Contents

[1. Introduction à la programmation. Algorithmique 3](#_Toc946811)

[2. Écriture et lecture 4](#_Toc946812)

[1. Écriture 4](#_Toc946813)

[2. Lecture 4](#_Toc946814)

[3. Les variables 5](#_Toc946815)

[4. Typage fort et faible 7](#_Toc946816)

[5. Le type Undefined 8](#_Toc946817)

[6. Expressions et operateurs 9](#_Toc946818)

[1. Expressions Arithmétiques et Operateurs Arithmétiques 10](#_Toc946819)

[2. Opérateur de concaténation 10](#_Toc946820)

[7. Les instructions conditionnelles (IFs) 11](#_Toc946821)

[1. Expressions Logiques: Operateurs Logiques de base (<,>,>=,<=,==,!=) 13](#_Toc946822)

[2. Operateurs Logiques ET (&&), OU (||), NOT (!) 13](#_Toc946823)

[3. L'opérateur == 15](#_Toc946824)

[4. Résultats pour x === y 18](#_Toc946825)

[Exercice: 19](#_Toc946826)

[5. Les IF imbriqués 20](#_Toc946827)

[8. Les Boucles 23](#_Toc946828)

[1. Boucle While 24](#_Toc946829)

[2. Compter en utilisant une boucle 26](#_Toc946830)

[3. Quand est-ce qu'on utilise de boucles ? 28](#_Toc946831)

[4. Boucle For 30](#_Toc946832)

[5. Boucles imbriqués 31](#_Toc946833)

[9. Les fonctions 33](#_Toc946834)

[Les Fonctions tels qu'expressions et tel qu'instructions 34](#_Toc946835)

[10. Les paramètres et le renvoi de valeurs dans les fonctions 35](#_Toc946836)

[11. Les tableaux (I) 37](#_Toc946837)

[Les tableaux et les boucles 40](#_Toc946838)

[12. Les tableaux (II) 41](#_Toc946839)

[13. First-class functions et High Order Functions 42](#_Toc946840)

[1. Stocker une fonction dans une variable ou une autre structure de données (array, objet...) 42](#_Toc946841)

[2. Passer une fonction comme paramètre à une autre fonction 43](#_Toc946842)

[3. Renvoyer une fonction dans un return 45](#_Toc946843)

[4. Arrow functions 46](#_Toc946844)

[14. Contexte Global d'Exécution, l'objet Window 48](#_Toc946845)

[15. Hoisting 49](#_Toc946846)

[16. Exécution de fonctions et pile (stack) d’appels 50](#_Toc946847)

[17. Les contextes d'exécution et la portée de variables (scope) 51](#_Toc946848)

[18. Var: portée de variables (Scope) 53](#_Toc946849)

[19. Let: portée de variables et blocs 56](#_Toc946850)

[20. Const 59](#_Toc946851)

[21. Les IIFE (Inmediatly Invoked Function Expression) 60](#_Toc946852)

[22. Closures 61](#_Toc946853)

[23. Les objets 63](#_Toc946854)

[1. Les objets littéraux (sans classe) 63](#_Toc946855)

[2. Les objets et les arrays 65](#_Toc946856)

[3. L'objet this 66](#_Toc946857)

[4. La fonction call et this 67](#_Toc946858)

[5. La fonction bind et this 69](#_Toc946859)

[6. Les classes 71](#_Toc946860)

[24. Résumé AJAX en JavaScript 73](#_Toc946861)

[25. Les Prototypes 74](#_Toc946862)

[26. Héritage avec Function Constructor 76](#_Toc946863)

[1. Héritage de propriétés 76](#_Toc946864)

[2. Héritage de méthodes 79](#_Toc946865)

[3. Partage de propriétés 81](#_Toc946866)

# Introduction à la programmation. Algorithmique

Pour que l’ordinateur réalise la fonction qu’on souhaite on doit créer un **programme**. Un programme est une suite d’instructions dans un langage de programmation (Javascript, PHP, ASP, Java, Python) qui conduit à un résultat donné (si tout va bien!)

Avant de coder un programme on a besoin de définir **la suite d’opérations que ce programme va réaliser dans un langage compréhensible par l’être humain**. Cette suite d’opérations s’appelle **algorithme**.

**Ex : on veut réaliser un programme qui lit deux valeurs du clavier et affiche la somme de ces deux valeurs.** On ne sait pas encore quel langage on va utiliser mais on peut déjà faire l’algorithme.

Algorithme :

1. Demander à l’utilisateur de saisir une valeur
2. Lire la valeur saisie par l’utilisateur
3. Demander à l’utilisateur de saisir une deuxième valeur
4. Lire la valeur saisie par l’utilisateur
5. Calculer la somme de ces deux valeurs
6. Afficher le résultat

Nous pouvons écrire des algorithmes pour de procédures qui n’ont rien à voir avec l’informatique : cuisiner un bon repas, ouvrir une porte fermée à clé, monter un meuble...

Observez que nous venons d'écrire un algorithme dans le « langage humain ». Logiquement la machine ne comprend pas ce langage. Pour que la machine comprenne, on doit créer le programme à partir de l’algorithme dans un langage de programmation dont vous avez surement entendu parler: Javascript, PHP, Java, C++,...

Dans le cadre de ce cours **on utilisera le langage Javascript**, qui est très similaire à celle de la plupart de langages modernes. Tous les langages de programmation sont basés sur quatre éléments :

1. les instructions de **lecture / écriture (READ/WRITE)**
2. les **variables** (variable= espace de stockage)
3. les **tests ou instructions conditionnelles (IF)**
4. les **boucles (WHILE, FOR)**

# Écriture et lecture

### Écriture

Les instructions d’**écriture** permettent au programme de « envoyer de données à un dispositif de sortie ». Dans notre contexte ça veut juste dire « afficher de données sur l’écran »

(D’un point de vue plus large on peut aussi « écrire » de données sur un disque dur, une carte réseau, une imprimante… pour le moment on se concentre sur l’écran !)

Pour afficher quoi qui ce soit dans la console on utilise **console.log**

**Exemple :**

console.log ("Bienvenu");

console.log (2\*5+6);

### Lecture

Les instructions de **lecture** permettent à l’utilisateur de « recevoir de données d’un dispositif d’entrée ». Dans notre contexte, c’est juste « obtenir les données qu’on saisit au clavier ».

Dans le milieu web, on obtient souvent les données des formulaires HTML. Vu qu'on ne sait pas encore manipuler les éléments de la page, on va utiliser la fonction **prompt** qui affiche une fenêtre où on peut saisir de valeurs. La fonction prompt renvoie la valeur saisie si on clique sur Accepter (toujours un string!) et **null** si on clique sur Annuler.

**Exemple :**

let nomUtilisateur = prompt("Tapez votre nom") ;

# Les variables. Création et affectation

Nous avons parlé de données (« prix », « notes »…) dans un programme informatique, on devra stocker provisoirement des valeurs pour pouvoir les manipuler. Ces valeurs proviennent de l’utilisateur (clavier), du disque dur, d’une base de données qui se trouve dans un autre ordinateur…

Les types de base en Javascript **(types primitifs)** sont:

1. **number** (ex : la taille d'une personne, un code postal, le chiffre pi)
2. **string** (ex: le nom d'une personne, une adresse, un numéro de téléphone)
3. **boolean** ou vrai/faux (ex : un compte bancaire bloquée ou pas bloquée)
4. **undefined**: un type spécial qui indique qu'une variable n'est pas définie
5. **null**: un type pour indiquer qu'une variable n'a pas de valeur (même si elle est définie en faisant **let laVariable**
6. **symbol**: un type spécial qui existe à partir de EC6 (à voir plus tard)

Pour stocker ces valeurs on utilise de **variables**.

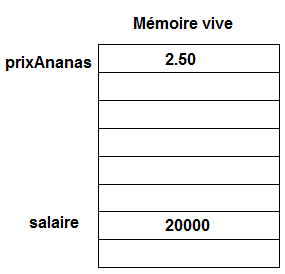
Une **variable peut être vue comme une boite dans la mémoire qui contient une valeur et qui est identifiée par un nom**. Sa valeur et son nom sont stockées dans la mémoire vive. Sa valeur peut être obtenue et modifiée (pas son nom !).

**Ex: quelques variables dans la mémoire vive**

**let** prixAnanas = 2.50;

var salaire= 20000 ;

**Note** : nous pouvons utiliser let ou var selon le contexte où on se trouve. La différence entre les deux sera étudiée plus tard, car ici les deux ont le même effet sur la mémoire. Pour le moment on va utiliser **let** car c'est considéré comme bonne pratique



On peut créer de variables et y stocker de valeurs. Une variable a toujours un nom, un type et une valeur.

Nous allons représenter une **affectation** **d'une variable** par le nom de la variable suivi d'un "=" dans tous les langages de programmation les plus connus. On écrit la nouvelle valeur de la variable juste après le symbole d'égal et sa valeur changera dans la mémoire vive !

On peut **affecter** une variable (changer sa valeur)autant de fois qu'on veut. Par exemple:

Le prix des ananas est maintenant 3 euros... (attention, il n'y a pas de let car la variable existait déjà, il y a juste une affectation)

prixAnanas = 3.00;

les ananas coutent maintenant 1 euro de moins...

prixAnanas = prixAnanas – 1;

et le prix est doublé!

prixAnanas = prixAnanas \* 2;

Considérons maintenant la variable **monPrenom**

let monPrenom = "Ismael";

Si je change mon prénom je dois juste faire:

monPrenom = "Achab";

Pour finir, je peux affecter une variable avec la valeur d'une autre variable. Considérons les variables **prixOrangeTable** et **prixOrangeJus**.

prixOrangeTable = 2.00;

prixOrangeJus = 1.80;

Si on fait :

prixOrangeTable = prixOrangeJus;

le prix des oranges de table sera maintenant de 1.80 euros.

Si on **continue** l'algorithme en faisant:

prixOrangeJus = prixOrangeTable;

la variable prixOrangeJus contiendra 1.80, vu qu’on a changé prixOrangeTable à 1.80 juste avant. **L'ordre des instructions déterminera toujours le résultat de l'algorithme, qui est lancé séquentiellement.**

# Typage fort et faible

En JS nous n'avons pas besoin d'indiquer quel est le type de la valeur qu'on va stocker dans la variable (en PHP non plus) car il s'agit d'un langage faiblement typé.

En C++ ou JAVA (typage fort) :

**int** age = 5;

age = "coucou"; // erreur, car age est declarée int

En JS (typage faible):

**let** age = 5;

age = "coucou";

// pas d'erreur, maintenant JS considère que le type actuel de // la variable est **string**

En JS, nous pouvons afficher le type d'une variable en utilisant la fonction **typeof**:

console.log (typeof (5));

console.log (typeof ("hello"));

console.log (typeof (false));

console.log (typeof (b))

# Le type Undefined

**Undefined est une valeur spéciale pour les variables qui n'ont pas été déclarées ou pour les variables qui n'ont jamais été affectées**:

let i;

console.log (i); // affiche "undefined"

On n'assignera pas undefined aux variables. Si on veut indiquer qu'une variable est vide on l'affectera la valeur **null.**

let i;

i = null;

console.log (i); // affiche "null"

Si vous ne déclarez pas du tout la variable, JS indiquera qu’il y a une « Reference error » car la variable n’existe pas du tout.

# Expressions et operateurs

Prenons quelques instructions qu'on a utilisées dans les algorithmes précédents :

prixOrangeTable = 2.00;

prixOrangeJus = 1.80;

prixAnanas = prixAnanas \* 2;

A gauche du signe **=** il y a **toujours** uniquement un nom de variable, mais à droit il y a ce qu'on appelle une **expression.**

**Une expression est un ensemble de valeurs relié par des opérateurs et équivalent à une seule valeur**. Voici quelques exemples:

**Expression: Valeur de l'expression:**

5+3 8

10-5\*2 0

8/2+3 7

Dans une expression on peut aussi utiliser des variables. Considérons que prixAnanas contient la valeur numérique 3.50 et prixOrangeJus contient 1.80

**Expression: Valeur de l'expression:**

prixAnanas\*2 7

6+prixAnanas 9.5

prixOrangeJus+2 3.80

prixOrangeJus+prixAnanas 5.30

A continuation on va étudier les expressions arithmétiques et les opérateurs arithmétiques.

### Expressions Arithmétiques et Operateurs Arithmétiques

Une expression arithmétique contient de **valeurs numériques connectées par des opérateurs arithmétiques**. Les quatre **opérateurs arithmétiques** classiques sont:

**+ : addition**

**- : soustraction**

**\* : multiplication**

**/ : division**

**Exemples :**

5 + 6 \* 7

23 / 4

prix \* 3

age - 20

prix \* tva - 30

On a le droit d’utiliser les parenthèses en respectant les mêmes règles qu’en mathématiques. La multiplication et la division ont « naturellement » priorité sur l’addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

Cela signifie qu’en informatique, **12 \* 3 + 5** et **(12 \* 3) + 5** valent strictement la même chose, à savoir 41. Pourquoi dès lors se fatiguer à mettre des parenthèses inutiles ?

