.

**À quoi servent les boucles ?**

Les boucles sont donc très utiles pour éviter les redondances dans un programme (ex : jouer 5 fois le même son, mettre tous les champs d’un formulaire en majuscules…), mais elles sont surtout indispensables dans de nombreuses applications courantes:

* Les jeux tour-par-tour consistent en une boucle qui se termine lorsqu’un joueur remporte la partie ;
* Les jeux d’action utilisent une boucle permettant de mettre à jour l’affichage (frame par frame, pour utiliser la terminologie exacte) en fonction des actions du/des joueur(s) ;
* Ainsi que les algorithmes de tri et de manipulation de données utilisés dans 99% des logiciels.

Javascript fournit quatre mots-clés pour définir des boucles : do, while, until et for. La forme de boucle la plus courante est for car c’est la plus générique / adaptable. Nous allons donc seulement travailler avec des boucles for dans le cadre de ce cours.

**Anatomie d’une boucle for en JavaScript**

Reprenons l’exemple de boucle que nous avons vu plus haut :

for ( let monNombre = 1 ; monNombre <= 10000 ; monNombre++ ) {

console.log( monNombre );

}

Cette boucle est définie par :

* l’usage du mot clé for;
* une liste d’instructions (saisie entre accolades {}) à répéter tant que la condition est vraie: console.log( monNombre ); (dans notre exemple, il n’y a qu’une seule instruction, mais on peut en mettre une infinité);
* une condition (expression conditionnelle, comme dans une condition if): monNombre <= 10000;
* une instruction d’itération qui sera exécutée après chaque itération de la boucle: monNombre++ (qui, ici, incrémente la valeur de monNombre, c’est à dire augmente sa valeur de 1);
* et une instruction d’initialisation qui ne sera exécutée qu’une seule fois: var monNombre = 1 (ici, on créée une variable monNombre et on lui affecte la valeur initiale 1).

On appelle **itération** chaque répétition de la boucle.

Pour synthétiser, voici la syntaxe à utiliser pour définir une boucle for en JavaScript :

for( /\* initialisation \*/ ; /\* condition \*/ ; /\* incrémentation \*/ ) {

/\* instructions à répeter \*/

}

À noter que, dans la plupart des cas, les boucles sont utilisées pour itérer :

* sur un intervalle (dans notre exemple: nombres entiers entre 1 et 10000),
* ou sur une énumération de valeurs (ex: un tableau/Array, comme on le verra plus tard).

**Traçage de l’exécution d’une boucle for**

Afin de mieux comprendre le fonctionnement de la boucle for et de la manière de saisir ces trois paramètres, nous allons interpréter une boucle comme le fait un navigateur web (ou tout autre interpréteur JavaScript).

Prenons la boucle for suivante :

console.log('on va boucler');

for ( let i = 0; i < 4; i++ ) {

console.log('i', i, i < 4);

}

console.log('on a fini de boucler');

Voici la manière dont elle va être interprétée et exécutée par la machine :

console.log('on va boucler'); // => affiche: on va boucler

// interprétation de la boucle => on commence par l'initialisation

let i = 0; // initialisation de la boucle, en affectant 0 à la variable i

// --- première itération de la boucle ---

i < 4 ? // condition vraie, car i vaut 0

console.log('i', i, i < 4); // => affiche: i 0 true

i++; // incrémentation => i vaut maintenant 1

// --- seconde itération de la boucle ---

i < 4 ? // condition vraie, car 1 < 4

console.log('i', i, i < 4); // => affiche: i 1 true

i++; // incrémentation => i vaut maintenant 2

// --- troisième itération de la boucle ---

i < 4 ? // condition vraie, car 2 < 4

console.log('i', i, i < 4); // => affiche: i 2 true

i++; // incrémentation => i vaut maintenant 3

// --- quatrième itération de la boucle ---

i < 4 ? // condition vraie, car 3 < 4

console.log('i', i, i < 4); // => affiche: i 3 true

i++; // incrémentation => i vaut maintenant 4

// --- cinquième itération de la boucle ---

i < 4 ? // condition fausse, car i==4 => fin de boucle

// boucle terminée => on interprète les instructions suivantes.

console.log('on a fini de boucler'); // => affiche: on a fini de boucler

Il est très pratique de décomposer une boucle de cette manière lorsqu’elle ne se comporte pas comme voulu.

