JAVASCRIPT

Guillaume Rodrigues

[NOM DE LA SOCIETE] [Adresse de la société]

Table des matières

JavaScript

Introduction

Qu'est-ce que JavaScript ?

JavaScript est un langage de programmation dynamique qui est principalement utilisé pour créer du contenu interactif sur les pages web. Initialement développé par Netscape en 1995 sous le nom de LiveScript, il a été rebaptisé JavaScript pour tirer parti de la popularité de Java à l'époque. Depuis lors, JavaScript est devenu l'un des piliers du développement web moderne.

Pourquoi utiliser JavaScript ?

- **Interactivité** : JavaScript permet de rendre les pages web interactives. Par exemple, vous pouvez créer des menus déroulants, des carrousels d'images, des formulaires dynamiques et bien plus encore.
- **Exécution côté client** : JavaScript s'exécute dans le navigateur de l'utilisateur, ce qui permet de réduire la charge sur le serveur et d'améliorer la réactivité des applications web.
- **Riche écosystème** : Avec des bibliothèques et frameworks populaires comme React, Angular et Vue.js, ainsi que des frameworks côté serveur comme Node.js, JavaScript offre beaucoup de flexibilité pour le développement full-stack.
- **Communauté active** : La grande communauté de développeurs JavaScript signifie qu'il existe une abondance de ressources, de tutoriels et de soutien disponible pour les débutants comme pour les experts.

Historique et évolution de JavaScript

- 1995 : Brendan Eich développe JavaScript en 10 jours chez Netscape.
- 1996 : Microsoft introduit JScript dans Internet Explorer 3.0.
- 1997 : Le langage est standardisé sous le nom d'ECMAScript par l'ECMA International.
- 2009 : Naissance de Node.js, permettant l'exécution de JavaScript côté serveur.
- 2015 : Publication de la version ECMAScript 6 (ES6/ECMA-2015), apportant des améliorations significatives comme les classes, les modules, et les promesses.

Le typage en JavaScript

JavaScript est un langage à typage dynamique et faible.

- **Typage dynamique** : En JavaScript, le type des variables est déterminé au moment de l'exécution et non à la compilation. Cela signifie que vous pouvez déclarer une variable sans spécifier son type, le type sera défini lorsque vous assignerez une valeur à votre variable.

```
let variable = 42;     // variable est un nombre
variable = "Bonjour"; // variable est maintenant une chaîne de
caractères
```

- **Typage faible** : En JavaScript, le typage est faible. Cela signifie qu'une variable peut changer de type au cours de l'exécution du code. Les conversions de types sont souvent effectuées implicitement, ce qui peut mener à des résultats inattendus.

```
let resultat = "5" + 2;  // resultat sera "52" car '2' est converti
en chaîne de caractères
let somme = "5" - 2;     // somme sera 3 car '5' est converti en
nombre
```

Cette nature flexible du typage en JavaScript offre une grande liberté, mais elle nécessite également une attention particulière pour éviter les erreurs de type.

Syntaxe de base

Dans le cas où votre code JS est dans un fichier séparé (la majeure partie du temps), l'extension du fichier doit être « .js ».

Les délimiteurs

Les délimiteurs sont utilisés pour structurer le code JavaScript et séparer les différentes instructions.

- **Point-virgule (;)** : Utilisé pour terminer une instruction. Bien que facultatif dans de nombreux cas grâce à l'ASI (Automatic Semicolon Insertion), il est recommandé de l'utiliser pour éviter les erreurs inattendues.
- **Accolades ({})** : Utilisées pour délimiter les blocs de code, comme les fonctions, les boucles, et les structures conditionnelles.

```
if (a > b) {
    console.log("a est plus grand que b");
}
```

- **Parenthèses (())** : Utilisées pour entourer les paramètres des fonctions et les conditions des structures de contrôle.

```
function addition(x, y) {
    return x + y;
}

while (a < b) {
    a++;
}
```

- **Crochets ([])** : Utilisés pour créer et accéder aux éléments des tableaux.

```
let tableau = [1, 2, 3];
console.log(tableau[0]); // Affiche 1
```

Variables

Déclaration et assignation des variables

Pour déclarer une **variable**, nous utilisons les mots clé « **let** » et « **var** ».

var : Déclare une variable avec une portée fonctionnelle ou globale.

let : Déclare une variable avec une **portée de bloc**.

Étant donné sa portée globale, l'utilisation de « **var** » peut induire des effets de bords imprévus (par exemple si on utilise une variable du même nom dans une fonction). De plus, la plupart des langages de haut niveau prennent en compte la portée des variables. Pour ces raisons, il est conseillé de privilégier l'utilisation du mot clé « **let** ».

Pour déclarer une **constante** (une variable à qui on ne peut assigner une valeur que lors de la déclaration), nous utiliserons le mot-clé « **const** ».

Pour **assigner une valeur** à une variable, comme dans de nombreux langages de programmation, nous utiliserons le symbole « **=** ».

