JavaScript
EcmaScript 6 – ES 6 – ES 2015

Edition février 2025

JAVASCRIPT MODERNE :

ECMASCRIPT-2015

Introduction

Tester le JavaScript : VS Code et Navigateur !

- VS Code, c'est la bonne solution !
- On peut tester en ligne, mais ce n'est pas une bonne idée !
  - ⇨ https://codepen.io/wlabarron/pen/yYrPRQ?editors=0011

Ré-introduction

- Le JavaScript est un des trois grands : JavaScript, Python, Java.
- Une connaissance approfondie de cette technologie est une **compétence importante pour les développeurs Web**.
- Créé en **1995** par Brendan Eich, un ingénieur de **Netscape**.
- Rapidement soumis à l'Ecma International, organisation de normalisation européenne =>première édition du **standard ECMAScript en 1997** (ES-1 = ES-1997).

- **ES6=ES2015 : sixième édition** qui apporte des nouveautés majeures, publié en juin 2015.
- **Conçu pour s'exécuter comme un langage de script dans un environnement hôte qui servent d'interpréteur JS** : c'est à cet environnement de fournir des mécanismes de communication avec le monde extérieur.
- **Les navigateurs** sont les premiers interpréteurs JS les plus communs.
- **Node.js** est le 2è interpréteur JS commun.
- **D'autres interpréteurs JS existent** :
  - dans Adobe Acrobat, Photoshop, les images SVG, le moteur de widgets de Yahoo!,
  - les bases de données NoSQL telles que Apache CouchDB,
  - les ordinateurs embarqués
  - des environnements de bureaux comme GNOME (interface graphique très populaire des systèmes d'exploitation GNU/Linux).
  - etc.

JavaScript d'origine

Principes

- JavaScript est un langage dynamique multi-paradigmes :
  - ⇨ procédural,
  - ⇨ objet,
  - ⇨ événementiel.
- Il dispose de différents types, opérateurs, objets natifs et méthodes.
- Sa syntaxe s'inspire des langages Java et C.
- Le JavaScript d'origine n'a pas de classes.
  - ⇨ Avec ES-2015, on peut écrire nos classes de POO : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Operators/class
- **Spécificité du JavaScript : les fonctions sont des objets**.
  - ⇨ On peut donc stocker ces fonctions dans des variables et les transmettre comme n'importe quel objet.

Les types :

Principaux types

- Number
- String
- Boolean
- Object : c'est l'objet JSON : {clé : valeur}
- Function
- Array
- Date
- RegExp : correspondances d'un texte avec un motif donné.
- null : absence de valeur explicitement donnée
- undefined : pas de valeur, ni de valeur null.
- Symbol : un peu comme une constante.
- Error : pour gérer les exceptions :

Tous les objets globaux de JavaScript

- https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux

Récupérer le type

- typeof : renvoie le type de la variable.

Les nombres

- Format IEEE 754 en double précision 64 bits
- Attention aux calculs mathématiques !
- Math.sin(), Math.PI, parseInt(), etc.
- NaN, Infinity, isFinite

Les chaines de caractères

- "bonjour".length; // 7
- "bonjour".charAt(0); // "b"
- "coucou tout le monde".replace("coucou", "bonjour"); // "bonjour tout le monde"
- "bonjour".toUpperCase(); // "BONJOUR"
- Toutes les méthodes de manipulations de chaines : méthodes

Booléen

- true ou false.
- Boolean(expression) convertit une expression en booléen
- false, 0, chaîne vide (""), NaN, null et undefined valent false. Tout le reste vaut true.
- Boolean(‘coucou’) est true

Symbole

- Apparu avec ES6/2015.
- const ANIMAL_DOG = Symbol();
- const ANIMAL_CAT = Symbol();
- Un symbole est une valeur unique et non modifiable.
- Un symbole peut donc s’utiliser comme clé sans risque de collision.
- Exemples : https://putaindecode.io/articles/es6-es2015-les-symboles/

Déclaration des variables – var et portée avant le ES6

Principes

- On créer une variable en lui donnant une valeur.
- La première fois, on parle de déclaration.

Comportement ES-2015 : let et const

- En ES-2015, les variables sont toujours déclarées soit avec « let », soit avec « const »

Comportement classique en JavaScript : var et rien -> à oublier !!!

