

# Javascript

## Les bases

# Developper Fullstack Javascript

Fullstack developer for about fourteen years, I like making software in all its aspects: user needs, prototypes, UI, frontend developments, API developments, ...

## Team & Projects


I work on DataClient teams where we work for tooling around data client and perform the data client quality.

## Hobbies


I like drawing, UX,  
Web Design




# SAMUEL GOMEZ

 samuel-gomez

 @gamuez

 www.samuelgomez.fr

 gamuez\_art

---

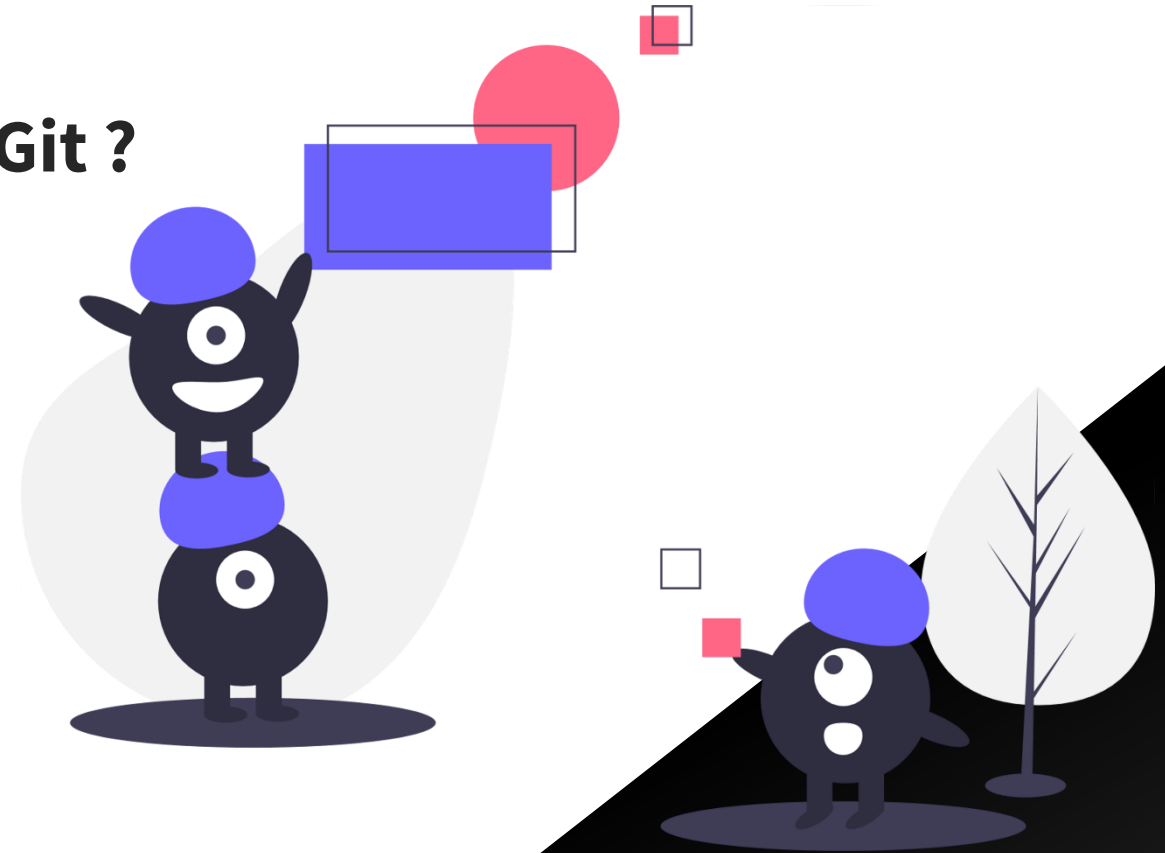
Qui n'a jamais fait de Javascript ?

---

Qui n'a jamais utilisé Git ?

---

Qui ne connaît pas npm?



## Séance 1 : Base de JavaScript

Rappel rapide des bases.

## Séance 2 : JavaScript avancé

Manipulation du DOM, appels HTTP, etc ...

## Séance 3 : Programmation fonctionnelle (bases)

Les concepts de bases.

## Séance 4 : Programmation fonctionnelle (avancée)

Les concepts plus avancés.