Le nom d'une variable doit respecter les règles suivantes :

- Contenir uniquement des caractères alphanumériques non-accentués (a-zA-Z0-9) ainsi que des underscore « _ ».
- Commencer par un caractère alphabétique (a-zA-Z)
- Bonnes pratiques (conseillé mais non-obligatoire)
    - Le nom des variables doit être le plus explicite possible pour faciliter la compréhension du code en limitant la quantité de commentaires nécessaire.
    - Le nom des variables doit être écrit en camelCase
    - Le nom des constantes doit être écrit en UPPERCASE

```
// Déclaration de la variable "maVariable"
let maVariable;
var maVariable; // Usage déconseillé

// Pour définir une constante (dont la valeur n'a pas vocation à être modifié) on utilise le mot clé "const"
const MA_CONSTANTE = "UNE VALEUR";

// Assignation de la valeur "Une chaine de caractères" à la variable maVariable. maVariable est maintenant de type "String"
maVariable = "Une chaine de caractères";

// Maintenant, on écrase la valeur précédente, et on assigne maintenant le nombre 2 (notez l'absence de quotes) à maVariable. Son type est maintenant Number
maVariable = 2;

maVariable = ["el1", 3, "toto"]; // C'est maintenant un tableau
```

Types de données

JavaScript supporte plusieurs types de données :

- **Nombres** : Représente les valeurs numériques.

```
let x = 5;
let y = 3.14;
```

- **Chaînes de caractères** : Représente les séquences de caractères, entourées de guillemets simples ou doubles.

```
let salut = "Bonjour";
let reponse = 'Oui';
```

- **Booléens** : Représente les valeurs « true » ou « false ».

```
let estVrai = true;
let estFaux = false;
```

- **Objets** : Collections de paires clé/valeur.

```
let personne = {
    prenom: "John",
    nom: "Doe",
    age: 25
};
```

- **Tableaux** : Listes ordonnées de valeurs.

```
let fruits = ["Pomme", "Banane", "Cerise"];
```

- **Null** : Représente une absence intentionnelle de valeur.

```
let vide = null;
```

- **Undefined** : Représente une variable déclarée mais non initialisée.

Opérateurs

1. Opérateurs arithmétiques : « + », « - » , « * », « / », « % » (modulo)
   Ces opérateurs permettent de réaliser opérations et calculs.

```
let somme = 5 + 3; // 8
let produit = 4 * 2; // 8
```

2. Opérateurs d'assignation : « = », « += », « -= », « *= », « /= », « %= », « ++ », « -- »

```
let a = 5;
a += 2; // a vaut maintenant 7
a++; // a vaut maintenant 8
a /= 2; //a vaut maintenant 4
```

3. Opérateurs de comparaison : « == », « != », « === », « !== », « > », « < », « >= », « <= »

```
let egal = (5 == "5"); // true (on compare uniquement la valeur)
let strictEgal = (5 === "5"); // false (on compare la valeur et le
type)
```

4. Opérateurs logiques : « && » (et), « || » (ou), « ! » (non)

```
let et = (true && false); // false
let ou = (true || false); // true
let non = !true; // false
```

Fonctions

Les fonctions sont des blocs de code réutilisables qui exécutent une tâche spécifique.

Les fonctions sont définies par leur nom (qui doit respecter les mêmes règles que les variables).

Les fonctions peuvent nécessiter des paramètres en entrée.

Les fonctions peuvent retourner une valeur.

Pour déclarer une fonction, nous utilisons généralement le mot clé « function ».

- Déclaration de fonction

```
function saluer() {
    console.log("Bonjour !");
}
```

- Expression de fonction

```
let saluer = function() {
    console.log("Bonjour !");
};
```

- Fonction fléchée (EcmaScript 6 : ES6)

```
let saluer = () => {
    console.log("Bonjour !");
};
```

- Appel d'une fonction

```
saluer();
```

- Quelques exemples, notamment avec des paramètres possédant une valeur par défaut

```
// Fonction sans paramètre
function saluer() {
    console.log("Bonjour !");
}

// La fonction attend un paramètre 'nom'
function saluer(nom) {
    console.log("Bonjour " + nom + "!");
}

// On peut donner des valeurs par défaut à nos paramètres de
fonction.
function saluer(nom = "Bryan") {
    console.log("Bonjour " + nom + "!");
}

saluer("Guillaume"); // La fonction me retournera "Bonjour Guillaume
!"
saluer(); //La fonction me retournera "Bonjour Bryan !"
```

Contrôle de flux

Structures conditionnelles : « if », « else if », « else »

```
if (a > b) {
    console.log("a est plus grand que b");
} else {
    console.log("a n'est pas plus grand que b");
}
```

Switch

```
switch (a) {
    case 1:
        console.log("a vaut 1");
        break;
    case 2:
        console.log("a vaut 2");
        break;
    default:
        console.log("a a une autre valeur");
}
```

Structures itératives (boucles) : « for », « while », « do...while »

```
for (let i = 0; i < 5; i++) {
    console.log(i); // Affiche 0 à 4
}

let j = 0;
while (j < 5) {
    console.log(j);
    j++;
}

let k = 0;
do {
    console.log(k);
    k++;
} while (k < 5);
```

Programmation orientée objet

En JavaScript, la plupart des valeurs manipulées sont des **objets**, qu'ils proviennent des fonctionnalités natives du langage, comme les tableaux, ou qu'ils soient fournis par les API du navigateur. Il est aussi possible de créer ses propres objets qui contiendront des propriétés avec des données ou des fonctions.

Un objet est un type de données complexe car il va nous permettre de renseigner énormément d'informations.

