Programmer avec JavaScript

[0. Prolégomènes 6](#_Toc105623736)

[0.1. Paradigme de programmation du langage JavaScript 6](#_Toc105623737)

[0.1.1. Programmation orientée objet à prototype : POOP 6](#_Toc105623738)

[0.1.1.1. Comparaison des modèles à classes et à prototypes 6](#_Toc105623739)

[0.1.1.2. Cinq notions de la POO vues par le prisme de la POOP 6](#_Toc105623740)

[0.1.1.2.1. POO 7](#_Toc105623741)

[0.1.1.2.2. Héritage 7](#_Toc105623742)

[0.1.1.2.3. Méthode 7](#_Toc105623743)

[0.1.1.2.4. Classe 8](#_Toc105623744)

[0.1.1.2.5. Instance 8](#_Toc105623745)

[0.1.1.3. Les propriétés en JavaScript 8](#_Toc105623746)

[0.1.2. Rappel sur la programmation orientée objet à classique : POO 8](#_Toc105623747)

[0.1.2.1. Les principaux (7) concepts du paradigme POO 8](#_Toc105623748)

[0.1.2.1.1. Classe 9](#_Toc105623749)

[0.1.2.1.2. Objet 9](#_Toc105623750)

[0.1.2.1.3. Encapsulation 9](#_Toc105623751)

[0.1.2.1.4. Abstraction 9](#_Toc105623752)

[0.1.2.1.5. Héritage 9](#_Toc105623753)

[0.1.2.1.6. Polymorphisme 9](#_Toc105623754)

[0.1.2.1.7. Interface 9](#_Toc105623755)

[0.1.3. JSBIN 10](#_Toc105623756)

[0.1.4. A faire ? 10](#_Toc105623757)

[1. UTILISER LES DONNEES ET LES TYPES DE DONNEES DANS JAVASCRIPT 11](#_Toc105623758)

[1.1. Déclarez des variables et modifiez leurs valeurs 11](#_Toc105623759)

[1.1.1. Qu'est-ce qu'une variable ? 11](#_Toc105623760)

[1.1.2. Créez une variable en la déclarant 11](#_Toc105623761)

[1.1.3. <code> 11](#_Toc105623762)

[1.1.4. Code css 11](#_Toc105623763)

[1.1.5. Code js 12](#_Toc105623764)

[1.1.6. Affichage 12](#_Toc105623765)

[1.1.7. Modifiez la valeur d'une variable 13](#_Toc105623766)

[1.1.7.1. Opérateurs arithmétiques - travail sur des nombres 13](#_Toc105623767)

[1.1.7.1.1. Addition et soustraction 13](#_Toc105623768)

[1.1.7.1.1.1. Ajouter ou soustraire un nombre d'une variable 13](#_Toc105623769)

[1.1.7.1.1.2. Incrément ou décrément de 1 14](#_Toc105623770)

[1.1.7.1.2. Multiplication et division 14](#_Toc105623771)

[1.1.7.1.2.1. Les opérateurs \*= et /= 14](#_Toc105623772)

[1.1.7.2. Pratiquez : utilisez des opérateurs 14](#_Toc105623773)

[1.1.8. Code js 15](#_Toc105623774)

[1.1.8.1. Mutabilité des variables 15](#_Toc105623775)

[1.1.9. Découvrez les constantes 15](#_Toc105623776)

[1.1.9.1. Exemple de constante : 15](#_Toc105623777)

[1.1.9.2. Pratiquez : constantes 15](#_Toc105623778)

[1.1.9.2.1. <code> 16](#_Toc105623779)

[1.1.9.2.2. Css 16](#_Toc105623780)

[1.1.9.2.3. js 16](#_Toc105623781)

[1.1.9.2.4. Affichage 17](#_Toc105623782)

[1.1.10. Résumé en 6 points 17](#_Toc105623783)

[1.2. Enregistrez vos données avec des types de données 17](#_Toc105623784)

[1.2.1. Qu'est-ce qu'un type ? 18](#_Toc105623785)

[1.2.2. Découvrez le type "number" 18](#_Toc105623786)

[1.2.2.1. Attention à l'arithmétique en virgule flottante ! 19](#_Toc105623787)

[1.2.3. Appréhendez les valeurs logiques 19](#_Toc105623788)

[1.2.4. Découvrez les chaînes de caractères 19](#_Toc105623789)

[1.2.4.1. La string interpolation 20](#_Toc105623790)

[1.2.4.2. Pratiquez : types de données 20](#_Toc105623791)

[1.2.4.2.1. Solution 20](#_Toc105623792)

[1.2.5. Comprenez les types en JavaScript 21](#_Toc105623793)

[1.2.6. Résumé 22](#_Toc105623794)

[1.3. Gérez la complexité avec la bonne collection (array, set, map) 22](#_Toc105623795)

[1.3.1. Utilisez un tableau pour enregistrer une liste ordonnée d'éléments 23](#_Toc105623796)

[**1.3.1.1.** Pratiquez : créez un **array** 23](#_Toc105623797)

[1.3.1.1.1. Solution 23](#_Toc105623798)

[1.3.2. Utilisez des valeurs plutôt que des références 25](#_Toc105623799)

[1.3.3. Travaillez sur les tableaux 26](#_Toc105623800)

[1.3.3.1. Le comptage d'éléments 26](#_Toc105623801)

[1.3.3.2. L'ajout d'éléments 27](#_Toc105623802)

[1.3.3.2.1. Ajouter l’élément au début du tableau 27](#_Toc105623803)

[1.3.3.2.2. Ajouter l’élément à la fin du tableau 27](#_Toc105623804)

[1.3.3.3. La suppression d'éléments 27](#_Toc105623805)

[1.3.4. Pratiquez : travaillez avec les **arrays** 27](#_Toc105623806)

[1.3.4.1. Solution 27](#_Toc105623807)

[1.3.5. Découvrez les sets et les maps 29](#_Toc105623808)

[1.3.5.1. Set 29](#_Toc105623809)

[1.3.5.2. Map 29](#_Toc105623810)

[1.3.6. Résumé 30](#_Toc105623811)

[1.4. Résumé de la partie 1 30](#_Toc105623812)

[1.5. Quiz 1 30](#_Toc105623813)

[2. GERER LA LOGIQUE D’UN PROGRAMME EN JAVASCRIPT 31](#_Toc105623814)

[2.1.1. Découvrez l'avantage de JavaScript : un langage, de nombreux environnements 31](#_Toc105623815)

[2.1.1.1. Découverte de JSBin 31](#_Toc105623816)

[2.1.1.2. Écrivez du code pour des pages web 32](#_Toc105623817)

[2.1.1.2.1. Fonctionnement de JavaScript pour le web : 32](#_Toc105623818)

[2.1.1.3. Écrivez du JavaScript pour des serveurs 33](#_Toc105623819)

[2.2. Contrôler le déroulement du programme avec if, else et switch 33](#_Toc105623820)

[2.2.1. Les instructions if/else 34](#_Toc105623821)

[2.2.1.1. Utilisez des valeurs boolean 34](#_Toc105623822)

[2.2.2. Pratiquez les conditions if/else 35](#_Toc105623823)

[2.2.2.1. Solution 35](#_Toc105623824)

[2.2.2.2. Utilisez des expressions de comparaison 37](#_Toc105623825)

[2.2.2.2.1. Opérateurs de comparaison en JavaScript 37](#_Toc105623826)

[2.2.2.2.2. Pratiquez : les instructions if/else 38](#_Toc105623827)

[2.2.2.2.2.1. Solution 39](#_Toc105623828)

[2.2.2.3. Découvrez l'égalité : == ou === 41](#_Toc105623829)

[2.2.2.4. Les conditions multiples 41](#_Toc105623830)

[2.2.2.4.1. Les 3 principaux opérateurs logiques 41](#_Toc105623831)

[2.2.2.5. Pratiquez : les opérateurs logiques 42](#_Toc105623832)

[2.2.2.6. Pratiquez : les opérateurs logiques 2 43](#_Toc105623833)

[2.2.2.7. Solution 43](#_Toc105623834)

[2.2.3. Le scope des variables 45](#_Toc105623835)

[2.2.4. Les instructions switch 47](#_Toc105623836)

[2.2.4.1. Instruction switch pour afficher une date personnalisée 48](#_Toc105623837)

[2.2.5. Résumé 49](#_Toc105623838)

[2.3. Utilisez la bonne boucle *for* ou *while*, pour repeter les taches 49](#_Toc105623839)

[2.3.1. Utilisez la boucle for pour savoir "combien de fois ?" 49](#_Toc105623840)

[2.3.2. Travaillez sur des tableaux : for…of et for…in 50](#_Toc105623841)

[2.3.2.1. Parcourir un tableau avec for 50](#_Toc105623842)

[2.3.2.2. Parcourir un tableau avec for…in 50](#_Toc105623843)

[2.3.2.3. Parcourir un tableau avec for…of 51](#_Toc105623844)

[2.3.2.3.1. Pratiquez : les boucles for 52](#_Toc105623845)

[2.3.2.3.1.1. Solution 52](#_Toc105623846)

[2.3.3. Utilisez la boucle while pour continuer jusqu'à ce qu'on dise d'arrêter 56](#_Toc105623847)

[2.3.4. En résumé 57](#_Toc105623848)

[2.4. Gérez des erreurs et des exceptions dans votre programme 57](#_Toc105623849)

[2.4.1. Trois types d'erreur 57](#_Toc105623850)

[2.4.1.1. Les erreurs de syntaxe 57](#_Toc105623851)

[2.4.1.1.1. Exemples d'erreurs de syntaxe 58](#_Toc105623852)

[2.4.1.2. Les erreurs de logique 58](#_Toc105623853)

[2.4.1.3. Les erreurs d'exécution 58](#_Toc105623854)

[2.4.1.3.1. Les gestions d’exceptons 59](#_Toc105623855)

[2.4.2. Résumé 60](#_Toc105623856)

[2.5. Résumé de la partie 2 60](#_Toc105623857)

[2.6. QUIZ 2 60](#_Toc105623858)

[3. LES FONCTIONS EN JAVASCRIPT 61](#_Toc105623859)

[3.1. La déclaration de fonction classique 61](#_Toc105623860)

[3.2. L’expression de fonction 62](#_Toc105623861)

[3.2.1. Named Function Expression (NFE) 62](#_Toc105623862)

[3.3. La fonction fléchée (Arrow function) 63](#_Toc105623863)

[3.3.1. Syntaxe de la fonction fléchée 63](#_Toc105623864)

[3.3.2. Cas de la fonction fléchée à argument unique 63](#_Toc105623865)

[3.3.3. Cas de la fonction fléchée sans argument 64](#_Toc105623866)

[3.3.4. Cas de la fonction fléchée avec plusieurs expressions 65](#_Toc105623867)

[3.4. Function anonyme 65](#_Toc105623868)

[3.5. IIFE 65](#_Toc105623869)

[3.6. La syntaxe objet : new *Function* 65](#_Toc105623870)

[3.7. Travaillez sur les fonctions 66](#_Toc105623871)

[3.7.1. Les deux types de fonctions en javascript 66](#_Toc105623872)

[3.7.2. Comprenez les fonctions 66](#_Toc105623873)

[3.7.2.1. Pratiquez la note moyenne 67](#_Toc105623874)

[3.7.2.1.1. Calculer une moyenne 67](#_Toc105623875)

[3.7.2.1.2. Parcourir un tableau 67](#_Toc105623876)

[3.7.2.2. Solution 68](#_Toc105623877)

[~~3.7.3. Ajoutez des tableaux et des objets dans les fonctions~~ 70](#_Toc105623878)

[3.8. Écrire des fonctions propres 70](#_Toc105623879)

[3.8.1. Pourquoi la propreté du code est-elle importante ? 71](#_Toc105623880)

[3.8.2. Être une fonction ou ne pas être 71](#_Toc105623881)

[3.8.2.1. La refactorisation du code 71](#_Toc105623882)

[3.8.2.1.1. Code non factorisé versus code factorisé 71](#_Toc105623883)

[3.8.2.1.1.1. Code de départ non factorisé 71](#_Toc105623884)

[3.8.2.1.1.2. Code factorisé sans la sortie console 72](#_Toc105623885)

[3.8.2.1.1.3. Code factorisé avec la sortie console 72](#_Toc105623886)

[3.8.3. Le principe DRY 73](#_Toc105623887)

[3.8.4. Pratiquez le principe DRY 73](#_Toc105623888)

[3.8.4.1. Solution 74](#_Toc105623889)

[3.8.4.2. Quelqu'un sait-il ce que fait ce truc ? 75](#_Toc105623890)

[3.8.5. Laissez des commentaires 77](#_Toc105623891)

[3.8.6. Écrivez du code avec style 78](#_Toc105623892)

[3.8.6.1. Qu'y a-t-il dans un nom ? 78](#_Toc105623893)

[3.8.6.1.1. Utilisez les conventions de nommage 78](#_Toc105623894)

[3.8.6.1.1.1. Choisissez un nom 78](#_Toc105623895)

[3.8.6.2. Mettez en forme votre code 78](#_Toc105623896)

[3.8.7. Pratiquez l’écriture de code propre 79](#_Toc105623897)

[3.8.7.1. Correction expliquée de l’exercice 80](#_Toc105623898)

[3.8.7.1.1. Code non DRY 80](#_Toc105623899)

[3.8.7.1.1.1. Code avant correction : 80](#_Toc105623900)

[3.8.7.1.1.2. Code après correction : 81](#_Toc105623901)

[3.8.7.1.1.3. *Explication :* 81](#_Toc105623902)

[3.8.7.1.2. Code mal nommé 81](#_Toc105623903)

[3.8.7.1.2.1. Code avant correction : 81](#_Toc105623904)

[3.8.7.1.2.2. Code après correction : 82](#_Toc105623905)

[3.8.7.1.2.3. Explication : 82](#_Toc105623906)

[3.8.7.1.3. Code mal mis en forme 82](#_Toc105623907)

[3.8.7.1.3.1. Code avant correction : 82](#_Toc105623908)

[3.8.7.1.3.2. Code après correction : 83](#_Toc105623909)

[3.8.7.1.3.3. Explication : 84](#_Toc105623910)

[3.8.8. En résumé 84](#_Toc105623911)

[3.9. Tester qu’une fonction fait ce qu’elle dit 84](#_Toc105623912)

[3.9.1. Comprenez l'importance des tests 84](#_Toc105623913)

[3.9.2. Qu'est-ce que le test?- **les tests unitaires** 84](#_Toc105623914)

[3.9.3. Découvrez les tests d'intégration 86](#_Toc105623915)

[3.9.4. Appréhendez les tests fonctionnels 87](#_Toc105623916)

[3.9.5. Les tests: comment ça se passe dans la pratique ? 87](#_Toc105623917)

[**3.9.6. Quid les tests de non régression ?** 89](#_Toc105623918)

[3.9.7. En résumé 89](#_Toc105623919)

[3.10. Déboguer une fonction 89](#_Toc105623920)

[3.10.1. Affichez la console 90](#_Toc105623921)

[3.10.1.1. Correction du bug 90](#_Toc105623922)

[3.10.2. Utilisez des outils pour développeur 91](#_Toc105623923)

[3.10.3. Déboguez avec un canard en plastique 93](#_Toc105623924)

[3.10.4. Pratiquez les console.log 93](#_Toc105623925)

[3.10.4.1. Correction 93](#_Toc105623926)

[3.10.5. En résumé 94](#_Toc105623927)

[4. La récursivité 95](#_Toc105623928)

[4.1. Découvrir la récursivité : l’appel de fonctions à l’intérieur d’elles-mêmes 95](#_Toc105623929)

[4.1.1. Une fonction qui s'appelle elle-même 95](#_Toc105623930)

[4.1.1.1. La fonction récursive 95](#_Toc105623931)

[4.1.1.2. Cas de base, ou base case 97](#_Toc105623932)

[4.1.1.2.1. Quel est le base case dans cet algorithme ? 97](#_Toc105623933)

[4.1.2. Pratiquez la récursivité 99](#_Toc105623934)

[4.1.2.1. Exercice factorielle récursive 99](#_Toc105623935)

[4.1.2.1.1. Enoncé 99](#_Toc105623936)

[4.1.2.1.2. Solutions 100](#_Toc105623937)

[4.1.2.1.2.1. Version en locale 100](#_Toc105623938)

[4.1.2.1.2.2. Version en ligne 100](#_Toc105623939)

[4.1.3. En résumé 101](#_Toc105623940)

[4.2. Résumé de la partie 102](#_Toc105623941)

[4.3. QUIZ 3 102](#_Toc105623942)

[5. POO en JavaScript 102](#_Toc105623943)

[5.1. Définissez des objets et leurs attributs avec des classes 102](#_Toc105623944)

[5.1.1. Découvrez les objets 103](#_Toc105623945)

[5.1.2. La notion de classe 103](#_Toc105623946)

[5.1.2.1. Exemple de classe 104](#_Toc105623947)

[5.1.2.2. Instance d’une classe 104](#_Toc105623948)

[5.1.2.3. Pratiquez : créez un objet 105](#_Toc105623949)

[5.1.2.3.1. Solution 105](#_Toc105623950)

[5.1.2.4. Accédez aux données d'un objet avec la notation pointée 106](#_Toc105623951)

[5.1.2.4.1. Pratiquez : récupérez des valeurs depuis un Object 107](#_Toc105623952)

[5.1.2.4.1.1. Solution 107](#_Toc105623953)

[5.1.3. Manipulez des classes 108](#_Toc105623954)

[5.1.3.1. Définition de la classe & constructeur 109](#_Toc105623955)

[5.1.3.2. Construction d’une classe 109](#_Toc105623956)

[5.1.3.2.1. Pratiquez : les classes 109](#_Toc105623957)

[5.1.3.2.1.1. Solution 110](#_Toc105623958)

[5.1.4. Résumé 112](#_Toc105623959)

[5.1.5. To do : UML 112](#_Toc105623960)

[5.2. Définir des méthodes d’instance et des propriétés 112](#_Toc105623961)

[5.2.1.1. Rappel sur la notion de propriété de classe 113](#_Toc105623962)

[5.2.2. Tirez parti des classes avec des méthodes d'instance 114](#_Toc105623963)

[5.2.2.1. Instance de classe newAccount() 114](#_Toc105623964)

[5.2.2.2. Méthode d'instance showBalance() 115](#_Toc105623965)

[5.2.2.3. Ajouter des méthodes capables de modifier les propriétés de l'instance 116](#_Toc105623966)

[5.2.3. Pratiquez les méthodes instance 116](#_Toc105623967)

[5.2.4. Solution 117](#_Toc105623968)

[5.2.5. Découvrez les méthodes statiques 120](#_Toc105623969)

[5.2.6. En résumé 122](#_Toc105623970)

[6. Les modules 122](#_Toc105623971)

[7. To do list 122](#_Toc105623972)

[8. Bibliographie 123](#_Toc105623973)

[9. Links 123](#_Toc105623974)

[10. Links 124](#_Toc105623975)

# Prolégomènes

## Paradigme de programmation du langage JavaScript

Le JavaScript est un **langage orienté objet à prototype : POOP.**

Le langage JavaScript est massivement **orienté objet**. Il suit un modèle basé sur le **prototypage** (contrairement au modèle de classes)[[1]](#footnote-1).

Le JavaScript est un langage orienté :

* Programmation objet à prototype (POOP)
* Programmation objet à classes (POO)

### Programmation orientée objet à prototype : POOP

**La programmation orientée prototype[[2]](#footnote-2)** est un style de programmation orientée objet} dans laquelle les classes ne sont pas explicitement définies, mais plutôt dérivées par ajout des propriétés et des méthodes à une instance d'une autre classe ou, moins fréquemment, par ajouts à un objet vide.

#### Comparaison des modèles à classes et à prototypes

(Comparaison issue de la même source que précédemment.)

**Objets à classes :**

* Une classe définie par son code source est **statique** ;
* Elle représente une définition abstraite de l'objet ;
* **Tout objet est instance d'une classe** ;
* L'héritage se situe au niveau des classes.

**Objets à prototypes :**

* Un prototype défini par son code source est **mutable** ;
* **Il est lui-même un objet au même titre que les autres** ;
* Il a donc une **existence physique en mémoire** ;
* Il peut être **modifié**, appelé ;
* Il est obligatoirement nommé ;
* Un prototype peut être vu comme un exemplaire modèle d'une famille d'objet ;
* **Un objet hérite des propriétés** (valeurs et méthodes) **de son** **prototype**.

En termes simples, ce type de style (POOP) permet de créer un objet sans définir sa classe[[3]](#footnote-3).

#### Cinq notions de la POO vues par le prisme de la POOP

Ce sont les notions[[4]](#footnote-4) suivantes :

1. POO
2. Héritage
3. Méthode
4. Classe
5. Instance

##### POO

Le terme programmation orientée objet est partagé par le JS, basé sur des objets et dont **l’héritage est prototypal** et par les langages OO “classiques”, basés sur les classes.

JavaScript mériterait le terme d’orienté objet car le langage est basé sur les objets dans sa forme la plus pure.

On pourrait résumer le paradigme JavaScript par  OOJS[[5]](#footnote-5) : orientée objet JavaScript.

##### Héritage

Alors qu’en POO classique, lorsqu’on instancie une classe, l’objet ainsi construit contient toutes les propriétés et méthodes de cette classe et éventuellement de ses classes parentes, il en va tout autrement en JavaScript.

En effet, en POOP le concept est plus limpide si l’on parle de **délégation** plutôt que d’héritage[[6]](#footnote-6).

L’objet n’hérite pas des propriétés d’autres objets (il n’ y a pas vraiment de classes en JS) dans le sens où il n’en contient pas une copie mais un **lien** vers un autre objet : son **prototype**.

L’idée de **délégation** prend tout son sens lorsque l’on comprend que notre objet délègue au prototype la responsabilité de trouver une propriété ou une méthode, s’il ne la possède pas lui-même.

##### Méthode

On mentionne souvent le terme méthode, mais JS ne possède pas de méthode au sens classique de l’OO.

Une méthode en OOJS est simplement une fonction rattachée à un objet en tant que propriété. Elle obéit aux mêmes règles que toute autre propriété, elle “hérite” de la même manière, la seule différence est qu’elle peut être appelée.

Il est bon de noter que lorsqu’une fonction contenue dans le prototype est exécutée, this fait référence à l’objet depuis lequel on appelle la fonction et non au prototype.

##### Classe

Le JS n’a pas de classes à proprement parler, il n’y a pas d’implémentations de classes dans le langage. Tout est objet et l’héritage est intégralement basé sur les prototypes.

ES6 introduit quelques mots clefs supplémentaires pour faciliter le travail en OO, notamment le mot clef class, mais celui-ci n’est qu’un sucre syntaxique et le fonctionnement interne reste inchangé.

##### Instance

Si le JS n’a pas de classes, on est en droit de se demander s’il a des instances.

En POO classique, une instance est un objet issue d’une classe. Comme souvent en JS, on utilisera les mots habituellement utilisé en POO, on parlera donc d’instance.