- On peut précéder la création du mot clé « var » ou pas.
- Toutes les variables créées **sans le mot clé « var »** sont des **variables globales** : elles sont utilisables **partout dans le code après leur déclaration, même dans les fonctions**.
- Toutes les variables déclarées **avec un « var » dans les fonctions** sont **locales aux fonctions** : elles ne sont pas utilisables en dehors des fonctions.

⇨

```
ga='ga';
var gb='gb'
gc='gc';
f();
gd='gd';
console.log(fa)
console.log(fb)      // fb n'existe pas

function f(){
      fa='fa'
      var fb='fb'      // fb est locale à la fonction
      console.log(ga)
      console.log(gb)
      console.log(gc)
//    console.log(gd) // gd est déclaré après l'appel à la fonction
}
```

Les opérateurs

- x += 5;
- x = x + 5;
- x++ ;
- "coucou" + " monde" // "coucou monde"
- "3" + 4 + 5; // "345"
- 3 + 4 + "5"; // "75"
- 4 + ""; // "4" : l'ajout d'une chaîne vide convertit en une chaîne.
- Les comparaisons en JavaScript se font à l'aide des opérateurs <, >, <= et >=
- 123 == "123"; // true, comparaison non typée par défaut
- 1 == true; // true
- 123 === "123"; //false, comparaison typée à 3 =
- 1 === true; // false
- && , || et logique de court-circuit.

Structures de contrôle

Tests

- if, else,
- switch… case… break… default,

Boucles

- while, do… while
- for (i=0 ;i<tab.length ; i++)
- for (value of tab)
- for (propriete in objet)
- break et continue

Les tableaux

- var tab = ["chien", "chat", "poule"];
- a.length; // 3

Les objets

```
objet = {}

obj = {
  nom: "Carotte",
  prix: 10,
  details: {
    couleur: "orange",
    taille: 12
  },
  toString :function(){
    return this.nom+" "+this.details.couleur+" taille "
        + this.details.taille+" : prix = "+this.prix
  }
};
console.log(obj.details.couleur)
console.log(obj["details"]["taille"])
console.log(obj.toString())
```

Les fonctions

- possibilité de fonctions récursives : RAS
- possibilité de fonctions internes : RAS
- Valeur par défaut des arguments.
- Tableau des arguments.
- Nombre indéfini d'arguments : « paramètres du reste ».
- Méthodes « apply » et « call » : apply permet de passer les arguments dans un tableau.

Les closures (fermetures) : compliqué et peu utilisé

Exemple

- Les closures sont un usage inhabituel assez inutile en programmation objet.

Exemple

```
function creerFonction(a, b) {
  var nom = "Mozilla";
  function afficheNom(b) {
    console.log(b+' '+nom);
  }
  return afficheNom;
}

var maFonction = creerFonction("Mozilla");
maFonction("coucou"); // coucou Mozilla
maFonction("super");  // super Mozilla
```

Principes

- On crée un objet fonction avec une première série de paramètres.
- Cet objet peut être utilisé avec une deuxième série de paramètres.
- C'est très étrange !
- **Ces fonctions se « souviennent » de l'environnement dans lequel elles ont été créées (on dit aussi que la fonction capture son « environnement »).**
- Précisions ici.

POO en JavaScript classique (on peut oublier tout car c'est repris en ES-2015

- Il y a plusieurs façons de faire de la POO en JavaScript classique.

Version 1

```
function Personne(prenom, nom) {
  this.prenom = prenom;
  this.nom = nom;
  this.nomComplet = function() {
    return this.prenom + ' ' + this.nom;
  }
  this.nomCompletInverse = function() {
    return this.nom + ' ' + this.prenom;
  }
}
var s = new Personne("Simon", "Willison");
console.log(s.nomComplet())
console.log(s.nomCompletInverse())
```

⇨ Défaut : chaque objet porte les fonctions. Elles ne sont pas partagées.

Version 2 : avec prototype

```
function Personne(prenom, nom) {
  this.prenom = prenom;
  this.nom = nom;
}
Personne.prototype.nomComplet = function() {
  return this.prenom + ' ' + this.nom;
}
Personne.prototype.nomCompletInverse = function nomCompletInverse()
{
  return this.nom + ' ' + this.prenom;
}
var s = new Personne("Simon", "Willison");
console.log(s.nomComplet())         //  Simon Willison
console.log(s.nomCompletInverse())  //  Willison Simon
```

⇨ Le prototype permet d'ajouter la fonction et qu'elle soit partagée.