Vocabulaire

- **Objet** : Comme dans la vie courante, un objet est quelque chose que l'on peut définir et qui va pouvoir réaliser des choses. Pour l'exemple, imaginons que l'on travaille sur un objet **Voiture**.
- **Propriétés/Attributs** : Les propriétés ou attributs sont les caractéristiques qui définissent notre objet. Dans le cadre de notre exemple, la Voiture pourrait avoir des propriétés telles que : couleur, marque, modèle, puissance.
  Pour accéder à une propriété, on peut utiliser les notations suivantes :

```
let obj = {propriete1: "valeur", propriete2: 3};
console.log(obj.propriete1); // Affiche "valeur"
obj.propriete1 = "Nouvelle valeur"; //On modifie la valeur de la propriété
console.log(obj.propriete1); // Affiche "Nouvelle valeur"

// On peut aussi accéder aux propriétés avec une syntaxe similaire à celle des tableaux :
console.log(obj['propriete2']); // Affiche 3
obj['propriete2'] = -12; // On modifie la valeur de la propriete2
console.log(obj['propriete2']); // Affiche -12
```

- **Méthodes** : Ce sont les actions que l'objet peut réaliser. En JavaScript, les méthodes sont des fonctions qui ne pourront être réalisées que par l'objet. Dans le cas de notre voiture, les méthodes pourraient être : « accélérer », « démarrer », « s'arrêter ».

```
// On créé un objet « voiture » qui a pour propriétés : une marque, une couleur, et qui a pour méthode une fonction « démarrer »
let voiture = {
  marque: 'Toyota',
  couleur: 'rouge',
  demarrer: function() {
    console.log('La voiture démarre');
  }
};
// Pour accéder à la propriété d'un objet, on utilise la syntaxe nomDeLObjet.nomDeLaPropriete
// On peut aussi y accéder par la syntaxe nomDeLObjet[nomDeLaPropriete]
console.log(voiture.marque); // Toyota
voiture.demarrer(); // La voiture démarre
```

- **Classe** : Une classe est le modèle d'un objet. Elle va nous permettre de définir les propriétés et méthodes de l'objet.
  Par convention, un nom de classe sera écrit en PascalCase et devra respecter les mêmes contraintes que les variables et fonctions (pas de caractères accentués, pas de caractère spécial sauf l'underscore « _ »).
- **Constructeur** : C'est une méthode d'une classe qui va nous permettre de créer une instance de classe.
- **Instance** : C'est un objet construit selon le modèle défini par une classe. Il s'agit de l'exemplaire d'une classe. On créé l'instance d'une classe en faisant « **new MaClasse()** ». En faisant ça, on appelle le constructeur de la classe.
- **« this »** : Le mot clé « this » est utilisé dans les classes. Il définit l'instance de l'objet sur laquelle sont exécutées les méthodes, ou sur laquelle nous souhaitons accéder à la valeur des propriétés.

```
class Voiture {
  constructor(marque, couleur) {
    this.marque = marque;
    this.couleur = couleur;
  }

  demarrer() {
    console.log('La voiture démarre');
  }
}

let maVoiture = new Voiture('Toyota', 'rouge');
console.log(maVoiture.marque); // Toyota
maVoiture.demarrer(); // La voiture démarre
```

Concepts de base

Encapsulation

L'**encapsulation** est comme une boîte fermée avec des informations à l'intérieur. Vous ne pouvez accéder à ces informations qu'avec des méthodes spécifiques. Par exemple, le moteur d'une voiture est encapsulé : vous ne pouvez pas y accéder directement, mais vous pouvez démarrer la voiture avec une clé.

En n'exposant que ce qui est nécessaire, cela nous permet d'éviter au maximum les erreurs de développement.

En JavaScript, l'encapsulation est limitée, mais nous verrons que ce concept peut être poussé encore plus loin dans d'autres langages.

```
class Voiture {
  constructor(marque, couleur) {
    this.marque = marque;
    this.couleur = couleur;
    this.vitesse = 0;

    this.accelerer = function() {
      this.vitesse += 10;
      console.log(`La vitesse est maintenant de ${this.vitesse} km/h`);
    };
  }
}
```

Héritage

L'**héritage** permet de créer de nouvelles classes basées sur des classes existantes. Cela favorise la réutilisation du code.

L'**héritage** est comme un enfant qui hérite des traits de ses parents. Une classe enfant hérite des propriétés et des méthodes d'une classe parent.

```
// On créé la classe parente Vehicule
class Vehicule {
  constructor(marque) {
    this.marque = marque;
  }

  demarrer() {
    console.log(`${this.marque} démarre`);
  }
}

// On créé la classe Moto qui hérite de Vehicule.
// On peut remarquer que cette classe ajoute une nouvelle méthode, propre à l'enfant : faireUnWheeling
class Moto extends Vehicule {
  faireUnWheeling() {
    console.log(`${this.marque} fait un wheeling!`);
  }
}

let maMoto = new Moto('Yamaha');
maMoto.demarrer(); // Yamaha démarre
maMoto.faireUnWheeling(); // Yamaha fait un wheeling!
```