En revanche, **12 \* (3 + 5)** vaut 12 \* 8 soit 96. Rien de difficile là-dedans, que du normal.

### Opérateur de concaténation

Cet opérateur permet de concaténer, autrement dit d’agglomérer, **deux chaînes de caractères**. En Javascript l'opérateur est le symbole "+":

let nom1 = "Joe";

let nom2 = "Mary";

let nom3 = nom2 + nom1; // on obtient "JoeMary"

Attention quand vous faites une addition si les types sont string!!

let hiffre1 = "5";

let chiffre2 = "87";

let autreVar = chiffre1 + chiffre2; // on obtient "587";

# Les instructions conditionnelles (IFs)

Nous avons découvert déjà deux éléments de la programmation : les **variables** et **la lecture/écriture**. Les **instructions conditionnelles** sont la troisième, et il nous restera uniquement les boucles !

Considérons l’algorithme qui gère l'accès à un site en tenant compte l'âge de l'utilisateur. On pourrait résumer l’algorithme en :

Demander à l'utilisateur de taper son âge

Lire l'âge de l'utilisateur

Vérifier que l'utilisateur a au moins 18 ans

Afficher un message de bienvenue au site

Mais qu’est-ce qui arrive si l'utilisateur est trop jeune ? On devrait agir différemment et montrer un message d’erreur à l’utilisateur au lieu d’entrer dans le système et afficher le bureau !

Dans la programmation on a la possibilité d’**établir deux façons d’agir selon que la situation se présente d’une manière ou d’une autre**, grâce aux **instructions conditionnelles**.

Voici la structure d’une condition en Javascript:

**if** (condition) {

// suite d'instructions 1

}

**else** {

// suite d'instructions 2 (alternative)

}

On utilise des **accolades** pour délimiter chaque section et la **condition** de l'IF se trouve **entre parenthèses**.

Si on reformule notre algorithme pour agir différemment selon la valeur de notre âge on obtient :

Demander à l'utilisateur de taper sa date de naissance

Lire l'âge de l'utilisateur

**if (**l'utilisateur a plus de 18 ans){

Afficher un message de bienvenue au site

**}**

**else {**

Afficher un message d’erreur « Reviens quand t'auras 18! »

}

Le fonctionnement d’une condition est comme suit :

1. On évalue la **condition** du **If**
2. Si la valeur obtenue est **VRAIE (TRUE)**, on lance la **suite d’instructions 1**  et on continue l’exécution de l’algorithme aprèsla dernière accolade du if (la suite d’instructions 1 ne s’exécute pas)
3. Si la valeur obtenue est **FAUSSE (FALSE)**, on lance la **suite d’instructions 2** et on continue l’exécution de l’algorithme aprèsla dernière accolade de l’if (la suite d’instructions 1 ne s’exécute pas)

**Faites cet exemple en Javascript en suivant votre intuition !**

### Expressions Logiques: Operateurs de comparaison (<,>,>=,<=,==,!=)

Une **condition** est une expression qui peut être **VRAIE** ou **FAUSSE. Les conditions sont des « expressions logiques »** (pas arithmétiques !). Voici quelques exemples de base :

age>18

prix<=300

nomUtilisateur == "Marie"

motPasse != "3JK2-35"

Observez que les expressions logiques contiennent **des valeurs connectées par des opérateurs logiques**. Les **opérateurs** **logiques de base** sont :

* strictement plus petit que (**<**)
* strictement plus grand que (**>**)
* plus petit ou égal à (**<=**)
* plus grand ou égal à (**>=**)
* égal à (**==**). **Notez la difference avec le = simple, qui indique l’affectation!**
* différent de (**!=**)

Il existe **de versions de ces opérateurs qui comparent aussi les types: ===, !==**

"500" == 500 // renvoie true

"500" === 500 // renvoie false

### Operateurs Logiques ET (&&), OU (||), NOT (!)

Dans certaines situations les conditions sont plus complexes que celles qu’on a vues jusqu’à présent. Considérez par exemple « l’âge est entre 12 et 18 ». On pourrait dire aussi : « l’âge est supérieure à 12 **ET** l’âge est inférieure à 18 ». On est en train de combiner deux conditions.

(age>12 **&&** age<18)

On vient d’utiliser un autre **opérateur logique**, le &&**.** Cette expression logique vaut TRUE si l'âge vaut entre 13 et 17 et false dans le reste des cas.

**Operateur && : l'expression Condition1 && Condition2** sera vraie uniquement si les deux conditions sont vraies

Ex : age>12 **&&** age<18

Sera vrai si l’âge est supérieur à 12 et inférieure à 18

**Operateur OU : l'expression** **Condition1 || Condition2** sera vraie si au moins une de conditions est vraie

Ex : nombreEnfants>2 **||** revenus<10000

**Operateur NOT: l'expression** **NOT (Condition1)** sera vraie si la condition est fausse.

Ex :

**!**(etatCompte == "bloquée")

**!**(etatCivil == "marié")

On peut se demander pourquoi utiliser NOT si on pourrait faire juste:

etatCompte != "bloquée"

etatCivil != "marié"

C’est dans la programmation pratique où cet opérateur logique va s’avérer très utile, et il sera souvent exprimé avec le symbole "**!**" au lieu de "NOT".

### L'opérateur ==

1. Si **x** et **y ont le même type**
   1. Si **Type(x)** est **String**

1. Si x est la même séquence que y, on obtient true : "hello" == "hello". Sinon, false.

* 1. Si **Type(x)** est **Boolean**

1. Si x a la même valeur que y, on obtient true : false == false. Sinon, false.

* 1. Si **Type(x)** est **Number**

1. Si x est la même valeur numérique que y, on obtient true car 2 == 2

2. Si x est une valeur numérique différente de y, on obtient false car 2 != 1

3. Si x est NaN (nombre invalide), on obtient false car NaN != NaN

4. Si y est NaN (nombre invalide), on obtient false car NaN != NaN

5. Si x est +0 et y est −0, on obtient true car 0 == 0

6. If x est −0 et y est +0, on obtient true car 0 == 0

* 1. Si **Type(x)** est **Undefined**, on obtient true car undefined == undefined
  2. Si **Type(x)** est **Null**, on obtient true car null == null
  3. Si **Type(x)** est **Object**, on obtient true si x et y font référence au même objet

1. Si **x et y n'ont pas le même type**
   1. Si l'**un de deux types est String et l'autre Number**, le String est converti à Number et après on réalise la comparaison

2=="5" // on convertira "5" en 5

"100"==8 // pareil. On convertira "100" en 100

* 1. Si l'**un de deux types est Boolean et l'autre Number**, le Boolean est converti à Number et après on réalise la comparaison (false devient 0 et true devient 1)

0==false // true

2==true // false

1==true // true

* 1. **Si l'un de deux types est null et l'autre undefined**, on obtient **true.** Si on compare undefined ou null avec un autre type on obtient toujours false

undefined == null // renvoie true

false == null // renvoie false

* 1. Si **x est String ou Number et y est un Object**, on réalise **x == valueOf (y)** or **x == toString (y)**

let one = {

valueOf: function () {

return 1;

},

toString: function () {

return "2";

}

};

alert(one == 1); // true

alert(one == "2"); // false

Si on efface valueOf, JS appelle toString et il compare one == "2" ou {} == "[object Object]", alors on obtient false

Pour savoir si un paramètre d'une fonction contient une valeur, on realisera cette vérification

function something(arg) {

if (arg == null) {

}

}

car JS ne cherchera pas la variable dans un autre contexte tel qu'elle ferait si on comparait avec undefined

* 1. **Les valeurs false, 0 et "" sont équivalents et comparables entre eux**

false == 0; // true

false == ""; // true

null == ""; // false

0 == ""; // true

* 1. **La comparaison avec NaN renvoie toujours faux**

Ex:

NaN == NaN renvoie false

NaN == false renvoie false

Pour savoir si une variable contient un Number valable on utilise la fonction isNaN:

let a = 10/2;

console.log (isNaN(a)); // false

let b = 5;

let c = "hello";

console.log (isNaN(b/c)); // Number / String est NaN, true

### Résultats pour x === y

Pour éviter les doutes dans les comparaisons avec **==** on peut utiliser **la comparaison stricte** **===**. C'est une comparaison plus stricte que celle de l'opérateur == car === **réalise une comparaison de types**. **Si le type n'est pas le même, JS renvoie faux au lieu de convertir les types**

Ex.:

false === false renvoie true

false === "false" renvoie false car un est false et l'autre String

**Si vous ne maitrisez pas l'utilisation de ==, utilisez toujours ===. C'est important quand-même de connaitre == pour comprendre le code d'autres developpeurs.**

#### Exemples opérateur de comparaison :

comparaisonTypes.html

Exercice:

**Évaluez les expressions suivantes**

3>5

4!=5

4=="4"

false=="false"

"bonjour" == "salut"

3<4 && 9<8

3<4 OR 9<8

9<2 && 4>3 && 1<8

9>2 && 8>5 && 8>=1

8>=8

9<2 OR 4<5 && 1<8

(9<2 OR 4<5) && 1<8

4<9

!(4<9)

!(9<2 OR 4<5)

!(9<2) OR 4<5

!(9<2) && !(4>10)

5==4 OR (5 < 4+3)

true && false

true && false OR (4<8)

### Les IF imbriqués

Considérez un algorithme qui nous indique l’état de l’eau selon sa température (solide, liquide, gazeuse).

Une première solution serait :

let temp= **prompt (**"Entrez la température de l’eau :"**)**;

**if** (temp <= 0 ) {

**console.log (**"C’est de la glace"**)**;

}

**if** (temp > 0 **&&** temp < 100 ) {

**console.log (**"C’est du liquide"**)**;

}

**if** (temp > 100 ) {

**console.log (**"C’est de la vapeur"**)**;

}

**console.log (**"Au revoir"**)**;

Construire une condition de cette manière n'est pas la meilleure solution : la machine est obligée de réaliser tous les tests (temp<=0 ? temp>0 et <100 ? temp >100 ?), même s’ils sont mutuellement exclusifs...

Nous pouvons structurer le test d’une autre façon pour obtenir le même résultat, grâce **à l'imbrication d'instructions conditionnelles** :

let temp= **prompt (**"Entrez la température de l’eau :"**)**;

**if** (temp <= 0){

**console.log (**"C’est de la glace");

}

**else if** (temp < 100){

**console.log (**"C’est du liquide");

}

**else**{

**console.log (**"C’est de la vapeur");

}

**console.log (**"Au revoir");

On explique ce code à continuation. Observez qu’il y a un nouveau mot « **else if** »

**Explication :**

On exécute la première instruction conditionnelle (Temp<=0)  et

1. Si c’est VRAI, la machine lance la suite d’instructions jusqu’au **else if** et après elle saut à l’instruction après la dernière accolade du if (console.log (« au revoir »). Le reste d'instructions conditionnelles ne se lanceront pas, on n’a plus besoin.
2. Si c’est FAUX, la machine fait le test du else if (Temp<100?).
   * Si la condition est Vrai, on lance la suite d’instructions correspondante et après on saut à l’instruction après la dernière accolade du if
   * Si la condition est Faux on sait déjà que c’est de la vapeur : la température n’est pas <0 ni <100… alors elle est >=100 !

Remarquez aussi que si les deux premiers conditions ne sont pas vraies nous ne sommes même pas obligés de lancer le troisième if: si temp n'est pas inférieure à 0 ni inférieure à 100…. temp est forcément >=100 et l’eau est forcément de la vapeur!

On aurait pu mettre autant de else ifs qu’on veut, il suffit de bien placer les accolades.

# Les Boucles

Les boucles sont la quatrième et dernière structure de programmation. Elles permettent de **répéter une suite d'instructions**: on les appelle **structures répétitives** ou **structures itératives**.

Considérez le cas d'un logiciel de console (pas une page web avec un formulaire) qui pose une question à l'utilisateur. L'utilisateur doit répondre Oui ("O") ou Non ("N"):

**let rep = prompt (**"Voulez-vous un chocolat ? (O/N)";

**if** (rep == "O"){

**console.log (**"Tenez votre chocolat";

}

**else if** (rep == "N") {

**console.log (**"Ok, pas de chocolat!";

}

Si l'utilisateur se trompe (ce qui arrive tout le temps!) et saisie une autre valeur que "O" ou "N", le programme n'affichera rien. On peut améliorer le programme: si l'utilisateur se trompe, on va lui demander de saisir une valeur à nouveau:

**let rep = prompt (**"Voulez-vous un chocolat ? (O/N)";

**if** (rep!="O" **&&** rep!="N"){

**console.log (**"Saisie erronée. Recommencez.";

rep= read();

}

**if** (rep == "O"){

**console.log (**"Tenez votre chocolat";

}

**else if** (rep == "N") {

**console.log (**"Ok, pas de chocolat!";

}

L'utilisateur a le droit à se tromper une fois en tapant autre que "O" ou "N", mais… qu'est-ce qui se passe s'il se trompe une deuxième fois? Si on veut permettre à l'utilisateur de se tromper plusieurs fois, on devrait utiliser plusieurs IF:

let rep **= prompt(**"Voulez-vous un chocolat ? (O/N)";

**if** (rep!="O" **&&** rep!="N"){

**console.log (**"Saisie erronée. Recommencez.";

rep= **prompt(**"Voulez-vous un chocolat ? (O/N)";

}

**if** (rep!="O" **&&** rep!="N"){

**console.log (**"Saisie erronée. Recommencez.";

rep= **prompt(**"Voulez-vous un chocolat ? (O/N)";

}

**if** (rep == "O"){

**console.log (**"Tenez votre chocolat";

}

**else if** (rep == "N") {

**console.log (**"Ok, pas de chocolat!";

}

.