- ***ajout d'une fonction avec prototype en cours de code***

```
Personne.prototype.nomEnMajuscules = function() {
  return this.nom.toUpperCase()
}
console.log(s.nomEnMajuscules());  //  WILLISON
```

- ***ajout d'une fonction avec prototype sur des classes natives***

```
var s = "Simon"; // s est une String
String.prototype.inverse = function() {
  var r = "";
  for (var i = this.length - 1; i >= 0; i--) {
    r += this[i];
  }
  return r;
}
console.log(s.inverse()); // "nomiS"
```

⇨ On pourrait aussi redéfinir la méthode « toString » qui est déjà présente sur les chaines de caractères.

Version 3 : avec la fonction static Object.create

- ***Classe : création d'une variable de type structure qui sert de prototype***

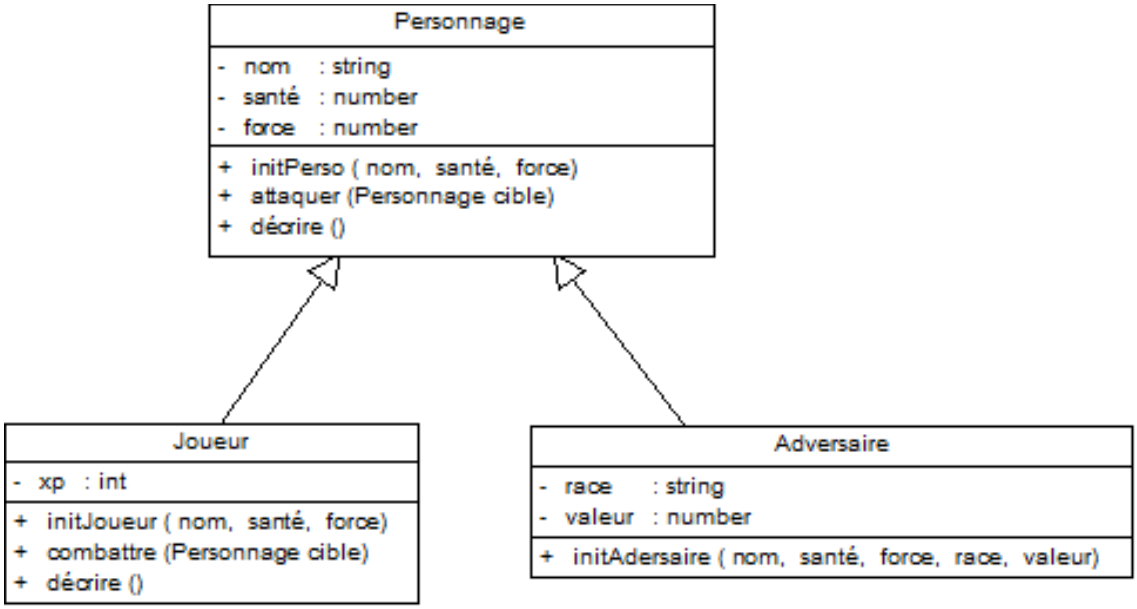
```
var Personnage = {
   init: function (nom, sante, force) {
      this.nom = nom;
      this.sante = sante;
      this.force = force;
      this.xp = 0;
   },

   toString: function () { // retourne la description à afficher
      return this.nom + " a " + this.sante +
           "points de vie, " + this.force +
           " en force et " + this.xp + " points d'expérience";
    }
};
```

- ***Objet : création d'un objet et utilisation de l'objet : fonction static Object.create***

```
var perso1 = Object.create(Personnage);
perso1.init("Aurora", 150, 25);
console.log(perso1.toString());
```

- ***Héritage :***




```
// On crée le prototype Personnage
var Personnage = {
        initPerso: function (nom, sante, force) {
            this.nom = nom;
            this.sante = sante;
            this.force = force;
        }
};

//on crée le prototype Joueur
var Joueur = Object.create(Personnage);      // Prototype du Joueur

// on ajoute la function initJoueur au prototype
Joueur.initJoueur = function (nom, sante, force) {
    this.initPerso(nom, sante, force);          // appelle initPerso
    this.xp = 0;                                // init la partie joueur
};

// on ajoute la function décrire au prototype
Joueur.decrire = function () {
    var description = this.nom + " a " + this.sante + " points de vie, " + this.force + "
       en force et " + this.xp + " points d'expérience";
    return description;
};

//idem avec Adversaire
```

