

# Apprendre à programmer avec JS

Version ESG (ou ES 2015)

## LES VARIABLES

- Nom → camelCase
- Type
- Valeur

```
let number Of Cats = 2;  
number Of Cats *= 6; // = 12
```

Opérateurs :

- + \* /

```
let number Of Likes = 10;  
number Of Likes -- ; // = 9
```

```
let cookies In Jar = 10;
```

```
let cookies Removed = 2;
```

```
let cookies Left In Jar = cookies In Jar - cookies Removed;
```

## LES CONSTANTES

```
const hoursPerDay = 24;  
console.log(hoursPerDay);  
↳ 24
```

Cela évite de remplacer des données essentielles.

## LES TYPES

### Types primitifs

- number → 42 || -42 || 42.424242
- string → let firstName = 'Raphy'
- boolean

↳ yes → let userIsSignedIn = true;

↳ no → let userIsAdmin = false;

```
console.log(typeof firstName);
```

↳ string

```
console.log(firstName + ' ' + 'Beaudet')
```

↳ Raphy Beaudet

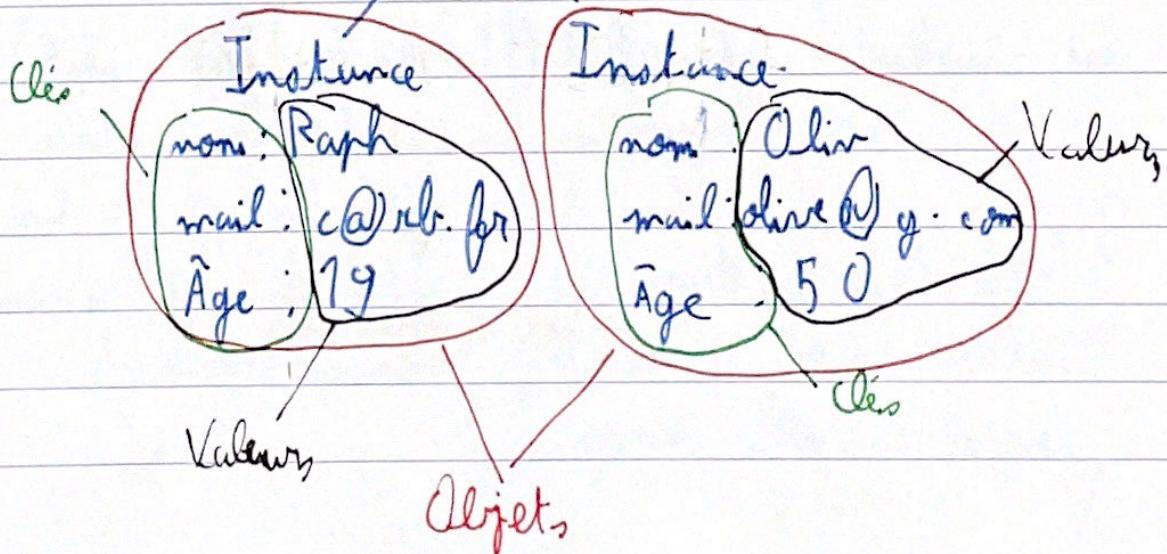
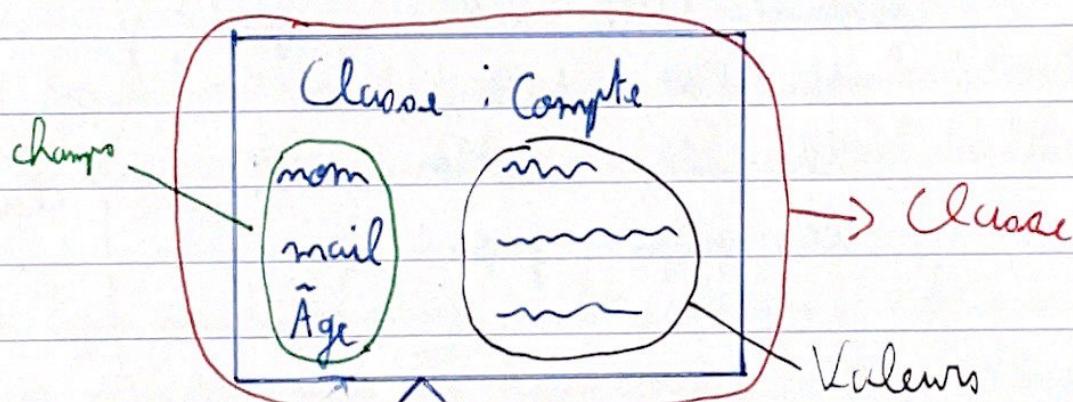
# LES OBJETS ET LES CLASSES