.

.

Pas terrible! Pourquoi?

La **solution** à ce problème passe par l'utilisation d'une **boucle**. Une boucle nous permet de répéter une suite d'instructions: dans notre cas on vuex répéter la lecture de la réponse de l'utilisateur jusqu'à elle nous convienne. Il y existe plusieurs types de boucles, dont les plus importants sont **while** et **for**. Bien qu'on puisse tout faire avec la boucle **while**, les boucles **for** nous aident à simplifier le code et on les utilise partout.

### Boucle While

**while** (condition) {

…

Instructions

}

**Boucle While:** Le programme arrive sur la ligne **While** et il examine la condition (ou une valeur booléenne directement). Si la condition est vraie il exécute les instructions à l'intérieur des accolades. Le processus se répète pendant que la condition soit vraie.

**Exemple:**

Le programme teste la valeur de la variable rep et répète le code dans la boucle pendant que la valeur est différente de "O" et "N".

En **Javascript (attention aux accolades!)**

let rep= **prompt (**"Voulez-vous un chocolat ? (O/N)";

**while** (rep!="O" **&&** rep!="N"){

rep= **prompt (**"Voulez-vous un chocolat ? (O/N)";

}

**if** (rep == "O"){

**console.log (**"Tenez votre chocolat";

}

**else if** (rep == "N") {

**console.log (**"Ok, pas de chocolat!";

}

### Compter en utilisant une boucle

On peut utiliser une boucle pour compter ou pour réaliser une action un nombre défini de fois.

**Exemple:** afficher les valeurs de 0 à 15

let compteur = 0;

**while** (compteur<= 15){

**console.log (**compteur;

compteur = compteur + 1;

}

Pour compter en utilisant une boucle while on doit d'abord **créer une variable compteur et l'initialiser**.

Cette variable doit être **incrémentée dans la boucle, ou on restera indéfiniment bloqués à l'intérieur!**. Si ça arrive, on est dans une **boucle infinie**. Cette erreur peut avoir de conséquences très importantes.

**On peut aussi utiliser ce système pour exécuter une suite d'instructions un nombre défini de fois. Dans ce cas, le compteur sert uniquement à déterminer le nombre de répétitions.**

**Exemple:** lire le nom de 5 personnes et leur dire bonjour à chacune. Dans ce cas, le compteur nous sert uniquement à répéter les instructions, mais on ne l'utilise nulle part ailleurs.

let compteur **= 0;**

**while** (compteur<= 5){

nom= prompt ("Tapez un nom");

**console.log (**"Bonjour " . nom;

compteur = compteur + 1;

}

**ATTENTION:**

J'insiste sur le fait que la condition de la boucle doit devenir fausse à un certain moment. Si non, vous créerez une boucle infinie qui, sauf dans certains cas très précis, bloquera votre application et peut avoir de conséquences graves.

**Exemple:** quel est le problème de cette boucle? Et la solution?

let compteur = 0;

**while** (compteur<= 5){

nom= prompt("Tapez un nom");

**console.log (**"Bonjour " . nom;

}

**Exemple:** quel est le problème de cette boucle? Et la solution?

**while** (compteur<= 50){

nom= prompt("Tapez un nom");

**console.log (**"Bonjour " . nom;

compteur + 1;

}

### Quand est-ce qu'on utilise de boucles ?

* Répéter une action x fois: condition dépendante d'un compteur

Ex: afficher dix fois le message "Bonjour"

* Répéter une action jusqu'au changement de la valeur d'une variable qui fasse partie dans la condition

Ex: voulez-vous un chocolat? Tapez (oui/non)

* Combiner les deux exemples précédant

Ex: voulez-vous un chocolat? Tapez (oui/non). L'utilisateur peut se tromper deux fois

* Répéter une action x fois et accumuler un résultat dans une ou plusieurs variables qu'on utilisera en dehors de la boucle

Ex: additionner les chiffres entre 10 et 20 (incluses)

/\* Exemple du café, sans compteur \*/

***let*** reponse **=** prompt **(**"Voulez vous un café?"**);**

***while*** **(**reponse **!=** "oui" **&&** reponse **!=** "non"**){**

reponse **=** prompt **(**"Svp. tapez oui ou non"**);**

**}**

// c'est oui ou non

***if*** **(**reponse **==** "oui"**){**

console.log **(**"Tenez votre café"**);**

**}**

***else*** **{**

console.log **(**"Pas de café!"**);**

**}**

/\* Exemple des bombons: compteur \*/

***let*** nombreBombons **=** 20**;**

***while*** **(**nombreBombons **>** 0**){**

console.log **(**"Il vous reste " **+** nombreBombons**);**

console.log **(**"Tenez un bombon"**);**

nombreBombons **=** nombreBombons **-** 1**;**

**}**

console.log **(**"Plus de bombons!"**);**

/\* Exemple d'essais: variable + compteur \*/

***let*** chiffreDeviner **=** 8**;**

***let*** nombreEssais **=** 10**;**

***let*** reponse **=** prompt **(**"Tapez un chiffre entre 1 et 10"**);**

***while*** **(**nombreEssais **>** 0 **&&** chiffreDeviner **!=** reponse**){**

reponse **=** prompt **(**"Essayez encore!"**);**

nombreEssais **=** nombreEssais **-**1**;**

**}**

***if*** **(**reponse **==** chiffreDeviner**){**

console.log **(**"Gagné!"**);**

**}**

***else*** **{**

console.log **(**"Perdu! Plus d'essais!"**);**

**}**

### Boucle For

La boucle For est une façon de simplifier le comptage à l'utilisateur. Observez bien cette boucle While, qu'on a déjà vu:

let compteur=0;

**while** (compteur< 5){

**console.log (**compteur);

compteur = compteur + 1;

}

Elle fait exactement la même chose que cette boucle **For:**

**for** (var compteur=0;compteur<5;compteur=compteur+1){

**console.log (**compteur;

}

Il y a juste deux détails concernant la syntaxe, mais la fonction est la même:

1. dans la deuxième boucle **la variable est initialisée dans la boucle même**
2. dans la deuxième boucle **on incrémente le compteur dans la boucle même**. il passera par tous les valeurs entre 0 et 4, alors la boucle s'exécutera 5 fois)

**Boucle For**

Quand le programme arrive par la première fois à la ligne **For**,il **assigne au compteur la valeur Initial, vérifie la condition (ex: compteur< valeurFinale)** et **lance une première fois la suite d'instructions.**

Une fois exécutée la suite d'instructions, il **incrémente le compteur** et **revérifie la condition.** Si c'est le cas, il lance à nouveau la suite d'instructions, incrémente le compteur et revient à la vérification. Si non, il sort de la boucle.

### Boucles imbriqués

De la même manière qu'on a des **if** successifs (un après l'autre) et imbriqués (un à l'intérieur de l'autre) on peut avoir de boucles successifs et de boucles imbriquées.

Faisons quelques exemples :

**Boucles successifs :**

**for** (let compteur1=1;compteur1<15;compteur1=compteur1+1){

**console.log (**"Il est passé par ici";

}

**for** (let compteur2=1;compteur2<5;compteur2=compteur2+1){

**console.log (**"Il repassera par là";

}

Ce premier cas est simple: il y aura quinze écritures consécutives de "il est passé par ici", puis cinq écritures consécutives de "il repassera par-là".

**Boucles imbriqués :**

**for** (let compteur1=1;compteur1<15;compteur1=compteur1+1){

**console.log (**"Il est passé par ici";

**for** (let compteur2=1;compteur2<5;compteur2=compteur2+1){

**console.log (**"Il repassera par là";

}

}

Ici, le programme écrira une fois "il est passé par ici" puis cinq fois de suite "il repassera par là", et ceci quinze fois en tout. A la fin, il y aura donc eu 15 x 5 = 75 passages dans la deuxième boucle (celle du milieu), donc 75 écritures

**Pourquoi imbriquer les boucles ?**

Une **boucle, c’est un traitement systématique, un examen d’une série d’éléments un par un** (par exemple, « prenons tous les employés de l’entreprise un par un »).

On peut imaginer que **pour chaque élément** ainsi considéré (pour chaque employé), on doive procéder à **un examen systématique d’autre chose** (« prenons chacune des commandes que cet employé a traitées »). Voilà un exemple typique de boucles imbriquées : on devra programmer une boucle principale (celle qui prend les employés un par un) et à l’intérieur, une boucle secondaire (celle qui prend les commandes de cet employé une par une).

Les boucles imbriquées sont **indispensables pour, par exemple, parcourir un tableau de deux (ou plus) dimensions**: pour chaque ligne du tableau on doit parcourir tous les éléments (chaque colonne). Les résultats d'une requête à une base de données sont traités de cette façon.

# Les fonctions

Une fonction nous permet de réutiliser le code. Dans sa forme la plus simple, une fonction a une déclaration et un appel :

1. **Déclaration**

function bonjour (nom){

console.log ("Bonjour "+nom);

}

1. **Appel**

// pas de valeur de retour pour cette fonction

bonjour ("Annie");

bonjour ("Melissa");

Une fonction peut optionnellement renvoyer une valeur avec **return**. On peut stocker la valeur renvoyé dans une variable qu'on utilisera à posteriori :

function calculePrixReduit (prix, reduction){

return (prix \* (1-reduction/100));

}

// on capture la valeur de retour dans une variable

prixReduit = calculePrixReduit (200,10);

console.log ("Le prix du produit en promo est de: " + prixReduit);

// ou on l'utilise directement dans une expression

if (calculePrixReduit (300,25)>200){

console.log ("Le produit reste cher !");

}

### Les Fonctions tels qu'expressions et tel qu'instructions

On peut créer une fonction de deux façons : comme instruction et comme expression.

Voici les deux exemples :

1. **Fonction tel qu'instruction**

***function*** bonjour **(){**

console.log **(**"Bonjour à toutes!"**);**

**}**

// Quand le moteur de JS lit cette déclaration, aucune

// valeur n'est générée

// Pour l'appeler

bonjour**();**

1. **Fonction tel qu'expression**

***var*** salut **=** ***function*** **(){**

console.log **(**"Salut à toutes!"**);**

**}**

// Quand le moteur de JS lit cette déclaration, une valeur est

// générée et stockée dans une variable

// Cette fois la fonction n'a pas de nom. Pour l'appeler on utilise le

// nom de la variable

// pour l'appeler:

salut**();**

Le fait de stocker de fonctions dans de variables nous permet, par exemple, d'avoir un array de fonctions.

Nous verrons d'autres utilités de stocker les fonctions comme expressions plus tard.

# Les paramètres et les valeurs de retour

Une fonction **peut avoir besoin de données d'entrée pour réaliser sa tâche**. Considérez par exemple une fonction qui calcule l'addition de deux chiffres : on doit fournir les deux chiffres à la fonction pour qu'elle réalise le calcule et puisse afficher un résultat.

Si on crée une fonction tel qu'instruction on doit fournir les paramètres comme **une liste de variables** (liste d noms génériques) **entre parenthèses juste après le nom de la fonction**.

Voici la déclaration :

***function*** additionAffiche **(**val1 **,** val2**){**

console.log **(**"J'affiche l'addition: "**);**

console.log **(**val1 **+** val2**);**

**}**

On fournira de valeurs pour ces paramètres quand on réalise l'appel :

additionAffiche **(**100**,**300**);** // affiche 400

additionAffiche **(**20**,**40**);** // affiche 60

On peut très bien créer la fonction en tant qu'expression, rien ne change sauf la position de la liste de paramètres (maintenant elle se trouvera après le mot **function**)

***let*** additionAffiche **=** ***function*** **(**val1**,**val2**){**

console.log **(**"J'affiche l'addition: "**);**

console.log **(**val1**+**val2**);**

**}**

additionAffiche**(**5**,**6**);** // cette fonction ne renvoie rien, elle affiche l'addition

console.log **(**additionAffiche**(**9**,**10**));** // la fonction est lancée mais on voit undefined car elle ne renvoie rien (pas de return)

Dans la plupart de cas, les fonctions ne font pas qu'afficher un résultat. Qu'est-ce qu'on pourrait faire si on voulait par exemple afficher le résultat des additions d'abord dans un DIV et après dans une table ? Ou si on voulait utiliser cette addition pour réaliser d'autres calculs sans rien afficher ?

On cherche maintenant de créer de fonctions **qui produisent un résultat et le renvoient pour qu'il soit utilisé dans le code principal** de notre script.