**Cependant, une instance n’est rien d’autre qu’un objet qui hérite du prototype de son constructeur**.

#### Les propriétés en JavaScript

Chaque propriété d’un objet possède des attributs.

En JavaScript Il existe deux types de propriétés qui ont certains attributs :

1. des propriétés de données (data *property*) et
2. des propriétés d'accesseur (*get ; set*).

### Rappel sur la programmation orientée objet à classique : POO

La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique. Elle consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique.[[7]](#footnote-7)

Les langages de programmation chronologiquement : Simula, LOGO, Smalltalk, Ada, C++, Objective C, Eiffel, Python, PHP, Java, Ruby, AS3, C#, VB.NET, Vala, Haxe, Swift.

#### Les principaux (7) concepts du paradigme POO

1. Classe
2. Objet
3. Encapsulation
4. Abstraction
5. Héritage
6. Polymorphisme
7. Interface

##### Classe

Une classe est un ensemble de code contenant des attributs et des méthodes permettant de construire des objets. Une classe contient plusieurs objets.

**Une classe doit être instanciée pour être utilisée.**

##### Objet

**Un objet est une instance d’une classe**

Un objet est un bloc de code mêlant des attributs et des méthodes.

Les attributs définissent les caractéristiques d’un objet d’une classe.

Les méthodes définissent quant à elles les fonctions propres aux instances d’une classe.

##### Encapsulation

L'encapsulation permet d’isoler les données brutes afin d’éviter des erreurs de manipulation ou de corruptions des données. L’encapsulation permet ainsi de cacher des méthodes et des attributs à l’extérieur de la classe.

##### Abstraction

Le concept d’abstraction consiste à cacher les détails inutiles à l’utilisateur final d’une classe. Il va ainsi pouvoir utiliser une classe dans son code de programmation tout en ne sachant pas comment celle-ci a été́ développée.

##### Héritage

Le concept d’héritage signifie qu’une classe B va hériter des mêmes attributs et méthodes qu’une classe A. Lorsqu’une instance de la classe B est construite, on peut alors appeler les méthodes présentes dans la classe A par la classe B. Cela va permettre de faire gagner du temps de développement.

##### Polymorphisme

lorsqu'une classe hérite des méthodes d’une classe parent, il est possible de surcharger une méthode, qui consiste à redéfinir la méthode de la classe parent pour que les deux classes ne fassent pas les mêmes taches.

##### Interface

Une interface est un ensemble de signatures de méthodes publiques d'un objet.

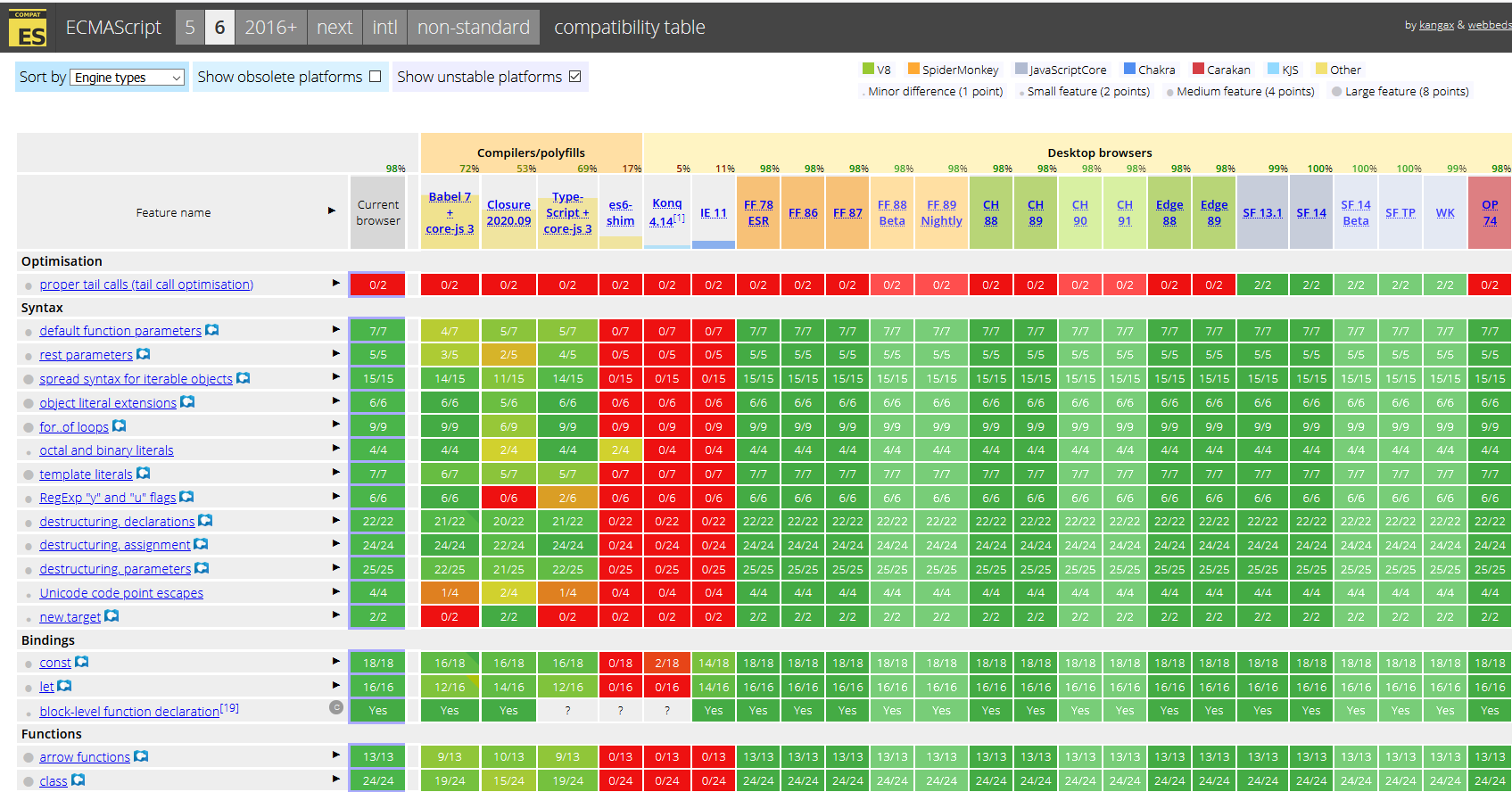
Il s'agit donc d'un ensemble de méthodes accessibles depuis l'extérieur d'une classe, par lesquelles on peut modifier un objet, ou plus généralement communiquer avec lui[[8]](#footnote-8).

### JSBIN

Pour tester les extraits de code JavaScript sans avoir à les intégrer dans une page web, vous pouvez utiliser [JS Bin](https://jsbin.com/?js,console)[[9]](#footnote-9) : c'est le même outil que j'utiliserai dans plusieurs séquences vidéo pour expliquer les notions et présenter des exemples simples.

**Vous allez bâtir des fonctionnalités courantes d'une application de diffusion vidéo pendant ce cours**

Dans ce cours, vous allez apprendre une version de JavaScript appelée ECMAScript 6 (ES6 ou ES2015). Même si ES6 a une compatibilité très élevée à la date de création de ce cours, il y a encore quelques situations où certaines des fonctionnalités ne sont pas actives. Pour en savoir plus, consultez ce [tableau de compatibilité](https://kangax.github.io/compat-table/es6/)[[10]](#footnote-10).



C'est un tableau impressionnant, mais ne vous inquiétez pas ! Même dans les cas où ES6 est incompatible, il y a beaucoup d'outils simples qui permettent de **transpiler** (traduire) du code ES6 automatiquement en code ES5 plus ancien.

**►Le Javascript est case sensitive (en français sensible à la casse)،**

**Commençons à coder !**

### A faire ?

1. Où écrire le code JavaScript ?
2. Historique de JavaScript ?
3. Commentaires en JavaScript ?
4. Codes caractères spéciaux ! (“ ” ¼ …)
5. <script src="script.js" async></script> ou defer ?[[11]](#footnote-11)

► JavaScript permet d'insérer des commentaires dans les programmes, et ceci de deux manières :

Syntaxe à double-slash (//) → sur une seule ligne

Syntaxe slash-étoile (/\* \*/) → sur plusieurs lignes

# UTILISER LES DONNEES ET LES TYPES DE DONNEES DANS JAVASCRIPT

## Déclarez des variables et modifiez leurs valeurs

### Qu'est-ce qu'une variable ?

Voici quelques règles générales pour la création de noms de variables.

1. Utiliser des noms descriptifs dans l'ensemble de votre code.
2. Éviter d'utiliser des abréviations ou de raccourcir des mots quand c'est possible.
3. Suivre une convention de nommage constante : ***camel case[[12]](#footnote-12).***

[CodePen](https://codepen.io/)

### Créez une variable en la déclarant

**Déclarer** une variable pour l’utiliser dans le code.

2 variables **numberOfSeasons** et **numberOfEpisodes**

Une image contenant texte

Description générée automatiquement

### <code>

<link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="info"></p>

</div>

### Code css

body {

background-color: #111;

color: #EEE;

font-family: 'Roboto Slab', serif;

}

.series-frame {

max-width: 600px;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #EEE;

margin: 30px;

padding: 10px;

}

### Code js

// Create your variables here

// ==========================================

let numberOfSeasons = 6;

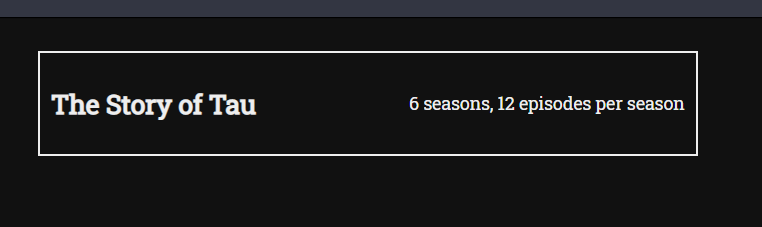
let numberOfEpisodes = 12;

// ==========================================

let paragraph = document.querySelector('#info');

paragraph.innerText = `${numberOfSeasons} seasons, ${numberOfEpisodes} episodes per season`

### Affichage



**Attention** : dans beaucoup de démonstrations JavaScript, vous pourrez croiser le mot-clé **var** plutôt que **let**. Bien qu'il y ait une différence subtile entre les deux (que nous détaillerons dans le chapitre sur la portée), pour l'instant vous pouvez simplement voir **var** comme l'ancienne version de **let** : c'est une autre façon de construire une variable.

### Modifiez la valeur d'une variable

La façon la plus simple de modifier la valeur d'une variable est simplement de la réaffecter :

Une image contenant texte

Description générée automatiquement

1. Ici, nous déclarons la variable **numberOfCats** et **l'initialisons** à la valeur 3.
2. Nous lui **réaffectons** ensuite la valeur 4, **sans le mot clé** **let[[13]](#footnote-13)**, parce que la variable a déjà été déclarée.

#### Opérateurs arithmétiques - travail sur des nombres

Les opérateurs arithmétiques permettent d'effectuer des opérations qui peuvent être extrêmement utiles, **même en dehors des contextes mathématiques**.

##### Addition et soustraction

Pour ajouter deux variables, utiliser le signe **+**

Une image contenant texte

Description générée automatiquement

Pour soustraire deux variables, utiliser le signe **–**

Une image contenant texte, périphérique

Description générée automatiquement

###### Ajouter ou soustraire un nombre d'une variable

Utiliser les opérateurs +**=**  et **-=**

Une image contenant texte

Description générée automatiquement

###### Incrément ou décrément de 1

Utiliser **++** ou **--**

Une image contenant texte

Description générée automatiquement

##### Multiplication et division

Les opérations de multiplication et de division utilisent les opérateurs \*et /

Une image contenant texte

Description générée automatiquement

###### Les opérateurs \*= et /=

Comme pour l'addition et la soustraction, il existe aussi les opérateurs **\*=** et **/=** pour multiplier ou diviser un nombre :

Une image contenant texte, périphérique, mètre

Description générée automatiquement

#### Pratiquez : utilisez des opérateurs

1. Dans l'éditeur JavaScript, créez deux variables, **episodeTime** et **commercialTime** ("temps de l'épisode" et "temps des pubs"), avec des valeurs appropriées (**45** et **5**, par exemple).
2. À l'aide des deux variables de temps que vous venez de construire, ainsi que du nombre d'épisodes et du nombre de saisons (variables pré-existantes), créez une variable nommée **totalShowTime** qui contient le temps de visionnage total de cette série.
3. Les variables **numberOfSeasons** et **numberOfEpisodes** ont déjà été déclarées pour vous.

Solution

### Code js

Une image contenant texte

Description générée automatiquement

#### Mutabilité des variables

Une variable est de base mutable c'est-à-dire qu’elle peut changer au cours du temps. On la déclare avec le mot clé let suivi d’un nom de variable et on lui affecte une valeur de départ. Par la suite, on va pouvoir changer la valeur de cette variable autant de fois que l’on souhaite.

### Découvrez les constantes

Dans de nombreux programmes, certaines **données ne seront pas modifiées pendant l'exécution du programme**.

Ce sont simplement des variables qui **ne seront pas mutables.**

#### Exemple de constante :

const numberOfBooks = 99;

#### Pratiquez : constantes

Voici un composant qui déconstruit une journée en heures, minutes et secondes.

Tout ce que vous avez à faire, c'est de construire les trois constantes qui donnent le nombre d'heures par jour, le nombre de minutes par heure, et le nombre de secondes par minute.

Dans l'espace alloué, créez les trois constantes suivantes :

**hoursPerDay**: heures par jour

**minutesPerHour**: minutes par heure

**secondsPerMinute**: secondes par minute

Initialisez-les avec les valeurs correctes.

##### <code>

<link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

<div class="my-form">

<label for="day-input">Number of days</label>

<input id="day-input" type="number" min="1">

<button id="calculate-button">Calculate!</button>

<p id="hours"></p>

<p id="minutes"></p>

<p id="seconds"></p>

</div>

##### Css

body {

background-color: #111;

color: #EEE;

font-family: 'Roboto Slab', serif;

}

.my-form {

padding: 20px;

}

##### js

// Create constants here

// =====================================

const hoursPerDay = 24;

const minutesPerHour = 60;

const secondsPerMinute =60;

// =====================================

const dayInput = document.querySelector('#day-input');

const calculateButton = document.querySelector('#calculate-button');

const hours = document.querySelector('#hours');

const minutes = document.querySelector('#minutes');

const seconds = document.querySelector('#seconds');

calculateButton.addEventListener('click', () => {

let days = dayInput.value;

let calcHours = days \* hoursPerDay;

let calcMinutes = calcHours \* minutesPerHour;

let calcSeconds = calcMinutes \* secondsPerMinute;

hours.innerText = `${calcHours} hours`;

minutes.innerText = `${calcMinutes} minutes`;

seconds.innerText = `${calcSeconds} seconds`;

});

##### Affichage

Une image contenant texte

Description générée automatiquement

### Résumé en 6 points

1. JavaScript est ***casse sensitive***
2. Donner un **nom**[[14]](#footnote-14) de variable en ***"camel case"***
3. **Déclarer** les variables par le mot clé **let**
4. **Initialiser** les variables avec l'opérateur **=** pour lui donner une **valeur** ;
5. Modifier le contenu d'une variable en la **réaffectant** ; ou avec des **opérateurs** ;
6. Utiliser des **constantes** pour éviter le remplacement d'éléments de données essentiels.

## Enregistrez vos données avec des types de données

Le type d'une variable.

Les langages de programmation disposent de **structures de données** natives.

**►JavaScript est un langage dont le typage est faible et dynamique**.

**Cela signifie qu'il n'est pas nécessaire de déclarer le type d'une variable avant de l'utiliser[[15]](#footnote-15).**

►Le type de la variable sera automatiquement déterminé lorsque le programme sera exécuté.

►Cela signifie également que **la même variable pourra avoir différents types au cours de son existence[[16]](#footnote-16).**

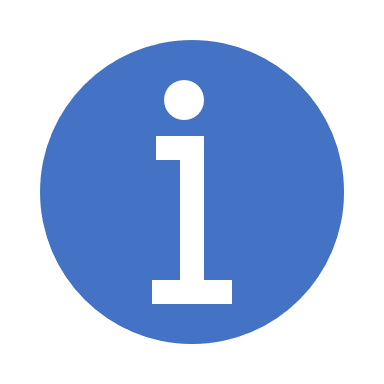
**⬄ Il n’est donc pas nécessaire de déclarer le type d'une variable en JavaScript.**

### Qu'est-ce qu'un type ?

**Le type d'une variable ou d'une constante** est tout simplement le **genre des données** qu'elle enregistre.

En JavaScript, il y a trois **types primitifs** principaux :

1. **number** (nombre) ;
2. **string** (chaîne de caractères) ;
3. **boolean** (valeur logique).

**Les types primitifs** sont les briques de base de chaque structure de données en JavaScript.

### Découvrez le type "number"

Les variables de type number peuvent être positives ou négatives. Elles peuvent aussi être des nombres entiers (1, 2, 3, etc.) ou décimaux (1,4 ; 67,34 ; etc.).

let positive = 23;

let negative = -23;

let decimal = 23.23232323;

Une image contenant texte

Description générée automatiquement

Figure : le type “number”

#### Attention à l'arithmétique en virgule flottante !

L'arithmétique en virgule flottante peut déclencher des erreurs très gênantes dans tous les langages de programmation :

Une image contenant texte

Description générée automatiquement

Figure : Attention à l'arithmétique en virgule flottante !

⬄Chaque fois que c'est possible, utilisez plutôt des calculs entiers (pour le calcul de prix, vous pouvez par exemple calculer en centimes, et non pas en euros ou en dollars).

### Appréhendez les valeurs logiques

Les variables de type ***boolean*** peuvent avoir une des deux valeurs **true** ou **false**.

Une image contenant texte

Description générée automatiquement

Figure : le type “boolean”

### Découvrez les chaînes de caractères

Les chaînes de caractères (chaînes, ou ***strings***, en anglais) sont la façon d'enregistrer du texte dans des variables JavaScript.

On peut enregistrer dans une variable de type **string** n'importe quelle chaîne de caractères, allant d'une seule lettre à un très grand nombre de lettres, plus de 134 millions.

Les variables de type string sont **encadrées** par des guillemets simples[[17]](#footnote-17) : **‘** ou doubles[[18]](#footnote-18) : **"**

Les chaînes peuvent être **concaténées** (ajoutées à la fin l'une de l'autre) par l’opérateur[[19]](#footnote-19) **+**

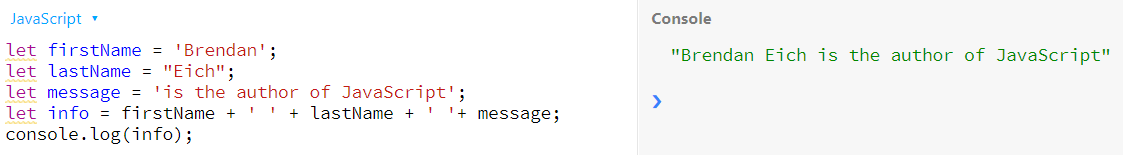


Figure : Le type “string”

#### La string interpolation

Il est possible depuis quelques années d’utiliser une nouvelle écriture qui simplifie la concaténation des variables et des chaînes de caractère : la [***string interpolation***](https://en.wikipedia.org/wiki/String_interpolation).

Pour construire une string interpolation on écrit du texte encadrée par le signe ’ et si on veut injecter une variable dans ce code on utilise l’expression ${maVariable}.

const myName = 'Alexander';

const salutation = ’Bienvenue sur mon site ${myName}!’;

console.log(salutation); //retournera “Bienvenue sur mon site Alexander!”

#### Pratiquez : types de données

Ce composant affichera le titre de l'épisode, sa durée en minutes, et si l'utilisateur a déjà regardé l'épisode ou non.

1. Créez une variable nommée **episodeTitle** qui contient une **chaîne de caractères** qui correspond au titre de votre épisode (employez votre créativité !).
2. Créez une variable appelée **episodeDuration** qui contient un **nombre** qui correspond à la durée de votre épisode en minutes.
3. Enfin, créez une variable **booléenne** nommée **hasBeenWatched** qui décrit si l'utilisateur a déjà regardé l'épisode ou non.

##### Solution

HTML :

<h3>Type de données</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="episode-info"></p>

</div>

CSS :

body{

background-color: #b7ffb7;

color: #000;

font-family: 'Roboto Slab', serif;

}

.series-frame{

max-width: 600px;

font-size: 0.9em;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #804040;

margin: 30px;

padding: 10px;

}

.my-form {

padding: 20px;

}

JS :

let episodeTitle = 'Gone with the Wind';

let episodeDuration = 243;

let hasBeenWatched = true;

// =========================================

document.querySelector('#episode-info').innerText = `Episode: ${episodeTitle}

Duration: ${episodeDuration} min

${hasBeenWatched ? 'Already watched' : 'Not yet watched'}`

Ecran :

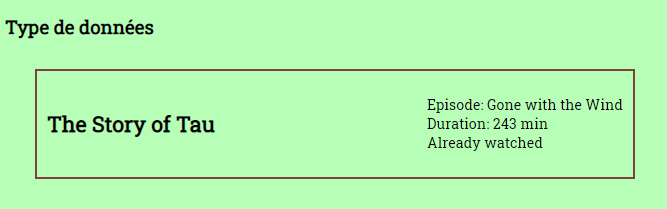


Figure : Type de données

### Comprenez les types en JavaScript

►JavaScript est un langage dit à **types dynamiques et à typage faible**.

**►Cela signifie que vous pouvez initialiser une variable en tant que nombre, puis la réaffecter comme chaîne, ou tout autre type de variable.**

►Ceci offre une grande souplesse, mais peut aussi conduire à un comportement inattendu si vous opérez sans précaution.

### Résumé

Les trois principaux types de données primitifs en JavaScript :

1. **number** (nombre) ;
2. **boolean** (valeur logique) ;
3. **string** (chaîne de caractères).

Il existe d'autres types de données plus complexes.

Techniquement, il y a trois autres types de données primitifs dans JavaScript : **null**, **undefined** et **symbol**.

Vous croiserez les deux premiers dans ce cours, mais les symboles sortiraient du sujet. Pour en savoir plus, consultez les [docs MDN](https://developer.mozilla.org/fr/docs/Web/JavaScript/Data_structures)[[20]](#footnote-20) pour les types.

**Le dernier standard ECMAScript définit 8 types de données :**

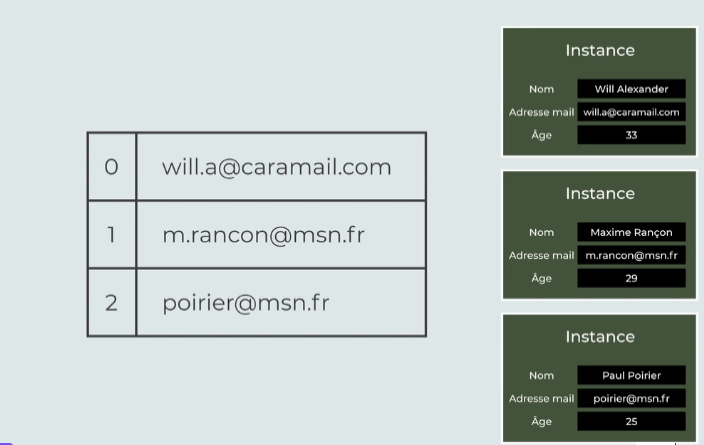
**►*Sept types de données primitifs:***

1. Boolean
2. Null
3. Undefined
4. Number
5. BigInt (proposition pour ES2020)
6. String : Chaîne de caractères
7. Symbol (type introduit avec ECMAScript 6)

►***Le type*** Object

## Gérez la complexité avec la bonne collection (array, set, map)

Le tableau (***array***).