Un objet contient

- Plusieurs informations
- Différents types
- Type complexe

Clé : chaîne de caractères qui explique une valeur

Les classes = c'est un plan qui permet de créer des objets. Elle contient des champs qui correspondent aux clés de l'objet final.



```
let myBook = {  
    title: "L'histoire de Tao",  
    numberOfPages: 250,  
    isAvailable: true  
};
```

```
let bookTitle = myBook.title; //="L'histoire de Tao"
```

Pour les classes :

```
class Book {  
    constructor(title, author, pages) {  
        this.title = title;  
        this.author = author;  
        this.pages = pages;  
    }  
}
```

Création classe

```
let myBook = new Book("Le roi", "RB", 45);  
→ création instance
```

```
{ title: "Le roi",  
  author: "RB",  
  pages: 45 }  
→ Voici l'objet créé
```

# ARRAY, SET, MAP

Un array permet de créer une liste de valeurs ou d'objets.

```
let firstGuest = 'Sarah Kate'  
let secondGuest = 'Will Alexander';  
let guests = [firstGuest, secondGuest];  
console.log(guests[0]);  
↳ Sarah Kate
```

Les types valeurs (types primitifs : string, number, boolean) et les types références (objets, tableaux) ont des différences.

Les types primitifs (string, number, boolean) sont passés par valeurs, mais les types complexes (objet, tableau), sont passés par références. Donc :

```
let guest = "Will Alexander"  
let guests = [guest]  
guest = 'Sarah Kate'  
console.log(guests);  
↳ Will Alexander
```

X

```
let guest = {  
    name: "Will Alexander"  
};  
let guests = [guest];  
guest.name = 'Sarah Kate';  
console.log(guests);  
↳ Sarah Kate
```

travailler sur les tableaux :

- `length` → nombre d'éléments
- `push("m")` → ajoute élément à la fin
- `unshift("m")` → ajoute élément au début
- `pop()` → supprime le dernier élément

A savoir : il existe également les sets et les maps.

JavaScript peut être utilisé dans de nombreux environnements : JS Bin (test en ligne de code), pages web, serveurs ...

## LES CONDITIONS

```
const numberofSeats = 30;
let numberofGuests = 25;
if (numberofGuests == numberofSeats) {
    // tous les sièges occupés
} else if (numberofGuests < numberofSeats) {
    // autoriser + d'invités
} else {
    // ne pas autoriser + d'invités.
```

Expressions de comparaison :

< <= == > = > !=

L'égalité

== != → vérifie valeur

== = != = → vérifie valeur et type

Conditions multiples

& & || !

let userLoggedIn = true;

let userHasPremiumAccount = false;

userLoggedIn & & userHasPremiumAccount; // true

!userLoggedIn; // false

Le scope des variables : Une variable déclarée au sein d'un bloc ne pourra pas être utilisée en dehors de celui-ci. Par contre, si si elle est déclarée au préalable, on pourra lui attribuer une valeur au sein d'un bloc.

L'instruction switch : sert à vérifier la valeur d'une variable par rapport à une liste de valeurs attendues

[création d'objets avec des noms d'utilisateurs et des clés "name", "age", "accountLevel"]

```
switch (firstUser.accountLevel){
```

case 'normal':

```
    console.log('Votre compte est normal');  
    break;
```

case 'premium':

```
    console.log('Votre compte est premium');  
    break;
```

default:

```
    console.log('Inconnu');
```

```
}
```

## LES BOUCLES

For → Répéter une action un certain nombre de fois.  
While → Répéter tant qu'une action est remplie.

Pour la boucle "for" :

```
const numberOfPassengers = 10;  
for (let i = 0; i < numberOfPassengers; i++) {  
    console.log("Passager embarqué!");  
}  
console.log("Tous les passagers sont embarqués");
```

La boucle "for" avec les tableaux :

for ... in

```
const passengers = [ ..., ..., ..., ...];  
for (let i in passengers) {  
    console.log("Embarquement du passager " + passengers[i]);  
}
```

for ... of → pour le cas où l'indice précis d'un élément n'est pas nécessaire pendant l'itération  
En plus, on peut gérer les objets !

```
const passengers = [object, object, object, object];  
for (let passenger of passengers) {  
    console.log("Nom du passager : " + passenger.name);  
}
```

Pour la boucle "while" :

```
let seatsLeft = 10;  
let passengersStillToBoard = 8;  
let passengersBoarded = 0;  
while (seatsLeft > 0 && passengersStillToBoard > 0){  
    passengersBoarded++;  
    passengersStillToBoard--;  
    seatsLeft--;  
}  
console.log(passengerBoarded); // = 8
```