Le mécanisme est simple :

1. La fonction prend (ou pas) de paramètres d'entrée
2. La fonction réalise un ensemble d'opérations et génére un résultat qui sera stocké dans une seule variable
3. Avec l'instruction **return** la fonction renvoie le résultat au code principal.
4. Dans le code principal on utilise le résultat généré par la fonction directement ou on le stocke dans une variable pour l'utiliser après

Voici un exemple :

***function*** additionRenvoie **(**val1**,**val2**){**

// la fonction génére un résultat mais n'affiche rien

// le résultat est stocké dans une variable

***let*** res **=** val1 **+** val2**;**

// la fonction renvoie cette variable

***return*** res**;**

**}**

On réalise l'appel : **L'appel à la fonction prendra la valeur de la variable spécifiée dans le return**

console.log **(**additionRenvoie**(**5**,**6**));** // pareil que faire "console.log (11)"

alert **(**additionRenvoie**(**10**,**20**));**

console.log **(** 30 **\*** 2 **+** additionRenvoie**(**10**,**20**));**

***var*** val3 **=** additionRenvoie **(**10**,**20**);** // on stocke le résultat pour l'utiliser après

***var*** val4 **=** additionRenvoie **(**10**,**90**)** **\*** additionRenvoie **(**10**,**20**);**

console.log **(**val3**);**

console.log **(**val4**);**

On aurait pu aussi déclarer la fonction en tant qu'expression. **Rien ne change dans les appels!** :

***let*** additionRenvoie **=** ***function*** **(**val1, val2**){**

console.log **(**"Je renvoie l'addition mais je ne l'affiche pas"**);**

***return*** **(**val1 + val2**);**

**}**

# Les tableaux

Considérez que vous avez besoin, dans votre application web d'achat online, de manipuler simultanément le prix de 10 produits différents dans un panier pour calculer le prix total.

Notre premier reflex est de créer une variable pour chaque prix: prix1, prix2 prix3… prix10.

Imaginez maintenant qu'il y a 100 produits: err…. ce système est impossible à gérer! On devrait créer à la main 100 variables!

Les **tableaux (arrays)** nous permettent de rassembler les variables indépendantes dans une seule variable, référencée par un seul nom.

Dans notre cas, on va créer un tableau nommé **listePrix** de 10 positions qui contiendra un prix dans chaque position.

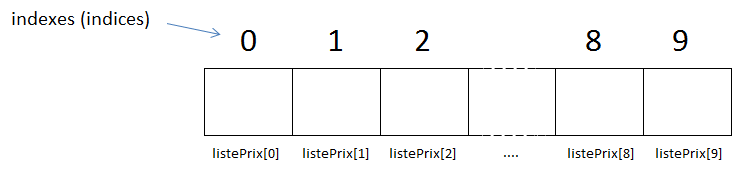
Pour créer l'array on va utiliser la ligne suivante:

**let listePrix = [];**

**Note: dans plein de langages de programmation on peut spécifier la taille initiale d'un array dans la forme:**

**listePrix = int[10]; // array de 10 entiers**

**Mais en Javascript les arrays sont dynamiques et on n'indique pas leur taille lors de sa création**

****

Pour accéder à chaque **valeur** de l'array (chaque prix dans notre cas), on utilise le nom de la variable suivie d'un index **(clé)** entre crochets. Voici quelques exemples:

listePrix[0] = 23; // modifier la 1ere valeur, accessible par la clé 0

listePrix[1] = 45; // modifier la 2ème valeur (clé 1)

// avec 45

**console.log (**listePrix[1]); // afficher la 2ème valeur

listePrix[1]= prompt("Tapez une valeur");

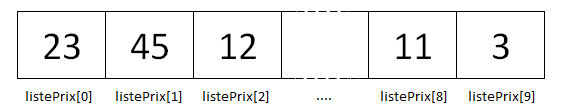
// stocker le contenu saisi

// par l'utilisateur dans la position 2

**console.log (**listePrix[1]); // afficher la deuxième valeur

listePrix[9] = listePrix[0]-20; // réduire le prix de la valeur

// dans la position 9

****

La notation de crochets nous sert à accéder au tableau.

On peut réaliser plusieurs opérations avec les tableau. Les plus importantes sont:

1. **Rajouter un élément au tableau**

listePrix.push (20); // rajoute un élément à la fin de l'array

listePrix [10]=90; // rajoute un élément (ou écrase l'existant)

// dans la position 10

1. **Accéder au contenu d'un élément du tableau**

Ex1: afficher le troisième prix

console.log (listePrix [2]\*10);

1. **Affecter les éléments du tableau**

Encore des exemples:

listePrix[5] = 50;

listePrix[5] = listePrix[3];

listePrix[9] = 2\*listePrix[2];

val= prompt("Tapez une valeur");

listePrix[2]= val;

1. **Obtenir la taille du tableau**

console.log (tab.length);

1. **Extraire le dernier élément du tableau**

val = tab.pop();

1. **Rajouter un élément dans une position "n"du tableau**

let pos = 2;

let nouveauPrix = 99;

listePrix.splice (2,0, 99);

"splice" n'est pas proprement "insérer", mais une sorte de "coller". Concrètement son format est :

**unArray.splice (position1, quantité, contenu)**

Le fonctionnement est un peu particulier : splice va **insérer** une valeur dans la position **position** et va **effacer** un nombre l'éléments (**quantité**)à partir de cette position. Du coup, pour insérer on doit juste lui indique d'effacer 0 éléments.

Exemples :

***let*** ages **=** **[** 10**,** 20**,** 50**,** 70**,** 90**,** 100**,**200**,**300**];**

ages.splice **(**3**,**0,999**);** // rajouter l'élément 999 après 50

1. **Effacer un élément d'un tableau**

Pour effacer des éléments d'un tableau, on va utiliser aussi la fonction splice. On doit juste omette le paramètre contenu.

Exemples :

***let*** ages **=** **[** 10**,** 20**,** 50**,** 70**,** 90**,** 100**,**200**,**300**];**

ages.splice **(**3**,**1**);** // effacer 70

ages.splice **(**1**,**4**);** // effacer 20, 50, 90 et 100

Nous n'avons pas une fonction "remove" ou "unset" tel que dans d'autres langages.

Voir la documentation ici :

<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/splice>

**Important!**

1. Un tableau **peut contenir des variables de toute sorte** (numériques, texte, booléens, même d'autres tableaux !). Normalement, tous les éléments d'un tableau sont du même type.
2. Les indices sont toujours **positifs et commencent par 0, pas par 1**! L'index 1 est déjà la deuxième position du tableau
3. Les **clés (ou** **indices) sont souvent de nombres entiers**. Si on utilise un index pas numérique (ex: produits['bananes'] = 30) l'array sera transformé en objet et certains méthodes qu'on applique sur les objets ne fonctionneront plus (ex: length)

### Les tableaux et les boucles

On apprécie vraiment les avantages des tableaux quand on les utilise avec de boucles :

**Ex: afficher les éléments du tableau listePrix** (déjà rempli)

**for** (compteur=0;compteur<10;compteur=compteur+1){

**console.log (**listePrix [compteur]);

}

Ex: lire 5 noms du clavier et les stocker dans un tableau

let listeNoms = **[]**;

**for** (compteur=0;compteur<5;compteur=compteur+1){

val= prompt ("Tapez un nom");

listeNoms[compteur]=val;

}

Pour simplifier la notation, normalement on utilise les noms **i**, **j**, **k** ou même **n** pour les variables compteur:

listeNoms = **[]**;

**for** (i=0;i<5;i=i+1){

val= prompt ("Tapez un nom");

listeNoms[i]=val;

}

# Les fonctions appliquées sur les tableaux

Considérons cet array :

let villes = ['Bruxelles', 'Rome'];

Voici quelques **fonctions** importantes qu'on peut appliquer sur les tableaux :

1. Créer une copie d'un array indépendante de l'array original

let villesCopie = villes.slice();

Si on modifie l'array villes, villesCopie ne sera pas modifié. Ceci n'arrive pas si on fait juste :

let villesReference = villes;

Dans ce cas on a créé une référence au tableau original (villes). Si on modifie l'un, l'autre sera aussi modifié. Testez-le par vous-mêmes.

1. Effacer le dernier élément (en même temps qu'on le stocke dans une variable)

let derniere = villes.pop();

1. Effacer le premier élément de l'array (shift efface le premier élément et le renvoie)

let premiere = villes.shift();

1. Rajouter un élément au début de l'array

villes.unshift('Varsovie');

1. Obtenir la position d'un élément de l'array

let pos = villes.indexOf('Rome');

1. Effacer un élément dans une position (position 2, la fonction renvoie l'élément effacé)

let elementEfface = villes.splice(pos, 2);

1. Effacer les éléments à partir d'une position (y incluse). La fonction renvoie un array contenant les éléments effacés

let noms = ['Sarah', 'Lucie', 'Alex', 'Jeanne'];

console.log(noms);

**pos** : position à partir de laquelle on commence à effacer

**nombreElements** : nombre d'éléments à effacer

let pos = 1, nombreElements = 2;

let elementsEffaces = noms.splice(pos, nombreElements);

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

# First-class functions et High Order Functions

Le terme "**first-class functions" est une expression pour indiquer que dans un langage on peut** :

1. **Stocker une fonction dans une variable ou une autre structure de données (array, objet...).** Notez que nous avons réalisé cette opération dans la section précédente.
2. **Passer les fonctions comme paramètre d'une autre fonction**
3. **Renvoyer une fonction dans un return**

Nous ne créons pas proprement une "first class function". Ce terme sert uniquement à spécifier que **dans un langage les fonctions ont la même catégorie que d'autres types du langage** (pour pouvoir les stocker dans de variables, les passer comme paramètres ou les renvoyer).

Par contre, **on peut créer une High Order Function**. Une High Order Function est tout simplement **une fonction qui manipule d'autres fonctions** (en les utilisant comme **variables**, **paramètres** ou dans un **return**)

**Un langage qui considère les fonctions comme éléments de première classe (First Class Functions) permet de créer des High Order Functions**

Voici quelques exemples de l'utilisation des fonctions comme des éléments de première classe.

### Stocker une fonction dans une variable ou une autre structure de données (array, objet...)

##### Stockage dans une variable

***let*** salut **=** ***function*** **(){**

console.log **(**"Salut à toutes!"**);**

**}**

// les appels se font de la même manière

salut**();**

##### Stockage dans un array

Exemple: D'abord on crée trois fonctions qu'on va stocker dans un array, puis on fait une boucle pour faire appel à chaque fonction de l'array

***let*** uneFonction **=** ***function*** **(){**

console.log **(**"J'aime le sushi!"**);**

**}**

***let*** autreFonction **=** ***function*** **(){**

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

console.log **(**i**\***2**);**

**}**

**}**

***let*** encoreAutreFonction **=** ***function*** **(){**

***let*** nom **=** prompt **(**"Vous êtes?"**);**

console.log **(**"Bienvenu " **+** nom**);**

**}**

***let*** operations **=** **[**uneFonction**,** autreFonction**,** encoreAutreFonction**];**

// on fait appel à chaque fonction dans l'array

***for*** **(**i**=**0**;**i**<**operations.length**;**i**++){**

operations**[**i**]();**

**}**

### Passer une fonction comme paramètre à une autre fonction

// version 1: on crée les fonctions d'abord

***let*** somme **=** ***function*** **(**a**,**b**){**

***return*** **(**a**+**b**);**

**}**

***let*** multiplication **=** ***function*** **(**a**,**b**){**

***return*** **(**a**\***b**);**

**}**

// voici une High Order Function, car elle reçoit une function comme paramètre

***let*** calculatrice**=** ***function*** **(**a**,**b**,**quoiFaire**){**

***return*** **(**quoiFaire**(**a**,**b**));**

**}**

//console.log ("La somme: " + calculatrice(8,9, somme));

//console.log ("La multiplication: " + calculatrice(8,9,multiplication));

// version 2: on crée les fonctions dans l'appel

***let*** calculatrice**=** ***function*** **(**a**,**b**,**quoiFaire**){**

***return*** **(**quoiFaire**(**a**,**b**));**

**}**

console.log **(**"La somme: " **+** calculatrice**(**8**,**9**,** ***function*** **(**a**,**b**){**

***return*** **(**a**+**b**);**

**}));**

console.log **(**"La multiplication: " **+** calculatrice**(**8**,**9**,*function*** **(**a**,**b**){**

***return*** **(**a**\***b**);**

**}));**

console.log **(**"La division est: " **+** calculatrice **(**10**,**2**,** ***function*(**a**,**b**){**

***return*** **(**a**/**b**);**

**}));**

### Renvoyer une fonction dans un return

// voici une High Order Function, car elle renvoie une function

***function*** genereFonction**(**message**){**

***let*** maFonc **=** ***function*** **(){**

console.log **(**"Je suis une fonction et je vais être renvoyée!"**);**

console.log **(**"Je veux aussi dire: " **+** message**);**

**}**

***return*** maFonc**;**

**}**

// à chaque fois j'obtient une nouvelle fonction "personnalisée"

***let*** maFonctionPerso **=** genereFonction **(**"hey hey hey!!!!"**);**

maFonctionPerso**();**

***let*** maFonctionPerso **=** genereFonction **(**"la la la la!!!!"**);**

maFonctionPerso**();** // observez qu'on fait le même appel qu'avant mais le résultat est différent

Important: le **hoisting** n'affecte pas les fonctions qui sont déclarées comme expression (ex: salut) (essayez d'appeler la fonction salut avant de l'avoir déclarée. Quelle est la différence avec la fonction bonjour?)