### Utilisez un tableau pour enregistrer une liste ordonnée d'éléments

►Pour construire un tableau vide et l'enregistrer dans une variable, utilisez **une paire de crochets** **carrés** : **[ ]**

**Exemples de tableaux :**

►

►

**L’indice d’un tableau démarre à 0 !**

#### Pratiquez : créez un **array**

Créez un ***Array*** stocké dans une variable **episodes**, et peuplez-le avec les trois variables **firstEpisode** , **secondEpisode** et **thirdEpisode** .

##### Solution

**HTML :**

<h3>Array</h3>

CSS :

body {

background-color: #111;

color: #EEE;

font-family: 'Roboto Slab', serif;

}

.series-frame {

max-width: 600px;

**display: flex;**

align-items: center;

box-sizing: border-box;

border: 2px solid #EEE;

margin: 30px;

padding: 10px;

}

.series-frame h2 {

margin-right: 20px;

}

**JS :**

class cEpisodes {

constructor(title, duration, hasBeenWatched) {

this.title = title;

this.duration = duration;

this.hasBeenWatched = hasBeenWatched;

}

}

let firstEpisode = new cEpisodes('Dark Beginnings', 45, true);

let secondEpisode = new cEpisodes('The Mystery Continues', 45, false);

let thirdEpisode = new cEpisodes('An Unexpected Climax', 60, false);

// Create your array here

// ====================================

let episodes = [firstEpisode, secondEpisode, thirdEpisode];

// ====================================

const body = document.querySelector('body');

for(let i = 0; i < 12; i++) {

let newDiv = document.createElement('div');

newDiv.classList.add('series-frame');

let newTitle = document.createElement('h2');

newTitle.innerText = 'The Story of Tau';

let newParagraph = document.createElement('p');

newParagraph.innerText = `${episodes[i].title}

${episodes[i].duration} minutes

${episodes[i].hasBeenWatched ? 'Already been watched' : 'Not yet watched'}`;

newDiv.append(newTitle);

newDiv.append(newParagraph);

body.append(newDiv);

}

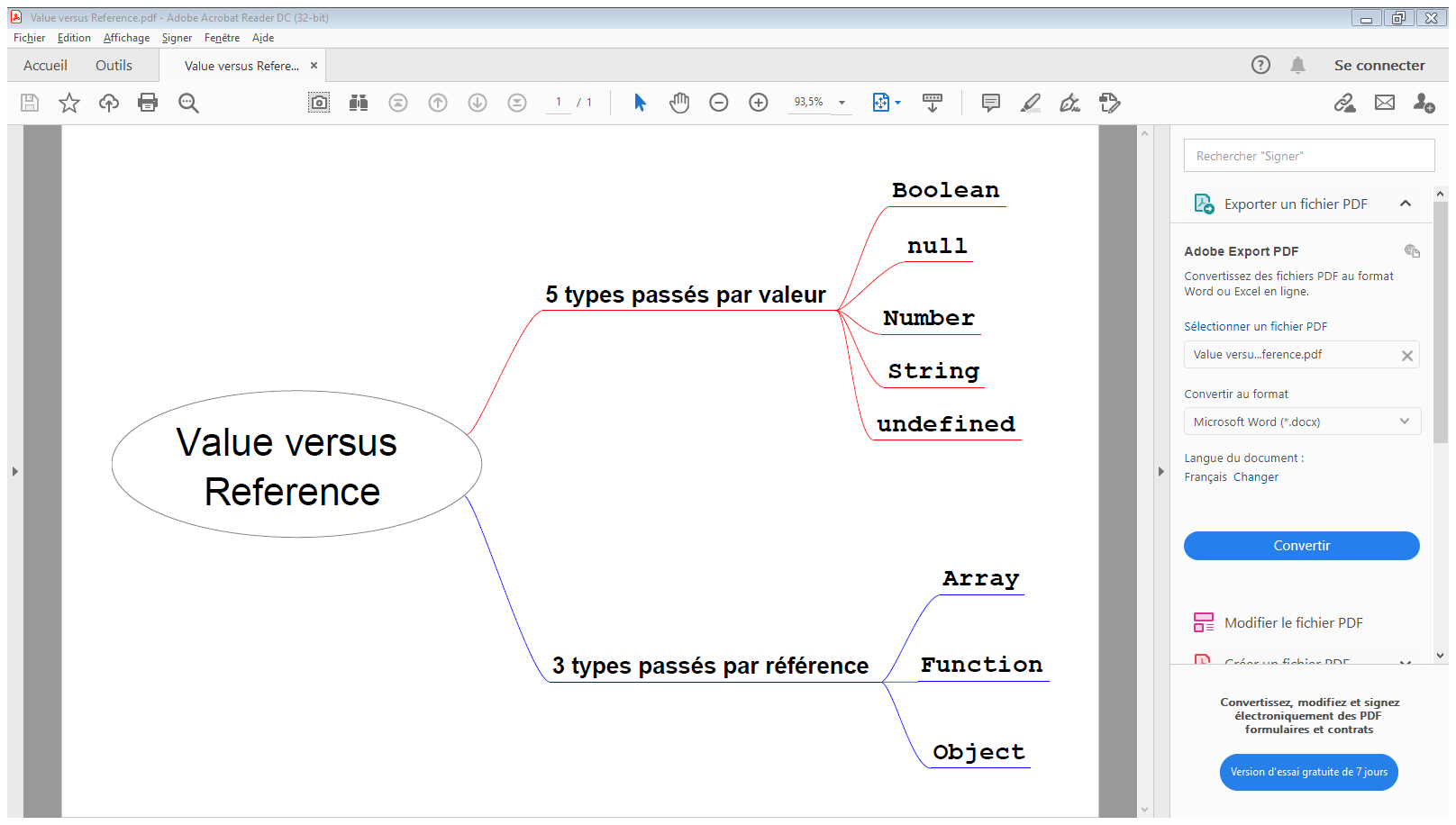
**Ecran :**

Une image contenant texte

Description générée automatiquement

### Utilisez des valeurs plutôt que des références

En JavaScript, **les types primitifs** tels que les nombres, les valeurs logiques et les chaînes **sont** **passés par** **valeur**.



Ceci signifie que quand vous faites :

**let** numberOfGuests = 20;

**let** totalNumberOfGuests = numberOfGuests; // 20

C’est la **valeur** 20 qui est copiée dans la nouvelle variable (totalNumberOfGuests), et aucun lien n'est maintenu entre les deux variables.

Ce n'est pas le cas avec les **objets** et **tableaux**, qui sont passés par **référence[[21]](#footnote-21)**.

Si vous n'y prenez pas garde, cela peut conduire à des comportements inattendus. Par exemple :

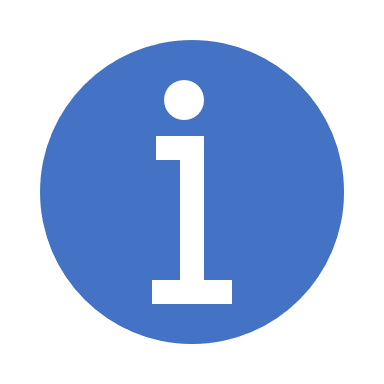
Une image contenant texte

Description générée automatiquement

Bien que nous ayons créé le tableau et passé l'objet **avant** la modification de cet objet, vous la voyez dans le tableau.

C'est parce que quand on utilise des **tableaux** et des **objets**, on **passe des références aux objets** plutôt que la valeur des données qu'ils contiennent.

Les variables **artistProfile[[22]](#footnote-22) (objet)** et **allProfiles (tableau)** présentées ci-dessus contiennent des **références à l'objet et au tableau en** **mémoire**.

**** [Voici un article](https://codeburst.io/explaining-value-vs-reference-in-javascript-647a975e12a0)[[23]](#footnote-23) pour en savoir plus sur les types à passage par valeur et les types à passage par référence en JavaScript.

### Travaillez sur les tableaux

Les tableaux en JavaScript sont très puissants et ont beaucoup d'attributs et de méthodes très utiles. Voici une brève introduction à quelques-uns d'entre eux.

#### Le comptage d'éléments

La propriété length d'un tableau indique le nombre d'éléments qu'il contient

let guests = ["Will Alexander", "Sarah Kate", "Audrey Simon"];

let howManyGuests = guests.length; // 3

#### L'ajout d'éléments

##### Ajouter l’élément au début du tableau

►Pour ajouter un élément **au début** d'un tableau, utilisez la méthode **unshift**

guests.unshift("Tao Perkington");

##### Ajouter l’élément à la fin du tableau

►Pour ajouter un élément **à la fin** d'un tableau, utilisez la méthode **push**

guests.push("Tao Perkington");

#### La suppression d'éléments

►Pour supprimer **le dernier** élément d'un tableau, appelez la méthode pop, **sans passer d’argument.**

guests.pop**()**;

### Pratiquez : travaillez avec les ***arrays***

* Dans cet exercice, un **Array** vide **episodes** a déjà été créé.
* En utilisant la méthode push du tableau **episodes**, ajoutez les épisodes un par un. Ajoutez deux fois le troisième épisode.
* Avec la méthode pop, retirez l'épisode superflu du tableau **episodes**.
* Créez une variable appelée **numberOfEpisodes** qui contient la length du tableau **episodes**.
* **Ajoutez et retirez des épisodes et vérifiez que tout fonctionne correctement.**

#### Solution

**HTML :**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

<link rel="stylesheet" href="style.css">

<script src="SCRIPT.js" async></script>

<title>Array</title>

</head>

<body>

<p>Number of episodes: <span id="episodes"></span></p>

</body>

</html>

**CSS :**

body {

background-color: #b7ffb7;

color: #000;

font-family: 'Roboto Slab', serif;

}

.series-frame {

max-width: 600px;

display: flex;

align-items: center;

box-sizing: border-box;

border: 2px solid #804040;

margin: 30px;

padding: 10px;

}

.series-frame h2 {

margin-right: 20px;

}

**JS :**

class cEpisode {

constructor(title, duration, hasBeenWatched) {

this.title = title;

this.duration = duration;

this.hasBeenWatched = hasBeenWatched;

}

}

let firstEpisode = new cEpisode('Dark Beginnings', 45, true);

let secondEpisode = new cEpisode('The Mystery Continues', 45, false);

let thirdEpisode = new cEpisode('An Unexpected Climax', 60, false);

// ====================================

let episodes = [];

**episodes.push(firstEpisode, secondEpisode, thirdEpisode, thirdEpisode);**

episodes.pop();

let numberOfEpisodes = episodes.length;

// ====================================

const body = document.querySelector('body');

document.querySelector('#episodes').innerText = numberOfEpisodes;

for(let episode of episodes) {

let newDiv = document.createElement('div');

newDiv.classList.add('series-frame');

let newTitle = document.createElement('h2');

newTitle.innerText = 'The Story of Tau';

let newParagraph = document.createElement('p');

newParagraph.innerText = `${episode.title}

${episode.duration} minutes

${episode.hasBeenWatched ? 'Already been watched' : 'Not yet watched'}`;

newDiv.append(newTitle);

newDiv.append(newParagraph);

body.append(newDiv);

}

**Ecran :**

Une image contenant texte

Description générée automatiquement

### Découvrez les sets et les maps

Après les tableaux, voyons les autres types de **collection** en JavaScript : **les ensembles (sets) et les cartes (maps)**.

#### Set

En JavaScript, un **set**, ou ensemble, est **une liste non ordonnée**.

Cela signifie que vous ne pouvez pas récupérer de façon fiable les éléments de liste, car ils n'ont pas d'indice fixe.

Vous pouvez cependant ajouter et supprimer des éléments, **les doublons ne sont pas autorisés** et vérifier si un ensemble contient ou non un élément particulier.

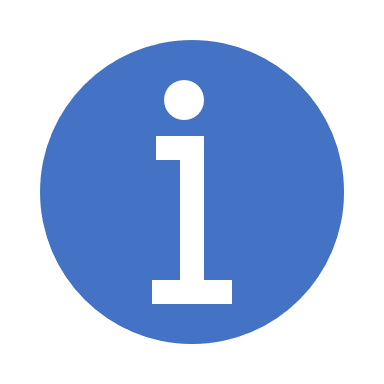
Les ensembles peuvent être pratiques pour enregistrer les utilisateurs en ligne à un instant donné, pour vérifier s'ils le sont ou non.

#### Map

Une **map** est **une liste ordonnée** de **paires clés/valeurs**.

Bien qu'elle ressemble à un objet, il y a des différences importantes. Par exemple, **les clés peuvent avoir n'importe quelle valeur (et pas seulement des chaînes)** dans les cartes.

Vous pouvez facilement trouver combien d'éléments une map contient, et il est possible de filtrer ou modifier dans certains cas les paires clés/valeurs.

Pour en savoir plus, voici des liens utiles :

* [sets](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Set)[[24]](#footnote-24)
* [maps](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Map)[[25]](#footnote-25).

### Résumé

* Connaître les **collections** ;
* La collection la plus courante en JavaScript : le **tableau** ;
* Construire des tableaux, à les remplir, et vu certains outils de base pour les manipuler ;
* Découverte des notions de **sets** et de **maps**.

## Résumé de la partie 1

Ce que sont les **variables** et les **constantes**, et à quoi elles servent ;

Pourquoi le **type** est important, et comment utiliser les différents types ;

Ce que sont les **objets**, et comment utiliser les **classes** pour les construire facilement et de façon fiable ;

L’utilité des **tableaux**, et les bases sur leur manipulation.

## Quiz 1

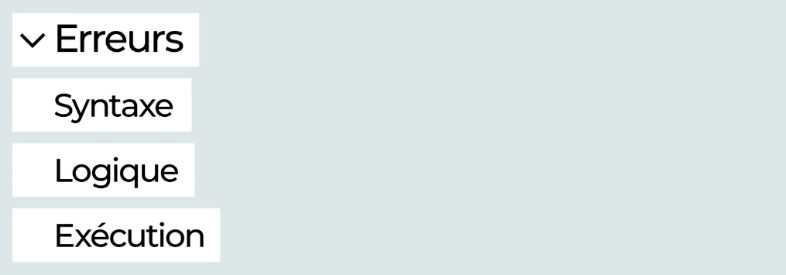
[First quiz](file:///D:\OPENCLASSROOMS\Apprenez%20à%20programmer%20avec%20JavaScript\QUIZs\Quiz_1\Vérifiez%20ce%20que%20vous%20avez%20appris%20sur%20les%20données%20et%20les%20types.pdf)

# GERER LA LOGIQUE D’UN PROGRAMME EN JAVASCRIPT

Une image contenant texte

Description générée automatiquement





Démarrez votre programme avec la fonction main

### Découvrez l'avantage de JavaScript : un langage, de nombreux environnements

L'avantage du JavaScript, c'est que ce langage vous permet de programmer dans de **nombreux environnements**.

Gardez cependant en tête que la façon d'exécuter le code est légèrement différente dans chaque environnement.

Nous allons ici explorer brièvement trois environnements différents : **JSBin**, les pages **web** et les **serveurs**.

#### Découverte de JSBin

**JSBin** est un excellent outil pour tester des extraits de code en JavaScript.

Dans **JSBin**, vous pouvez écrire du code ligne par ligne dans le volet JavaScript, et utiliser le bouton ***Run*** pour l'exécuter.

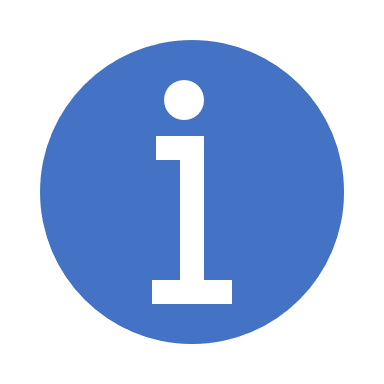
**JSBin** parcourt alors vos lignes de code et **les exécute l'une après l'autre** lorsque vous cliquez sur le bouton **Run**.

**►L'ordre est donc important.**

#### Écrivez du code pour des pages web

##### Fonctionnement de JavaScript pour le web :

1. On écrit du code, l'enregistre dans un ou **plusieurs fichiers,**
2. Puis on **importe** ces fichiers par une balise spéciale **dans le fichier HTML.**
3. Le navigateur **exécute** ensuite automatiquement le code dans ces fichiers, en général dans l'ordre de leur importation.

Il existe un cours complet sur JavaScript pour le web, donc quand vous aurez fini celui-là, pourquoi ne pas y jeter un œil ?

[Écrivez du JavaScript pour le web](https://openclassrooms.com/fr/courses/5543061-ecrivez-du-javascript-pour-le-web)[[26]](#footnote-26)

[](https://openclassrooms.com/fr/courses/5543061-ecrivez-du-javascript-pour-le-web)

**Pré-requis :**

Afin d'être capable de suivre ce cours jusqu'au bout, vous devrez déjà avoir des connaissances en **HTML**, **CSS** et **JavaScript**. Vous devez commencer par ces 2 cours :

1. [Apprenez à créer votre site web avec HTML5 et CSS3](https://openclassrooms.com/fr/courses/1603881-apprenez-a-creer-votre-site-web-avec-html5-et-css3)[[27]](#footnote-27)
2. [Apprenez à coder avec JavaScript](https://openclassrooms.com/fr/courses/2984401-apprenez-a-coder-avec-javascript)[[28]](#footnote-28)  **(Ce cours)**

#### Écrivez du JavaScript pour des serveurs

Il fut un temps où JavaScript ne pouvait s'utiliser que dans des pages web (en avant-plan), mais ce n'est plus vrai ! Il peut maintenant aussi s'utiliser en **arrière-plan,** pour gérer l'accès à certaines ressources.

Prenez par exemple un service de **messagerie** **mail**.

Vous avez une page web qui vous permet d'accéder à vos messages, **mais ils ne sont pas enregistrés dans cette page**.

**Ils sont enregistrés dans une base de données,** et la page que vous utilisez **accède à cette base de données à travers un serveur, qui peut être écrit en JavaScript !**

Néanmoins, les **serveurs** doivent être actifs à tout moment, et seuls certains éléments de code doivent être exécutés à certains moments. Par exemple, le code qui vous permet d'accéder à vos messages **ne doit être exécuté que si** et quand vous décidez d'ouvrir votre client de messagerie !

Dans ce cas, un environnement tel que **Node.js[[29]](#footnote-29)[[30]](#footnote-30)**, couplé à un code JavaScript particulier fait que cela se produit. **Le serveur réagit** à votre ouverture du client de messagerie, vérifie l'authentification **et envoie** vos messages.

## Contrôler le déroulement du programme avec if, else et switch

**Le déroulement du programme** est un terme général qui décrit **l'ordre[[31]](#footnote-31)** dans lequel s'exécutent vos lignes de code.

Ceci signifie que certaines lignes seront lues une seule fois, certaines plusieurs fois, et d'autres complètement ignorées, selon la situation.

Une image contenant texte

Description générée automatiquement

On va s’intéresser aux **instructions conditionnelles.**

**Une image contenant texte

Description générée automatiquement**

### Les instructions if/else

L'instruction  **if / else** est une des plus universelles en programmation.

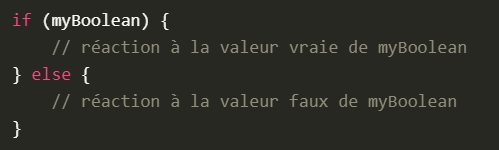
C'est ce qu'on appelle une **instruction conditionnelle**, parce qu'elle vérifie si certaines conditions sont réunies, et réagit en conséquence.

Voyons les **types de condition** qui peuvent être utilisés dans ces instructions :

1. Boolean
2. Expressions de comparaison
3. Conditions multiples

#### Utilisez des valeurs boolean

En JavaScript, si on utilise des **boolean** simples pour les instructions **if / else**, la syntaxe se présente comme suit :



Donc, pour vérifier si un utilisateur est connecté, vous pouvez procéder comme suit :

Une image contenant texte

Description générée automatiquement

### Pratiquez les conditions if/else

Pour réaliser cet exercice, voici comment vous allez procéder :

Vous allez lire le code JavaScript, qui est intégralement commenté, pour bien comprendre à quoi sert chaque élément, et surtout les variables à votre disposition. Il y a sûrement des éléments que vous ne comprenez pas car vous n’avez pas encore vu comment manipuler du HTML en JavaScript, mais ici vous n’avez qu’à exploiter les variables et fonctions déjà codées.

Vous allez écrire une condition if/else entre les lignes 34 et 38 qui appellera la fonction refuser() si l'âge saisi correspond à un mineur, et la fonction valider() s’il est majeur.

Attention, l'âge de la majorité a été stocké dans la variable ageMajorite, donc il faut utiliser cette variable dans la condition.

Une fois que vous aurez codé la bonne condition, vous pourrez enchaîner les tests en saisissant différents âges et en validant.

#### Solution

**HTML**

<h1>Quel est votre age?</h1>

<input id="age" type="text" placeholder="entrez votre age">

<input id="bouton" type="button" value="confirmer">

<h2 id="message"></h2>

**CSS**

body{

text-align:center;

}

**JS**

//On pointe sur l'élément de message

const espaceMessage = document.getElementById("message");

//On pointe sur l'élément de bouton

const bouton = document.getElementById("bouton");

//On pointe sur l'élément de champ de saisie de l'age

const ageInput = document.getElementById("age");

//On défini la variage age qu'on utilisera et un variable définissant l'age de la majorité

let age;

let ageMajorite = 18;

//Cette fonction affichera le message de validation

function valider(){

espaceMessage.innerHTML = "Vous êtes autorisé à entrer";

}

//Cette fonction affichera un message d'erreur

function refuser(){

alert("Cette espace est interdit aux personnes mineurs");

}

function onConfirm(){

//On récupère la saisie de l'age et on transforme le texte en nombre entier

age = parseInt(ageInput.value);

//Si la saisie n'est pas un nombre, on affiche un message d'erreur

if(isNaN(age)){

alert("Ceci n'est pas un nombre");

return;

}

**//=======Code ======**

**if(age < ageMajorite){**

**refuser();**

**}else{**

**valider()**

**}**

**//=======/Code =====**

//On vide le champ de saisie

ageInput.value = "";

}

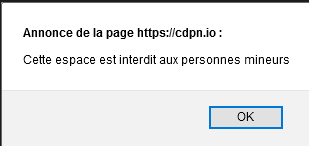
//On écoute l'action de click sur le bouton et on appelle la fonction onConfirm

bouton.addEventListener('click', onConfirm);

**Affichage**

Une image contenant texte

Description générée automatiquement



#### Utilisez des expressions de comparaison

**Les expressions** **de comparaison** vous permettent de comparer deux valeurs par les opérateurs suivants :

##### Opérateurs de comparaison en JavaScript

|  |  |  |  |
| --- | --- | --- | --- |
| **Opérateurs de comparaison en JavaScript** | | | |
| N° | Opérateur | Signification | **Remarque** |
| 1 | < | Inférieur à |  |
| 2 | <= | Inférieur ou égal à |  |
| 3 | == | Égal à | **Égalité simple**. Vérifie la **valeur**, **mais pas le type** |
| 4 | === | Égal à | **Égalité stricte**. Vérifie à la fois la **valeur et le type**. |
| 5 | >= | Supérieur ou égal à |  |
| 6 | > | Supérieur à |  |
| 7 | != | Différent de | Inégalité simple. Vérifie la **valeur**, **mais pas le type** |
| 8 | !== | Différent de | Inégalité stricte. Vérifie à la fois la **valeur et le type**. |

Tableau : Opérateurs de comparaison en JavaScript

Par exemple :

Une image contenant texte

Description générée automatiquement

Il est possible[[32]](#footnote-32) **chaîner** les instructions **if /else** pour réagir à des conditions potentielles multiples :

Une image contenant texte

Description générée automatiquement

**Le chaînage d'instructions** permet de prévoir différents résultats en fonction des différentes situations.

##### Pratiquez : les instructions if/else

Dans cet exercice, j'ai légèrement modifié notre classe Episode . Au lieu de préciser si un épisode a été visionné au moment de construire l'instance, on va simplement donner le nombre de minutes visionnées par l'utilisateur, en le passant comme argument au ***constructor*** .