Polymorphisme

Le **polymorphisme** signifie "plusieurs formes". Le polymorphisme permet d'utiliser une méthode de différentes manières. C'est la capacité d'utiliser une méthode d'une classe mère dans des classes filles avec des comportements différents.

```
class Animal {
  parler() {
    console.log('L\'animal fait un bruit');
  }
}

class Chien extends Animal {
  parler() {
    console.log('Le chien aboie');
  }
}

class Chat extends Animal {
  parler() {
    console.log('Le chat miaule');
  }
}

let animaux = [new Animal(), new Chien(), new Chat()];
animaux.forEach(animal => animal.parler());
// L'animal fait un bruit
// Le chien aboie
// Le chat miaule
```

Exercices pratiques

1. Créer une classe « Personne » avec des propriétés : « nom », « age » et une méthode « sePresenter » qui affiche un message de présentation.
2. Créer une classe « Etudiant » qui hérite de « Personne » et ajoute une propriété « ecole » et une méthode « etudier ».

Prototypes

JavaScript propose également une façon unique de gérer les objets : les prototypes.

Plutôt que de définir une classe, nous allons définir une fonction qui va définir notre objet. Elle prendra le rôle de constructeur, mais devra définir également les différentes méthodes de l'objet.

```
function Animal(espece, age, poids) {
    // Le fonctionnement ici est très similaire au constructeur. On définit les propriétés de l'objet tout en lui attribuant les valeurs envoyées par l'utilisateur
    this.espece = espece;
    this.age = age;
    this.poids = poids;

    // Dans cette fonction, on peut aussi définir les méthodes de notre objet
    this.manger = function(poidsNourriture) {
        this.poids += poidsNourriture;
        console.log("J'ai mangé, je pèse maintenant "+ this.poids + "kg.");
    };

    this.afficher = function(){
        console.log("Je suis un " + this.espece + ", j'ai " + this.age + " ans et je pèse " + this.poids + "kg.");
    };
}

let animal = new Animal("Chien", 4, 12);

animal.afficher();
// Affichera "Je suis un Chien, j'ai 4 ans et je pèse 12kg.""

animal.manger(1);
// Affichera "J'ai mangé, je pèse maintenant 13kg."
```

On peut aussi définir les méthodes d'un objet en dehors de sa fonction de création. De la même façon, on modifier une méthode d'un prototype existant. Pour cela, il faut accéder au prototype de notre objet.

```
Animal.prototype.seDeplacer = function(duree){
    let poidsPerdu = Math.floor(duree/60);
    this.poids -= poidsPerdu;
    console.log("Je me suis déplacé pendant " + duree + " minutes, je pèse maintenant "+ this.poids + "kg.");
};

animal.seDeplacer(120);
// Affichera "Je me suis déplacé pendant 120 minutes, je pèse maintenant 11kg."
```

Les prototypes vont aussi nous permettre de gérer l'héritage.
Pour cela, nous allons déjà devoir rattacher toutes les méthodes de l'objet parent sur le prototype et non les déclarer directement dans le constructeur.

En effet, nous allons devoir faire hériter notre objet enfant du prototype de son parent pour qu'il hérite des méthodes du parent.

Dans le constructeur, la méthode « call » va nous permettre d'appeler le constructeur du parent (équivalent du mot clé « super » lorsqu'on utilise la POO par le biais de Classes).

Exemple d'implémentation

```
     function Animal(espece, age, poids) {
        // Le fonctionnement ici est très similaire au constructeur. On définit les propriétés de
l'objet tout en lui attribuant les valeurs envoyées par l'utilisateur
        this.espece = espece;
        this.age = age;
        this.poids = poids;
    }

    // Lorsque l'on veut faire de l'héritage, nous sommes obligés de déclarer nos méthdodes directement
sur le prototype. Il faut donc les sortir du constructeur.
    Animal.prototype.manger = function (poidsNourriture) {
        this.poids += poidsNourriture;
        console.log("J'ai mangé, je pèse maintenant " + this.poids + "kg.");
    };

    Animal.prototype.afficher = function () {
        console.log("Je suis un " + this.espece + ", j'ai " + this.age + " ans et je pèse " +
this.poids + "kg.");
    };

    let animal = new Animal("Chat", 4, 12);

    animal.afficher();  // Affichera "Je suis un Chat, j'ai 4 ans et je pèse 12kg.""
    animal.manger(1);   // Affichera "J'ai mangé, je pèse maintenant 13kg."

    function Chien(age, poids, race, nom) {
        // La méthode de protype "call" nous permet de récupérer tous les attributs du prototype Animal
au sein de notre objet Chien. La méthode call attend que l'on envoie l'objet sur lequel on travail. On
récupère l'objet courant avec le mot clé "this"
        Animal.call(this, "Chien", age, poids);

        this.race = race;
        this.nom = nom;

    };

    let dog = new Chien(5, 6, "Beagle", "Snoopy");

    // On ne peut pas utiliser les méthodes afficher et manger car pour l'instant notre objet Chien
n'en hérite pas
    // dog.afficher();
    // dog.manger(1);

    //En revanche on peut accéder aux propriétés
    console.log(dog.age);       // Affiche 5
    console.log(dog.poids);     // Affiche 6
```

```
// Pour pouvoir hériter des méthodes d'Animal sur l'objet Chien, il faut dire que le prototype de notre objet Chien est le même que celui de l'objet Animal
    // Pour cela on utilise la méthde Object.create() pour dupliquer le prototype de Animal et l'assigner dans le proto de Chien
    Chien.prototype = Object.create(Animal.prototype);
    // En revanche, on ne veut pas que les construteurs soient les mêmes. On redéfinit donc le constructeur de Chien
    Animal.prototype.constructor = Animal;

    dog = new Chien(5, 6, "Beagle", "Snoopy");

    // On hérite bien des méthodes du parent
    dog.afficher();     // Affiche "Je suis un Chien, j'ai 5 ans et je pèse 6kg."
    dog.manger(1);      // Affiche "J'ai mangé, je pèse maintenant 7kg."


// On peut maintenant surcharger la méthode afficher pour l'adapter à notre Objet enfant
    Chien.prototype.afficher = function () {
        console.log("Je suis un " + this.espece + " de la race " + this.race + ", je m'appelle " + this.nom + ", j'ai " + this.age + " ans et je pèse " + this.poids + "kg.");
    }

    dog.afficher();
    // Affiche "Je suis un Chien de la race Beagle, je m'appelle Snoopy, j'ai 5 ans et je pèse 7kg."

    //Que se passe-t-il si nous attachons une nouvelle méthode à l'objet Animal ?
    Animal.prototype.dormir = function () {
        console.log(this.espece + " dors.");
    };

    dog.dormir();
    // Affiche "Chien dors."

    // On remarque que les méthodes sont mises à jour à la volée et deviennent donc directement accessibles à toutes les instances déjà créées de l'objet
```