### Arrow functions

A partir d'ECMA6 on peut créer les fonctions en utilisant une syntaxe simplifiée :

**Exemple** : fonction en ECMA5

controlerEntree **=** ***function*** **(**age**)** **{**

***if*** **(**age **>=** 18**){**

console.log **(**"Ok vous pouvez entrer"**);**

**}**

***else*** **{**

console.log **(**"Vous êtes trop jeune"**);**

**}**

**}**

**Voici la même fonction en ECMA6 (arrow function)**

controlerEntree **=** **(**age**)** **=>** **{**

***if*** **(**age **>=** 18**){**

console.log **(**"Ok vous pouvez entrer"**);**

**}**

***else*** **{**

console.log **(**"Vous êtes trop jeune"**);**

**}**

**}**

Cette notation **s'avère utile dans plein de situations.**

Exemple : la gestion de l'événement du click sur un bouton:

bouton1.addEventListener **(**'click'**,** ***function*** **(**evenement**)** **{**

console.log **(**"vous avez cliqué: " **+** evenement.currentTarget.id**);**

**});**

Devient :

bouton1.addEventListener **(**'click'**,** **(**evenement**)** **=>** **{**

console.log **(**"vous avez cliqué: " **+** evenement.currentTarget.id**);**

**});**

Voici d'autres exemples :

***let*** listeAuteurs **=** **[**auteur1**,** auteur2**];**

listeAuteurs.forEach **(*function*** **(**unAuteur**){**

unAuteur.afficherConsole**();**

**});**

listeAuteurs.forEach **((**unAuteur**)** **=>** **{**

unAuteur.afficherConsole**();**

**});**

***let*** calculeSomme **=** ***function*** **(**val1**,** val2**){**

***return*** **(**val1 **+** val2**);**

**}**

// arrow function de base

***let*** calculeSomme = (val1,val2) => {

***return*** **(**val1 **+** val2**);**

}

// simplification de l'arrow function

***let*** calculeSomme **=** **(**val1**,** val2**)** **=>** **(**val1 **+** val2**);**

console.log **(**calculeSomme **(**4**,**5**));**

***let*** calculeTVA = (prix, tauxTVA) => (prix + prix \* tauxTVA/100);

Vous avez d'autres exemples dans le dossier "Arrow Functions" (Aide-mémoire)

# Contexte Global d'Exécution, l'objet Window

Pour tous les scripts, le **moteur JS** **crée** un **objet** **global window** (on étudiera les objets plus tard, considérer pour le moment qu'un objet est juste un autre type de données qui contient de propriétés - variables). Toutes les variables et les fonctions qu'on créera dans le code principal seront attachées à cet objet[[1]](#footnote-1). En plus, l'objet global a déjà, par défaut, un ensemble de propriétés.

Pour visualiser ce fait, ouvrez le fichier contexteGlobal1.html. Ouvrez la console et tapez "window" et Enter. Cliquez sur l'objet Window. Cherchez la propriété "nom" et la fonction "bonjour" parmi les propriétés de l'objet window.

Vous pouvez tapez **windows.nom** sur la console et le moteur JS vous affichera sa valeur. Vous pouvez aussi tapez **bonjour()** pour faire appel à la fonction.

#### Exemples Contexte Global:

#### contexteGlobal1.html

# Hoisting

En JS on peut souvent trouver ce genre de code:

hello(); // appel à la fonction avant de la déclarer

function hello (){

console.log ("hello!!");

}

Ce code fonctionne… comment c'est possible? **Comment est-ce que le moteur de JS peut réaliser l'appel de la fonction si on l'a pas encore déclarée?** Dans la plupart de langages un code comme celui-ci provoquerai une erreur!

On dit souvent qu'en JS les déclarations de fonctions et de variables dans un script sont **"déplacées en haut"** du script, mais **c'est faux**.

En réalité **le moteur de JS cherche toutes les déclarations de variables et de fonctions et** **réalise** **pour chacune** **juste une réservation de mémoire** : il ne donne pas encore de valeurs aux variables, qui restent **undefined** (démonstration dans hoisting1.html).

Ce phénomène s'appelle **hoisting**. Ouvrez les fichiers d'exemple qui contiennent toutes les explications pertinentes.

#### Exemples Hoisting

hoisting1.html

hoisting2.html

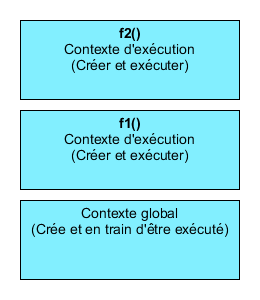
**Une fois le moteur a créé le contexte d'exécution et réalisé le hoisting, le moteur passe à la phase d'exécution ligne par ligne du code.**

# La pile (stack) d’appels des fonctions

Si on fait appel à une fonction (ex: f1) dans **le "contexte global" (pour le moment "le code principal"),** JS créera un nouveau contexte d'exécution pour cette fonction et l'empilera sur le contexte global. Si dans cette fonction on fait appel à une autre fonction (ex: f2), JS créera encore un nouveau contexte et l'empilera sur les deux précédents.

Quand le moteur finira l'exécution de f2 il enlèvera de la pile son contexte d'exécution et continuera l'exécution de f1. Pareillement, quand le moteur finira l'exécution de f1 il enlèvera de la pile son contexte et il continuera l'exécution du contexte global.

Voici la pile d'appels (f2 empilée sur f1 empilée sur le contexte global):



#### Exemples stack appels:

#### fonctionsStack1.html

#### fonctionsStack2.html

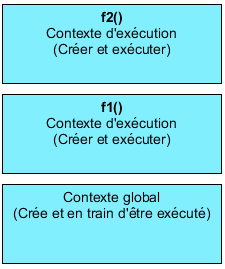
# Les contextes d'exécution des fonctions

Le code JS est lancé toujours dans un **contexte d'exécution**. Ce contexte peut être :

1. Le contexte d'exécution **global**
2. Le contexte d'exécution **d'une fonction**

**Les contextes d'exécution s'empilent les uns sur les autres quand on fait appel aux fonctions (d'abord on a que le contexte global dans la pile)**

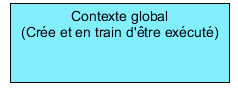
**Exemple :** un code principal où on a fait appel à f1. Dans f1 on a fait appel à f2 (voir le fichier *porteeVariables1.html* dans le dossier *"portée de variables"*)

****

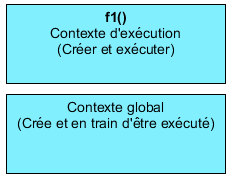
Quand f2 finira son exécution elle disparaitra de la pile. Pareil pour f1.

Notez que si on avait appelé f1 et f2 depuis le code principal (au lieu d'appeler f2 dans f1), on aurait eu jamais plus de deux contextes dans la pile. Voici comment ça se passerait pour les contextes dans ce cas :

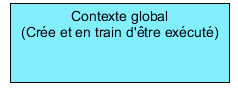
On commence l'exécution du code principal et la pile contient uniquement le contexte global :



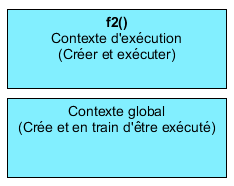
Voici la pile après quand on appelle f1 et elle est en train de s'exécuter :



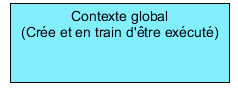
Quand f1 fini elle disparait de la pile …



Et voici la pile juste après l'appel à f2, pendant qu'elle s'exécute



Pour finir, voici la pile après que f2 a fini !



# Portée de variables (Scope) avec Var

La **portée** (scope) d'une variable détermine **la zone du code où est accessible**. Pour résumer, on pourrait dire qu'**une variable** (**créé avec var**)est **accessible uniquement à l'intérieur du bloc de la fonction où elle a été créé** (délimité par les accolades). Chaque fonction crée son propre **scope**.

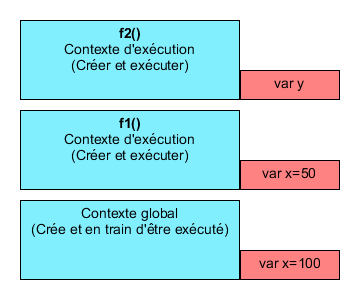
**En général, quand on utilise une variable** JS cherche sa valeur physiquement **dans le bloc du code où on a essayé de l'utiliser**. S'il ne la trouve pas, la recherche continue dans bloc de code **immédiatement supérieur.**

Dans le cas de **var, le bloc supérieur** **sera une fonction ou le code principal**, voir les codes d'exemple juste ci-dessous.

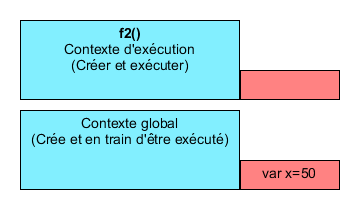
S'il le moteur ne la trouve toujours pas, il continue à monter dans les blocs de code supérieurs jusqu'au bloc principal du code. Si la variable ou la fonction n'est pas trouvée, JS affichera une erreur (ReferenceError).

Exemples :

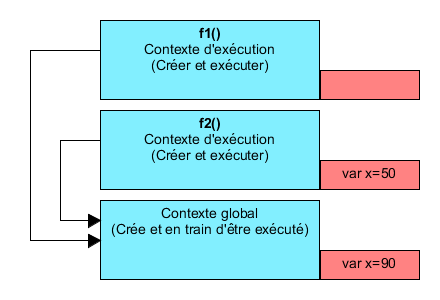
#### PorteeVariables1.html



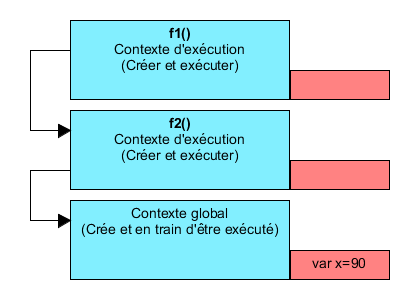
#### PorteeVariables2.html



#### PorteeVariables3.html



#### PorteeVariables4.html



Ici, le bloc du code immédiatement supérieur à f2 sera f1, car f1 a été créé à l'intérieur de f2. Même si ça a l'air artificiel, on aura souvent de fonctions crées à l'intérieur d'une fonction.

Les flèches indiquent **quel est le bloc qui JS considèrera immédiatement supérieur**. JS cherchera ici les valeurs des variables **qui n'ont pas été déclarées dans le code des fonctions**.

Ici, le bloc immédiatement supérieur tant de f1 comme de f2 est le code principal car c'est là où leur code a été écrit

**Exemple** :

Dans le code de f1, x n'est pas déclaré et sa valeur sera cherchée dans le contexte immédiatement supérieur (f2). Dans le code de f2, x est déclaré et JS ne cherche pas alors sa valeur dans le code principal. **Mais** si on utilisait une variable dans f2 qui n'a pas été déclarée chez elle, JS cherchera sa valeur dans le code principal

# Portée de variables (Scope) et blocs avec Let

Nous avons vu comment le scope fonctionne pour les déclarations avec **var**, mais à partir de ES6 on peut déclarer les variables en utilisant le mot réservé **let**.

Les "**blocs**" pour une variable déclarée avec **var** **sera toujours la fonction où elle a été déclarée**. Si on utilise **let**, par contre, la variable **sera attachée au bloc du code** (délimité par **{}**) où elle se trouve, **pas forcément à la fonction mais un bloc plus petit. Il peut s'agir d'une boucle, un if ou une autre structure délimitée par des accolades ({})**

**Exemple :**

***function*** f1**(){**

***for*** **(*var*** i **=** 0**;** i **<** 5**;** i**++){**

console.log **(**i**);**

**}**

console.log **(**i**);**

**}**

***function*** f2**(){**

***for*** **(*let*** j **=** 0**;** j **<** 5**;** j**++){**

console.log **(**j**);**

**}**

console.log **(**j**);** // référence error car j n'existe pas en dehors du for

**}**

f1**();**

f2**();**

**En résumé** : quand on crée une variable en utilisant **var**, elle sera accessible :

* + dans le contexte global si on la crée dans le code principal
  + dans le contexte de la fonction (n'importe où on l'a crée dans la fonction, même si elle est créée à l'intérieur d'une boucle ou un if)

**let**, par contre, nous permet de créer une variable dans un bloc plus petit qu'une fonction (voir une **boucle** ou un **if**, par exemple).