## Séance 5 : Les bases de React

JSX, composants, ...

## Séance 6 : Gestion d'état

Etat local, Context API, Fetch, ...

## Préambule

- Présentation
- Niveau de connaissance
- Déroulé de la formation

## Introduction

- C'est quoi ?
- Historique
- ECMAScript
- TC39
- Quels usages ?

## Valeurs, opérateurs et affectation

- Les types de valeur
- Les opérateurs
- Exercices
- Corrigés

## Les fonctions

- Concept
- Déclaration
- Exécution
- Arguments
- Portée et contexte
- Exercices
- Corrigés

## Structures de contrôle

- Les conditions
- Les boucles
- Exercices
- Corrigés

## Objets et tableaux

- Les objets
- Les tableaux
- Exercices
- Corrigés

## Introduction

# C'est quoi ?

Moteurs JavaScript



SpiderMonkey



V8 (Google)



Le JS a été créé en 1995 par **Brendan Eich** chez Netscape Communications.

D'abord appelé Mocha puis **Livescript**, il a été créé à l'origine pour être exécuté côté serveur puis fut utilisé sur le navigateur Netscape qui deviendra **Firefox**.

A ses débuts, le JS n'était pas très évolué, il a évolué au fil des années et a envahi toutes les couches du développement logiciel.



# Introduction

# ECMAScript

**ECMAScript** est un ensemble de normes concernant les langages de programmation de type script et standardisées par **Ecma International** dans le cadre de la spécification ECMA-262.

Il s'agit donc d'un **standard**, dont les spécifications sont mises en œuvre dans différents langages de script, comme **JavaScript** ou **ActionScript**.

C'est un langage de **programmation orienté prototype**.

ECMAScript propose actuellement 7 versions : ..., ES5, ES6, ES7

Beaucoup de changements ont été amenés dans la version 6 qui faciliteront la mise en œuvre des concepts de programmation fonctionnelle.



Comme expliqué précédemment, le **JS évolue** et des propositions d'amélioration sont faites régulièrement.

Pour qu'une **proposition** soit **validée et intégrée** à la prochaine release du **standard**, la proposition doit passer par un ensemble d'étapes de validation, tout cela dirigé par le comité **TC39**.

Ce comité se réunit deux fois par mois afin de discuter des différentes propositions qui sont en en cours de validation.

Il y a **5 étapes** (ou **stade**) de validation, de **Stage 0** pour la soumission de l'idée à **Stage 4** pour la validation définitive du standard.

Il est possible d'utiliser certains niveaux de standardisation dans son code avant que celui-ci n'atteigne le stade final. Pour cela, il faudra utiliser un outil de compilation : **BabelJS**.

# Pour quel usage ?

Tout. Enfin presque.

Evidemment, pour le **Web** avec la manipulation du DOM, la gestion des évènements, les animations, les styles.

On peut développer des logiciels soit des WebApp type **PWA**, des **applications mobiles** proches du natif (IONIC ou React Native).

Côté serveur, avec **NodeJS**, par exemple créer une **API Rest**

De la **3D** avec le WebGL et une librairie très connue comme ThreeJS.

De l'**IOT**, du multimédia, des jeux vidéo, ...

# Valeurs, opérateurs et affectation

# Les type de valeurs

En JS, Il y a six types de valeurs de base : les **nombres** (number), les **chaînes de caractères** (string), les **booléens** (boolean), les **objets** (object), les **fonctions** (function) et les **valeurs indéfinies** (undefined).

Pour connaitre le type d'un élément, il suffit d'utiliser l'opérateur « **typeof** ».

Le type « **number** » est le type numérique, les nombres peuvent être utilisés directement dans un script JS. Ils servent notamment à effectuer des opérations arithmétiques.

Le type « **Boolean** » est le type booléen (true ou false), il est utile pour vérifier si une valeur ou une comparaison est vraie ou non.

# Les type de valeurs

Le type « **string** » est le type chaîne de caractères, à mettre entre doubles ou simples guillemets. Certains caractères doivent être « échappés » pour pouvoir être interprétés :

```
"Un caractère de saut de ligne est écrit ainsi \"\\n\"."
```

Il est possible de concaténer des chaînes de caractères à l'aide l'opérateur « + » :

```
"con" + "cat" + "é" + "ner"
```

L'**ES6** apporte une meilleure façon de faire, on le verra dans le chapitre sur l'ES6.