Ressources

L'utilisation de prototypes est propre à JavaScript. Bien que la gestion de l'héritage soit un peu plus complexe que l'héritage classique, sa force est sa souplesse et la possibilité de pouvoir modifier le modèle de certains objets même si nous ne le pouvons pas directement. De plus, c'est une notation qui est assez utilisée.

Objet en JavaScript : https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects

Prototypes Objets en JS : https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Object_prototypes

Héritage prototypique : https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Classes_in_JavaScript

Structures de données en JavaScript

Outre les types primitifs que nous avons vu précédemment (number, string, boolean, ...), le langage JavaScript, comme tout langage de haut niveau, propose également d'autres structures de données. Ces structures ne sont ni plus ni moins que des objets !

Pour une vision globale des types de données en JavaScript, vous pouvez aller voir la documentation officielle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Data_structures

String : les chaines de caractères

Un objet **String** est utilisé afin de représenter et de manipuler une chaîne de caractères.

Vous les avez déjà utilisées, mais voyons quelques exemples afin d'appréhender les méthodes qui peuvent être utiles.

Exemples d'utilisation

```
// On peut créer des chaines de caractères de plusieurs façons. Quelque soit la façon de construire votre chaine, à la fin, vous aurez un objet String donc toutes les méthodes rattachées à cet objet seront utilisables.
// façon la plus simple, on utilise le type primitif
const string1 = "Une chaîne de caractères primitive";
const string2 = "Là encore une valeur de chaîne de caractères primitive";
const string3 = `Et ici aussi`;

//Ou bien on utilise le constructeur de l'objet String
const string4 = new String("Un objet String");

// On peut itérer sur une chaine de caractères de la même façon qu'on le ferait sur un tableau
for (let i = 0; i < string2.length; i++) {
    console.log(string2[i]);    //Affichera en console la chaine de caractères, caractère par caractère
}

//On peut aussi accéder au caractère utf-16 de la chaine, par son index avec la méthode 'charAt()'
console.log(string1.charAt(1));     //Affichera le 2eme caractère de la chaine 1 : 'n'

// La méthode charCodeAt fonctionne de la même façon, mais retourne cette fois le code UTF-16 du caractère (nombre entier)
console.log(string1.charCodeAt(1));     // Retourne 110 : le code UTF-16 du caractère 'n'

// La méthode concat permet de concaténer plusieurs chaines de caractères entre elles.
// On peut aussi utiliser l'opérateur de concaténation '+'
const str1 = 'Hello';
const str2 = 'World';

console.log(str1.concat(' ', str2));
// Expected output: "Hello World"

console.log(str2.concat(', ', str1));   // Equivalent à console.log(str2 + ', ' + str1)
// Expected output: "World, Hello"

// Méthode .includes() : La méthode includes() détermine si une chaîne de caractères est contenue dans une autre et renvoie true ou false selon le cas de figure.
const sentence = 'The quick brown fox jumps over the lazy dog.';
const word = 'fox';
console.log(
  `The word "${word}" ${
    sentence.includes(word) ? 'is' : 'is not'
  } in the sentence`,
);
// Expected output: "The word "fox" is in the sentence"

// Méthode .indexOf() : prend en paramètre un élément (caractère ou chaine de caractères) et renvoie l'index de la 1ère occurence de cet élément dans la chaine
// Il existe aussi la méthode .lastIndexOf() qui fonctionne de façon similaire, mais parcourt la chaine de caractères en sens inverse.
const paragraph = "I think Ruth's dog is cuter than your dog!";

const searchTerm = 'dog';
const indexOfFirst = paragraph.indexOf(searchTerm);

console.log(`The index of the first "${searchTerm}" is ${indexOfFirst}`);
// Expected output: "The index of the first "dog" is 15"

console.log(
  `The index of the second "${searchTerm}" is ${paragraph.indexOf(
    searchTerm,
    indexOfFirst + 1,
  )}`,
);
```

```
// Expected output: "The index of the second "dog" is 38"

// Méthode .trim() : la méthode trim est très utilisée pour nettoyer une chaine de caractères des blancs situés au début ou a la fin de la chaine.
// Les blancs sont tous les caractères d'espacement : espace, tabulation, espace insécable, etc
//En revanche, ça ne touche pas aux espaces présents à l'intérieur de la chaine
// Attention, comme beaucoup des méthodes sur les chaines de caractère, elle considère la chaine comme immuable et ne modifiera donc pas la chaine initiale. Elle retourne simplement la chaine nettoyée
let uneChaineANettoyer = '     Une chaine avec des espaces avant et après       ';
console.log(uneChaineANettoyer.trim());     //Affichera "Une chaine avec des espaces avant et après"

// Méthode replace : permet de chercher le 1er élément correspondant dans la chaine de caractères et le remplacer par un autre.
let chaine = "Les chiens ne font pas des chats, chatperlipopette";
console.log(chaine.replace('chat', 'chien'));   //Affichera "Les chiens ne font pas des chiens, chatperlipopette"

// Il existe aussi la méthode replaceAll qui va remplacer toutes les occurences