## APPRÉHENDER LES ERREURS

- Erreurs de syntaxe : faute d'écriture dans le code.
- Erreurs logiques : erreurs de logique dans le programme.
- Erreurs d'exécution : quelque chose d'inattendu, souvent extérieur
  - ↳ 

```
if (dataExists && dataIsValid) {  
    // utiliser données  
} else {  
    // gérer erreur  
}
```

On peut utiliser des blocs `try / catch` pour essayer un code et détecter les erreurs éventuelles.

```
try {  
    // code susceptible à l'erreur  
} catch (error) {  
    // réaction aux erreurs  
}
```

## LES PARAMÈTRES ET LES VALEURS DE RETOUR

Fonction : bloc de code qui effectue une tâche précise.

```
function somme(nombre1, nombre2) {  
    return nombre1 + nombre2;  
}  
  
somme(3, 4);
```

Arguments

Écrire une fonction en JS :

```
const add = (firstNumber, secondNumber) => {  
    return firstNumber + secondNumber;  
}  
const result = add(4, 7);
```

Pour les objets et les tableaux dans les fonctions, comme ~~they~~ leurs variables contiennent des références, quand on ~~put~~ on passe un dans une fonction, on ne fait pas une copie mais on passe par la référence d'origine.

## MÉTHODES D'INSTANCE ET CHAMPS

### Méthodes d'instance et champs

Méthodes d'instance → Fonction associée à une classe, opérant à l'intérieur de chaque instance d'une classe.

→ Ces méthodes sont dites "d'instance" car elles n'agissent que dans le contexte de l'instance de la classe qui les contient.

TIPS : les "this." permettent de faire référence à l'instance de la classe.

```

class BankAccount {
    constructor(owner, balance) {
        this.owner = owner;
        this.balance = balance;
    }
    showBalance() {
        console.log(`Solde: ${this.balance} EUR`);
    }
    deposit(amount) {
        console.log(`Dépot de ${amount} EUR`);
        this.balance += amount;
        this.showBalance();
    }
}

```

\*

Il existe également les méthodes statiques. Contrairement aux méthodes d'instance, elle n'est pas liée à une instance particulière d'une classe, mais à la classe elle-même.

\* [crier nouvelle instance "newAccount"] \*

newAccount.deposit(50);

```

class BePolite {
    [ méthode statique ]
    static sayHelloTo(name) {
        console.log("Hello " + name + " !");
    }
    [ méthode statique ]
}
[ appel méthode ]
BePolite.sayHello("Raph");
[ appel méthode ]

```

On a pas besoin d'ajouter un constructeur à notre classe car on ne va pas l'instancier.

Pour exemple, en JS, l'objet Math contient beaucoup de méthodes utiles :

```

const randomNumber = Math.random(); // nbr aléa [0;1]
const roundMeDown = Math.floor(495.966); // arrondis le bas (495)

```

L'avantage d'utiliser des méthodes de classe statiques est de pouvoir regrouper par catégorie ou par type.

## ÉCRIVEZ DES FONCTIONS PROPRES

Si on doit répéter deux fois ou plus du code identique, alors il faut créer une fonction. Par ailleurs, une fonction doit faire une tâche précise, qu'il à appeler une fonction dans une fonction.

→ L'objectif est de faire des fonctions les plus petites possible.

Il est nécessaire de laisser des commentaires lorsque c'est utile.

↪ Sur une ligne : //

↪ Sur plusieurs lignes : /\*

\*\* Commentaire ligne 1

\*\* Commentaire ligne 2

\*/

Quelques autres règles basiques : pas de majuscule sur la première lettre d'une variable contrairement aux classes, choisir des noms explicites, faire des tabulations et mettre des espaces.

# TESTER QU'UNE FONCTION FAIT CE QU'ELLE DIT

## • Tests unitaires :

```
const testSimpleWordCount = () => {
  const testString = 'I have four words!';
  if (getWordCount(testString) !== 4) {
    console.error('Simple getWordCount failed!');
  }
}
```

## • Architecture de test (solution préférable) :

Les architectures et bibliothèques de test permettent l'écrire automatiquement des suites de tests complètes de notre code.

Voici à quoi ça pourrait ressembler :

```
describe('getWordCount()', function() {
  it('should find four words', function() {
    expect(getWordCount('I have four words!')).to
      .equal(4);
  });
});
```

## • Tests d'intégration

Vérifient les multiples fonctions ou classes pour s'assurer qu'elles travaillent ensemble.

## • Tests fonctionnels (E2E)

Vérifient des scénarios complets en contexte. Ex : Un utilisateur se connecte, ouvre ses notifications et les marque toutes comme lues.