Il y a des parenthèses vides pour les instructions **if** et **else**. Complétez ces conditions pour que la valeur de la propriété **hasBeenWatched** soit assignée correctement.

Observez. Le premier épisode devrait afficher "Déjà regardé", et les deux autres "Pas encore regardé".

Si l'utilisateur a commencé à regarder un épisode mais ne l'a pas encore terminé, on considère l'épisode comme n'étant pas encore regardé.

###### Solution

**HTML**

<!DOCTYPE html>

<html lang="en">

    <head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

    <link rel="stylesheet" href="style.css">

    <script src="script.js" async></script>

    <title>if / Else</title>

    </head>

    <body>

        <h3>if / Else</h3>

    </body>

</html>

**CSS**

body {

background-color: #b7ffb7;

color: #000;

font-family: 'Roboto Slab', serif;

}

.series-frame{

max-width: 600px;

/\*font-size: 0.9em;\*/

display: flex;

/\*justify-content: space-between;\*/

align-items: center;

box-sizing: border-box;

border: 2px solid #804040;

margin: 30px;

padding: 10px;

}

.my-form {

padding: 20px;

}

.series-frame h2 {

margin-right: 20px;

}

**JS**

class Episode {

    constructor(title, duration, minutesWatched) {

      this.title = title;

      this.duration = duration;

      // Add conditions here

      // =================================

      if (minutesWatched == duration) {

        this.hasBeenWatched = true;

      } else if (minutesWatched < duration) {

        this.hasBeenWatched = false;

      }

      // =================================

    }

  }

  let firstEpisode = new Episode('Dark Beginnings', 45, 45);

  let secondEpisode = new Episode('The Mystery Continues', 45, 10);

  let thirdEpisode = new Episode('An Unexpected Climax', 60, 0);

  let episodes = [firstEpisode, secondEpisode, thirdEpisode];

  const body = document.querySelector('body');

  for(let i = 0; i < 3; i++) {

    let newDiv = document.createElement('div');

    newDiv.classList.add('series-frame');

    let newTitle = document.createElement('h2');

    newTitle.innerText = 'The Story of Tau';

    let newParagraph = document.createElement('p');

    newParagraph.innerText = `${episodes[i].title}

  ${episodes[i].duration} minutes

  ${episodes[i].hasBeenWatched ? 'Already been watched' : 'Not yet watched'}`;

    newDiv.append(newTitle);

    newDiv.append(newParagraph);

    body.append(newDiv);

  }

**Ecran**



#### Découvrez l'égalité : == ou ===

En JavaScript, toutes les égalités ne sont pas nées égales :

Il y a deux façons de vérifier si deux valeurs sont égales en JavaScript : **==**  et **===**, aussi appelées égalité simple et égalité stricte :

►L’égalité simple vérifie la valeur, mais pas le type. Donc ceci :

**5 == "5"** renvoie la valeur **true** :

►En revanche, l'égalité stricte vérifie à la fois la valeur et le type. Donc ceci  :

5 === "5" renvoie **false**, car on compare un **number** à un **string**.

►De même, il y a deux opérateurs d'inégalité  **!=** et  **!==,** avec la même distinction.

#### Les conditions multiples

Dans certaines situations, on souhaite vérifier plusieurs conditions pour un même résultat ; par exemple dans la même instruction **if**. Pour cela, il existe des **opérateurs logiques** :

##### Les 3 principaux opérateurs logiques

|  |  |  |  |
| --- | --- | --- | --- |
| **Opérateurs logiques en JavaScript** | | | |
| N° | Opérateur | Signification | **Description** |
| 1 | && | ET logique | Vérifier si deux conditions sont **toutes les deux** vraies |
| 2 | || | OU logique | Vérifier si **au moins une** condition est vraie |
| 3 | ! | NON logique | Vérifier si une condition n'est **pas** vraie |

Tableau : Les 3 principaux opérateurs logiques

Une image contenant texte

Description générée automatiquement

#### Pratiquez : les opérateurs logiques

Modifions encore un peu la logique de la classe ***Episode*** . Le composant de votre collègue nécessite maintenant des épisodes avec les **propriétés** suivantes :

**title** : le titre de l'épisode (**string**)

**duration** : la durée de l'épisode en minutes (**number**)

**watchedText** : un **string** contenant "*Pas encore regardé*", "*En train de regarder*" ou "*Déjà regardé*" **selon le nombre de minutes visionnées**

**continueWatching** : un **boolean** qui sera **true** **si l'utilisateur a commencé à regarder l'épisode** mais pas encore terminé, et false sinon

C'est à vous de jouer ! Suivez les étapes suivantes :

Avec une ou plusieurs instructions **if/else**, **ajoutez la logique requise** au ***constructor*** de la ***classe*** **Episode** .

Est-ce possible de raccourcir votre code ou de le rendre plus lisible ?

N'oubliez pas d'utiliser le mot-clé **this** pour ajouter des propriétés à la ***classe***.

#### Pratiquez : les opérateurs logiques 2

Nous reprenons l’exercice précédent avec une checkbox servant à indiquer si le contrôle parental est activé ou non. On se retrouve avec 2 situations :

Le contrôle parental est activé et il faut être majeur pour accéder au contenu.

Le contrôle parental est désactivé et tout le monde accède au contenu.

Il y a donc une notion de contrôle parental qui se retrouve dans une nouvelle variable dans le code.

Votre mission est donc d’écrire la condition de la même manière que l’exercice précédent au niveau des lignes 40 à 42, en prenant en compte la nouvelle variable isControlParentalActive .

Avant de commencer à coder votre solution, n’hésitez pas à regarder comment la variable isControlParentalActive est obtenue. Mais si vous ne comprenez pas, ce n’est pas grave, l’objectif ici est avant tout d’exploiter sa valeur dans une condition.

#### Solution

**HTML**

<h1>Quel est votre age?</h1>

<div id="control">

<input type="checkbox" id="parental" name="parental" checked>

<label for="parental">Contrôle parental activé</label>

</div>

<input id="age" type="text" placeholder="entrez votre age">

<input id="bouton" type="button" value="confirmer">

<h2 id="message"></h2>

**CSS**

body{

text-align:center;

}

#control{

padding-bottom:10px;

}

**JS**

//On pointe sur l'élément de message

const espaceMessage = document.getElementById("message");

//On pointe sur l'élément de bouton

const bouton = document.getElementById("bouton");

//On pointe sur l'élément de champ de saisie de l'age

const ageInput = document.getElementById("age");

//On pointe sur l'élément de la checkbox de contrôle parental

const parentalInput = document.getElementById("parental");

//On défini la variage age qu'on utilisera et un variable définissant l'age de la majorité

let age;

let isControlParentalActive;

let ageMajorite = 18;

//Cette fonction affichera le message de validation

function valider(){

espaceMessage.innerHTML = "Vous êtes autorisé à entrer";

}

//Cette fonction affichera un message d'erreur

function refuser(){

alert("Cette espace est interdit aux personnes mineurs");

}

function onConfirm(){

//On récupère la saisie de l'age et on transforme le texte en nombre entier

age = parseInt(ageInput.value);

//Si la saisie n'est pas un nombre, on affiche un message d'erreur

if(isNaN(age)){

alert("Ceci n'est pas un nombre");

return;

}

//On récupère la valeur du checkbox de contrôle parentale. "true" si coché ou "false" si décoché

isControlParentalActive = parentalInput.checked;

//====Code à rédiger ici======

//si non majeur et que le controle parental est activé appeller la fonction refuser(), sinon apeller la fonction valider()

//=======/Code à rédiger ici======

**if (age < ageMajorite && isControlParentalActive){**

**refuser();**

**}else{**

**valider()**

**}**

//On vide le champ de saisie

ageInput.value = "";

}

//On écoute l'action de click sur le bouton et on appelle la fonction onConfirm

bouton.addEventListener('click', onConfirm);

**Affichage**

**►cas 1**

**Une image contenant texte

Description générée automatiquement** **Une image contenant texte

Description générée automatiquement**

**►cas 2**

**Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement**

### Le scope des variables

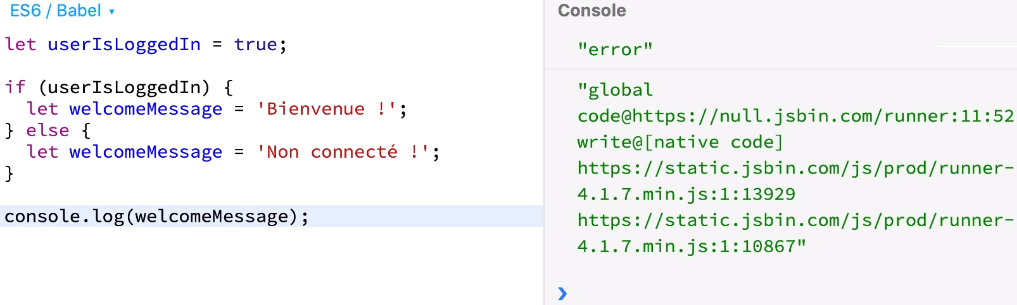
En JavaScript, les variables créées par ***let*** ou ***const*** ne peuvent être vues ou utilisées qu'à l'intérieur du **bloc de code** dans lequel elles sont déclarées.

**►Un bloc de code**, aussi souvent appelé bloc tout court, est **une section de code incluse entre accolades { }.**

Vous rencontrerez certainement le mot-clé ***var*** pour la création de variables.

Les variables déclarées ainsi n'ont pas un scope de bloc **mais un scope de fonction** ; donc elles n'ont pas tout à fait le même comportement que celles que je décris dans ce cours. Pour plus d'informations, je vous conseille ceci [en français](https://www.azur-web.com/astuces/javascript-differences-var-let-const)[[33]](#footnote-33) ou cet article[[34]](#footnote-34) ([en anglais](https://www.geeksforgeeks.org/difference-between-var-and-let-in-javascript/)) :

Ce phénomène est appelé **scope de bloc**, ou **block** **scope**. Voici un exemple :



Quand vous essayez d'afficher le contenu de **welcomeMessage** sur la console, vous obtenez une erreur : pour le code en dehors des blocs (aussi appelé portée extérieure ou parente), il n'y a pas de variable **welcomeMessage** !

**Explications :**

Dans ce code, nous avons deux blocs de code issus de l'instruction **if / else**. Ils sont mis en évidence dans l'image ci-dessous :

Une image contenant texte

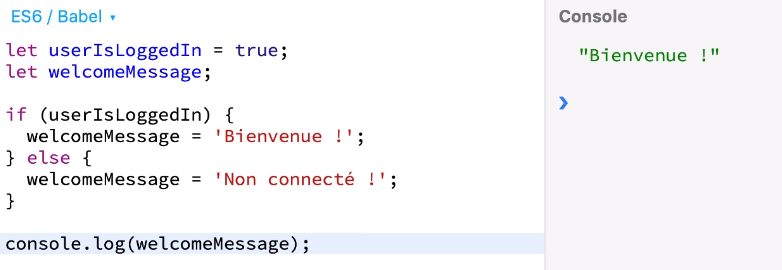
Description générée automatiquement

Figure : Scopes de variables surlignés

Les deux blocs orange et rose déclarent une variable par **let** **welcomeMessage**. Néanmoins, ces variables ne sont disponibles **qu'à l'intérieur des blocs** où elles sont déclarées.

►Donc, comment obtenir le comportement voulu ? Comment passer des valeurs vers l'extérieur depuis l'intérieur d'un bloc de code ?

Dans ce cas, une méthode pourrait être de déclarer la variable dans la **portée extérieure**, puis de la modifier à l'intérieur des blocs **if / else** :



Ici, du fait que la variable est déclarée **dans le scope parent**, elle est **disponible partout,** accessible et peut être modifiée correctement.

### Les instructions switch

l'instruction **switch** :

Supposons que vous ayez quelques objets utilisateurs et souhaitiez vérifier quel type de compte à chacun, pour envoyer un message personnalisé :

**let firstUser = {**

**name: "Will Alexander",**

**age: 33,**

**accountLevel: "normal"**

**};**

**let secondUser = {**

**name: "Sarah Kate",**

**age: 21,**

**accountLevel: "premium"**

**};**

**let thirdUser = {**

**name: "Audrey Simon",**

**age: 27,**

**accountLevel: "mega-premium"**

**};**

Vous pouvez ici utiliser une instruction **switch**, qui prend la variable à vérifier et une liste de valeurs comme différents cas :

**switch (firstUser.accountLevel) {**

**case 'normal':**

**console.log('You have a normal account!');**

**break;**

**case 'premium':**

**console.log('You have a premium account!');**

**break;**

**case 'mega-premium':**

**console.log('You have a mega premium account!');**

**break;**

**default:**

**console.log('Unknown account type!');**

**}**

* Après le code à exécuter dans chaque instruction case, ajoutez l'instruction **break**.
* Si vous ne le faites pas, JavaScript continuera l'exécution des cas suivants (en cascade) jusqu'à rencontrer une instruction **break** ou la fin de l'instruction **switch**.
* Comme vous pouvez le voir dans l'exemple, vous pouvez aussi ajouter un cas **default** qui ne sera exécuté que si la variable que vous vérifiez ne correspond à aucune des valeurs répertoriées.

Ici, également, les instructions **switch** utilisent aussi des accolades. C'est un autre exemple de **bloc de code**, **donc les variables éventuellement déclarées à l'intérieur d'une instruction switch ne seront pas disponibles pour le code environnant !**

#### Instruction switch pour afficher une date personnalisée

**function EuDate(){**

**var dt = new Date();**

**var day = dt.getDay();**

**var fullDay = "";**

**switch(day){**

**case 0 : fullDay = "Sunday";**

**break;**

**case 1 : fullDay = "Monday";**

**break;**

**case 2 : fullDay = "Tuesday";**

**break;**

**case 3 : fullDay = "Wednesday";**

**break;**

**case 4 : fullDay = "Thursday";**

**break;**

**case 5 : fullDay = "Friday";**

**break;**

**case 6 : fullDay = "Saturday";**

**break;**

**}**

**var myDate = fullDay + ' ' + dt.getDate() + '/' + (dt.getMonth() +1) + '/' + dt.getFullYear();**

**console.log(myDate);**

**}**

**EuDate();**



### Résumé

1. instructions **if / else** ;
2. Les différents **types de** ***condition*** pouvant être utilisés pour les instructions **if/else** ;
3. Regrouper les différentes conditions avec des ***opérateurs logiques*** ;
4. La ***portée des variables***, et les conséquences qu'elle a sur la ***structure du code*** ;
5. l'instruction **switch** pour comparaison à une liste de valeurs attendues.

## Utilisez la bonne boucle *for* ou *while*, pour repeter les taches

En programmation, il y a **des ensembles d'instructions à répéter plusieurs fois**.

►Parfois, vous **connaîtrez à l'avance le nombre de répétitions** → **FOR**

**►d'autres fois non**. Il est aussi possible que le nombre de fois n'importe pas, et que vous souhaitiez **répéter le code jusqu'à atteindre une certaine condition**. → **WHILE**

Pour tous ces cas, nous utiliserons des boucles.

* **Freeplane**

### Utilisez la boucle for pour savoir "combien de fois ?"

Supposons que vous ayez à faire embarquer 10 passagers, et que pour l'instant vous **n'accordiez pas d'importance à l'ordre** de leur embarquement. Utilisez une boucle for pour les embarquer un par un jusqu'à atteindre 10 :

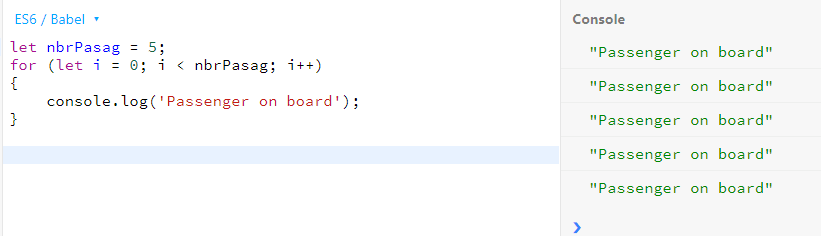
**let nbrPasag = 5;**

**for (let i = 0; i < nbrPasag; i++)**

**{**

**console.log('Passenger on board');**

**}**



Le travail sur des indices c'est bien, mais si vous souhaitiez utiliser un **tableau** de passagers pour les embarquer dans un ordre précis ?

### Travaillez sur des tableaux : for…of et for…in

Dans beaucoup de cas[[35]](#footnote-35), vous aurez un tableau et devrez le parcourir, pour faire quelque chose sur chaque élément.

Il peut s'agir de mettre à jour les prix sur un menu, de **traiter les données d'une base de données**, ou d'embarquer des passagers dans l'ordre.

#### Parcourir un tableau avec for

Une image contenant texte, intérieur, capture d’écran

Description générée automatiquementL'ancienne façon de parcourir un tableau dans une boucle était d'utiliser la boucle **for** vue précédemment avec sa propriété **length**. Par exemple, avec un tableau appelé **passengers** :

**let passengers = ['toto', 'titi','zozo'];**

**for (let i = 0; i < passengers.length; i++)**

**{**

**console.log ('Passager embarqué !');**

**}**

Bien que ceci fonctionne, il y a **2 autres** façons de parcourir en boucle des tableaux.

#### Parcourir un tableau avec for…in

**const passengers = [**

**"Will Alexander",**

**"Sarah Kate'",**

**"Audrey Simon",**

**"Tao Perkington"**

**]**

**for (let i in passengers) {**

**console.log("Embarquement du passager " + passengers[i]);**

**}**

Une image contenant texte

Description générée automatiquement

La boucle **for … in** est très comparable[[36]](#footnote-36) à l'exemple de boucle **for** normale, mais elle est plus facile à lire.

#### Parcourir un tableau avec for…of

Pour les cas où l'indice précis d'un élément n'est pas nécessaire pendant l'itération, vous pouvez utiliser une boucle **for … of** :

**const passengers = [**

**"Will Alexander",**

**"Sarah Kate",**

**"Audrey Simon",**

**"Tao Perkington"**

**]**

**for (let passenger of passengers) {**

**console.log("Embarquement du passager " + passenger);**

**}**

Une image contenant texte

Description générée automatiquement

Ceci atteint exactement le même résultat, mais de façon plus lisible, car vous n'avez pas à vous inquiéter des indices et des tableaux : vous recevez simplement chaque élément dans l'ordre.

**C'est encore plus utile si le tableau est un peu plus complexe et contient par exemple des objets :**

**const passengers = [**

**{**

**name: "Will Alexander",**

**ticketNumber: 209542**

**},**

**{**

**name: "Sarah Kate",**

**ticketNumber: 169336**

**},**

**{**

**name: "Audrey Simon",**

**ticketNumber: 779042**

**},**

**{**

**name: "Tao Perkington",**

**ticketNumber: 703911**

**}**

**]**

**for (let passenger of passengers) {**

**console.log('Embarquement du passager ' + passenger.name + ' avec le ticket numéro ' + passenger.ticketNumber);**

**}**

Une image contenant texte

Description générée automatiquement

##### Pratiquez : les boucles for

Maintenant, vous devez implémenter du code qui remettra à zéro l'information de visionnage pour tous les épisodes. Vous allez accomplir cette tâche en itérant le tableau **episode** en assignant la valeur **false** à la propriété **hasBeenWatched** de chaque épisode.

Le tableau **episodes** a déjà été déclaré pour vous.

Itérez le tableau **episodes** et assignez la valeur **false** à la propriété **hasBeenWatched** de chaque épisode, en utilisant une boucle **for**, **for**…**in** ou **for**…**of**.

Maintenant, faites en sorte que tous les épisodes soient considérés comme ayant été vus. Cela fonctionne t-il également ?

###### Solution

**HTML :**

<!DOCTYPE html>

<html lang="en">

    <head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

    <link rel="stylesheet" href="style.css">

    <script src="script.js" async></script>

    <title>JS-Boucles For</title>

    </head>

    <body>

        <h3>Boucles For</h3>

    </body>

</html>

**CSS :**

body{

    background-color: #b7ffb7;

    color: #000;

    font-family: 'Roboto Slab', serif;

}

.series-frame {

    max-width: 600px;

    display: flex;

    align-items: center;

    box-sizing: border-box;

    border: 2px solid tomato;

    margin: 30px;

    padding: 10px;

  }

  .series-frame h2 {

    margin-right: 20px;

  }

**JS :**

//  <h3>Boucles For</h3>

class cEpisode {

  constructor(title, duration, hasBeenWatched) {

    this.title = title;

    this.duration = duration;

    this.hasBeenWatched = hasBeenWatched;

  }

}

let episodes = [

  new cEpisode('Dark Beginnings', 45, true),

  new cEpisode('The Mystery Continues', 45, false),

  new cEpisode('An Unexpected Climax', 60, false)

];

// ======================

**for (let episode of episodes){**

**episode.hasBeenWatched = false;**

}

// ======================

const body = document.querySelector('body');

for(let episode of episodes) {

  let newDiv = document.createElement('div');

  newDiv.classList.add('series-frame');

  let newTitle = document.createElement('h2');

  newTitle.innerText = 'The Story of Tau';

  let newParagraph = document.createElement('p');

  newParagraph.innerText = `${episode.title}

${episode.duration} minutes

${episode.hasBeenWatched ? 'Already been watched' : 'Not yet watched'}`;

  newDiv.append(newTitle);

  newDiv.append(newParagraph);

  body.append(newDiv);

}

Affichage



### Utilisez la boucle while pour continuer jusqu'à ce qu'on dise d'arrêter

Une boucle **while** vérifie si une condition est vraie. Si c'est le cas, la boucle se poursuit ; sinon elle s'arrête.