Le type « **expression rationnelle** » est utile pour le traitement des chaînes de caractères.

Voici la forme que cela peut prendre :

```
var masque = /^[a-z0-9st]+$/i ;
```

Les **objets**, les **tableaux** et les **fonctions** seront vus après.

# Les opérateurs

## Opérateurs d'affectation

L'affectation de base utilise le signe « = »

## Opérateur de comparaison

Une bonne connaissance des opérateurs vous permettra une programmation plus efficace et succincte.

## Egalité / inégalité

On utilise les opérateurs binaires suivant « ==, ===, !=, !== » pour effectuer des comparaisons.

```
(1=== 2-1) // true
(1===2) // false
(1 == '1') ?
(1 === '1') ?
('mot') !== 'm'+ 'o'+ 't') // false
('mot1' !== 'mot2') // true
```

# Les opérateurs

## Plus grand que / plus petit que

Pour tester cela, on utilise les opérateurs binaires suivant « **>**, **<**, **>=**, **<=** » pour effectuer des comparaisons

```
1<1 //false  
1<= 1 //true  
'1' <= 1 //true
```

On peut également comparer des chaînes de caractères

```
'b'>'a' // true  
'aaa'>'b' // false
```

Même si cela est possible, il est déconseillé d'effectuer des comparaisons sur des types différents.



# Les opérateurs

## Opérateurs arithmétiques

Binaires: multiplication ( \* ), division ( / ), modulo ( % )

```
var a = 4 ; var b = 2 ;  
var c = a * b ; // 8  
var d = a / b ; // 2  
var e = a % 2 ; // 0
```

Unaire : ++ ou --

```
var a = 1 ;  
a++ ; // 2  
a-- ; // 1
```

# Les opérateurs

Il existe également l'opérateur « - », qui retourne le nombre opposé à son opérande :

```
var a = 3 ;  
a=-a ; // -3
```

L'opérateur « + » n'a pas grand intérêt d'un point de vue arithmétique, cela dit il est très pratique pour effectuer une conversion de type en numérique :

```
var a = '3' ;  
console.log(typeof +a) ;
```

Ce qui équivaut à un `parseInt` ou `Number()`.

# Les opérateurs

## Opérateurs logiques

Le NON logique « ! » renvoie la valeur booléenne inverse :

```
var a = true ; console.log( !a) // false  
var a = undefined ; console.log( !a) // true
```

Un usage fréquent de cet opérateur est son double emploi. De cette manière, on convertit toute valeur en valeur strictement booléenne :

```
console.log( !!'test') // true  
console.log( !!5) // true  
console.log( !!0) // false
```

# Les opérateurs

## Opérateurs logiques

Le OU logique « || » :

Cet opérateur binaire permet de retourner la valeur du premier opérande si l'opération de transtypage est vraie sinon la deuxième.

```
console.log( 'test' || 0 ) // test  
console.log( "" || 'test2' ) // test2
```

Cela permet le plus souvent de définir une valeur par défaut à une variable.

On peut également l'utiliser pour l'exécution conditionnelle d'une fonction : `a > b || a / b`

# Les opérateurs

## Opérateurs logiques

Le ET logique « **&&** » :

Il retourne son premier opérande si celui-ci peut être converti en false, sinon renvoie le deuxième.

```
console.log( 'test' && 0 ) // 0
```



## Exercices variables



# Les fonctions



# Introduction

Afin de **limiter la complexité** d'un programme, on encapsule certaines brides de sa logique dans des **fonctions**.

Les fonctions ont aussi d'autres rôles :

- Eviter les répétitions
- Simplifier la lisibilité du code
- Régir la portée des variables
- Servir de base à tous les objets
- Reprendre la main sur l'exécution

# Concept et Déclaration

## Concept

Les **fonctions** sont **un ensemble d'instructions** pour lesquelles on peut définir des paramètres variables appelés **arguments**. Ces instructions effectuent des opérations et retournent optionnellement un résultat.