# DÉBOGUER UNE FONCTION

## • Afficher la console

```
const getWordCount = (stringToTest) => {
  const wordArray = stringToTest.split(' ');
  console.log("Word array in getWordCount: " + wordArray);
  console.log(wordArray);
  return wordArray.length;
}
```

→ Ici, on s'aperçoit grâce à la console que notre fonction a un problème.

On décide qu'on aurait du écrire `stringToTest.split('')` à la place de `stringToTest.split(' ')`.

## • Utiliser des outils pour développer

Pour le web, il existe des outils sur les navigateurs (Chrome, Firefox, Safari et Edge) qui permettent d'ajouter des points d'arrêt pas à pas (breakpoints) à notre code. On parcourt l'exécution ligne après ligne et on peut vérifier les valeurs des variables à chaque étape. On peut même ignorer certains morceaux de code pour voir comment l'application réagit.

Hors web, on a des outils sur les environnements de développement intégrés (Visual Studio, WebStorm) qui comprennent aussi des débogeurs.

## • Le canard en plastique

Tout bête : on explique notre code à un canard en plastique.

La réflexion à voix haute permet de voir simplement les problèmes.

## LA RÉCURSIVITÉ : L'APPEL DE FONCTIONS À L'INTÉRIEUR D'ELLES-MÊMES

Une fonction récursive est une fonction qui s'appelle elle-même d'une façon ou d'une autre.

Exemple : Trouver un prénom précis dans un tableau composé de beaucoup de prénoms.

→ Solution 1 : On parcourt le tableau du début jusqu'à ce qu'on trouve le prénom... Mais ce n'est pas optimisé.

→ Solution 2 : On va à la moitié de notre tableau et, comme il est rangé par ordre alphabétique, on détermine s'il fait partie de la partie supérieure ou inférieure. Ensuite, on rappelle la même fonction avec un intervalle entre deux index réduits, jusqu'à ce qu'on trouve le nom!

```
const binarySearch = (array, thingToFind, start, end) => {
    if (start > end) {
        return false;
    }
    let mid = Math.floor((start + end) / 2);
```

```
if (thingToFind < array[mid]) {  
    // il faut rechercher dans la première moitié  
    return binarySearch(array, thingToFind, start, mid-1);  
    // mid-1 car on a pas besoin de re-vérifier  
} else {  
    // il faut rechercher dans la deuxième moitié  
    return binarySearch(array, thingToFind, mid+1, end);  
}  
}
```

Pour éviter une boucle infinie, il faut toujours un base case pour dire à la fonction de s'arrêter. Dans notre cas, il s'agit de :

```
| if (start > end) {  
|     return false;  
| }
```

Notre fonction pas à pas :

#1 start = 0

end = 7

mid = 3

Anne	Bernard	Clemence	Justine	Lopard	Renard	Tao	Valerie
			Justine	Lopard	Renard	Tao	Valerie

Justine < Simon

#2 start = 4

end = 7

mid = 5

Anne	Bernard	Clemence	Justine	Lopard	Renard	Tao	Valerie
				Lopard	Renard	Tao	Valerie

Renard < Simon

#3 start = 6

end = 7

mid = 6

Anne	Bernard	Clemence	Justine	Lopard	Renard	Tao	Valerie
					Renard	Tao	Valerie

Tao > Simon

#4 start = 6

end = 5

start > end

return false

# RÉSUMÉ

variable : let / constante : const / type : - number  
- string  
- ...

## POO

classe : class MaClasse / instance : let monInstance = new MaClasse();  
attribut : this.couleur = couleur;  
méthode : fonction associée à une classe

## Collection

array : let monArray = ['a', 'b', 'c']; console.log(monArray[0]);  
object : let monObject = {nom: "Jean", prénom: "Dany", age: 26};  
console.log(monObject.age);

opérateurs logiques : &&, ||, <, >, >=, <=, ==, ===, != et !

exception : try { fonctionQuiRetourneUneException(); } catch(e) {  
console.log("il y a une exception : " + e.getMessage()); }

## Fonction

paramètre : function maFonction(parametre1) {  
 console.log(parametre1);  
}

argument : let monArgument = "Bonjour";  
maFonction(monArgument);

return : function additionner(valeur1, valeur2) {  
 return valeur1 + valeur2;  
}

let resultat = additionner(12, 13);

récursivité : function factorielle(number) {  
 if (number <= 1) return 1;  
 else return (number \* factorielle(number - 1));  
}

Exemple : factorielle(4);  
→  $4 \times 3 \times 2 \times 1 = 24$

(C'est une formule mathématique)