En revenant à l'exemple de l'embarquement d'avion : supposons que vous ayez des variables représentant le ***nombre de sièges restants*** et le ***nombre de passagers restants***. Vous souhaitez continuer l'embarquement jusqu'à l'épuisement du nombre de sièges ou de passagers :

let seatsLeft = 10;

let passengersStillToBoard = 8;

let passengersBoarded = 0;

while (seatsLeft > 0 && passengersStillToBoard > 0) {

    passengersBoarded++; // un passager embarque

    passengersStillToBoard--; // donc il y a un passager de moins à embarquer

    seatsLeft--; // et un siège de moins

}

console.log('Number of passengers abord: ' + passengersBoarded );// imprime 8, car il y a 8 passagers pour 10 sièges

Une image contenant texte

Description générée automatiquement

Cette boucle **while** poursuit son exécution jusqu'à ce que l'un des nombres **seatsLeft** et **passengersStillToBoard** atteigne zéro, et à ce point elle se termine.

### En résumé

1. boucle **for**, pour un nombre d'itérations fixe ;
2. boucle **while**, quand le nombre d'itérations nécessaire est inconnu.

## Gérez des erreurs et des exceptions dans votre programme

**Ne pas faire d'erreur lorsque l'on code est pratiquement impossible !**

### Trois types d'erreur

**En JavaScript, comme dans beaucoup d'autres langages de programmation**, on distingue généralement trois types d'erreur.

#### Les erreurs de syntaxe

Les erreurs de **syntaxe** (ou erreurs **d'analyse**) surviennent quand vous avez fait une faute d'écriture dans votre code.

Beaucoup d'éditeurs de texte ou d'environnements de développement intégrés[[37]](#footnote-37) mettent automatiquement en évidence les erreurs de syntaxe.

Un IDE est généralement spécifique à un langage, à une architecture ou à un domaine, et certains sont gratuits, comme [Visual Studio Code](https://code.visualstudio.com/)[[38]](#footnote-38), alors que d'autres ne le sont pas, comme la suite [JetBrains](https://www.jetbrains.com/)[[39]](#footnote-39).

##### Exemples d'erreurs de syntaxe

Une image contenant texte

Description générée automatiquement

Figure : Trois exemples d'erreurs de syntaxe

Solutions : 1) il manque un +, 2) la parenthèse n'est pas fermée, 3) l'accolade n'est pas fermée !

#### Les erreurs de logique

Les erreurs logiques sont un peu plus **vicieuses**. Elles surviennent quand vous avez fait des erreurs dans la logique de votre programme. Voici quelques exemples :

* Affectation d'une valeur erronée à une variable ;
* Mélange de conditions dans les instructions **if** ;
* Ordre incorrect d'écriture des lignes ou des blocs de code.
* Oubli d'incrémentation de l'indice d'une boucle, pouvant potentiellement conduire à **une boucle infinie** ;

Avec ce genre d'erreur, votre programme pourra avoir un comportement inattendu dans le meilleur des cas, **ou planter dans le pire des cas !**

**Les erreurs logiques sont aussi plus difficiles à trouver et à corriger parce que le code n'est pas faux** ; il ne fait simplement pas ce que vous souhaitez qu'il fasse !

#### Les erreurs d'exécution

Les erreurs d'exécution sont un peu différentes. Elles tendent à survenir quand quelque chose d'inattendu se produit dans votre application.

Il s'agit souvent de quelque chose associé aux **ressources extérieures** (connexions réseau, appareils physiques, etc.) ou à une saisie/erreur humaine.



Une façon de traiter les erreurs potentielles consiste à utiliser une instruction **if / else** pour vérifier la validité des données :

Une image contenant texte

Description générée automatiquement

Figure : vérifier la validité des données

On peut aussi utiliser des blocs **try** / **catch** pour essayer (**try**) un code pouvant potentiellement renvoyer une erreur, et détecter (**catch**) les erreurs éventuelles survenues :

Une image contenant texte

Description générée automatiquement

Figure : bloc try / catch

##### Les gestions d’exceptons

Dans **certaines situations**, vous pouvez prévoir du ***code de traitement d'erreur***. De cette façon, une erreur ne fera pas planter votre programme, et pourra être corrigée.

On parle de gestion d’exceptions ou de : **Exception handling**.

Avec plus **d’expérience** en développement JavaScript, on saura **anticiper** où les erreurs peuvent survenir, et comment s’ y préparer et **les traiter correctement.**

### Résumé

* **Erreurs de syntaxe** — erreurs dans l'utilisation de JavaScript ;
* **Erreurs logiques** — erreurs dans l'application du déroulement du programme, boucle infinie ;
* **Erreurs d'exécution** — souvent causées par des ressources externes telles que réseaux, bases de données ou utilisateurs ; elles peuvent être traitées en sachant où elles peuvent survenir, et par des techniques de traitement d'erreur telles que les blocs **try / catch**.

## Résumé de la partie 2

Dans cette partie du cours, nous avons parlé de la logique de programme. Nous avons vu :

Certains **environnements** où JavaScript peut être utilisé, et la façon dont son **exécution** peut être différente d'un environnement à l'autre ;

Les **instructions** **conditionnelles** telles que **if/else** et **switch** ;

Les **boucles** pour des sections répétitives de code, telles que **for** et **while** ;

Les différents types **d'erreurs**, et comment les traiter.

## QUIZ 2

[Second quiz](file:///D:\OPENCLASSROOMS\Apprenez%20à%20programmer%20avec%20JavaScript\QUIZs\Quiz_2\Vérifiez%20ce%20que%20vous%20avez%20appris%20sur%20la%20logique%20informatique.pdf)

# LES FONCTIONS EN JAVASCRIPT

(ÉCRIRE DU CODE PROPRE ET FACILE A MAINTENIR)

Il existe quatre syntaxes différentes nous permettant de créer une [fonction en JavaScript](https://www.pierre-giraud.com/javascript-apprendre-coder-cours/fonction-flechee/)[[40]](#footnote-40).

1. une déclaration de fonction ;
2. une expression de fonction ;
3. une fonction fléchée ;
4. la syntaxe objet : new Function.

## La déclaration de fonction classique

La **syntaxe** de déclaration de fonction est la suivante :

* On déclare une fonction avec le **mot clef** function,
* on donne un **nom** à notre fonction
* et on fait suivre ce nom par un **couple de parenthèses**
* et un **couple d’accolades**

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

**function sayHello(message){**

**console.log(message);**

**}**

**sayHello('Hello !');**

Une image contenant texte

Description générée automatiquement

Ou encore :

**const coutFormation = 1000;**

**function prixTTC(prixHT){**

**let cout = prixHT \* 1.2;**

**return cout;**

**}**

**console.log('Cout total de la formation → ' + prixTTC(coutFormation));**

**Une image contenant texte

Description générée automatiquement**

## L’expression de fonction

La syntaxe de l’expression de fonction est la suivante :

* similaire à celle-ci-dessus, plus :
* on **assigne** la fonction à une **variable**

**let sayHello = function(message){**

**console.log(message);**

**}**

**sayHello('Hello !');**

Une image contenant texte

Description générée automatiquement

### Named Function Expression (NFE)

On peut donner un nom à la fonction. Dans ce cas-là, on parlera de « NFE » pour « ***Named Function Expression*** » ou « **expression de fonction nommée** » en français.

Ajouter un nom à une expression de fonction permet à la fonction **de faire référence à elle-même** en interne en étant sûr de s’appeler 🡺 [**récursivité**](https://www.pierre-giraud.com/javascript-apprendre-coder-cours/fonction-anonyme-auto-invoquee-recursive/)**[[41]](#footnote-41).**

**let sayHello = function myNameIsPerson(anyName){**

**if(anyName){**

**console.log('Hello' + String.fromCharCode(32) + anyName);**

**}**

**else{**

**myNameIsPerson('unknown !')**

**}**

**}**

**sayHello('Ulysses');**

**sayHello();**

Une image contenant texte

Description générée automatiquement

En effet, en utilisant une expression de fonction classique, la variable qui contient l’expression de fonction peut changer de valeur dans le script et rendre notre fonction inutilisable.

De plus, le nom passé aux NFE n’est pas accessible depuis l’extérieur de la fonction. Cela garantit qu’en l’utilisant dans notre expression de fonction, il fera toujours bien référence à la fonction actuelle.

## La fonction fléchée (Arrow function)

C’est une notation apparue depuis ES6[[42]](#footnote-42).La fonction fléchée utilise le signe => qui lui a donné son nom à cause de sa forme de flèche.

Rappel de la syntaxe de la fonction classique :

//La déclaration de fonction classique :

function remainder(x, y){

return x % y;

}

console.log(remainder(17,5));

### Syntaxe de la fonction fléchée

**►**la syntaxe de la fonction fléchée est très **concise** :

* Il faut **obligatoirement** mettre la fonction dans une **variable**
* La fonction fléchée **n’utilise pas** de mot clé **function,** il est **remplacé par : =>**
* La fonction fléchée **n’a pas besoin du couple d’accolades** classique aux fonctions pour fonctionner
* n’a **pas besoin non plus d’une expression return** puisque celle-ci va automatiquement évaluer l’expression à droite du signe **=>** et retourner son résultat.

**let remainder = (x,y) => (x % y);**

**console.log(remainder(17,5));**

Une image contenant texte

Description générée automatiquement

### Cas de la fonction fléchée à argument unique

Si la fonction fléchée n’a besoin que d’un argument pour fonctionner, alors le couple de parenthèses devient facultatif.

Rappel de la syntaxe de la fonction classique :

function squareRoot(x) {

return Math.sqrt(x);

}

console.log(squareRoot(49));

►Syntaxe de la fonction fléchée :

**let squareRoot = x => Math.sqrt(x);**

**console.log(squareRoot(49));**

Une image contenant texte

Description générée automatiquement

### Cas de la fonction fléchée sans argument

Si la fonction fléchée n’a besoin d’aucun argument pour fonctionner, alors on met soit des **parenthèses vides** ou bien un **underscore** (\_)

Rappel de la syntaxe de la fonction classique :

function sayHello(){

console.log('Hello !');

}

sayHello();

Une image contenant texte

Description générée automatiquement

►Syntaxe de la fonction fléchée :

**const sayHello = () => console.log('Hello !');**

**sayHello();**

Ou :

**const sayHello = \_ => console.log('Hello !');**

**sayHello();**

Une image contenant texte

Description générée automatiquement

### Cas de la fonction fléchée avec plusieurs expressions

**Dans ce cas les parenthèses et les accolades redeviennent obligatoires.**

Rappel de la syntaxe de la fonction classique :

function hypotenuse(a,b){

let sumPow = Math.pow(a,2) + Math.pow(b,2);

return Math.sqrt(sumPow);

}

console.log(hypotenuse(3,4))

Une image contenant texte

Description générée automatiquement

►Syntaxe de la fonction fléchée :

**let hypotenuse = (a,b) => {**

**let sumPow = Math.pow(a,2) + Math.pow(b,2);**

**return Math.sqrt(sumPow);**

**}**

**console.log(hypotenuse(3,4));**

Une image contenant texte

Description générée automatiquement

## Function anonyme

## IIFE

<https://medium.com/simplon-saint-gaudens/i-i-f-e-m%C3%A8s-ques-aqu%C3%B2-cee4c441026b>

## La syntaxe objet : new *Function*

Le constructeur Function crée un nouvel objet Function. En JavaScript, chaque fonction est un objet Function.

**Syntaxe : new Function ([arg1[, arg2[, ...argN]],] corpsFonction)**

Attention à la syntaxe du constructeur new **F**unction : F majuscule !

**Exemple addition :**

**let addition = new Function('a', 'b', 'return a + b');**

**console.log(addition(5,7));**

Une image contenant texte

Description générée automatiquement

**Exemple racine carrée :**

**let rootSquare = new Function('x', 'return Math.sqrt(x)');**

**console.log(rootSquare(144));**



## Travaillez sur les fonctions

### Les deux types de fonctions en javascript

En Javascript, il existe deux types de fonctions :

1. les **fonctions propres à Javascript[[43]](#footnote-43)**: **prédéfinies[[44]](#footnote-44)** On les appelle des "**méthodes**". Elles sont associées à un objet bien particulier comme c'était le cas de la méthode Alert() avec l'objet window.
2. les fonctions **écrites par le développeur** pour les besoins du script. C'est à celles-là que nous nous intéressons ici.

### Comprenez les fonctions

Une **fonction** est un bloc de code auquel vous attribuez un nom.

Quand vous **appelez** cette fonction, vous exécutez le code qu'elle contient

►Quand vous créez ou déclarez une fonction, vous indiquez la liste des variables dont elle a besoin pour effectuer son travail : vous **définissez les paramètres de la fonction**

►Á l'appel de la fonction, vous lui attribuez des valeurs pour ses paramètres. Les valeurs sont **les arguments** d'appel.

►La fonction peut vous donner un résultat : une **valeur de retour**.

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

**console.log('First value : ' + val1);**

**console.log('Second value : ' + val2);**

**}**

**displayTwoValues(13,'Hello ☺');**

Une image contenant texte

Description générée automatiquement

#### Pratiquez la note moyenne

Avant de commencer l’exercice, nous allons juste revoir deux notions : le calcul d’une moyenne et comment parcourir un tableau.

##### Calculer une moyenne

Lorsque l’on a un ensemble de valeurs, le calcul de la moyenne[[45]](#footnote-45) se fait en 3 étapes :

1. on compte le nombre de valeurs que l’on a (qu’on nommera nombreValeurs);
2. on fait la somme de toutes les valeurs (qu’on nommera sommeValeurs ) ;
3. on calcule la moyenne en faisant moyenne = sommeValeurs / nombreValeurs

##### Parcourir un tableau

Lorsque l’on a un tableau (par exemple monTableau ) on utilise la boucle **for of**.

Par exemple :

**let myTable = [1,"two",3,"for",5];**

**for(let itemTable of myTable){**

**console.log(itemTable);**

**}**

Une image contenant texte

Description générée automatiquement

Vous pouvez manipuler **itemTable** comme bon vous semble (par exemple, l’ajouter dans un autre tableau, l'additionner avec une variable globale, etc.).

Votre application de streaming permet aux utilisateurs de noter les séries sur 5 étoiles. Votre collègue a construit le composant pour afficher la note **moyenne** pour chaque série, mais elle a besoin que vous écriviez la **fonction qui va calculer cette moyenne**.

Elle vous explique que **la fonction doit prendre un tableau de nombres comme argument**, et **retourner un nombre qui correspond à la note moyenne** calculée.

#### Solution

**HTML**

<link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="tau-info">Average rating: <span id="tau-score"></span></p>

</div>

<div class="series-frame">

<h2>The Hero of Old Meldrum</h2>

<p id="colin-info">Average rating: <span id="colin-score"></span></p>

</div>

<div class="series-frame">

<h2>The Bugs of Isla Clara</h2>

<p id="clara-info">Average rating: <span id="clara-score"></span></p>

</div>

**CSS**

body {

background-color: #111;

color: #EEE;

font-family: 'Roboto Slab', serif;

}

.series-frame {

max-width: 600px;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #EEE;

margin: 30px;

padding: 10px;

}

**JS**

**const calculateAverageRating = (ratings) => {**

**if(ratings.length === 0) {**

**return 0;**

**}**

**let sum = 0;**

**for (let rating of ratings) {**

**sum += rating;**

**}**

**return sum / ratings.length;**

**}**

const tauRatings = [5, 4, 5, 5, 1, 2];

const colinRatings = [5, 5, 5, 4, 5];

const tauAverage = calculateAverageRating(tauRatings);

const colinAverage = calculateAverageRating(colinRatings);

if (tauAverage && colinAverage) {

document.querySelector('#tau-score').innerText = tauAverage.toFixed(1) + ' stars';

document.querySelector('#colin-score').innerText = colinAverage.toFixed(1) + ' stars';

document.querySelector('#clara-score').innerText = `${calculateAverageRating([]) === 0 ? 'No ratings' : calculateAverageRating([]) + ' stars'}`

}

**Explication du code**

let sum = 0;

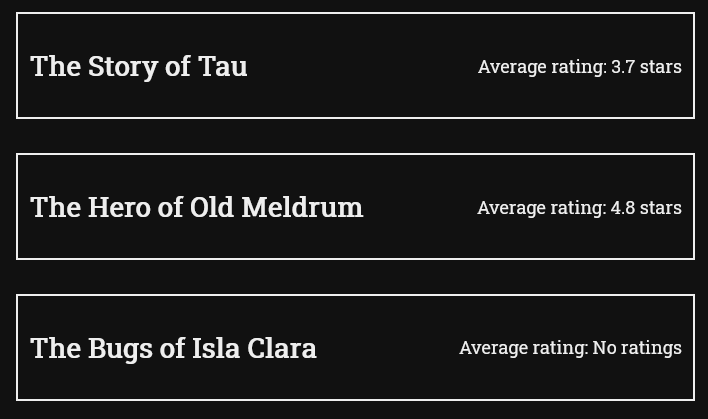
for (let rating of ratings) {

**sum += rating;**

}

►On parcourt le tableau ratings et pour **chaque élément on additionne la valeur avec la valeur de** sum et le résultat est stocké dans sum **.**

**Affichage**



### ~~Ajoutez des tableaux et des objets dans les fonctions~~

## Écrire des fonctions propres

**Si, au moins deux lignes de code se répètent, alors il faut en faire une fonction !**

Une image contenant texte

Description générée automatiquement

**Clean code !**

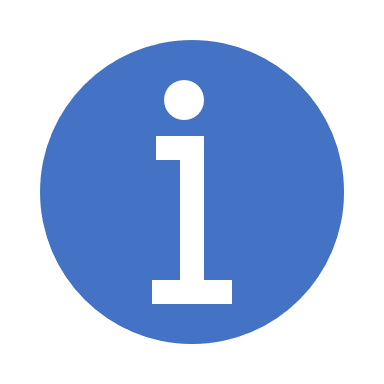
Une image contenant texte

Description générée automatiquement

### Pourquoi la propreté du code est-elle importante ?

**Le forme c'est le fond qui remonte à la surface[[46]](#footnote-46) [[47]](#footnote-47)**

Le code, si vous n'y avez pas jeté un œil depuis quelques mois et que vous ne l'avez pas écrit proprement, il peut **être très long de se rappeler ce que fait quelle fonction**, et comment les choses s'organisent entre elles.

**Proverbe de développeur : codez toujours comme si la personne qui devra maintenir ce code était un psychopathe violent sachant où vous vivez.**

### Être une fonction ou ne pas être

Si vous vous trouvez à écrire plusieurs fois le même code, vous devriez probablement le **refactoriser dans une fonction**.

#### La refactorisation du code

**La refactorisation du code** consiste à modifier la structure d'un élément de code sans changer son comportement.

##### Code non factorisé versus code factorisé

Observons le code avant et après la refactorisation.

###### Code de départ non factorisé

Observons le code suivant :

**if (firstUser.online) {**

**if (firstUser.accountType === "normal") {**

**console.log("Hello " + firstUser.name + "!");**

**} else {**

**console.log("Welcome back premium user " + firstUser.name + "!");**

**}**

**}**

**if (secondUser.online) {**

**if (secondUser.accountType === "normal") {**

**console.log("Hello " + secondUser.name + "!");**

**} else {**

**console.log("Welcome back premium user " + secondUser.name + "!");**

**}**

**}**

**if (thirdUser.online) {**

**if (thirdUser.accountType === "normal") {**

**console.log("Hello " + thirdUser.name + "!");**

**} else {**

**console.log("Welcome back premium user " + thirdUser.name + "!");**

**}**

**}**

Nous répétons ici la même tâche plusieurs fois pour des utilisateurs différents.

C'est une **bonne occasion de factoriser votre code** dans une fonction[[48]](#footnote-48) :

###### Code factorisé sans la sortie console

**const sendWelcomeMessageToUser = (user) => {**

**if (user.online) {**

**if (user.accountType === "normal") {**

**console.log("Hello " + user.name + "!");**

**} else {**

**console.log("Welcome back premium user " + user.name + "!");**

**}**

**}**

**}**

**sendWelcomeMessageToUser(firstUser);**

**sendWelcomeMessageToUser(secondUser);**

**sendWelcomeMessageToUser(thirdUser);**

###### Code factorisé avec la sortie console

**function userOnlineType(user,online, accountType){**

**if (user){**

**if(online){**

**if(accountType === "normal"){**

**console.log("Hello Mister " + user + " !" );**

**}else{**

**console.log("Welcome back premium user " + user + " !" );**

**}**

**}else{**

**console.log("User is offline !" );**

**}**

**}else{**

**console.log("END !" );**

**// console.log("Welcome unknown !" );**

**}**

**}**

**userOnlineType('Person',true,'');**

Une image contenant texte

Description générée automatiquement

Nous avons créé une fonction **sendWelcomeMessageToUser** : un nom qui **exprime clairement ce que fait la fonction[[49]](#footnote-49)**.

Dans ce cas, nous avons **réduit la quantité de code** de 23 lignes à 13, tout en le rendant **plus lisible**.

Grâce à un nom de fonction descriptif, vous pouvez indiquer que nous envoyons un message de bienvenue à trois utilisateurs.

### Le principe DRY

**DRY** – Don't Repeat Yourself : Ne vous répétez pas.

C’est un principe de programmation qui **réduit au minimum la répétition du code**.

### Pratiquez le principe DRY

Ce code importe des séries depuis la base de données, en génère les informations pour les composants du site, et les met dans un tableau à exporter.