## Déclaration

Une fonction peut être déclarée grâce au mot-clé « **function** » suivi du nom de la fonction, puis de ses arguments entre parenthèses.

```
function inverse (n) {  
    return -n;  
}
```

### Execution

Une fois déclarée, nous pouvons utiliser la fonction et l'exécuter :

```
inverse(5) ; // retourne -5
```

### Déclarations et exécutions alternatives

```
// par assignation  
var inverse = function (n) {  
    return -n ;  
};
```

```
//Assignation d'une fonction existante  
var inverse2 = inverse;
```

# Les fonctions

## Exécution

### Déclarations et exécutions alternatives

Une fonction peut également être **anonyme**, mais elle n'a d'utilité que si elle exécutée directement, assignée à une valeur ou passée en paramètre d'une autre fonction.

```
(function(n){  
    return -n;  
})(1) ;
```

# Arguments

## Arguments

Les **arguments** sont déclarés par leur nom et séparés par des virgules. Le nombre d'arguments acceptables est limité à la taille maximale d'un tableau. Sans parler de cette limite technique, il est évident qu'un nombre d'arguments trop important est déjà une mauvaise pratique.

Les arguments déclarés sont indicatifs et n'empêcheront pas la fonction de s'exécuter.

```
function concatener(a,b) {  
    return "+a+b ;  
}  
console.log(concatener('a','b')) ; // ab  
console.log(concatener('a')) ; // a undefined
```

# Arguments et retour

## Arguments

On peut également utiliser la variable “arguments” qui est pratique lorsque l’on ne connaît pas le nombre d’arguments en entrée.

```
function concatener(s) {  
  for (var i = 1, j = arguments.length; i < j; i++) {  
    s += " + arguments[i];  
  }  
  return s;  
}
```

## Retour de fonction

Comme vu précédemment, une fonction peut retourner une valeur grâce à **return**.

```
function ajouterUn(a) {  
  return a + 1 ;  
}  
console.log(ajouterUn(2)) ; // 3
```

# Portée et contexte

## Portée et contexte

Lorsqu'une fonction est exécutée, des règles définissent les variables auxquelles les fonctions ont accès. Il y a 3 couches :

### Les variables locales

Une fonction peut naturellement accéder à ses variables, soit provenant des arguments soit déclarer en interne.

```
function a(i) {  
  var j = 2;  
  return i + j;  
}
```

### Les variables de portée

Une fonction déclarée dans une autre fonction peut accéder aux variables locales de la fonction parente et récursivement jusqu'à trouver le contexte global.



# Portée et contexte

```
function a() {  
  var i = 3;  
  function b() {  
    return i + 3;  
  }  
  return b();  
}  
a() // 6
```

## Le contexte d'une fonction

Une fonction peut également accéder à son contexte identifié par le mot-clé **“this”**.  
Par défaut, le contexte d'une fonction est le contexte global.

```
var j = 3;  
function a(i) {  
  return i + this.j;  
}  
a(1) // 4
```

# Portée et contexte

il n'est pas très utile pour une fonction d'avoir pour contexte, le contexte global car celui-ci est accessible de n'importe où. Le contexte d'une fonction peut être substitué de plusieurs manières. D'abord par l'assignation à une propriété d'un objet, on parle alors de **méthode**.

On peut également appliquer un contexte à l'aide de la fonction **bind**

```
var objet2 = {};  
objet2.n = 1; // ici, on assigne la valeur 1 à n du contexte de l'objet2  
function inverse() {  
    return -this.n; // à ce moment this correspond au contexte global  
}  
console.log(inverse.bind(objet2)());
```



[Codingame : Les fonctions](#)  
[Les fonctions- CodeSandbox](#)



# Structures de contrôle

## Les structures de contrôle

# Les conditions

### If ... else

La structure de contrôle bien connue est le if...else...

```
if (condition) {  
    // Si vrai exécution de ce code  
} else {  
    // Sinon exécution de ce code  
}
```

```
if (condition) {  
    // Si vrai exécution de ce code  
} else if (condition2) {  
    // Sinon si vrai exécution de ce code  
} else {  
    // Sinon exécution de ce code  
}
```

# Les structures de contrôle

## Les conditions

### Switch

La structure de contrôle **switch...case** est utile pour éviter l'utilisation du **if...else** avec trop de cas

```
switch (a) {  
    case 0:  
        // exécution du code  
        break;  
    case 1:  
        // exécution du code  
        break;  
    default:  
        // exécution du code  
}
```

Le switch évalue l'expression et exécute l'instruction dont la clause « case » est égale au résultat de cette évaluation. La clause « default » permet d'exécuter une instruction par défaut.