**Interrompre l’exécution d’une boucle : break**

Dans certains cas, il est pratique d’interrompre l’exécution d’une boucle, pendant l’exécution d’une de ses itérations (tel que définie entre les accolades de définition de la boucle).

for (let i = 0; i < 10; i++) {

let commande = prompt('entrez une commande');

if (commande === 'quitter') {

break; // => on sort de la boucle for, la suite programme continue de s'exécuter (après ses accolades du for)

}

}

Cependant, l’usage de break est non recommandé, car il rend la logique plus complexe à comprendre en lisant le code. Il est généralement possible et plus élégant d’intégrer la condition de sortie dans la condition de la boucle.

let commande;

for (let i = 0; i < 10 && commande !== 'quitter'; i++) {

commande = prompt('entrez une commande');

}

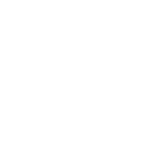
Ici, on a intégré la condition à l’aide de l’opérateur logique &&, donc la boucle continuera d’itérer tant que i < 10 *ET* que commande !== 'quitter'.

**Exercice 3 Les boucles**

**LES FONCTIONS**

**Introduction**

Comme en mathématiques, une fonction transforme des paramètres (en “entrée”) en une valeur de résultat (la “sortie”), lorsqu’elle est appelée. Avant de pouvoir l’appeler, on la définit par une suite d’instructions qui déterminera cette valeur de résultat, en fonction des paramètres qui lui seront passés.



Définir une fonction permet de regrouper des instructions JavaScript, afin qu’elles puissent être exécutées à différentes occasions, sans avoir à dupliquer le code correspondant.

Par exemple, sur le web, les fonctions JavaScript sont utilisées par le développeur pour définir le comportement que doit suivre le navigateur lorsque l’utilisateur effectue certaines actions (ex: saisie dans un champ, clic sur un bouton, soumission d’un formulaire).

**Définition et appel de fonction**

On définit une fonction de la manière suivante :

function nomDeLaFonction (parametre1, parametre2, parametre3 ...) {

// instructions javascript

// pouvant utiliser les paramètres parametre1, parametre2 et parametre3

return resultat;

}

Par exemple :

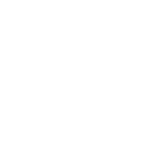
function multiplierParDeux (nombre) {

return nombre \* 2;

}

Pour exécuter une fonction, il faut *l’appeler* en citant son nom, et en lui fournissant des valeurs pour chacun des paramètres entre parenthèses.

Par exemple :



var resultat = multiplierParDeux(3); // => le paramètre nombre vaut 3 => la variable resultat vaudra 6

Comme pour une variable, l’appel à une fonction sera remplacé la valeur qu’elle renvoie, au moment de l’exécution du programme. Contrairement aux variables, cette valeur dépendra de la valeur des paramètres passés à la fonction.

Ainsi, il est possible de passer le résultat de l’appel d’une fonction en paramètre d’une fonction.

Exemple de substitution d’un appel de fonction par sa valeur de retour :

resultat = multiplierParDeux(multiplierParDeux(3)); // équivaut à: resultat = multiplierParDeux(3 \* 2); // qui équivaut à: resultat = (3 \* 2) \* 2; // qui vaut finalement resultat = 12;

Et, avec une autre valeur passée en paramètre:

var resultat = multiplierParDeux(multiplierParDeux(4)); // équivaut à:

var resultat = multiplierParDeux(4 \* 2); // qui équivaut à:

var resultat = (4 \* 2) \* 2; // qui vaut finalement:

var resultat = 16;

**Importance de return**

Quand on exécute une fonction depuis une console JavaScript, la valeur retournée par cette fonction est affichée dans la console. Il ne faut pas pour autant confondre le mot clé return et la fonction console.log.

En effet :

* console.log peut être appelée plusieurs fois depuis un même définition de fonction, mais chaque appel de fonction ne peut résulter qu’en une seule valeur de retour spécifiée par return,
* l’usage de return permet à l’appelant d’une fonction de disposer de la valeur résultante comme il le souhaite: il peut par exemple décider d’afficher cette valeur dans la console, ou dans un alert, ou même de la stocker dans une variable. Or, si la définition de cette fonction affiche la valeur résultante dans la console au lieu d’utiliser return, il sera impossible pour l’appelant de récupérer cette valeur résultante dans son programme.

**Exercice 4 Les fonctions**

**Bugs et tests unitaires : comment tester une fonction**

Appeler une fonction ajoute de l’incertitude et parfois de l’imprévisibilité au comportement du code, car cela revient à déléguer une fonctionnalité à une autre partie du code (la définition de la fonction appelée).