Cependant, pour l'instant, le même code est répété pour chaque série. L'endroit parfait pour une fonction ![[50]](#footnote-50)

* Créez une **fonction qui vous permet de refactoriser ce code**, afin de le raccourcir et de le rendre plus lisible.
* Appelez votre **nouvelle fonction pour chaque série** pour créer le composant correspondant, en passant les **arguments appropriés**.

class Show {

constructor(title, numberOfSeasons, episodesPerSeason) {

this.title = title;

this.numberOfSeasons = numberOfSeasons;

this.episodesPerSeason = episodesPerSeason;

}

}

const tau = new Show('The Story of Tau', 5, 12);

const meldrum = new Show('The Hero of Old Meldrum', 3, 6);

const clara = new Show('The Bugs of Isla Clara', 6, 15);

const shows = [tau, meldrum, clara];

// Modify the following code

// ======================

const tauTitleText = tau.title;

const tauSeasonsText = tau.numberOfSeasons + ' seasons';

const tauEpisodesText = tau.episodesPerSeason + ' episodes per season';

const tauComponent = {

titleText: tauTitleText,

seasonsText: tauSeasonsText,

episodesText: tauEpisodesText

};

const meldrumTitleText = meldrum.title;

const meldrumSeasonsText = meldrum.numberOfSeasons + ' seasons';

const meldrumEpisodesText = meldrum.episodesPerSeason + ' episodes per season';

const meldrumComponent = {

titleText: meldrumTitleText,

seasonsText: meldrumSeasonsText,

episodesText: meldrumEpisodesText

};

const claraTitleText = clara.title;

const claraSeasonsText = clara.numberOfSeasons + ' seasons';

const claraEpisodesText = clara.episodesPerSeason + ' episodes per season';

const claraComponent = {

titleText: claraTitleText,

seasonsText: claraSeasonsText,

episodesText: claraEpisodesText

};

const showComponents = [tauComponent, meldrumComponent, claraComponent];

// ======================

// Modify the code above

#### Solution

class Show {

constructor(title, numberOfSeasons, episodesPerSeason) {

this.title = title;

this.numberOfSeasons = numberOfSeasons;

this.episodesPerSeason = episodesPerSeason;

}

}

const tau = new Show('The Story of Tau', 5, 12);

const meldrum = new Show('The Hero of Old Meldrum', 3, 6);

const clara = new Show('The Bugs of Isla Clara', 6, 15);

const shows = [tau, meldrum, clara];

// Modify the following code

// ======================

const generateComponent = (show) => {

const titleText = show.title;

const seasonsText = show.numberOfSeasons + ' seasons';

const episodesText = show.episodesPerSeason + ' episodes per season';

return {

titleText,

seasonsText,

episodesText

};

}

const tauComponent = generateComponent(tau);

const meldrumComponent = generateComponent(meldrum);

const claraComponent = generateComponent(clara);

const showComponents = [tauComponent, meldrumComponent, claraComponent];

#### Quelqu'un sait-il ce que fait ce truc ?

Voyons la fonction suivante :

const printStringStats = (stringToTest) => {

const wordArray = stringToTest.split(" ");

const wordCount = wordArray.length;

let letterCount = 0;

for (let word of wordArray) {

word.replace(/[.,\/#!$%\^&\\*;:{}=\-\_`~()]/g, "");

letterCount += word.length;

}

const averageWordLength = parseFloat((letterCount / wordCount).toFixed(2));

const stringStats = {

wordCount: wordCount,

letterCount: letterCount,

averageWordLength: averageWordLength

};

console.log(stringStats);

}

Bien qu'il y ait un nom descriptif, et que vous puissiez voir qu'il s'agit d'imprimer le nombre de mots, le nombre de lettres et la longueur moyenne des mots d'une chaîne.

Il se passe tellement de choses qu'il est difficile de suivre si vous ne comprenez pas chaque ligne.

Cette fonction fait trop de choses d'un coup !

Voyons maintenant cette refactorisation :

**const getWordCount = (stringToTest) => {**

**const wordArray = stringToTest.split(" ");**

**return wordArray.length;**

**}**

**const getLetterCount = (stringToTest) => {**

**const wordArray = stringToTest.split(" ");**

**let totalLetters = 0;**

**for (let word of wordArray) {**

**word.replace(/[.,\/#!$%\^&\\*;:{}=\-\_`~()]/g, "");**

**totalLetters += word.length;**

**}**

**return totalLetters;**

**}**

**const getAverageWordLength = (stringToTest) => {**

**return parseFloat((getLetterCount(stringToTest) / getWordCount**

**}**

**const printStringStats = (stringToTest) => {**

**console.log({**

**wordCount: getWordCount(stringToTest),**

**letterCount: getLetterCount(stringToTest),**

**averageWordLength: getAverageWordLength(stringToTest)**

**})**

**}**

Bien que cette version soit plus longue, il y a trois fonctions :

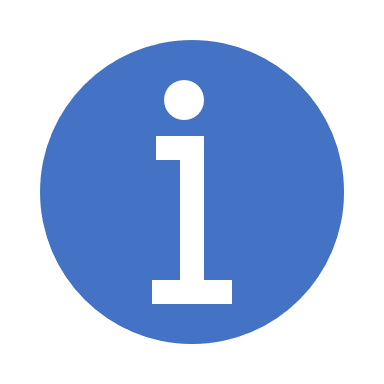
**getWordCount** (qui renvoie le nombre de mots),

**getLetterCount** (qui renvoie le nombre de lettres) et

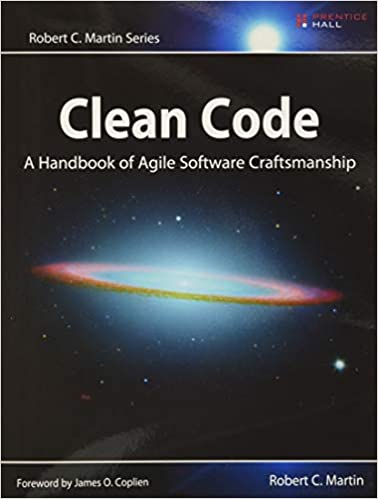
**getAverageWordLength** (qui renvoie la longueur moyenne des mots)

Ces trois fonctions appelées par la fonction **printStringStats** (qui imprime les statistiques de la chaîne de caractères).

Vous pouvez ne pas comprendre la complexité de chaque fonction, mais vous pouvez bien comprendre ce qui se passe parce qu'elles **ne font qu'une seule chose**, **mentionnée** **dans leur nom**.

*"La première règle des fonctions est qu'elles devraient être petites. La deuxième règle des fonctions est qu'elles devraient être encore plus petites.*

*Robert C. Martin, Clean Code A Handbook of Agile Software Craftsmanship*



### Laissez des commentaires

**Ne commentez pas chaque ligne ou fonction.** Les commentaires doivent clarifier ce qui n'est pas immédiatement apparent à la lecture du code.

Ajoutons des commentaires à notre exemple précédent :

**const getWordCount = (stringToTest) => {**

**const wordArray = stringToTest.split(" ");**

**return wordArray.length;**

**}**

**const getLetterCount = (stringToTest) => {**

**const wordArray = stringToTest.split(" ");**

**let totalLetters = 0;**

**for (let word of wordArray) {**

**// retire la ponctuation pour ne compter que les lettres**

**word.replace(/[.,\/#!$%\^&\\*;:{}=\-\_`~()]/g, "");**

**totalLetters += word.length;**

**}**

**return totalLetters;**

**}**

**/\*renvoie la longueur moyenne des mots**

**et arrondie à deux chiffres après la virgule\*/**

**const getAverageWordLength = (stringToTest) => {**

**return parseFloat((getLetterCount(stringToTest) / getWordCount(stringToTest)).toFixed(2));**

**}**

**const printStringStats = (stringToTest) => {**

**console.log({**

**wordCount: getWordCount(stringToTest),**

**letterCount: getLetterCount(stringToTest),**

**averageWordLength: getAverageWordLength(stringToTest)**

**})**

**}**

Comme vous pouvez le voir, toutes les fonctions ou lignes n'ont pas de commentaire :

* le commentaire sur une seule ligne explique la raison de l'appel, d'apparence complexe, à **replace** ;
* le commentaire sur plusieurs lignes explique que la longueur moyenne du mot sera renvoyée à 2 décimales près, car ce n'est pas immédiatement apparent.

*Laisser juste le nombre correct de commentaires est une forme d'art !*

**Assurez-vous donc de commenter votre code !**

### Écrivez du code avec style

Adopter un style **cohérent** est nécessaire pour que le code reste propre, et bien qu'il n'y ait pas de **bonne façon** absolue de faire les choses.

►Il y a un certain nombre de **conventions** qui peuvent vous faciliter la vie.

#### Qu'y a-t-il dans un nom ?

##### Utilisez les conventions de nommage

►Les noms de **variable et de fonction** dans JavaScript sont écrits en **camel case :**

**getWordCount** , **numberOfCats.**

Les noms de **classe** sont écrits en **Pascal case :**

**PremiumAccount , SpecialGuest**

###### Choisissez un nom

Des noms **clairs et descriptifs** pour les variables et les fonctions sont plus faciles à lire que des noms raccourcis, obscurs ou aléatoires.

Ainsi, **vous n'aurez plus à écrire autant de commentaires !**

#### Mettez en forme votre code

**Mise en retrait** → tabulation, en gardant le même style partout.

**Espacement** → aérer le code

**Positionnement des accolades :**

**Une image contenant texte

Description générée automatiquement**

**Pour écrire votre propre code, choisissez le style que vous préférez et tenez-vous-y !**

### Pratiquez l’écriture de code propre

Voici un code js avec plusieurs fonctions et différentes mauvaises pratiques rendant le code pas propre. Votre objectif est de créer **une fonction pour alléger votre code**.

//-----CODE NON DRY------

//variable de différentes personnes

let personne1 = "Jean";

let personne2 = "Paul";

let person3 = "Marcel";

//On met la première lettre en majuscule, on salue la première personne et on donne le nombre de lettre dans son prénom

personne1 = personne1[0].toUpperCase() + personne1.substr(1);

const longueurPrenom1 = personne1.length;

console.log(`Bonjour ${personne1}, ton prénom contient ${longueurPrenom1} lettres`);

//On met la première lettre en majuscule, on salue la deuxième personne et on donne le nombre de lettre dans son prénom

personne2 = personne2[1].toUpperCase() + personne2.substr(1);

const longueurPrenom2 = personne2.length;

console.log(`Bonjour ${personne2}, ton prénom contient ${longueurPrenom2} lettres`);

//On met la première lettre en majuscule, on salue la troisième personne et on donne le nombre de lettre dans son prénom

personne3 = personne3[2].toUpperCase() + personne3.substr(1);

const longueurPrenom3 = personne3.length;

console.log(`Bonjour ${personne3}, ton prénom contient ${longueurPrenom3} lettres`);

//-----CODE MAL NOMMÉ------

//tableau des ages des élèves dans la classe

const lrf = [14, 14, 15, 14, 16, 14, 14, 13];

// Nombre d'élèves

const kf = lrf.length;

// variable pour calculer la somme des ages

let mf = 0;

for(let df of lrf){

mf += df;

}

//moyenne d'age dans la classe

const mld = mf / kf;

console.log('Il y a ' + kf + " élèves dans la classe et la moyenne d'age est " + mld);

//-----CODE MAL MIS EN FORME------

const temperature = 25;

if(temperature < 10){ console.log("Il fait très froid"); }

else if(temperature < 0){

console.log(

"Il fait froid"

);

}else if(temperature < 10){

console.log( "Il fait frais");

}else if(temperature < 20){

console.log("Il fait doux");

}else if(temperature < 30){

console.log("Il fait bon");

}else{

console.log("Il fait chaud");

}

#### Correction expliquée de l’exercice

##### Code non DRY

###### Code avant correction :

//variable de différentes personnes

let personne1 = "Jean";

let personne2 = "Paul";

let person3 = "Marcel";

//On met la première lettre en majuscule, on salue la première personne et on donne le nombre de lettre dans son prénom

personne1 = personne1[0].toUpperCase() + personne1.substr(1);

const longueurPrenom1 = personne1.length;

console.log(`Bonjour ${personne1}, ton prénom contient ${longueurPrenom1} lettres`);

//On met la première lettre en majuscule, on salue la deuxième personne et on donne le nombre de lettre dans son prénom

personne2 = personne2[1].toUpperCase() + personne2.substr(1);

const longueurPrenom2 = personne2.length;

console.log(`Bonjour ${personne2}, ton prénom contient ${longueurPrenom2} lettres`);

//On met la première lettre en majuscule, on salue la troisième personne et on donne le nombre de lettre dans son prénom

personne3 = personne3[2].toUpperCase() + personne3.substr(1);

const longueurPrenom3 = personne3.length;

console.log(`Bonjour ${personne3}, ton prénom contient ${longueurPrenom3} lettres`);

###### Code après correction :

//variable de différentes personnes

let personne1 = "Jean";

let personne2 = "Paul";

let personne3 = "Marcel";

function saluer(prenom){

//On met la première lettre en majuscule, on salue la personne et on donne le nombre de lettre dans son prénom

const monPrenom = prenom[0].toUpperCase() + prenom.substr(1);

const longueurPrenom = monPrenom.length;

console.log(`Bonjour ${monPrenom}, ton prénom contient ${longueurPrenom} lettres`);

}

//On salue les 3 personnes

saluer(personne1);

saluer(personne2);

saluer(personne3);

###### Explication :

On crée une fonction unique saluer(prenom) qui va faire ce traitement basé sur la variable prenom . Il ne reste plus qu’à appeler la fonction 3 fois avec les 3 personnes en paramètre.

##### Code mal nommé

###### Code avant correction :

//tableau des âges des élèves dans la classe

const lrf = [14, 14, 15, 14, 16, 14, 14, 13];

// Nombre d'élèves

const kf = lrf.length;

// variable pour calculer la somme des âges

let mf = 0;

for(let df of lrf){

mf += df;

}

//moyenne d'âge dans la classe

const mld = mf / kf;

console.log('Il y a ' + kf + " élèves dans la classe et la moyenne d'âge est " + mld);

###### Code après correction :

//tableau des âges des élèves dans la classe

const agesElevesDeClasse = [14, 14, 15, 14, 16, 14, 14, 13];

// Nombre d'élèves

const nombreEleves = agesElevesDeClasse.length;

// variable pour calculer la somme des âges

let sommeAges = 0;

for(let age of agesElevesDeClasse){

sommeAges += age;

}

//moyenne d'âge dans la classe

const moyenneAgesDeClasse = sommeAges / nombreEleves;

console.log('Il y a ' + nombreEleves + " élèves dans la classe et la moyenne d'âge est " + moyenneAgesDeClasse);

###### Explication :

Après correction on peut voir que l’on **nomme les variables avec des noms explicites**. Maintenant, juste en lisant les variables on comprend la logique du code.

##### Code mal mis en forme

###### Code avant correction :

const temperature = 25;

if(temperature < 10){ console.log("Il fait très froid"); }

else if(temperature < 0){

console.log(

"Il fait froid"

);

}else if(temperature < 10){

console.log( "Il fait frais");

}else if(temperature < 20){

console.log("Il fait doux");

}else if(temperature < 30){

console.log("Il fait bon");

}else{

console.log("Il fait chaud");

}

###### Code après correction :

const temperature = 25;

if(temperature < 10){

console.log("Il fait très froid"); }

else if(temperature < 0){

console.log("Il fait froid");

}else if(temperature < 10){

console.log("Il fait frais");

}else if(temperature < 20){

console.log("Il fait doux");

}else if(temperature < 30){

console.log("Il fait bon");

}else{

console.log("Il fait chaud");

}

###### Explication :

Ici on a appliqué une bonne **indentation**, on a bien mis en forme les sauts de ligne pour les blocs et les ***brackets*** ({}) et on a évité les espaces et sauts de ligne superflus.

### En résumé

Dans ce chapitre, nous avons étudié quelques techniques et **meilleures pratiques** de **codage propre** :

* **refactorisation** **du code** en fonctions – **quand du code se répète**, ou quand une **fonction fait trop de choses** ;
* les fonctions qui ne font qu'**une seule chose** sont souvent plus **claires** que les fonctions à plusieurs usages ;
* laisser des **commentaires appropriés** rend votre code beaucoup plus facile à comprendre ;
* il est très important d'avoir une **convention de nommage** stricte avec des **noms clairs et descriptifs** ;
* vous avez vu des exemples de **mise en forme du code**, et différents choix possibles.

*Dans le chapitre suivant, nous allons nous intéresser à la façon de tester les fonctions pour éviter les surprises !*

## Tester qu’une fonction fait ce qu’elle dit

### Comprenez l'importance des tests

TL;DR – Le test est vraiment important !

**Tests => trois types essentiels. → freeplane**

### Qu'est-ce que le test?- **les tests unitaires**

Test unitaire :

**Le test unitaire** vérifie des unités individuelles (en général des fonctions uniques ou des classes) en leur fournissant une entrée et en s'assurant qu'elles donnent la sortie attendue.

C'est une autre raison pour écrire des fonctions courtes qui n'ont qu'un seul usage : il est plus facile d'écrire des tests pour elles !

En général, chaque unité est testée sur un **cas simple**, puis sur un ou plusieurs **cas limites.**

Si par exemple, vous prenez quelques fonctions du chapitre précédent :

Quels cas testeriez-vous pour chaque fonction ici ?

* getWordCount – Vous pourriez vérifier une chaîne dont vous connaissez le nombre de mots (**cas simple**), puis peut-être une **chaîne vide**, et une chaîne qui ne contiendrait **que des espaces** (**cas limite).**
* getLetterCount – Vous pourriez vérifier une chaîne dont vous connaissez le **nombre de lettres (cas simple),** puis essayer une chaîne ne contenant **que des signes de ponctuation (cas limite)**.

Vous pourriez écrire ces tests comme code accessoire :

//-------------UNIT TEST-----------------

const testSimpleWordCount = () => {

const testString = 'I have four words';

if (getWordCount(testString) !== 4) {

console.error('Simple getWordCount failed!');

}

}

const testEdgeWordCount = () => {

const testString = ' ';

if (getWordCount(testString) !== 0) {

console.error('Edge getWordCount failed!');

}

}

const testSimpleLetterCount = () => {

const testString = 'I have twenty one letters!';

if (getLetterCount(testString) !== 21) {

console.error('Simple getLetterCount failed!');

}

}

const testEdgeLetterCount = () => {

const testString = '")(&;//!!';

if (getLetterCount(testString) !== 0) {

console.error('Edge getLetterCount failed!');

}

}

Ce sont des tests simples, et ils peuvent convenir pour des vérifications rapides, mais il est généralement préférable d'utiliser une **architecture de test**.

**Les architectures et bibliothèques de test** permettent d'écrire automatiquement des suites de tests complètes de votre code, à l'aide de fonctions et de syntaxe spécifiques.

Voici à quoi pourraient ressembler les deux tests ci-dessus dans certaines architectures :

describe('getWordCount()', function() {

it('should find four words', function() {

expect(getWordCount('I have four words!').to.equal(4));

});

it('should find no words', function() {

expect(getWordCount(' ').to.equal(0));

});

});

Les tests unitaires constituent généralement entre 60 et 80 % de l'ensemble des tests des projets JavaScript.

### Découvrez les tests d'intégration



Image : Pas de tests d'intégration

**Les tests d'intégration** vérifient les multiples fonctions ou classes pour s'assurer qu'elles travaillent ensemble comme elles sont censées le faire.

L'image ci-dessus montre ce qui se passe quand les unités individuelles fonctionnent correctement : les deux tiroirs s'ouvrent correctement séparément ;

Mais que leur intégration dans le système qui les entoure cause un problème de fonctionnement.

### Appréhendez les tests fonctionnels

**Les tests fonctionnels**, aussi appelés de bout en bout (E2E)[[51]](#footnote-51), vérifient des scénarios complets en contexte.

Par exemple, un utilisateur se connecte à votre application, ouvre ses notifications et les marque toutes comme lues. Ces tests vérifient aussi les ressources externes que votre projet peut utiliser, par exemple un système de paiement tiers.

### Les tests: comment ça se passe dans la pratique ?

**Pratiquer les tests unitaires demande un peu d'expérience** et d’utiliser des outils dédiés. Il existe différents ***frameworks*** dédiés aux tests unitaires et fonctionnels pour les applications JavaScript frontend et/ou backend.

On retrouve souvent les frameworks :

[Jasmine](https://jasmine.github.io/)[[52]](#footnote-52), [Mocha.js](https://mochajs.org/)[[53]](#footnote-53), [Karma](https://karma-runner.github.io/6.3/index.html)[[54]](#footnote-54), etc.

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Pour utiliser ces outils, il est souvent nécessaire de **mettre en place une configuration particulière et un système de compilation de votre code**.

**Il vous est donc nécessaire d’appréhender plus largement JavaScript avant de pratiquer les tests.**

►Une fois que vous aurez une **bonne compréhension du JavaScript** et un peu **d'expérience**, vous pourrez consulter le cours : [Testez l’interface de votre site](https://openclassrooms.com/fr/courses/3504461-testez-linterface-de-votre-site)[[55]](#footnote-55).

Une image contenant texte

Description générée automatiquement

### **Quid les tests de non régression ?**

### En résumé

**Trois types de tests :**

1. les tests unitaires ;
2. les tests d'intégration ;
3. les tests fonctionnels (E2E).

*Nous allons voir comment corriger du code quand tout va mal : le* ***débogage*** *!*

## Déboguer une fonctionUne image contenant texte, équipement électronique Description générée automatiquement



### Affichez la console

Observons une version défectueuse d'une fonction d'un chapitre précédent .

Pour une raison quelconque, cette fonction renvoie des valeurs curieuses.

►Utilisons un **affichage de console** pour voir ce qui se passe :

**const getWordCount = (stringToTest) => {**

**const wordArray = stringToTest.split('');**

**/\*----------------DEBUG------------\*/**

**console.log("Word array in getWordCount: ");**

**console.log(wordArray);**

**/\*----------------/DEBUG------------\*/**

**return wordArray.length;**

**}**

**getWordCount("I am a fish");**

Une image contenant texte

Description générée automatiquement

#### Correction du bug

Plutôt que de répartir la chaîne en mots, elle les a répartis en lettres !

Une observation plus attentive de la fonction montre une erreur à l'appel de **split()** : ce devrait être **stringToTest.split(' ')** , et non pas **stringToTest.split('')** :

**const getWordCount = (stringToTest) => {**

**const wordArray = stringToTest.split(' ');**

**/\*----------------DEBUG------------\*/**

**console.log("Word array in getWordCount: ");**

**console.log(wordArray);**

**/\*----------------/DEBUG------------\*/**

**return wordArray.length;**

**}**

**getWordCount("I am a fish");**

Une image contenant texte

Description générée automatiquement

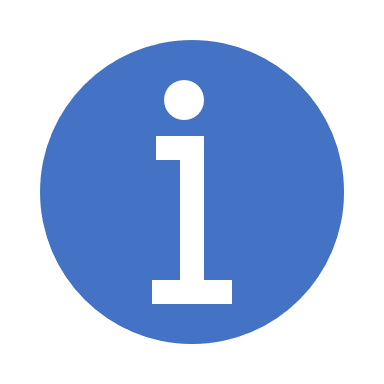
L'utilisation de la console convient bien dans les cas simples et isolés comme celui-ci, mais dans les projets plus complexes, ce serait beaucoup plus difficile et plus **chronophage**.

Dans ce cas, il faut des armes plus puissantes.

### Utilisez des outils pour développeur

Pour écrire du JavaScript pour des sites web, vous pouvez utiliser les outils pour développeur intégrés dans les quatre navigateurs: Chrome, Firefox, Safari et Edge.

La plupart des environnements de développement intégrés (EDI) comportent aussi un débogueur. Notamment [Visual Studio Code](https://code.visualstudio.com/)[[56]](#footnote-56) (gratuit) ou [WebStorm](https://www.jetbrains.com/fr-fr/webstorm/)[[57]](#footnote-57).

Si vous voulez en savoir plus sur Visual Studio Code, vous pouvez suivre ce cours, "[Adoptez Visual Studio Code comme environnement de développement](https://openclassrooms.com/fr/courses/5641796-adoptez-visual-studio-comme-environnement-de-developpement)[[58]](#footnote-58)".

Une image contenant texte

Description générée automatiquement

*L'affichage de console, c'est bien, les débogueurs, c'est mieux, mais quand tout le reste a échoué, il existe une dernière solution.*

### Déboguez avec un canard en plastique



Image :Quand tout va mal, parle au canard ☺

Quand votre code a un **bug** que vous ne pouvez pas trouver et expliquer, vous pouvez parler à un **canard en plastique** que vous conservez sur votre bureau.