Voici un exemple :

***function*** f1**(){**

***for*** **(*var*** i **=** 0**;** i **<** 5**;** i**++){**

console.log **(**i**);**

**}**

console.log **(**i**);**

**}**

***function*** f2**(){**

***for*** **(*let*** j **=** 0**;** j **<** 5**;** j**++){**

console.log **(**j**);**

**}**

console.log **(**j**);** // référence error car j n'existe pas en dehors du for

**}**

f1**();**

f2**();**

Voici un autre exemple :

// avec var

***function*** maFonction**()** **{**

***var*** nom **=** "Marie"**;**

***if*** **(**nom **==** "Marie"**)** **{**

varnom **=** "Louise"**;** // c'est la même variable, on est toujours dans la fonction

// même si on a créé un block pour l'if

console.log**(**nom**);** // on verra Louise, on a juste écrasé la valeur de nom

**}**

console.log**(**nom**);** // on verra Louise, on vient de l'écraser

**}**

maFonction**();**

// avec let

***function*** maFonctionLet**()** **{**

***let*** nom **=** "Marie"**;**

***if*** **(**nom **==** "Marie"**)** **{**

***let*** nom **=** "Louise"**;** // c'est une nouvelle variable car on est dans un nouveau bloc (le if)

console.log**(**nom**);** // on verra Louise, on a juste écrasé la valeur de nom

**}**

console.log**(**nom**);** // on verra Marie car les variables sont indépendantes même si elles portent le même nom

**}**

maFonctionLet**();**

# Const

**const** nous permet de créer une stocker une valeur en utilisant un identifiant. En principe c'est la même chose qu'une variable, mais on ne peut pas l'affecter deux fois.

Observez ces exemples:

// utilisation de const

***const*** TVA **=** 21**;**

//TVA = 15; // erreur car on ne peut pas affecter à nouveau une constante

***const*** Pi **=** 3.14**;**

// on ne peut pas affecter à nouveau, mais l'objet affecté il peut lui même changer

***const*** lesNoms **=** **[**'Jean'**,** 'David'**,** 'Paul'**];**

lesNoms.push **(**'Luc'**);**

console.log **(**lesNoms**);**

Tel que les variables créées avec **let** , les variables créés avec **const** seront accessibles uniquement dans le bloc où elles ont été déclarées. Rappelez-vous que les variables créées avec **var** seront accessibles dans la fonction où elles ont été créés (ou dans le contexte global si elles ont été créées dans le code principal).

# Les IIFE (Inmediatly Invoked Function Expression)

Une IIFE est une function comme n'importe quelle autre : la seule différence est qu'**elle est appelée au même moment de sa création**.

Une IIFE permet d'**isoler de variables contre les interférences extérieures** (voir exemples de code dans le dossier IIFE des Aide Mémoire, commencez par *introIIFE.html*). Ceci est important si vous écrivez un code qui sera largement utilisé par quelqu'un d'autre. Vous voulez éviter qu'une partie de votre code puisse réécrire accidentellement des variables ou des méthodes qui se trouvent ailleurs dans votre code.

# Closures

**"Closure"** **est** **le phénomène qui permet à une fonction de pouvoir accéder à toutes les variables du contexte où elle a été déclarée même si on la lance en dehors de son contexte**

Voici un exemple :

// exemple1: closure

***function*** message **(**contenuMessage**){**

***var*** nombreFois **=** 5**;**

***return*** ***function*** **(**personne**){**

// notez que "nombreFois" et "contenu" sont accessibles dans cette fonction naturellement

***for*** **(*var*** i**=**0**;** i**<**nombreFois**;**i**++){**

console.log **(**contenuMessage **+** " " **+** personne**);**

**}**

**}**

**}**

***var*** messager1 **=** message **(**"J'ai faim"**);**

// observez que la fonction anonyme retient la valeur de "contenu" (param) et de "nombreFois" (variable interne) même si elle est appelée dans le contexte global

messager1 **(**"Lucie"**);**

messager1 **(**"Alice"**);**

Dans ce code :

1. On appelle la fonction ***message***, dont le contexte s'empile sur le contexte global. Le contexte de la fonction ***message*** contient les variables ***contenu*** (juste un paramètre) et ***nombreFois*** (juste une variable interne à la fonction)
2. La fonction ***message*** **renvoie une autre fonction** (anonyme) **sans l'exécuter**. "message" **finit son exécution après le return** et **elle est enlevée de la pile d'appels,** mais…
3. Les variables ***contenu*** et ***nombreFois* (de son contexte) ne sont pas effacées de la mémoire : elles restent accessibles à la fonction anonyme** (ou à d'autres fonctions créées à l'intérieur si c'était le cas)
4. La fonction anonyme est stockée dans une variable
5. On appelle la fonction anonyme (maintenant **messager1**) depuis le contexte global, et grâce au phénomène de la **closure** elle peut quand-même accéder à ses variables **contenu** et **nombreFois** tel que si on l'appelait à l'intérieur de **message**

**En résumé :**

Une fonction créée à l'intérieur d'une autre aura toujours accès aux variables de son contexte **même si on l'exécute en dehors de son contexte**.

Dans l'exemple : la fonction anonyme a accès aux variables de la fonction *message* **où elle a été déclarée** même si on l'exécute depuis le contexte global

Voici un autre exemple :

// exemple 2

***function*** obtenirModificateurStyle **(**element**){**

// "element" est accésible depuis la fonction ci-dessous

***return*** ***function*** **(**attribut**,** nouvelleValeur**){**

element.style**[**attribut**]** **=** nouvelleValeur**;**

**}**

**}**

// nous allons obtenir une fonction pour modifier div1

// observez que la fonction divModif peut accéder à l'élément

***var*** divModif **=** obtenirModificateurStyle **(**div1**);**

divModif **(**"color"**,** "red"**);**

divModif **(**"background-color"**,** "yellow"**);**

divModif **(**"width" **,**"400px"**);**

// nous allons obtenir une fonction pour modifier une liste

***var*** ulModif **=** obtenirModificateurStyle **(**ul1**);**

ulModif **(**"background-color"**,**"orange"**);**

Dans ce cas, la valeur d'**element** reste accessible quand on appelle **divModif** et **ulModif**

Voici un autre exemple, cette fois un cas problématique :

***function*** generateurFonctions **(){**

***var*** tab **=** **[];**

***for*** **(*var*** i **=** 0**;** i **<** 3**;** i**++){**

tab.push **(**

***function*** **() {**

console.log **(**i**);**

**}**

**)**

**}**

***return*** tab**;**

**}**

***var*** mesFonctions **=** generateurFonctions**();**

mesFonctions**[**0**]();** // on obtient 3

// le problème : la fonction à accès aux valeurs ACTUELLES de i

mesFonctions**[**2**]();**

Pour comprendre cet exemple il faut savoir que **la fonction interne dans la boucle est en train d'être créée, pas exécuté. Au moment de son exécution, la fonction cherche la valeur actuelle de i**, pas la valeur fournie pendant sa création

Regardez les aide-mémoire sur le **infamous loop** pour mieux comprendre cet exemple. Comme exercice, essayez d'appliquer les solutions proposées pour l'infamous loop à cet exemple qu'on vient de réaliser (le **let** et la **IIFE**)

# Les objets

**Objet: Structure de données qui contient**

* Des **variables** ("**propriétés**") primitives (Strings, Numbers, Booleans…)
* Des **fonctions** ("**méthodes**") qui agissent sur ces variables
* D'autres objets éventuellement (aussi "propriétés")

Nous pouvons créer les objets de plusieurs façons en JavaScript, mais commençons pas la plus simple: les **objets littéraux**.

### Les objets littéraux (sans classe)

Voici un objet qui contient uniquement de propriétés primitives, pas de méthodes ni d'autres objets. Observez que l'objet est un ensemble de couples propriété-valeur entourées des accolades:

let personne = {

nom: "Elena",

dateNaissance: "20/2/2010",

membre: false,

salaire: 30000

};

Voici un autre exemple, en utilisant deux notations différentes:

***let*** film1 **=** **{**

titre**:** "Problemos"**,**

realisateur**:** "Eric Judor"**,**

dateSortie**:** "10/5/2017"**,**

afficherFilm**:** ***function*** **(){**

console.log **(**"Le film " **+** this.titre **+** " a été réalisé par " **+** this.realisateur **+** " et est sorti le " **+** this.dateSortie**);**

**}**

**}**

// ou on peut utiliser la notation d'ECMA6 pour les méthodes

***let*** film2 **=** **{**

titre**:** "Matrix"**,**

realisateur**:** "The Wachowskis"**,**

dateSortie**:** "1/1/1999"**,**

afficherFilm**:** ***function*** **(){**

console.log **(**"Le film " **+** this.titre **+** " a été réalisé par " **+** this.realisateur **+** " et est sorti le " **+** this.dateSortie**);**

**}**

**}**

film1.afficherFilm**();**

film2.afficherFilm**();**

On peut afficher la totalité de l'objet dans la console

console.log (personne);

console.log (film1);

On peut **accéder** aux propriétés d'un objet en utilisant l'opérateur ".", voici quelques exemples:

// **Afficher** une propriété

console.log ("La personne s'appelle: " + **personne.nom**);

// **Accéder** à une propriété

if (**personne.membre** === true){

console.log ("Bienvenue " + **personne.nom**);

}

else {

console.log ("Vous n'êtes pas membre");

}

// **Modifier, les propriétés et les méthodes**

personne.nom = "Marie";

console.log ("La personne a changé de nom et s'appelle " + personne.nom + " maintenant");

console.log ("La personne gagne: " + personne.salaire);

personne.salaire = personne.salaire \* 2;

console.log ("Et maintenant la personne gagne: " + personne.salaire);

// on peut aussi changer le

personne.travailler = function (){

console.log ("Je travaille");

}

### Les objets et les arrays

Les arrays et les objets sont très proches : en fait l'arrray est un sous-type d'objet ! La différence la plus évidente est que pour accéder aux valeurs des arrays on utilise des indices numériques (ex: noms[1]) et pour accéder aux valeurs d'un objet on utilise des indices string (ex: voiture['marque'] ou voiture.marque)

Si on crée un objet :

let evenement = {

type: "film",

dateEvenement: "20/2/2030",

titre: "Terminator X",

prix: 10

};

Accéder avec des indexes numériques ne fait pas du sense :

console.log (evenement [2]); // undefined

### L'objet this

**this** est une variable crée par JS dont le contenu change selon la situation:

* Si on utilise **this** dans le contexte global ou dans une fonction, on obtient une référence à l'objet **Window**
* Si on utilise **this dans** **une méthode** d'un objet, on a deux cas de figure :
  + Si on l'utilise **directement dans la méthode**, on obtient une référence à **l'objet** qui contient cette méthode
  + Si on l'utilise **à l'intérieur d'une fonction créé dans la méthode**, on obtient (de façon inattendue) une référence à l'objet **Window**.

On peut contourner ce "problème" en faisant une copie de l'objet this (**let self = this**) dans la méthode

Parcourez les exemples dans le dossier **this** des aide-mémoire.

### La fonction call et this

La fonction **call** nous permet d'appeler une méthode d'un objet **en choisissant quelle sera la valeur de this dans cette méthode**.

// Exemple 1 (sans call): Nous avons un objet de base contenant de

// méthodes et de propriétés

***var*** personne **=** **{**

prenom**:** "Marie"**,**

nom**:** "Curie"**,**

afficherInfos**:** ***function*** **(){**

***return*** this.prenom **+** " " **+** this.nom**;**

**},**

marcher**:** ***function*** **(){**

console.log **(**"Allons y on bouge!"**);**

**}**

**}**

personne.afficherInfos**();**

personne.marcher**();**

// Exemple 2 (call): on force maintenant l'objet this

// à être un autre objet de notre choix

// Dans cet exemple nous avons un objet qui contient

// uniquement une fonctionnalité (de méthodes)

// On veut lui envoyer un autre objet en paramètre

// contenant de valeurs pour de propriétés

// pour qu'il exécute de méthodes sur ces propriétés

***var*** utilisePersonne **=** **{**

afficherInfos**:** ***function*** **(){**

console.log **(*this*);**

console.log **(**this.prenom **+** " " **+** this.nom**);**

**},**

marcher**:** ***function*** **(){**

console.log **(**"Allons y on bouge!"**);**

**}**

**}**

***var*** personne1 **=** **{**

prenom**:** "Marie"**,**

nom**:** "Curie"

**}**

***var*** personne2 **=** **{**

prenom**:** "Ada"**,**

nom**:** "Lovelace"

**}**

utilisePersonne.afficherInfos**();** // ça ne marchera pas, "utilisePersonne" n'a pas de propriétés

utilisePersonne.afficherInfos **(**personne1**);** // ça ne marche pas non plus, la méthode car afficherInfos n'a pas de paramètres

**utilisePersonne.afficherInfos.call (personne2);** // ok! call enverra l'objet "personne2" qui deviendra "this" pour la méthode

// Exemple 3 (call):

// Même système mais en utilisant le DOM.

// Dans ce cas nous avons un objet capable de

// rajouter des éléments à une liste.