Pour éviter que la clause ne s'exécute systématiquement, il faut « sortir du switch en ajoutant un « break » à la fin de chaque case.

## Les structures de contrôle

# Les boucles

Les différentes structures de contrôles de type boucle permettent d'exécuter les mêmes instructions de manière itérative avec chacune des quelques spécificités.

### **for(exp1 ;exp2 ;exp3) instruction**

La boucle for s'initialise avec la 1ère expression, exécute l'instruction, évalue la 2ème expression, si exp2 égale à false la boucle s'arrête sinon exp3 est évalué et instruction s'exécute de nouveau.

```
for (var i = 0; i < 10; i += 1) {  
  console.log(i);  
} // 0 1 2 3 4 5 6 7 8 9
```



## Les structures de contrôle

# Les boucles

### while(expression) instruction

Ici, tant que l'évaluation de « expression » équivaut à true, instruction est exécutée :

```
var i = 0;
while (i < 10) {
  console.log(i++);
} // 0 1 2 3 4 5 6 7 8 9
```

### Do instruction while(expression)

La boucle **do...while** est semblable au while mis à part le fait que l'instruction est exécutée avant l'évaluation de l'expression.

```
let i = 1;
do {
  console.log(i++);
} while (i < 5);
```

donc le code de la boucle sera toujours exécuté au moins une fois.

# Les boucles

### for(var identifiant in objet) instruction

Cette boucle itère sur les noms de propriété d'un objet.

```
for (var props in this) {  
  console.log(prop + '=' + this[prop]);  
}
```

### Break et continue

On peut décider à tout moment de vouloir sortir d'une boucle ou d'une itération.

Pour cela, on a deux mots-clés à disposition :

L'instruction **continue** met fin à l'itération et passe à la suivante.

L'instruction **break** permet de sortir d'une boucle.



[Exercices tests conditionnels](#)  
[structures de controle – CodeSandbox](#)



# Objets et tableaux

## Objets et tableaux

# Les objets

### Qu'est qu'un objet ?

Un objet est la traduction d'un **concept** qui peut être concret ou abstrait (une voiture ou un événement d'interface). Au lieu de subdiviser son programme en fonctions, le développeur peut créer des objets qui encapsulent des données et les fonctionnalités qui lui sont rattachées.

```
var carte = {};  
carte.couleur = 'rouge';
```

### Propriétés

Pour pouvoir regrouper les données qui le composent, un objet peut stocker des variables appelées **propriétés ou attributs**. Ces propriétés peuvent être des valeurs primitives ou des références à d'autres objets.

```
var jeu = {};  
jeu.cartes = [Object.create(carte), Object.create(carte), Object.create(carte)];  
console.log(jeu);
```

# Objets et tableaux

## Les objets

### Méthodes

Une fonction est un objet, son nom une variable qui contient une référence à cet objet. On peut donc assigner à une propriété d'un objet, une fonction

```
var carte = {};  
carte.couleur = 'rouge';  
carte.nbReturn = 0;  
carte.retourner = function() {  
  this.nbReturn ++;  
}  
  
carte.retourner();  
console.log('nbreturn', carte.nbReturn); // nbReturn 1
```

# Les tableaux

### Les propriétés

`Array.length` : permet de connaître la longueur d'un tableau

```
console.log(myArray.length);
```

Chaque élément d'un tableau est accessible par un **indice numérique** représentant sa position.

```
var myArray = ['a', 5, {}];  
console.log(myArray[1]); // 5
```

On constate que la numérotation démarre à **0**.

# Les tableaux

## Les méthodes

Il existe une multitude de méthodes pour manipuler les tableaux.

**Array.concat** : concaténation de tableaux

```
var myArray = ['a', 5, {}];  
var myArray2 = [6, 'b'];  
console.log(myArray.concat( myArray2));
```

**Array.join** : regroupe les éléments d'un tableau en chaîne.

```
var myArray = ['a', 5, 'b'];  
console.log(myArray.join()); // a,5,b
```

Vous trouverez toute la documentation sur le site :

[https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets\\_globaux/Array](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array)





[Codingame : tableaux](#)

[Tableaux et objets – CodeSandbox](#)





# MERCI