**Vous expliquez votre code ligne par ligne en termes simples**, que le canard peut comprendre.

►**La réflexion à voix haute et l'explication de votre code en termes simples** vous permettra souvent de voir finalement : **"Comment n'ai-je pas vu ça ? Il y a un bug !".**

► **Le raisonnement** sur votre code, à voix haute et en termes simples, peut être utile pour le **débogage et la factorisation** !

### Pratiquez les console.log

L’objectif est de comprendre quelle est **l’erreur de logique qui pose souci.**

**Le code a pour objectif de convertir des années en mois**

(dans une année, il y a 12 mois). Si on saisit 10 ans dans le champ Année et qu’on clique sur Convertir, on se retrouve avec une phrase nous indiquant que dans 10 années il y a 80 mois, ce qui est faux car la bonne réponse est 120 mois.

* Le calcul des mois se fait à la **ligne 25**, et on souhaite comprendre ce qui ne va pas.
* Votre mission ici est de faire des **console.log au niveau des lignes 27 à 31** pour **analyser** les variables utilisées pour le calcul de la ligne 25.
* Suite à l'analyse, vous pourrez comprendre ce qui ne va pas et **effectuer la correction** nécessaire.
* [CodePen](https://codepen.io/quentin-laurent/pen/gOLEmey)[[59]](#footnote-59)

#### Correction

À la ligne 28, nous ajoutons le code suivant :

console.log('annee', annee);

console.log('params', params);

Ce qui donnera dans la console, si on simule une conversion pour la valeur 10 :

> annee 10

> params (20) [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

On en déduit que la valeur de annee est cohérente. En ce qui concerne params , on observe que c’est un tableau de valeur allant de 20 à 1. Dans notre calcul, on souhaite récupérer la valeur du tableau qui vaut 12. On trouve cette valeur à l’index 8.

Si on reprend le calcul de mois :

const mois = annee \* params[12];

On constate que l’on récupère l’index 12 de params , et non le 8. Donc on remplace la ligne 25 par :

**const mois = annee \* params[8];**

Maintenant notre application donne des valeurs cohérentes.

**Il ne faut pas avoir peur de faire des console.log pour avoir une vision plus large de la logique**,

Mais attention à ne pas oublier de supprimer ces lignes de votre code avant une mise en ligne !

### En résumé

Trois techniques de débogage différentes :

1. **affichage de console** – parfait pour des erreurs petites, simples et isolées, mais pas idéal dans des projets plus grands et plus complexes ;
2. **utilisation d'outils pour développeur** – qu'il s'agisse des outils intégrés dans la plupart des navigateurs, ou du **débogueur** de **votre environnement de développement intégré, c'est à coup sûr la façon fiable de déboguer du code** ;
3. débogage avec un canard en plastique – quand les choses vont terriblement mal.

*Nous allons nous intéresser à un type de fonction qui, utilisé correctement, peut améliorer spectaculairement les* ***performances*** *d'un projet.*

# La récursivité

## Découvrir la récursivité : l’appel de fonctions à l’intérieur d’elles-mêmes

Une image contenant texte

Description générée automatiquement

**►Algorithme de recherche binaire[[60]](#footnote-60)**

### Une fonction qui s'appelle elle-même

#### La fonction récursive

Une fonction récursive est une fonction qui s'appelle elle-même d'une façon ou d'une autre.

Exemple de la recherche binaire :

Le but de l'exercice : chercher un élément dans un array **trié[[61]](#footnote-61)** pour savoir s'il s'y trouve.

Une approche basique (et plutôt lente) serait la suivante :

const findElement = (array, thingToFind) {

for (let element of array) {

if (element === thingToFind) {

return true;

}

}

return false;

}

On avance, élément par élément, dans le tableau. Si on trouve un élément qui correspond à ce que l'on recherche, la fonction renvoie true . Si on arrive à la fin du tableau sans l'avoir trouvé, on passe à la ligne suivante et la fonction renvoie false .

C'est plutôt clair comme approche, mais c'est lent ! Le temps pris pour chaque recherche se prolonge de **manière linéaire[[62]](#footnote-62)** avec des listes plus longues !

Imaginons une autre approche :

const binarySearch = (array, thingToFind, start, end) => {

let mid = Math.floor((start + end) / 2);

if (array[mid] === thingToFind) {

return true;

}

if (thingToFind < array[mid]) {

// il faut rechercher dans la première moitié

return binarySearch(array, thingToFind, start, mid - 1); // on utilise (mid - 1) car on sait que l'on n'a pas besoin de l'élément mid, il a déjà été vérifié !

} else {

// il faut rechercher dans la deuxième moitié

return binarySearch(array, thingToFind, mid + 1, end);

}

}

►On sait que le **tableau est trié[[63]](#footnote-63)**, donc[[64]](#footnote-64) on peut savoir, pour un élément donné, si ce que l'on recherche risque de se trouver plus haut ou plus bas dans la liste.

Par exemple, si on recherche le nombre 42 et que l'on tombe sur 32, on sait qu'il faudra chercher plus bas.

►Du coup, commençons par analyser **l'élément médian[[65]](#footnote-65)** de la liste.

On peut faire la somme de **l'index de début** et de **l'index de fin**, et diviser par deux pour trouver cet élément et arrondissons vers le bas pour nous assurer de trouver un nombre entier :

**let mid = Math.floor((start + end) / 2);**

**Mais pourquoi utiliser les index de début et de fin plutôt que la propriété length du tableau ?**

►Utiliser les index nous permettra de **réutiliser le même code** sur des sélections de plus en plus petites du tableau.

►Maintenant que l'on a l'élément médian du tableau, vérifions si, par chance, on est tombé juste :

**const binarySearch = (array, thingToFind, start, end) => {**

**let mid = Math.floor((start + end) / 2);**

**if (array[mid] === thingToFind) {**

**return true;**

**}**

**}**

La fonction retournera **true** si on a trouvé l'élément.

* Si on n'a pas eu de chance, ce n'est pas grave : puisque le tableau est trié, on sait dans quelle moitié du tableau chercher !
* Donc, on a juste à exécuter exactement la même fonction sur la partie en question !
* Il suffit de modifier soit
  + l'index de fin (pour chercher dans la première moitié) soit
  + l'index de début (pour chercher dans la deuxième moitié) :

**const binarySearch = (array, thingToFind, start, end) => {**

**let mid = Math.floor((start + end) / 2);**

**if (array[mid] === thingToFind) {**

**return true;**

**}**

**if (thingToFind < array[mid]) {**

**// il faut rechercher dans la première moitié**

**return binarySearch(array, thingToFind, start, mid - 1); // on utilise (mid - 1) car on sait que l'on n'a pas besoin de l'élément mid, il a déjà été vérifié !**

**} else {**

**// il faut rechercher dans la deuxième moitié**

**return binarySearch(array, thingToFind, mid + 1, end);**

**}**

**}**

#### Cas de base, ou base case

La fonction continuera à s'appeler elle-même jusqu'à trouver ce que l'on recherche.

Mais il manque quelque chose !

**Qu'est-ce qui se passe si ce que l'on recherche n'existe pas dans le tableau ?**

Il faut ce que l'on appelle un cas de base, ou **base case**, pour **dire à la fonction de s'arrêter.**

##### Quel est le base case dans cet algorithme ?

On saura que l'algorithme est arrivé au bout si on a essayé de l'appeler avec un index de *début qui est supérieur à l'index de fin*.

Pourquoi ?

Eh bien parce que peu à peu, on divise le tableau, encore et encore, jusqu'à tomber sur une **sélection d'un seul élément** : on aura donc **start = mid = end** .

Du coup, quand la fonction se rappellera encore, elle utilisera soit

* **start = mid + 1** , soit
* **end = mid - 1** , selon notre recherche.

**On aura donc start > end ,** et la fonction peut retourner **false** , **car on sait qu'elle est arrivée au bout sans trouver ce que l'on recherche.**

On met donc ce ***base case*** : **start > end** au début de la fonction pour vérifier s'il s'agit du dernier appel :

**const binarySearch = (array, thingToFind, start, end) => {**

**//Base case**

**if(start > end){**

**return false;**

**}**

**// /Base case**

**let mid = Math.floor((start + end) / 2);**

**if (array[mid] === thingToFind) {**

**return true;**

**}**

**if (thingToFind < array[mid]) {**

**// il faut rechercher dans la première moitié**

**return binarySearch(array, thingToFind, start, mid - 1); // on utilise (mid - 1) car on sait que l'on n'a pas besoin de l'élément mid, il a déjà été vérifié !**

**} else {**

**// il faut rechercher dans la seconde moitié**

**return binarySearch(array, thingToFind, mid + 1, end);**

**}**

**}**

**let myArray = ['ab', 'cd','ef'];**

**let toFind = 'ef';**

**console.log(binarySearch(myArray,toFind,0,2));**

Une image contenant texte

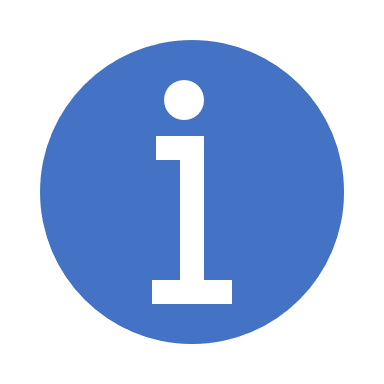
Description générée automatiquement

Et voilà ! Une fonction **récursive**, qui s'appelle elle-même, qui effectue une recherche d'élément dans un **tableau trié[[66]](#footnote-66)**, et qui renvoie **true** si l'élément s'y trouve, ou **false** s'il ne s'y trouve pas, grâce au **base case** !

**ATTENTION** ! Sans base case (ou avec un base case incorrect), vos fonctions récursives peuvent causer des **infinite loops** et des **stack** **overflows[[67]](#footnote-67)**, car elles vont continuer à s'appeler à l'infini, donc faites bien attention !

**WARNING** *! Without a* **base case** *(or with an incorrect base case), your recursive functions can cause infinite loops and* **stack overflows***, because they will continue to call each other ad infinitum, so be careful!*

**Cet algorithme s'appelle la recherche binaire, et il s'agit d'un exercice qui est souvent demandé en entretien d'embauche.**

N'hésitez pas également à explorer d'autres exemples courants [d'algorithme récursif](https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif)[[68]](#footnote-68) :

***merge sort, index sort, tree traversal***… il y en a énormément !

### Pratiquez la récursivité

Quand on parle de récursivité, il y a une fonction qui fait vraiment partie des **grands classiques** : la fonction mathématique “***factorielle***” !

Concrètement, la factorielle d'un nombre n est définie comme n fois la factorielle du nombre ( ***n-1)*** , et la ***factorielle de 1 est 1*** .

**Voici la décomposition de la factorielle de 3 pour mieux comprendre :**

**factorielle(3) = 3 \* factorielle(3 - 1)**

**factorielle(3) = 3 \* factorielle(2) = 3 \* ( 2 \* factorielle(2 - 1) )**

**factorielle(3) = 3 \* factorielle(2) = 3 \* ( 2 \* factorielle(1) ) = 3 \* 2 \* 1 = 6**

**⬄ factorielle(n) = n \* factorielle(n-1)**

**C’est la notation récursive de la factorielle.**

Quand on regarde la décomposition de factorielle(3), on constate ***qu’on appelle de nouveau la factorielle avec la valeur n-1, donc on est bien dans de la récursivité***.

#### Exercice factorielle récursive

##### Enoncé

Rendez-vous sur le [CodePen](https://codepen.io/quentin-laurent/pen/QWGRKMo)[[69]](#footnote-69). Votre mission est de coder la fonction factorielle ligne 13 avec en paramètre ***number*** .

**Point important** : on considère que la factorielle fonctionne pour les nombres supérieurs à 1, sinon la factorielle vaudra 1.

##### Solutions

###### Version en locale

function factorielle(number){

if(number<=1){

return 1;

}else{

return number\*(factorielle(number-1)) ;

}

}

console.log(factorielle(5));

Une image contenant texte

Description générée automatiquement

###### Version en ligne

**HTML :**

<h1>Caclculer la factorielle</h1>

<input id="number" type="text" placeholder="entrez nombre">

<input id="bouton" type="button" value="Calculer">

<h2 id="message"></h2>

**CSS :**

body{

text-align:center;

}

**JavaScript :**

//On pointe sur l'élément de message

const espaceMessage = document.getElementById("message");

//On pointe sur l'élément de bouton

const bouton = document.getElementById("bouton");

const numberInput = document.getElementById("number");

let number;

//======FONCTION À ECRIRE======

function factorielle(number){

if(number <= 1) return 1;

else return (number \* factorielle(number-1));

}

//======/FONCTION À ECRIRE======

function onCalcul(){

//On récupère la saisie de l'année et on transforme le texte en nombre entier

number = parseInt(numberInput.value);

//Si la saisie n'est pas un nombre, on affiche un message d'erreur

if(isNaN(number)){

alert("Ceci n'est pas un nombre");

return;

}

espaceMessage.innerHTML = "La factorielle de " + number + " est " + factorielle(number);

}

//On écoute l'action de click sur le onConvert et on appelle la fonction onConvert

bouton.addEventListener('click', onCalcul);

Une image contenant texte

Description générée automatiquement Une image contenant texte

Description générée automatiquement

**CodePen :** [**https://codepen.io/quentin-laurent/pen/oNYVwbb**](https://codepen.io/quentin-laurent/pen/oNYVwbb)

* **à implémenter !**

### En résumé

* les **fonctions récursives** sont des fonctions qui s'appellent elles-mêmes ;
* une fonction récursive a besoin d'un **cas de base**, ou base case, pour qu'elle puisse savoir quand son travail est terminé ;

## Résumé de la partie

* les **fonctions** et leurs entrées (paramètres et arguments) ainsi que leurs sorties (**valeur de retour**) ;
* les **méthodes d'instance,** et la puissance supplémentaire qu'elles peuvent ajouter à vos classes ;
* écrire du **code propre** pour rendre vos applis plus faciles à comprendre et à maintenir
* un **aperçu** des tests pour vous assurer que votre code fonctionne, et des techniques de débogage quand ce n'est pas le cas ;
* exploré les **fonctions récursives**, qui s'appellent elles-mêmes pour travailler sur des structures de données arborescentes.

## QUIZ 3

# POO en JavaScript

<https://www.freecodecamp.org/news/javascript-classes-how-they-work-with-use-case/>

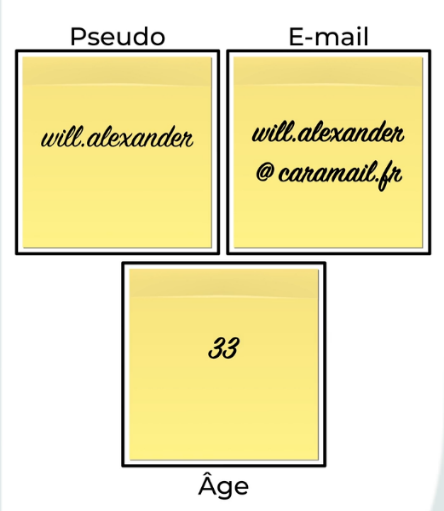
## Définissez des objets et leurs attributs avec des classes

Vous remarquez des **points communs** entre différents objets et vous notez ces informations pour construire une **représentation mentale d'une catégorie d'objets**.

Cette **liste mentale d'attributs** sert de **modèle** **pour cet objet**. En programmation, on l'appelle une **classe.**

Pour construire une classe, vous pouvez choisir le nom de votre choix. C'est pour cela qu'on l'appelle un **type nommé**. Vous le verrez, les classes permettent aussi de regrouper beaucoup de détails ; c'est pourquoi elles s'appellent aussi des **types complexes**.

Une image contenant texte

Description générée automatiquement  Une image contenant texte

Description générée automatiquement

### Découvrez les objets

Les objets JavaScript sont écrits en **JSON** (JavaScript Object Notation).

* Ce sont des séries de **paires clés/valeurs**
* séparées par des virgules,
* entre des accolades.

Les objets peuvent être enregistrés dans une variable :

Une image contenant texte

Description générée automatiquement

Chaque **clé** (« nom » « alias ») **est une chaîne** : un **string**.

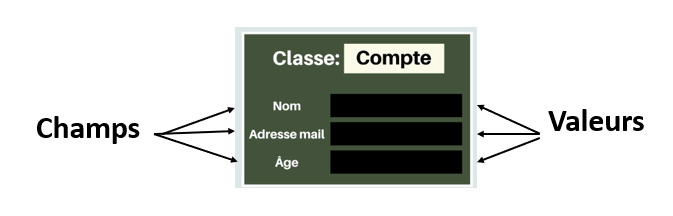
Et les **valeurs** associées peuvent avoir **tout type de données.**

Construire des objets présente un avantage essentiel : cela permet de **regrouper les attributs** d'une chose unique à un même emplacement.

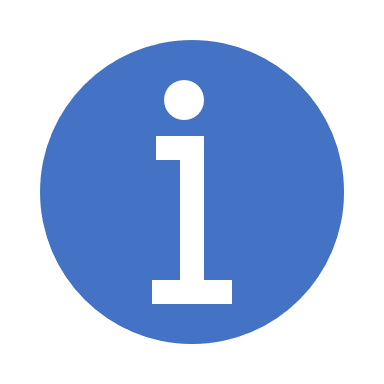
### La notion de classe

Une image contenant texte

Description générée automatiquement



**►La** [**classe**](https://fr.wikipedia.org/wiki/Classe_(informatique)) **(ou** [**class**](https://en.wikipedia.org/wiki/Class_(computer_programming))**)**  contient des ***champs*** qui correspondent aux ***clés*** de l’objet final

Le nom d’une **C**lasse doit commencer par une majuscule.

À partir de l’édition de 2015 (ECMAScript 6), une [syntaxe de définition de classes](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes) a été ajoutée, simplifiant l’utilisation de son mécanisme d’héritage prototypal pour le développement orienté objet.

#### Exemple de classe

[class Point](https://fr.wikipedia.org/wiki/Classe_(informatique)#Javascript):

**class Point {**

**constructor(x, y) {**

**this.\_x = x;**

**this.\_y = y;**

**}**

**getX() {**

**return this.\_x;**

**}**

**getY() {**

**return this.\_y;**

**}**

**isOrigin() {**

**return this.\_x === 0 && this.\_y === 0;**

**}**

**translate(pt) {**

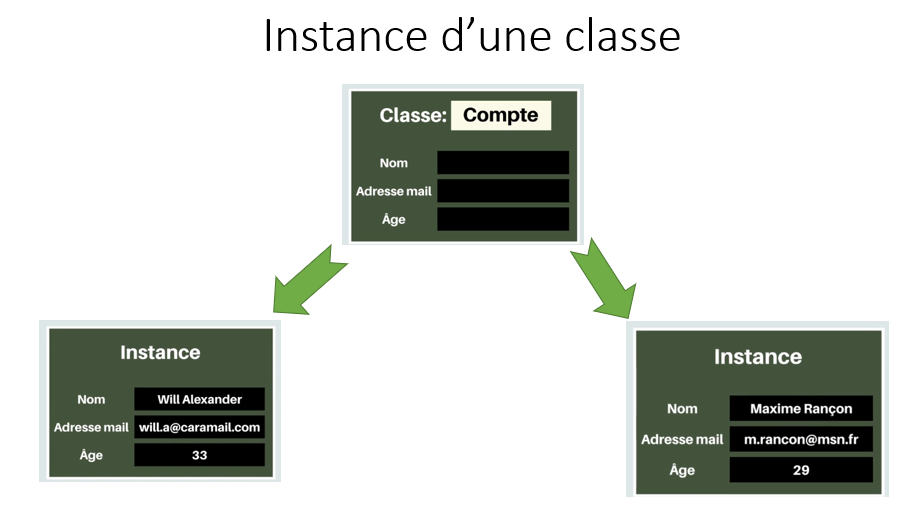
**return new Point(this.\_x + pt.\_x, this.\_y + pt.\_y);**

**}**

**}**

#### Instance d’une classe

Chaque objet construit à partir de cette **classe est appelé instance de cette classe.**

****

#### Pratiquez : créez un objet

Créez un objet et stockez-le dans une variable objet appelée **episode** . Utilisez bien des accolades et mettez les trois attributs suivants :

**title** : le titre de l'épisode

**duration** : la durée de l'épisode

**hasBeenWatched** : si l'épisode a été visionné ou non

Associez des valeurs appropriées à chaque attribut.

##### Solution

**HTML :**

<h3>Les objets</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="episode-info-o"></p>

</div>

**CSS :**

body{

background-color: #b7ffb7;

color: #000;

font-family: 'Roboto Slab', serif;

}

.series-frame{

max-width: 600px;

font-size: 0.9em;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #804040;

margin: 30px;

padding: 10px;

}

.my-form {

padding: 20px;

}

**JS :**

// =========================================

let oEpisode = {

title : 'Gone with the Wind',

duration : 243,

hasBeenWatched : true

};

// =========================================

document.querySelector('#episode-info').innerText = `Episode: ${oEpisode.title}

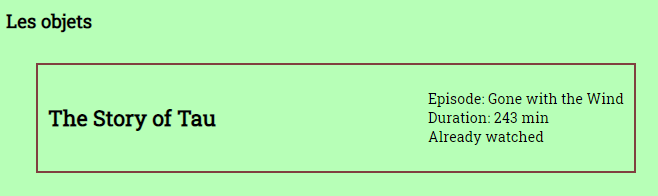
Duration: ${oEpisode.duration} min

${oEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`

Une image contenant texte

Description générée automatiquement

**Ecran :**



#### Accédez aux données d'un objet avec la notation pointée

Une fois qu'un **objet est enregistré dans une variable**, vous pouvez accéder à ses données.

Pour cela, utilisez **le nom de la variable qui contient l'objet**, **un point (.),** **puis le nom de la clé** dont vous souhaitez récupérer la valeur.

Une image contenant texte

Description générée automatiquement

##### Pratiquez : récupérez des valeurs depuis un Object

**Reprendre l’objet vu précédemment et y ajouter :**

1. Créez les trois variables suivantes :
   1. episodeTitle : le titre de l'épisode (string)
   2. episodeDuration : la durée de l'épisode (number)
   3. episodeHasBeenWatched : si l'épisode a été regardé ou non (boolean)
2. Assignez-y les valeurs correspondantes de l'objet episode en utilisant la notation "dot".

###### Solution

**HTML :**

<h3>Accéder aux données d'un objet</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="episode-info-aO"></p>

</div>

CSS : inchangé

**JS :**

let oEpisod = {

title : 'Gone with the Wind',

duration : 243,

hasBeenWatched : true

};

// Create variables here

// =====================================

let episodeTitle2 = oEpisod.title ;

let episodeDuration2 = oEpisod.duration ;

let episodeHasBeenWatched2 = oEpisod.hasBeenWatched ;

// =====================================

document.querySelector('#episode-info-aO').innerText = `Episode: ${episodeTitle2}

Duration: ${episodeDuration2} min

${episodeHasBeenWatched2 ? 'Already watched' : 'Not yet watched'}`

Une image contenant texte

Description générée automatiquement

**Ecran :**

Une image contenant texte

Description générée automatiquement

### Manipulez des classes

La construction d'un objet à la main, par la notation à accolades vue précédemment, convient bien à des objets simples et uniques.