// On force "this" à être la liste

***var*** manipuleListe **=** **{**

// on force le this à être la liste qu'on veut modifier

rajouterLiTexte **:** ***function*** **(**texteLi**){**

elem **=** document.createElement **(**"li"**);**

elem.appendChild **(**document.createTextNode **(**texteLi**));**

this.appendChild **(**elem**);**

**}**

**}**

// le prémier paramétre est l'objet qu'on veut qui devienne "this"

// le deuxième paramétre est le paramètre propre à la méthode (texteLi dans ce cas)

**manipuleListe.rajouterLiTexte.call (maListe, "classique");**

**manipuleListe.rajouterLiTexte.call (maListe, "salsa");**

// manipuleListe.rajouterLiTexte (maListe, "classique"); // erreur car this est le propre objet "manipuleListe"

Si la méthode qui doit être appelée à de **paramètres, nous les rajouterons** **après** le paramètre de l'objet qui deviendra **this**.Voici un exemple :

// Exemple 4: call avec de paramétres

***var*** persoImage **=** **{**

modifierTaille **:** ***function*** **(**uneImage**,** width**,** height**){**

uneImage.style.width **=** width**;**

uneImage.style.height **=** height**;**

// si on veut pouvoir envoyer uniquément width, height ou aucun de deux:

//uneImage.style.width = (typeof (width) !== 'undefined') ? width : uneImage.width;

//uneImage.style.height = (typeof (height) !== 'undefined') ? height : uneImage.height;

**}**

**}**

***var*** img **=** document.createElement **(**"img"**);**

img.src **=** "./lince.jpg"**;**

// voici les paramètres: l'objet (pour this) et les paramètres propres à la méthode

**persoImage.modifierTaille (img, "250px", "200px");**

document.body.appendChild **(**img**);**

### La fonction bind et this

La fonction **bind** crée une nouvelle fonction qui aura comme objet **this** un objet de notre choix.

**Important** : **Bind crée une nouvelle fonction, tandis que call c'est juste un appel.**

Avant de passer aux exemples, observons une erreur très fréquente parmi les développeurs de JS. Dans cet exemple on va essayer d'extraire une méthode d'un objet existant (faire une copie de la méthode dans une variable).

// un objet normal

***var*** personne **=** **{**

prenom**:** "Marie"**,**

nom**:** "Curie"**,**

afficherInfos**:** ***function*** **(){**

***return*** this.prenom **+** " " **+** this.nom**;**

**},**

marcher**:** ***function*** **(**vitesse**){**

console.log **(**this.prenom **+** " " **+** this.nom **+** " se déplace à " **+** vitesse **+** " km l'heure!"**);**

**}**

**}**

personne.marcher **(**70**);** // Marie Curie marche à 70 km/h, tout va bien

// Exemple 1: on extrait une méthode de l'objet personne

***var*** monMarcher **=** personne.marcher**;**

// quand on l'utilise, this pointe vers l'objet window car

// la méthode n'est plus dans le contexte de l'objet original

monMarcher **(**15**);** // undefined undefined bouge à .....

Voici l'utilité de **bind** : au moment de créer la fonction on lui **fixe une valeur pour this** :

// Exemple 2 (bind): on peut résoudre le problème en indiquant l'objet qui sera this

***var*** monMarcher2 **=** personne.marcher.bind**(**personne**);**

monMarcher2**(**80**);** // le this est fixé à "personne"

// Exemple 3 (bind): on peut aussi utiliser bind avec un autre objet, pas forcement

// avec celui dont on extrait la méthode

***var*** p **=** **{**

prenom**:** "Ada"**,**

nom**:** "Lovelace"

**}**

***var*** monMarcher3 **=** personne.marcher.bind **(**p**);**

monMarcher3**(**60**);** // le this est fixé à "p" même si on a extrait la méthode de "personne"

// Exemple 4: Valeurs par défaut: on peut utiliser bind aussi pour créer de fonctions qui ont des

// arguments par défaut. Ici on n'a pas un objet littéral mais juste une fonction

***function*** majuscules**(){**

console.log **(**arguments**);** // Observez le contenu de cet objet... qui n'est pas un array!

// console.log (typeof (arguments));

// Le moteur de JS crée un array à partir des arguments (valeurs isolées séparées par virgules)

// et applique toUpperCase à chaque valeur.

// Finalement il renvoie l'array modifié

***return*** Array.from **(**arguments**).**map **(*function*** **(**val**){** ***return*** val.toUpperCase**()});**

**}**

***var*** changeMajuscules **=** majuscules**;**

console.log **(**changeMajuscules**(**"Lola"**,**"Marie"**,**"Joanna"**));**

***var*** changeMajuscules2 **=** majuscules.bind **(*null*,** "Rudolph"**,** "Salvatore"**);**

// dans ce cas on aura déjà deux valeurs par défaut

console.log **(**changeMajuscules2**(**"Loulou"**,** "Sardinita"**));**

**Resumé de call et bind** :

* **call** fait un appel à une fonction en fixant l'objet **this**, **bind** fait une copie de la fonction en fixant l'objet **this**
* **call** et **bind** peuvent recevoir d'autres paramètres en plus (séparés par virgules du prémier, qui indique toujours qui sera le **this**).
  + On rajoute de paramètres à **call** pour que la fonction qui est appelée les utilise
  + On rajoute de paramètres à **bind** pour fixer les valeurs dans la fonction qui a été créé en appelant bind (voir exemple 4)

**Note:** il existe encore une fonction **apply** qui fonctionne de la même manière que **call** mais qui reçoit les paramètres extra dans un array et pas séparés par virgules

### Les classes

Dans certaines circonstances nous avons besoin de créer plusieurs objets qui partagent les mêmes caractéristiques (mêmes propriétés et mêmes méthodes. Exemple : plusieurs clients, voitures, amis, messages, évènements, repas…). À partir du ECMA 6 nous disposons des **classes**. Une classe, dans la plupart de langages, est une sorte de prototype d'objet (une sorte de "moule à tarte") qui **nous permet de générer plein d'objets** qui **partagent les mêmes propriétés et les mêmes méthodes**.

Considérons qu'on veut faire un site web pour donner des avis sur de films. Nous allons avoir besoin de stocker les données des films dans des structures de données (concrètement dans des objets). Sans une classe on aurait un code comme celui-ci:

***let*** film1 **=** **{**

titre**:** "Problemos"**,**

realisateur**:** "Eric Judor"**,**

dateRealisation**:** "10/5/2017"**,**

afficherFilm**:** ***function*** **(){**

console.log **(**"Le film " **+** this.titre **+** " a été réalisé par " **+** this.realisateur **+** " en " **+** this.dateRealisation**);**

**}**

**}**

***let*** film2 **=** **{**

titre**:** "Matrix"**,**

realisateur**:** "The Wachowskis"**,**

dateRealisation**:** "1/1/1999"**,**

afficherFilm**:** ***function*** **(){**

console.log **(**"Le film " **+** this.titre **+** " a été réalisé par " **+** this.realisateur **+** " en " **+** this.dateRealisation**);**

**}**

**}**

Etc…

film1.afficherFilm**();**

film2.afficherFilm**();**

Si on a plein de films ce code devient un chaos. On peut alors créer notre moule à films (classe):

// le moule à films!

***class*** Film **{**

// le constructeur: méthode spéciale qui est appelée à chaque fois qu'on crée un objet film (voir plus bas)

constructor **(**titre**,** realisateur**,** dateSortie**)** **{**

this.titre **=** titre**;**

this.realisateur **=** realisateur**;**

this.dateSortie **=** dateSortie**;**

**}**

// le reste de méthodes (notez qu'on n'écrit plus le mot "function")

afficherFilm **()** **{**

console.log **(**"Le film " **+** this.titre **+** " a été réalisé par " **+** this.realisateur **+** " et est sorti le " **+** this.dateSortie**);**

**}**

**}**

// créer les objets est beaucoup plus facile maintenant

***let*** f1**=** ***new*** Film **(**"Problemos"**,** "Eric Judor"**,** "10/5/2017"**);**

***let*** f2**=** ***new*** Film **(**"Matrix"**,** "The Wachowskis"**,** "1/1/1999"**);**

f1.afficherFilm**();**

f2.afficherFilm**();**

Une fois le concept est clair, on doit mentionner que en JavaScript **une classe est un objet qui nous permet de créer d'autres objets,** pas seulement un template comme dans les autres langages. **JavaScript n'a pas de classes comme un "type de données"**

En fait le mot **class** a été introduit pour attirer de developpeurs d'autres langages. Pour créer des structures d'objets en JavaScript on utilisera une autre méthode qu'on explique dans les chapitre 26

# Les Prototypes

Un **objet est un ensemble de propriétés et de méthodes**. En JavaScript, tout objet possède aussi *un seul* ***objet******prototype (accessible via la propriété \_\_proto\_\_)****.* Ce prototype peut être un objet ou *null.* Voici un exemple :

// un objet a ses propriétés, méthodes et un prototype accessible via \_\_proto\_\_

***var*** personne **=** **{**

nom **:** "Marie"**,**

age **:** 25**,**

danser **:** ***function*** **(){**

console.log **(**"Je m'amuse bien en dansant"**);**

**}**

**}**

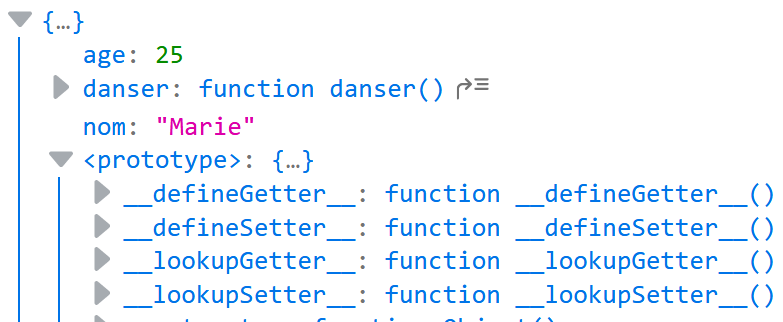
console.log **(**personne.nom**);**

personne.danser**();**

console.log **(**personne.\_\_proto\_\_**);**

console.log **(**personne**);**

Vous pouvez voir le contenu du prototype (\_\_proto\_\_) dans la console si vous déployez l'objet et puis cliquez sur **<prototype>**.



Disons pour le moment que le prototype détermine, en gros, une partie de la structure d'un certain objet. Le prototype de notre objet personne est l'objet générique **Object**

Voici un autre exemple plus clarifiant. Cette fois le prototype de notre objet contiendra de méthodes.

On sait qu'un array est un objet : affichons son prototype !

***var*** monArray **=** **[**4**,**5**,**6**];**

console.log **(**monArray**);**

console.log **(**monArray.\_\_proto\_\_**);** // l'objet Array

console.log **(**monArray.\_\_proto\_\_.\_\_proto\_\_**);** // l'objet Object

console.log **(**monArray.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_**);** // null

Observez le résultat dans la console : un array est un objet dont le prototype est l'objet **Array** (contenant toutes les méthodes d'array, regardez par vous-mêmes dans la console!). A son tour, le prototype d'Array est l'objet **Object** qui se trouve à la racine de tous les objets

Un **prototype (\_\_proto\_\_)** est un simple objet qui **peut avoir à son tour un prototype**. Ce fait porte le nom de **chaine de prototypes**

**La chaine de prototypes s'utilise comme mécanisme d'héritage et pour partager des propriétés** **entre les objets**. Le prototype d'un objet contient **l'ensemble de propriétés et des méthodes qui seront hérités par tous les objets basés sur ce prototype.**

Exemple : tous les objets basés sur l'objet Array contiendront **length**, **push, pop** etc …

Note : En JS nous n'avons pas de vraies classes (le mot **class** est juste une sorte d'enveloppe pour rendre JS similaire à d'autres langages orientés objets)

# Héritage avec Function Constructor

### Héritage de propriétés

Un **function constructor** est une **fonction** **qui nous permet de créer d'autres objets contenant un même ensemble de propriétés**. Une fonction est elle-même un objet, alors on peut lui affecter des propriétés !

Avant d'entrer dans le concept de function constructor, sachez que :

1. **Les fonctions sont des objets en JS**!!
2. Comme tous les objets, les fonctions ont un **prototype** accessible par **\_\_proto\_\_** mais aussi …
3. Toutes les fonctions ont une **propriété** **prototype** (**qui, bizarrement, n'est pas son prototype !**) qui est **un objet** (voler.prototype dans ce cas)
4. Ce dernier objet à une propriété **constructeur** qui pointe vers l'objet fonction original (**voler**)

On verra plus bas l'utilité de ces deux dernières propriétés, **prototype** et **constructor**

***function*** voler **(){**

console.log **(**"J'aime bien voler libre"**);**

**}**

voler **();** // appel

console.log **(**voler**);** // affiche l'objet fonction

console.log **(**voler.\_\_proto\_\_**);** // affiche le prototype de la fonction

console.log **(**voler.prototype**);** // affiche la propriété "prototype" de la fonction



Considérons maintenant **que nous avons un objet duquel on veut hériter certaines propriétés**. Une façon possible d'implémenter l'héritage en JS est de créer une **function constructor.**

**Exemple** : nous allons créer plain d'objets chocolat de types et prix différèrent. On peut créer un function constructor contenant les propriétés qui auront tous les chocolats, et puis créer les objets avec l'opérateur new :

// 1. Function constructor

***var*** ChocolatGen **=** ***function*** **(**type**,** prixKg**){**

// tous les objets crées par ce constructeur

// auront ces deux propriétés

this.type **=** type**;**

this.prixKg **=** prixKg**;**

**}**

***var*** choco1 **=** ***new*** ChocolatGen **(**"blanc"**,** 50**);**

***var*** choco2 **=** ***new*** ChocolatGen **(**"noir"**,** 100**);**

console.log **(**"choco 1 est du type : " **+** choco1.type**);** // blanc

console.log **(**"choco 1 coûte : " **+** choco1.prixKg**);** // 50

Jusqu'à ici les objets seront créés et contiendront les **propriétés** type et prix, tout va bien !