console.log(chaine.replaceAll('chat', 'chien'));    //Affichera "Les chiens ne font pas des chiens, chienperlipopette"

//Méthode split() : Prend un caractère séparateur en paramètre. Le split permet de découper la chaine selon ce séparateur et retourner un tableau des éléments ainsi obtenus.
console.log(chaine.split(' ')); //retournera un tableau ["Les","chiens", "ne", "font pas des chats,", "chatperlipopette"]

// Méthode substring :  retourne une sous-chaîne de la chaîne courante, entre un indice de début et un indice de fin.
console.log(chaine.substring(4,10));    // retournera la chaine "chiens"

// Méthodes utilisant des expressions régulières
//Méthodes match() : renvoie le tableau des correspondances entre la chaîne courante et une expression rationnelle.
const paragraph2 = 'The quick brown fox jumps over the lazy dog. It barked.';
const regex = /[A-Z]/g;
const found = paragraph2.match(regex);

console.log(found);
// Expected output: Array ["T", "I"]

// Méthode matchAll :  renvoie un itérateur contenant l'ensemble des correspondances entre une chaîne de caractères d'une part et une expression rationnelle d'autre part (y compris les groupes capturants).
const regexp = /t(e)(st(\d?))/g;
const str = 'test1test2';

const array = [...str.matchAll(regexp)];
console.log(array[0]);
// Expected output: Array ["test1", "e", "st1", "1"]
console.log(array[1]);
// Expected output: Array ["test2", "e", "st2", "2"]
```

Date

Vous verrez qu'en informatique de gestion gérer les dates est toujours quelque chose qui peut être complexe. Heureusement la plupart des langages orienté objet vont nous mettre à disposition différents types d'objets autour des dates afin de nous faciliter le travail.

En JS, nous utiliserons principalement l'objet Date. Cet objet est en réalité un « datetime » car il va stocker à la fois la date et l'heure.

Attention, les dates créées en JS peuvent poser des problèmes à l'analyse car, si elle n'est pas directement précisée, la timezone sera gérée par le navigateur et pourra donc différer en fonction de la configuration du navigateur et/ou de l'OS (operating système) de l'utilisateur.

Le constructeur de l'objet Date peut fonctionner avec différentes entrées :

- Une chaine de caractères d'horodatage :
  - Respectant la norme RFC 2822 IETF sur les horodatages (https://datatracker.ietf.org/doc/html/rfc2822#page-14)
  - Ou au format ISO8601 (https://262.ecma-international.org/11.0/#sec-date.parse)
- Plusieurs paramètres renseignant l'année, le mois, le jour, heure, minute, secondes, millisecondes.

Exemples d'utilisation des différents constructeurs

```
        let date;
        //Nous créeons un nouvel objet date sans envoyer de paramètre au
constructeur. La date créée comme ça aura pour valeur la date et l'heure à
laquelle le construteur est appelé. Et la timezone sera celle du client
        date = new Date();

        // Le constructeur peut aussi prendre des paramètres afin de
définir une date précise (utile lorsqu'on utilise une date stockée en base
de données et qu'on souhaite en créer un objet dans le langage de prog
utilisé)

        // Voici les différents constructeurs qui peuvent être utilisés
pour créer un objet date.
        // Il est important d'aller voir la documentation pour savoir
comment doit être formatée la chaine lorsqu'on créé notre objet date depuis
une date sous forme de chaine de caractères
        // Pour cela, la chaine devra respecter la norme RFC 2822 IETF sur
les horodatages (https://datatracker.ietf.org/doc/html/rfc2822#page-14) ou
le format ISO8601 (https://262.ecma-international.org/11.0/#sec-date.parse)
        /*
        new Date();
        new Date(value);
        new Date(dateString);
        new Date(year, monthIndex);
        new Date(year, monthIndex, day);
        new Date(year, monthIndex, day, hours);
        new Date(year, monthIndex, day, hours, minutes);
        new Date(year, monthIndex, day, hours, minutes, seconds);
        new Date(year, monthIndex, day, hours, minutes, seconds,
milliseconds);
        */

        // Ainsi, pour créer un objet date représentant le 1er janvier 2024
dans la timezone "central european summer time", je pourrais faire
        date = new Date("Mon Jan 1 2024 00:00:00 GMT+0100");
        // Ou
        date = new Date("Jan 1 2024 12:00:00 GMT+0100");
        // ou si on ne veut que la date
        date = new Date("Jan 1 2024");

        // Au format ISO8601
        date = new Date("2024-01-01");
        // DateTime au format ISO8601
        date = new Date("2024-01-01 12:00:00:00");

        // Ou en utilisant les différents constructeurs possibles pour
renseigner l'année, le mois, le jour de la date.
        date = new Date(2024,0);    // La date sera 1er janvier 2024 car
les mois sont indexés à partir de 0
        date = new Date(2024,0,1);    // La date sera la même car les mois
sont indexés à partir de 0 mais les jours sont indexés à partir de 1
        date = new Date(2024,0,1, 12); //1er janvier 2024 a 12h00
        date = new Date(2024,0,1, 12, 30);   //1er janvier 2024 a 12h30
        date = new Date(2024,0,1, 12, 30, 30);   //1er janvier 2024 a 12h30
et 30 secondes
```