Array.prototype – méthodes filter, every, some (plus très utile dans ES-2015)

- Array utilise la technique du prototype :
- https://developer.mozilla.org/fr/search?q=prototype
- Associé au prototype on trouve de nombreuses méthodes qu'on peut redéfinir, comme par exemple :
  - ⇨ **filter()**
  - ⇨ **every()**
  - ⇨ **some()**

JavaScript moderne : EcmaScript-2015

Bibliographie ES-2015

Réintroduction à JavaScript

https://developer.mozilla.org/fr/docs/Web/JavaScript/Une_réintroduction_à_JavaScript

ES-6 – ES-2015

- http://ccoenraets.github.io/es6-tutorial/
- https://apprendre-a-coder.com/es6/

L'essentiel de ES-2015

- ES-2015 = ES-6 = ES6/2015 : une révolution pour JavaScript.
- C'est le JavaScript moderne, comme le HTML-5, le CSS-3, le Python-3, le PHP-7 (décembre 2015).
- https://gist.github.com/gaearon/683e676101005de0add59e8bb345340c

- L'essentiel des modifications de ES-2025 se résument aux 5 points ci-dessous :
  - ⇨ **let** et **const** : mieux contrôler ses variables. On définit les variables avec des let et des const.
  - ⇨ **fonction fléchée** : arrow function. C'est une nouvelle syntaxe qui rend le code plus lisible dans certains cas, et qui permet un accès facilité au « this » des objets.
  - ⇨ **Déstructuration** : facilité d'écriture pour récupérer les composants d'un objet.
  - ⇨ **Classe** : on peut créer des classes à peu près comme en Python.
  - ⇨ **Le mode strict** : on force une erreur si la syntaxe ES-2015 n'est pas respectée.

let

Principes

- let : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/let

Portée

- let permet de déclarer une variable dont la portée est celle du bloc courant
  - ⇨ Une constante déclarée dans une fonction n'est visible que dans la fonction.

Utilité

- Avoir des variables locales explicites.

Usage

- On va déclarer toutes les variables en let : une variable est locale par défaut.
  - ⇨ On évite les variables globales.

Variables globales : rappels

- D'une façon générale, il vaut mieux éviter les globales.
- Les variables globales de niveau fichier (ou module) peuvent être déclarées avec un var ou sans mot clé.
- Les variables globales qui apparaissent dans les fonctions sont déclarées sans mot clé. Elles peuvent être utile pour conserver un état pour un retour dans la fonction à condition de ne pas les utiliser en dehors de la fonction.

const

Principes

- **const** : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/const
- **Pour un non objet** : on ne peut plus ni la redéclarer, ni modifier sa valeur.
  - ⇨ C'est une constante classique.
- **Pour un objet** (donc les tableaux) : on ne peut pas le redéclarer, mais on peut modifier son contenu ;
  - const obj={ … }
  - const tab=[]

Portée

- const permet de déclarer une variable dont la portée est celle du bloc courant
  - ⇨ Une constante déclarée dans une fonction n'est visible que dans la fonction.

Utilité

- Quand un objet est déclaré en constante, on ne peut plus le redéclarer, donc le redéfinir.
  - ⇨ On ne peut plus le transformer en entier, le passer à null, etc.
  - ⇨ C'est une protection syntaxique

Usage

- On va déclarer tous les objets en constante !

fonction fléchée-1: fat arrow : =>

https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486

De :

```
const people=[], adults = [], minors = []
people[0]={name:"adulte", age:30}; people[1]={name:"enfant",
age:10}

people.forEach(function (person) {
  if (person.age >= 18) {
    adults.push(person)
  } else {
    minors.push(person)
  }
})
console.log(adults)
console.log(minors)
```

À :

```
people.forEach((person) => {
  if (person.age >= 18) {
    adults.push(person)
  } else {
    minors.push(person)
  }
})
```

Ça n'est qu'une question de syntaxe : l'écriture est plus compacte.