Afin de se rassurer sur le bon fonctionnement d’une fonction et éviter les *bugs*, il est important de tester les fonctions qu’on utilise.

Un *bug* est un comportement imprévu causant des anomalies et/ou l’interruption de l’exécution du programme. Il est généralement causé par une erreur d’implémentation ou une réalisation trop naïve (c.a.d. ne couvrant pas certains cas qui peuvent se produire).

Exemple d’implémentation naïve pouvant causer un bug :

function multiplierParDix (nombre) {

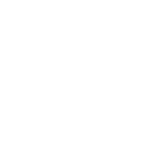
return nombre + '0'; // on ajoute un zéro à la fin du nombre

}

multiplierParDix(2); // => 20 => OK

multiplierParDix(3); // => 30 => OK

multiplierParDix(0.5); // => 0.50 => BUG! on voulait obtenir 5 dans ce cas



Dans l’exemple ci-dessus, nous avons effectué trois tests d’appel de notre fonction multiplierParDix, et l’un d’eux nous a permis de détecter un bug dans notre fonction.

Afin de réduire le nombre de bugs potentiels d’une fonction, et donc de se rassurer sur son bon fonctionnement, il est important d’écrire et exécuter plusieurs tests unitaires, et penser intelligemment aux *cas limites*, les cas qui pourraient le plus probablement causer un bug.

Écrire un test unitaire pour une fonction consiste à:

* Décrire un exemple d’usage de cette fonction, en précisant la valeur résultante attendue pour certaines valeurs de paramètres,
* Implémenter l’appel de cette fonction, et comparer la valeur résultante à celle qui est attendue.

Lors de l’exécution du test unitaire, si la valeur de la comparaison détermine si la fonction fonctionne comme prévu sur l’exemple de ce test.

Par exemple, on pourrait définir les trois tests unitaires suivants pour valider notre fonction multiplierParDix:

multiplierParDix(2) === 20; // => false (car '20' différent de 20)

multiplierParDix(3) === 30; // => false (car '30' différent de 30)

multiplierParDix(0.5) === 5; // => false (car '0.50' différent de 0.5)

Avec la définition de la fonction multiplierParDix fournie plus haut, aucun de ces tests ne *passe*. C’est à dire que chaque test d’égalité sera false.

En revanche, les tests unitaires *passeront* avec la définition suivante de cette même fonction:

function multiplierParDix (nombre) {

return nombre \* 10;

}

multiplierParDix(2) === 20; // => true => OK

multiplierParDix(3) === 30; // => true => OK

multiplierParDix(0.5) === 5; // => true => OK

À retenir : Un test unitaire est un exemple d’appel permettant de vérifier qu’une fonction se comporte comme prévu dans un cas donné, en comparant le résultat effectivement retourné au résultat qui devrait être retourné.

**Valeur et affectation d’une fonction**

Nous avons vu que l’appel d’une fonction consiste à mentionner son nom, suivi de paramètres exprimés entre parenthèses. Et que cet appel est remplacé par la valeur retournée par son exécution.

// définition de la fonction multiplierParDeux()

function multiplierParDeux (nombre) {

return nombre \* 2;

}

// appel de la fonction multiplierParDeux(), en passant le nombre 3 en paramètre

var resultat = multiplierParDeux(3);

// la valeur retournée par l'appel de la fonction (6) est affectée à résultat

Ainsi, multiplierParDeux(3) est remplacé par sa valeur de retour: 6, après l’exécution de la fonction multiplierParDeux à laquelle on a passé la valeur littérale 3 comme valeur du paramètre appelé nombre.

Pour rappel, une variable Javascript est remplacée par la dernière valeur qui lui a été affectée. Ainsi, si la valeur 6 a été affectée à la variable maVariable à l’aide de l’instruction maVariable = 6;, les mentions suivantes de maVariable seront remplacée par sa valeur 6.

En Javascript, une fonction est une valeur, au même titre qu’un nombre ou une chaîne de caractères. Elle peut donc aussi être attribuée à une variable.

Ainsi, il est possible d’affecter la fonction multiplierParDeux à la variable maVariable:

maVariable = multiplierParDeux;

…et de l’appeler de la manière suivante :

maVariable(3); // => retourne la valeur 6;

Il est donc aussi possible d’affecter une fonction anonyme à une variable :

var multiplierParTrois = function (nombre) {

return nombre \* 3;

};

… ce qui est équivalent à écrire :

function multiplierParTrois (nombre) {

return nombre \* 3;

}

**Exercice 6 chi fou mi**