Méthodes statiques

Les méthodes statiques ne nécessitent pas d'instance d'objet pour être exécutées.

- **Date.now()** : renvoie le nombre de millisecondes écoulées depuis le 1er janvier 1970 00:00:00 UTC.
- **Date.parse()** : Prend en paramètre une chaine d'horodatage (comme dans le constructeur) et renvoie le nombre de millisecondes, du 1er janvier 1970 00:00:00 UTC à cette date.
- **Date.UTC()** : accepte des paramètres similaires à ceux du constructeur Date et renvoie le nombre de millièmes de seconde depuis le 1er janvier 1970, 00:00:00, temps universel. Autrement dit, elle renvoie la date en UTC.

```
let utcDate = new Date(Date.UTC(2024,0,1));
console.log(utcDate.toUTCString());      // 1er janvier 2024 à 00h00:00 en
UTC
```

Travailler avec les dates : méthodes importantes

Travailler avec les dates est toujours un peu complexe. En effet, souvent la date est en fait un DateTime or, en fonction des traitements que l'on souhaite faire, on ne veut pas toujours que 01/01/2024 à 00h00 soit différent ou inférieur à 01/01/2024 à 00h30.

De plus la comparaison des dates peut poser un problème. En effet, l'opérateur d'égalité stricte en JavaScript, lorsqu'il compare des objets, renverra true seulement si les 2 objets sont les mêmes (donc qu'ils font référence au même emplacement mémoire).

Pour plus de détails :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Operators/Strict_equality.

Souvent, il faudra donc utiliser des fonctions pour convertir les dates avant d'effectuer des comparaisons ou des traitements dessus.

Pour la liste de toutes les méthodes, voir ici :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Date

Je vous invite à expérimenter ces différentes méthodes.

Les méthodes notables sont :

- getDate(), getDay(), getFullYear(), getHours(), getMilliseconds(), getMinutes(), getMonth(), getSeconds() : Ces méthodes sont assez explicites. Elles renvoient la donnée dans la timezone locale. Faites en revanche attention à l'utilisation de getMonth(), les mois commencent à l'index 0 !
- Il existe aussi le pendant de toutes ces méthodes en UTC : getUTCDate(), getUTCDay(), getUTCFullYear(), ...
- **getTime()** : renvoie le temps écoulé, en millisecondes, depuis le 1er janvier 1970 à 00h00 UTC.
- **toDateString()** : Renvoie la date formatée selon la RFC2822 (ex « Mon Jan 01 2024 »)
- **toIsoString()** : Renvoie la datetime au format ISO (ex : '2024-01-01T00:00:00.000Z')
- **toJSON()** : renvoie la datetime au format JSON (ex : '2024-01-01T00:00:00.000Z')

Comme vu précédemment, nous ne pouvons pas utiliser d'opérateur d'égalité facilement pour comparer une date. En revanche, nous pouvons utiliser les opérateurs de comparaison ('<', '<=', '>', '>=').

Exemples d'utilisation

```
let date1 = new Date(2024, 0, 1);
let date2 = new Date(2024, 5, 1);

// Un test simple. Mais attention, les dates sont comparées en prenant
également en compte les heures, minutes  et secondes
if (date1 > date2) {
    console.log("date 1 est > à date 2");
} else if (date2 > date1) {
    console.log("date 2 est > à date1");
} else {
    console.log("les dates sont égales");
}

// On pourrait aussi faire une fonction nous disant si les dates sont
égales
function isEqual(date1, date2) {
    //A nous de choisir le degré de finesse que l'on souhaite dans la
comparaison
    return date1.getDay() === date2.getDay() && date1.getMonth() ===
date2.getMonth() && date1.getYear() === date3.getYear();   //Ici, on teste
le jour, mois et année seulement
}
```

Ressources

https://www.freecodecamp.org/news/javascript-date-comparison-how-to-compare-dates-in-js/

Collections indexées

Tableaux (arrays)

Les tableaux sont des objets de haut-niveau semblables à des listes.