##### Explication en détail

**Mais** pour continuer **on doit comprendre un ensemble de choses qui se passent "en cachette":**

Pour commencer notez que, si on lance cette fonction normalement :

ChocolatGen **(**"blanc"**,** 50**);**

On serait en train de modifier l'objet **window** (car this fait référence à window dans ce contexte !).

C'est pour cette raison que pour créer nos objets basés sur notre constructor function on utilisera l'opérateur **new**.

**new** crée un objet vide et lance le code de la fonction. L'objet crée son propre contexte et **son objet this**. new évite que **this** fasse référence au contexte global.

**new** crée des objets basés sur la fonction constructrice (ici **ChocolatGen**). À l'intérieure de la fonction **on a affecté les valeurs des propriétés de l'objet qu'on est en train de créer** (le **this**, qui sera choco1 ou choco2). Ces valeurs sont reçues en paramètre (comme dans plein d'autres langages de programmation). Tous les objets créés à partir de la fonction auront ces propriétés.

Pour illustrer tout ce discours, observez le schéma suivant :



Notez que :

* Toutes les fonctions possèdent une propriété **prototype (qui n'est pas son Prototype accéssible via \_\_proto\_\_!).** Cette propriété **prototype** **est un objet** qu'on appellera ChocolatGen.prototype
* L'**objet** ChocolatGen.**prototype** a une propriété **constructor** qui, à son tour, qui pointe vers ChocolatGen. Le constructor déterminera la structure de tous les objets qu'il crée
* Les prototypes (\_\_proto\_\_) des objets crées avec **new** (choco1 et choco2) pointent vers cet objet (ChocolatGen.prototype)**. Si on modifie à la main cet objet les prototypes des objets seront modifiés aussi !**
* Les objets sont créés grâce à que **new** appelle le constructeur (la fonction ChocolatGen)
* Le prototype (\_\_proto\_\_) de ChocolatGen pointe vers le prototype générique de **Function (Function.prototype)**
* Le prototype (\_\_proto\_\_) de function pointe vers le prototype d'**Object (Object.prototype)**
* Le prototype (\_\_proto\_\_) de ChocolatGen.prototype pointe aussi vers **Object.prototype**

Les propriétés sont héritées directement dans les objets crées avec le function constructor :

console.log **(**"Choco 1 est du type : " **+** choco1.type**);** // blanc

### Héritage de méthodes

Quand on pense à hériter de méthodes, la première idée qui nous vient à l'esprit est de le faire de la même manière qu'avec les propriétés, c’est-à-dire :

***var*** ChocolatGen2 **=** ***function*** **(**type**,** prixKg**){**

// tous les objets crées par ce constructeur

// auront ces deux propriétés

this.type **=** type**;**

this.prixKg **=** prixKg**;**

**this.manger = *function* (){**

**console.log ("am am am le chocolat " + this.type + " comment c'est bon!!");**

**};**

**}**

***var*** choco3 **=** ***new*** ChocolatGen2 **(**"blanc"**,** 50**);**

***var*** choco4 **=** ***new*** ChocolatGen2 **(**"noir"**,** 100**);**

console.log **(**choco3**);**

console.log **(**choco4**);**

Si on agit de cette manière on obtient un schéma comme celui-ci :



**Chaque objet contient à l'intérieur le code de la fonction, ce qui est une mauvaise pratique car on consomme inutilement de la mémoire!** (Regardez vous-même dans la console le contenu des objets)

La bonne méthode est de **rajouter les méthodes à hériter dans l'objet prototype (**ChocolatGen.prototype). Le code de la méthode sera toujours accessible depuis chaque objet mais on ne fera pas un doublon du code.

C'est très simple à faire :

// Héritage de méthodes. On modifie directement l'objet prototype

ChocolatGen.prototype.manger **=** ***function*** **(){**

console.log **(**"am am am le chocolat " **+** this.type **+** " comment c'est bon!!"**);**

**};**

choco1.manger**();**

// Observez que pour ChocolatGen nous n'avons pas le code de la méthode pour chaque objet

console.log **(**choco3**);**

console.log **(**choco4**);**

// il se trouve dans le prototype!

console.log **(**choco3.\_\_proto\_\_**);**

console.log **(**choco4.\_\_proto\_\_**);**

Si nous voulons créer un héritage basé sur les prototypes, on doit **rajouter les méthodes et les propriétés**. Si on avait créé la méthode dans le constructeur en soi, le code de la méthode se trouverait dans chaque objet (à éviter). Voici un exemple :

// Observez que pour ChocolatGen nous n'avons pas le code de la méthode pour chaque objet

console.log **(**choco1**);**

console.log **(**choco2**);**

// il se trouve dans le prototype!

console.log **(**choco1.\_\_proto\_\_**);**

console.log **(**choco2.\_\_proto\_\_**);**

console.log **(**ChocolatGen2**);**

### Partage de propriétés

Si on veut **partager de propriétés** entre tous les objets basés sur un constructeur on doit les rajouter aussi au prototype

// Partage de propriétés (l'héritage éxiste déjà avec les propriétés définies dans le constructeur)

ChocolatGen.prototype.origin **=** "Belgique"**;**

**Observez que la propriété** "origin" **sera partagée par toutes les instances** de ChocolatGen, car elle est dans le prototype de tous les objets :



console.log **(**choco1.origin**);** // la propriété est définie dans le prototype SAUF si on la défini pour

// chaque objet. Dans ce cas, JS prendra la valeur propre à l'objet!

ChocolatGen.prototype.origin **=** "Equateur"**;**

choco2.origin **=** "Seville"**;** // on écrasse la valeur par défaut, hérité du prototype

console.log **(**choco1.origin**);** // prends la valeur du prototype ("Equateur")

console.log **(**choco2.origin**);** // prends sa propre valeur

console.log **(**choco1.\_\_proto\_\_.origin**);** // prends la valeur du prototype

console.log **(**choco2.\_\_proto\_\_.origin**);** // prends la valeur du prototype, qui n'est plus la même que la copie de l'objet

choco1.origin **=** "Chine"**;**

ChocolatGen.prototype.origin **=** "Italie"**;**

console.log **(**choco1.origin**);** // affiche sa propre valeur

console.log **(**choco2.origin**);** // affiche sa propre valeur

Cette documentation est basée sur: <http://dmitrysoshnikov.com/ecmascript/javascript-the-core/>

# Javascript Asynchrone : fondements de base

L’idée de base d’un code asynchrone est d’**éviter les blocages** provoqués par de fonctions qui prennent du temps à finir leur tâche et qui empêchent l’exécution de la suite du code de la page.

**Exemple :** si on lance une fonction qui obtient de données d’une BD ou qui traite une image on souhaite continuer à lancer le code de la page. On considère, bien entendu, que la suite du code n’a pas besoin des données générées par cette fonction.

Quand on utilise les WEB Apis (évènements du DOM, XMLHttpRequest, un timer avec setTimeout…) on lance **des opérations dont nous ignorons la durée exacte.** Combien du temps va prendre le serveur de BD pour nous fournir nos données ? combien du temps pour lire un fichier ?

Quand une opération de ce type finit son exécution **un événement est généré et le moteur JS essaiera de lancer le code associé à cet évènement** (ex : on affichera dans le DOM les données obtenues de la BD une fois qu’on est certains de les avoir reçues !). **Ce code est un callback qui sera lancé au plus vite possible une fois la fonction qui prend du temps a fini son exécution.**

**Entretemps**, le code de l’application continue à s’exécuter de façon **synchrone**.

Pour comprendre en profondeur ce mécanisme, on doit d’abord étudier deux éléments gérés para le moteur de JS : la **queue de messages** etla **boucle d’événements**.

* La **queue de messages** contient toutes les appels aux callbacks. Les appels passent de l’environnement de WEB APIs à la queue de messages.
* La **boucle d’évènements (event loop)** vérifie constamment cette queue. **Si la pile d’exécution est vide**, le moteur **lancera** le premier callback de la queue. Tant que la pile n’est pas vide, **le callback devra attendre**.

# Javascript Asynchrone - Les promeses

**Promise :** une promese est un objet qui surveille la finalisation d’un certain événement asynchrone (timer, AJAX, accès à une BD, etc…) dans l’application, en plus de determiner quoi faire après cette finalisation.

La promese détermine quoi faire avec une valeur qu’on recevra dans le futur et qui sera le résultat de l’événement.

Ex : demander le chargement d’une image du serveur. Quand l’image sera reçue completement, una action sera réalisée (**resolve**). Si la demande echoue, une autre action sera lancée (**reject**)

Une promese peut se trouver dans quatre états possibles :

1. **Pending** - en attente)
2. **Settled** - executée)
   1. **Resolved** - succes dans l’execution, **résultat disponible**
   2. **Rejected** - erreur dans l’execution, le résultat indisponible

## Production et consommation d’une promesse

Nous allons voir la structure de base d’une promesse pour avoir un modèle général. On testera des exemples dans la section suivante.

Pour utiliser une promesse, on doit la créer et la consommer :

1. **Création d’une promesse :** le constructeur reçoit une function, l’**executor, qui lancera l’opération asynchrone** à réaliser (AJAX, BD, etc…) et qui **reçoit deux callbacks**



1. **Consommer (utiliser) une promesse :** on lance la promesse et on attend le résultat de l’opération asynchrone pour l’utiliser dans…
   * le callback **resolve** implémenté dans la section **then**
   * le callback **reject** implémenté dans la section **catch**



L’executor a deux callbacks paramètres callbacks : **resolve** et **reject**. Pendant la **consommation de la promese** (plus bas), le callback **resolve** sera appelé si l’opération à réaliser (obtenir image, données BD, etc… ) a été complétée avec succès.

Autrement, le callback **reject** sera appelé. Toutes les deux méthodes reçoivent un paramètre qui contient le **résultat** de l’exécution de l’opération asynchrone.

## Exemple d’utilisation : appel AJAX avec une promesse

Exemple phppromiseXphp en HTDOCS

Sans promise

Avec promise

Avec enchainement de promeses sans parametres

Avec enchainement de promeses avec de parametres (la promesse est renvoyée dans une fonction)

(--- AÑADIR ESQUEMAS --)

Pile d’exécution

# Modules

Séparer des morceaux de code qui ont une rélation en morceaux. Chaque module expose une fonctionnalité qui peut être utilisée par d’autres modules. Les autres modules ne connaissent pas l’implementation.

En JS on utilise le pattern Module : on doit connaitre les IIFEs et les closures.

Un module renvoie **un objet** contenant toutes les méthodes qui doivent être accesibles depuis l’exterieur.

Module pattern

* Public and private

Ejemplos de módulos

# Architecture

1. Créer au moins un module pour la UI, un pour la logique et un module controller (IIFEs). Ce dernier reçoit les autres deux en paramètre



## Module UI

1. **Le module UI doit avoir une méthode pour obtenir les entrées dont on a besoin et renvoyer un objet contenant ces entrées**



Cette option est valable, mais au lieu d’accéder aux identifiants des classes directement on peut créer un objet contenant tous les idéntifiants du DOM et le rendre publique (getDOMStrings())



1. **Depuis le module controller on obtient cet objet**



## Fonction Init

Dans le module controller :



En dehors on lance la fonction init :



# Les operateurs REST et SPREAD

# Résumé AJAX en JavaScript

1. Créer une page principale
2. Créer les éléments de la page dont on a besoin
3. Créer un bouton (sans formulaire)
4. Créer une page de traitement qui enverra la réponse
5. Créer le script
6. Créer l'événement pour le bouton
7. Créer l'objet XHR
8. Créer l'événement **onreadystatechange**

- Créer le code en cas d'erreur

- Créer le code en cas su succès: manipuler la réponse (xhr.responseText)

1. Créer le code pour appeler le serveur (open et send)

**Localiser les erreurs:**

- Aller dans le navigateur sur l'onglet NETWORK

- Cliquer dans la réponse du serveur (404, 200, 304 ...)

- Cliquer dans l'onglet RESPONSE pour voir la réponse du serveur

- Il peut avoir des erreurs de JavaScript qu'on peut voir dans la console

**Codes de readyState:**

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState

**Codes de status HTTP:**

<https://en.wikipedia.org/wiki/List_of_HTTP_status_codes>

1. sauf si on les declare avec **let.** Elle sera toujours dans le contexte principal mais elle ne sera pas un propriété de l'objet Window [↑](#footnote-ref-1)