Mais vous aurez souvent besoin de **beaucoup d'objets du même type**. C'est là que les **classes** sont utiles.



Figure 6: La classe est comme un plan

#### Définition de la classe & constructeur

Une classe est un **modèle**[[70]](#footnote-70) pour un objet dans le code. Elle permet de **construire plusieurs** **objets du même type,** appelés ***instances*** de la même classe ; plus facilement, rapidement et en toute fiabilité.

#### Construction d’une classe

1. Pour construire une classe dans JavaScript, utilisez le mot clé **class**, suivi par un nom
2. Encadrez ensuite le code de la classe **entre accolades**
3. Utiliser un ***constructor***
   * Utiliser lemotclé **this** avecla notation dot, pour attribuer les valeurs reçues **aux clés** correspondantes.

Le ***constructor*** d'une classe est la fonction qui est appelée quand on crée une nouvelle instance de cette classe avec le mot clé **new**.

Ce qui donne :

Une image contenant texte

Description générée automatiquement

Maintenant que la classe est terminée, on peut construire des instances de la classe ***Book*** par le mot clé **new** :



Gardez cette notion de classe en tête, on y reviendra un peu plus loin en abordant la notion de **propriété de classe** et les **méthodes d’instance**.

##### Pratiquez : les classes

Cette fois-ci, il y a **trois composants épisodes** 🡺

1. **Donc** la manière logique de procéder serait de construire **une classe Episode**
2. Et puis de construire **trois instances de cette classe** — une pour chaque épisode.

Créez une **classe Episode** avec le mot-clé **class**.

Créez un ***constructor*** pour votre classe **Episode** qui devra accepter trois arguments[[71]](#footnote-71) :

**title** : le titre de l'épisode (string)

**duration** : la durée de l'épisode (number)

**hasBeenWatched** : si l'épisode a été regardé ou non (boolean)

Avec le mot-clé **this** , assurez-vous que le ***constructor*** assigne correctement les arguments reçus aux champs correspondants de chaque nouvelle instance.

Avec le mot-clé **new** , créez trois instances de la classe **Episode** et stockez-les dans les trois variables :

**firstEpisode**

**secondEpisode**

**thirdEpisode**

N'oubliez pas de passer des arguments appropriés à chaque instance.

###### Solution

**HTML :**

<h3>Les classes</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="first-episode-info"></p>

</div>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="second-episode-info"></p>

</div>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="third-episode-info"></p>

</div>

**CSS :**

.series-frame{

    max-width: 600px;

    /\*font-size: 0.9em;\*/

    display: flex;

    /\*justify-content: space-between;\*/

    align-items: center;

    box-sizing: border-box;

    border: 2px solid #804040;

    margin: 30px;

    padding: 10px;

}

.my-form {

    padding: 20px;

  }

.series-frame h2 {

  margin-right: 20px;

}

**JS :**

class cEpisode{

constructor(title, duration, hasBeenWatched){

this.title = title;

this.duration = duration;

this.hasBeenWatched = hasBeenWatched;

}

}

let oFirstEpisode = new cEpisode('Gone with the Wind', 243, true)

let oSecondEpisode = new cEpisode("2001: A Space Odyssey", 156, true)

let oThirdEpisode = new cEpisode('A Beautiful Mind', 135, true)

// =====================================

document.querySelector('#first-episode-info').innerText = `Episode: ${oFirstEpisode.title}

Duration: ${oFirstEpisode.duration} min

${oFirstEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`;

document.querySelector('#second-episode-info').innerText = `Episode: ${oSecondEpisode.title}

Duration: ${oSecondEpisode.duration} min

${oSecondEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`;

document.querySelector('#third-episode-info').innerText = `Episode: ${oThirdEpisode.title}

Duration: ${oThirdEpisode.duration} min

${oThirdEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`;

**Une image contenant texte

Description générée automatiquement**

**Ecran :**

Une image contenant texte

Description générée automatiquement****

### Résumé

1. Les **objets** avec les paires clés/valeurs en notation **JSON**. Ils permettent d'enregistrer plusieurs éléments de données associés dans une même variable ;
2. La **notation** **pointée** (**dot**) qui donne accès aux valeurs d'un objet et la possibilité de les modifier ;
3. **Les** **classes**, et comment l'utilisation de classes peut vous permettre de **construire** **des** **objets** plus facilement et de façon plus lisible.

### To do : UML

<https://openclassrooms.com/fr/search?page=1&query=UML>

[Une image contenant texte

Description générée automatiquement](https://openclassrooms.com/fr/courses/4055451-modelisez-et-implementez-une-base-de-donnees-relationnelle-avec-uml)

## Définir des méthodes d’instance et des propriétés

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

#### Rappel sur la notion de propriété de classe

Il n’est pas rare d’utiliser le terme “**attribut**” au lieu de "**propriété**", mais cela représente bien la même chose.

Prenons l’exemple d’une classe qui représenterait un compte bancaire, et que l’on appellerait du coup BankAccount .

On pourrait trouver comme propriétés owner (permettant d’identifier le propriétaire du compte) et balance (permettant de connaître le montant disponible sur le compte).

Une image contenant table

Description générée automatiquement

Ainsi, lorsque que l’on crée une instance de BankAccount avec un propriétaire et un montant initial en argument, on pourra exploiter par la suite ces **propriétés** pour afficher leurs valeurs, les utiliser pour des calculs, les modifier, etc.

### Tirez parti des classes avec des méthodes d'instance

Maintenant que vous avez commencé à découvrir les **fonctions** vous pouvez ajouter des **méthodes d'instance** à des classes, pour augmenter leur puissance et leur utilité.

**Une méthode d'instance est une fonction faisant partie d'une classe, et qui agit sur une instance de cette classe**

Reprenons notre exemple de classe BankAccount :

**class BankAccount {**

**constructor(owner, balance) {**

**this.owner = owner;**

**this.balance = balance;**

**}**

**}**

#### Instance de classe newAccount()

►Vous pouvez ensuite créer une **instance de cette classe** appelée newAccount :

**const newAccount = new BankAccount("Bill Gates", 98000000000);//2020-07-31**

**►Pour rappel : une instance d'une classe est un objet.**

…►Et un objet est un type par **référence** :



..►donc vous pouvez toujours apporter des modifications à l'instance de newAccount

…►la partie constante **désigne une référence[[72]](#footnote-72) à cette instance**

Telle quelle, l'instance n'est pas très utile. Vous pourriez afficher son solde à la console par newAccount.balance

#### Méthode d'instance showBalance()

**class BankAccount {**

**constructor(owner, balance) {**

**this.owner = owner;**

**this.balance = balance;**

**}**

**showBalance() {**

**console.log("Account balance: " + this.balance + " $");**

**}**

**}**

La nouvelle **méthode** ci-dessus, déclarée par son nom suivi par (), utilise le mot clé this pour accéder à la **propriété** balance de l'instance, et l'afficher sur la console.

Ceci signifie que vous pouvez utiliser la notation dot sur l'instance newAccount pour appeler sa méthode showBalance() :

**let newAccount = new BankAccount("Bill Gates", 98000000000); //2020-07-31**

**newAccount.showBalance();**

Une image contenant texte

Description générée automatiquement

#### Ajouter des méthodes capables de modifier les propriétés de l'instance

Vous pouvez aussi ajouter des méthodes capables **de modifier les propriétés** de l'instance. Dans ce cas, ajoutez les méthodes deposit() (dépôt) et withdraw() (retrait) à la classe, en n'oubliant pas que les deux ont besoin d'un paramètre amount (montant).

**class BankAccount {**

**constructor(owner, balance) {**

**this.owner = owner;**

**this.balance = balance;**

**}**

**showBalance() {**

**console.log("Solde: " + this.balance + " EUR");**

**}**

**deposit(amount) {**

**console.log("Dépôt de " + amount + " EUR");**

**this.balance += amount;**

**this.showBalance();**

**}**

**withdraw(amount) {**

**if (amount > this.balance) {**

**console.log("Retrait refusé !");**

**} else {**

**console.log("Retrait de " + amount + " $");**

**this.balance -= amount;**

**this.showBalance();**

**}**

**}**

**}**

Dans le corps d'une classe, le mot clé this fait référence à **l'instance créée de la classe**. Dans cet exemple, il fait référence à newAccount .

### Pratiquez les méthodes instance

Plutôt que d'utiliser une fonction pour calculer la note moyenne pour une série, on a décidé que la classe Show (ou "Série") **devrait gérer les nouvelles notes automatiquement**

Comme vous pouvez le constater, la classe Show contient un Array ratings , pour les notes, qui est initialisé vide, et une propriété averageRating (note moyenne) de zéro.

1. Déclarez une méthode d’instance dans la classe Show appelée addRating() . Elle prendra une rating (la note), qui sera un nombre entre 1 et 5, comme paramètre.
2. La méthode addRating() doit :

* ajouter la note reçue au tableau ratings de l'instance ;
* recalculer la valeur averageRating de l'instance.

### Solution

**HTML**

<link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

<button id="refresh">Check ratings</button>

**CSS**

body, button {

background-color: #111;

color: #EEE;

font-family: 'Roboto Slab', serif;

}

.series-frame {

max-width: 600px;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #EEE;

margin: 30px;

padding: 10px;

}

#refresh {

float: right;

margin-right: 20px;

font-size: 1.2em;

font-weight: 200;

padding: 10px;

}

#refresh:hover {

background-color: #000090;

cursor: pointer;

}

**JS**

**class Show {**

**constructor(title, numberOfSeasons) {**

**this.title = title;**

**this.numberOfSeasons = numberOfSeasons;**

**this.ratings = [];**

**this.averageRating = 0;**

**}**

**addRating(rating) {**

**this.ratings.push(rating);**

**let sum = 0;**

**for (let rating of this.ratings) {**

**sum += rating;**

**}**

**this.averageRating = sum / this.ratings.length;**

**}**

**}**

// ======================

const tau = new Show('The Story of Tau', 5);

const colin = new Show('The Hero of Old Meldrum', 3);

const clara = new Show('The Bugs of Isla Clara', 6);

const shows = [tau, colin, clara];

const body = document.querySelector('body');

const refresh = document.querySelector('#refresh');

refresh.addEventListener('click', () => {

removeShows();

addRandomRatings();

updateShows();

})

const updateShows = () => {

for (let show of shows) {

const showPane = document.createElement('div');

showPane.classList.add('series-frame');

const showHeading = document.createElement('h2');

showHeading.innerText = show.title;

const showDetails = document.createElement('p');

const seasons = document.createElement('p');

seasons.innerText = show.numberOfSeasons + ' seasons';

const ratings = document.createElement('p');

ratings.innerText = show.averageRating > 0 ? show.ratings.length + ' ratings\n' + show.averageRating.toFixed(1) + ' stars' : 'No ratings yet';

showDetails.append(seasons);

showDetails.append(ratings);

showPane.append(showHeading);

showPane.append(showDetails);

body.append(showPane);

}

};

const removeShows = () => {

const children = [];

for (let childNode of body.childNodes) {

children.push(childNode);

}

for (let child of children) {

if (child.tagName == 'DIV') {

body.removeChild(child);

}

}

}

const addRandomRatings = () => {

for (let show of shows) {

if (Math.random() >= 0.2) {

const numberOfRatings = Math.floor(Math.random() \* 4 + 1);

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

const rating = Math.floor(Math.random() \* 5 + 1);

show.addRating(rating);

}

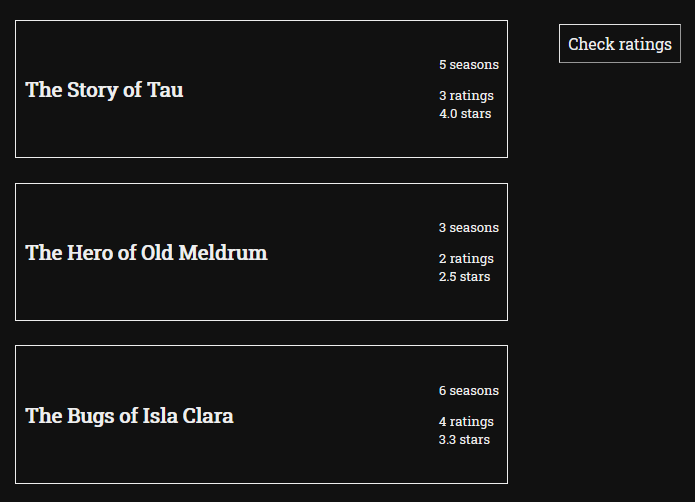
}

}

}

updateShows();

**Affichage**

****

### Découvrez les méthodes statiques

Il existe un type spécial de méthode pouvant être ajoutée à une classe : la méthode statique. Elle est différente des méthodes d'instance parce qu'elle n'est pas liée à une instance particulière d'une classe, mais à la classe elle-même.

Utilisez-la pour créer des **méthodes utilitaires** (**helper en anglais**) où vous n'aurez pas besoin d'une instance d'une classe pour les utiliser. Vous pourrez vous en servir comme **boîte à outils de fonctions** que vous utiliserez souvent.

Par exemple, en JavaScript, l'objet Math contient beaucoup de méthodes utiles :

**let x = 5.95;**

**const randomNumber = Math.random();**

**const roundDown = Math.floor(x);**

**const roundTop = Math.ceil(x);**

**const maxData = Math.max(23,17,83,2,13);**

**const minData = Math.min(23,17,83,2,13);**

Une image contenant texte

Description générée automatiquement

**console.log(randomNumber);**

**console.log(roundDown);**

**console.log(roundTop);**

**console.log(maxData);**

**console.log(minData);**

Vous n'avez pas besoin de créer par new une instance de l'objet Math pour utiliser ces méthodes ; il suffit de les appeler sur l'objet Math global.

►Vous pouvez créer vos propres méthodes statiques par le mot clé **static** !

**class BePolite {**

**static sayHello(){**

**console.log("Hello ☺");**

**}**

**static sayHelloTo(anyName){**

**console.log("Hello" + String.fromCharCode(32) + anyName + " ☺");**

**}**

**static remainder(dividend, divider){**

**return dividend % divider;**

**}**

**}**

**let modulo = BePolite.remainder(17,5);**

**BePolite.sayHello();**

**BePolite.sayHelloTo("Person");**

**console.log(modulo);**

Une image contenant texte

Description générée automatiquement

Vous n'avez pas besoin d'ajouter un **constructor** à votre classe, car vous n'allez pas l'instancier.

Vous avez une méthode qui :

* imprime un message **générique** ;
* accepte un argument et l'utilise pour créer un message ;
* renvoie une valeur à partir des arguments que vous lui envoyez.

Toutes **ces fonctionnalités pourraient être des fonctions**,

mais l'avantage d'utiliser des méthodes de classe statiques est par exemple de **pouvoir les regrouper par catégorie ou par type.**

### En résumé

nous nous sommes intéressés à **deux types de méthodes de classe** :

1. les **méthodes d'instance**, **qui agissent** sur les instances individuelles d'une classe ;
2. les **méthodes statiques**, qui ne **s'appuient pas** sur une instance d'une classe.

*Dans le chapitre qui suit, nous allons voir comment écrire des fonctions, et du code propre qui soit facile à lire, à maintenir et à adapter !*

# Les modules

<https://grafikart.fr/tutoriels/javascript-modules-1354>

# To do list

Prototype

Closure

# Bibliographie

BERSINI H. PROGRAMMATION ORIENTEE OBJET. Cours Et Exercices En Uml 2, Avec Java, C#, C++, Python, Linq

Une image contenant texte, noir, équipement électronique

Description générée automatiquement

POWERS Shelley Débuter en JavaScript



djamelchabane@gmail.com

# Links

<https://buzut.net/programmation-orientee-objet-javascript/>

# Links

1. https://developer.mozilla.org/fr/docs/Glossary/OOP [↑](#footnote-ref-1)
2. https://fr.wikipedia.org/wiki/Programmation\_orient%C3%A9e\_prototype [↑](#footnote-ref-2)
3. https://developer.mozilla.org/fr/docs/Glossary/Prototype-based\_programming [↑](#footnote-ref-3)
4. https://buzut.net/programmation-orientee-objet-javascript/ [↑](#footnote-ref-4)
5. https://davidwalsh.name/javascript-objects-deconstruction [↑](#footnote-ref-5)
6. https://davidwalsh.name/javascript-objects [↑](#footnote-ref-6)
7. https://fr.wikipedia.org/wiki/Programmation\_orient%C3%A9e\_objet [↑](#footnote-ref-7)
8. https://fr.wikipedia.org/wiki/Interface\_(programmation\_orientée\_objet) [↑](#footnote-ref-8)
9. https://jsbin.com/?js,console [↑](#footnote-ref-9)
10. https://kangax.github.io/compat-table/es6/ [↑](#footnote-ref-10)
11. https://www.alsacreations.com/astuce/lire/1562-script-attribut-async-defer.html [↑](#footnote-ref-11)
12. Ou Pascal [↑](#footnote-ref-12)
13. A la différence d’autres langages comme VB, par exemple. [↑](#footnote-ref-13)
14. Nommer les choses, car au commencement était le verbe. [↑](#footnote-ref-14)
15. Contrairement au C, C++, C#, par exemple. [↑](#footnote-ref-15)
16. https://developer.mozilla.org/fr/docs/Web/JavaScript/Structures\_de\_donn%C3%A9es [↑](#footnote-ref-16)
17. ‘ Code ascii = 039 [↑](#footnote-ref-17)
18. «  Code ascii = 034 [↑](#footnote-ref-18)
19. + Code ascii = 043 [↑](#footnote-ref-19)
20. https://developer.mozilla.org/fr/docs/Web/JavaScript/Data\_structures [↑](#footnote-ref-20)
21. C’est un pointeur comme en C, par exemple. [↑](#footnote-ref-21)
22. Erreur sur le nom des variables , ligne 538 du code source HTML. [↑](#footnote-ref-22)
23. https://codeburst.io/explaining-value-vs-reference-in-javascript-647a975e12a0 [↑](#footnote-ref-23)
24. https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global\_Objects/Set [↑](#footnote-ref-24)
25. https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global\_Objects/Map [↑](#footnote-ref-25)
26. https://openclassrooms.com/fr/courses/5543061-ecrivez-du-javascript-pour-le-web [↑](#footnote-ref-26)
27. https://openclassrooms.com/fr/courses/1603881-apprenez-a-creer-votre-site-web-avec-html5-et-css3 [↑](#footnote-ref-27)
28. https://openclassrooms.com/fr/courses/2984401-apprenez-a-coder-avec-javascript [↑](#footnote-ref-28)
29. Node.js est une plateforme logicielle libre en JavaScript orientée vers les applications réseau événementielles hautement concurrentes qui doivent pouvoir monter en charge.(Wikipédia) [↑](#footnote-ref-29)
30. Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.( https://nodejs.org/en/) [↑](#footnote-ref-30)
31. Dans le cas d’un programme séquentiel [↑](#footnote-ref-31)
32. C’est même souvent le cas le plus fréquent en réalité ! [↑](#footnote-ref-32)
33. https://www.azur-web.com/astuces/javascript-differences-var-let-const [↑](#footnote-ref-33)
34. https://www.geeksforgeeks.org/difference-between-var-and-let-in-javascript/ [↑](#footnote-ref-34)
35. C’est le cas réel ! [↑](#footnote-ref-35)
36. Elle rappelle le for…each du VB. [↑](#footnote-ref-36)
37. Environnement de développement intégré ou **IDE** (Integrated Development Environment) [↑](#footnote-ref-37)
38. https://code.visualstudio.com/ [↑](#footnote-ref-38)
39. https://www.jetbrains.com/ [↑](#footnote-ref-39)
40. https://www.pierre-giraud.com/javascript-apprendre-coder-cours/fonction-flechee/ [↑](#footnote-ref-40)
41. https://www.pierre-giraud.com/javascript-apprendre-coder-cours/fonction-anonyme-auto-invoquee-recursive/ [↑](#footnote-ref-41)
42. ECMAScript Edition 6 (ES6) : juin 2015 [↑](#footnote-ref-42)
43. https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Functions#fonctions\_pr%C3%A9d%C3%A9finies [↑](#footnote-ref-43)
44. Toutes les fonctions Javascript : https://waytolearnx.com/2019/09/liste-des-fonctions-javascript.html [↑](#footnote-ref-44)
45. Quelle moyenne ? (Harmonique, géométrique, arithmétique ou quadratique ?) [↑](#footnote-ref-45)
46. Victor Hugo [↑](#footnote-ref-46)
47. the form is the background that rises to the surface [↑](#footnote-ref-47)
48. Ou dans une classe [↑](#footnote-ref-48)
49. « Ce que l'on conçoit bien s’énonce clairement et les mots pour le dire arrivent aisément » Nicolas Boileau-Despréaux (1674). [↑](#footnote-ref-49)
50. Et donc un bouton ! [↑](#footnote-ref-50)
51. End to end. [↑](#footnote-ref-51)
52. https://jasmine.github.io/ [↑](#footnote-ref-52)
53. https://mochajs.org/ [↑](#footnote-ref-53)
54. https://karma-runner.github.io/6.3/index.html [↑](#footnote-ref-54)
55. https://openclassrooms.com/fr/courses/3504461-testez-linterface-de-votre-site [↑](#footnote-ref-55)
56. https://code.visualstudio.com/ [↑](#footnote-ref-56)
57. https://www.jetbrains.com/fr-fr/webstorm/ [↑](#footnote-ref-57)
58. https://openclassrooms.com/fr/courses/5641796-adoptez-visual-studio-comme-environnement-de-developpement [↑](#footnote-ref-58)
59. https://codepen.io/quentin-laurent/pen/gOLEmey [↑](#footnote-ref-59)
60. Examen lors de l’entretien d’embauche ! [↑](#footnote-ref-60)
61. Effectivement, il faut que le tableau soit, au préalable, trié. (*Sorted by alphabetical order*) [↑](#footnote-ref-61)
62. Complexité algorithmique [↑](#footnote-ref-62)
63. Donc indexé [↑](#footnote-ref-63)
64. Donc prérequis : algorithme de tri [↑](#footnote-ref-64)
65. Recherche dichotomique (*Dichotomous search*) [↑](#footnote-ref-65)
66. Donc il faudra donc penser, au préalable, à trier le tableau. [↑](#footnote-ref-66)
67. https://stackoverflow.com/ [↑](#footnote-ref-67)
68. https://fr.wikipedia.org/wiki/Algorithme\_r%C3%A9cursif [↑](#footnote-ref-68)
69. https://codepen.io/quentin-laurent/pen/QWGRKMo [↑](#footnote-ref-69)
70. Quid d’UML ? [↑](#footnote-ref-70)
71. Ou attributs [↑](#footnote-ref-71)
72. Équivalent du pointeur en C. [↑](#footnote-ref-72)