fonction fléchée-2 : fat arrow : =>

https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486

De :

```
tab = people.map(function (person) {
  return person.name
})
```

À :

```
tab = people.map(person => person.name)
```

L'écriture est encore plus compacte : on écrit directement ce qui est retourné derrière le =>

https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486

- map : la fonction map() retourne un tableau avec les valeurs passées en paramètres.
- https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/map

this et =>

- En JS classique, le this dans une fonction fait référence à l'objet qui appelle la fonction.
- Dans une fonction anonyme classique, le this fait référence à l'objet global (la fenêtre).
- Dans les fonctions fléchées, le this fait référence à l'objet le plus proche : c'est beaucoup plus pratique.
- Notez dans le code ci-dessous que le premier forEach ne permet pas l'accès à « name » :

```
const name = 'Extérieur'

const test = {
  name: 'Intérieur',
  testerThis () {
    let tab=[1]

    console.log("IN : testerThis")
    console.log(this)                // this c'est l'objet test
    console.log(this.name)

    tab.forEach(function(elt) {       // 1 seul élément dans tab
      console.log("TAB : forEach - name n'existe pas")
      console.log(this)              // this c'est la fenêtre
      console.log(elt+':'+this.name)   // name n'existe pas
    })

    tab.forEach((elt) => {            // 1 seul élément dans tab
      console.log("TAB => : forEach")
      console.log(this)              // this c'est l'objet test
      console.log(elt+':'+this.name)
    }, 0)
  }
}

test.testerThis ()
```

déstructuration

https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718497

- La déstructuration nous permet de récupérer plus facilement plusieurs informations au sein d'un objet ou d'un tableau.

Sur un objet

- A l'ancienne :

```
personne = {firstName:'bertrand', lastName:'liaudet'}
const firstName = personne.firstName
const lastName = personne.lastName
console.log(firstName, lastName)
```

- Syntaxe ES6

```
personne=firstName:'bertrand', lastName:'liaudet'}
const { firstName, lastName } = personne
console.log(firstName, lastName)
```

Sur un tableau

- A l'ancienne sur un tableau

```
let fullName='Bertrand Liaudet'
const names = fullName.split(' ')
const firstName = names[0]
const lastName = names[1]
console.log(firstName, lastName)
```

- Syntaxe ES6

```
let fullName='Bertrand Liaudet'
const [firstName, lastName] = fullName.split(' ')
console.log(firstName, lastName)
```

POO et class

Classe de base

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Operators/class

```
// Classe Rectangle
class Rectangle {
  constructor(longueur, largeur) {
    this.longueur = longueur;
    this.largeur = largeur;
  }

  // Méthode pour calculer l'aire du rectangle
  aire() {
    return this.longueur * this.largeur;
  }

  // Méthode pour calculer le périmètre du rectangle
  perimetre() {
    return 2 * (this.longueur + this.largeur);
  }

  // Méthode pour afficher les caractéristiques du rectangle
  afficher() {
    console.log(`Rectangle - longueur: ${this. longueur} - largeur: ${this. largeur}`);
    console.log(`Aire: ${this.aire()}`);
    console.log(`Périmètre: ${this.perimetre()}`);
  }
}
```

Utilisation de base

```
// Exemple d'utilisation
let rectangle = new Rectangle(10, 5);
rectangle.afficher(); // Affiche les informations du rectangle
```

Héritage de base

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes/extends

```
// Classe Carré qui hérite de Rectangle
class Carre extends Rectangle {
  constructor(coté) {
    // Appel du constructeur de la classe parente (Rectangle) : en premier
    super(coté, coté);
    // suite :
    this.coté = coté
  }

  // Méthode pour afficher les caractéristiques du carré
  afficher() {
    console.log(`Carré - Côté: ${this.coté}`);
    console.log(`Aire: ${this.aire()}`);
    console.log(`Périmètre: ${this.perimetre()}`);
  }
}
```

Utilisation de base

```
// Exemple d'utilisation
const carre = new Carre(5);
carre.afficher(); // Affiche les informations du carré
```

Le mode strict

- Il existe un mode laxiste et un mode strict.
- Le mode strict consiste à imposer la norme ES-2015 :

```
"use strict"
```

⇨ https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

- Le mode strict rend explicite certaines erreurs silencieuses. Il améliore ainsi la qualité et la maintenabilité du code.
  ⇨ Par exemple : on ne peut plus utiliser de « var ».