Constructeur

Plusieurs syntaxes sont possibles pour déclarer et instancier nos tableaux. Voici quelques exemples.

```
//Creation d'un tableau
let fruits = ["Apple", "Banana"];

console.log(fruits.length); //La propriété length affiche la taille du
tableau (le nombre d'éléments)
// 2

// Pour créer un tableau vide
let unTableau = [];
//ou en utilisant le constructeur de l'objet Array
let unAutreTabeau = new Array();
// Le constructeur peut aussi prendre en paramètres les éléments composants
ce tableau. Par exemple
fruits = new Array("Apple", "Banana", "Watermelon");
//Il peut aussi prendre en paramètre la longueur du tableau
let unTableauTaille10 = new Array(10);
```

Méthodes statiques

- Array.isArray() : prend en paramètre un élément, renvoie ‘true’ si c’est un tableau, ‘false’ sinon
- Array.of() : prend en paramètres un nombre variable d’éléments et retourne un tableau à partir de cet élément.
  Il peut aussi accepter en paramètre une fonction de transformation qui sera appliquée sur chaque élément du tableau initial. Dans ce cas la méthode nous retournera le tableau après transformation.
- Array.from() : prend en paramètre un tableau ou autre objet itérable et retourne une copie de ce tableau.

Méthodes

Comme tout objet, l’objet Array possède des méthodes qui lui sont propres. Vous en connaissez déjà quelques-unes. Pour une vue plus exhaustive, voir la doc officielle : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Array.

- **.push()** : prend un élément en paramètre et l’ajoute en fin de tableau.
- **.pop()** : renvoie le dernier élément du tableau et le supprime du tableau.
- **.at()** : prend un entier (index) en paramètre et retourne l’élément du tableau situé à cet index. Ainsi, « array.at(i) » est similaire à « array[i] »
- **.concat()** : permet de fusionner plusieurs tableaux en un seul. Ne modifie pas les tableaux envoyés en paramètre. Retourne le tableau fusionné.

- **.fill()** : Prend en paramètre une valeur, et éventuellement un index de départ et un index de fin, et remplit le tableau (en totalité ou entre les bornes renseignées) avec la valeur.
- **.find(), .findLast()** : Prend un élément en paramètre et renvoie, en fonction de la méthode utilisée, l'index du 1er ou dernier élément identique dans le tableau.
- **.includes()** : Prend un élément en paramètre et renvoie true si l'élément est contenu dans le tableau, false sinon.

Tableaux typés (typed arrays)

Ces tableaux ont été ajoutés avec ECMAScript 2015. Ils fournissent un mécanisme pour lire et écrire des données binaires brutes dans des tampons mémoires.

Ne pas confondre les tableaux typés et les tableaux « classiques » (Array). En effet, la méthode Array.isArray() renverra false lorsqu'elle sera utilisée sur un tableau typé. De plus, certaines des méthodes des tableaux « classiques » ne sont pas disponibles pour les tableaux typés (par exemple push et pop).

Nous ne rentrerons pas plus en détails sur les tableaux typés, mais vous pouvez aller voir par vous-même le fonctionnement si besoin : https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Typed_arrays.

Les collections de type « clé/valeur » : Map, Set, WeakMap, WeakSet

Ces structures de données utilisent des clés pour référencer des objets. Elles ont été introduites avec ECMAScript 2015. Set et WeakSet représentent des ensembles d'objets, Map et WeakMap associent une valeur à un objet.

Il est possible d'énumérer les valeurs contenues dans un objet Map mais pas dans un objet WeakMap. Les WeakMap quant à eux permettent certaines optimisations dans la gestion de la mémoire et le travail du ramasse-miettes.

Map : Exemple d'utilisation

Un objet **Map** contient des paires de clé-valeur et mémorise l'ordre dans lequel les clés ont été insérées. N'importe quel type de valeur (primitive ou objet) peut être utilisée comme clé ou comme valeur.

Nous allons voir rapidement le fonctionnement de l'objet Map. Pour connaitre toutes les méthodes et possibilités, voir la documentation : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Map.

```
// Création d'un objet Map
const map1 = new Map();

// Ajout de paires "clé/valeur".
map1.set('a', 1);
map1.set('b', 2);
map1.set('c', 3);

console.log(map1.get('a')); //Récupération de la valeur de la clé 'a'
// Expected output: 1

map1.set('a', 97);      // Les clés sont uniques. Donc ici, on écrase la valeur
initiale

console.log(map1.get('a'));
// Expected output: 97

console.log(map1.size);
// Expected output: 3

map1.delete('b');   // Suppression de l'entrée ayant la clé 'b'

console.log(map1.size);
// Expected output: 2

// La méthode has() renvoie un booléen permettant de déterminer si l'objet Map en
question contient la clé donnée.
console.log(map1.has('b'));     //Affichera true
console.log(map1.has('d'));     //Affichera false

//La méthode keys() renvoie un objet Iterator qui contient les clés de chaque
élément de l'objet Map, dans leur ordre d'insertion.
const keys = map1.keys();

console.log(keys.next().value);
// Expected output: "a"

console.log(keys.next().value);
// Expected output: 'b''

// La méthode values() renvoie un objet Iterator qui contient les valeurs de chacun
des éléments contenu dans l'objet Map donné, dans leur ordre d'insertion.
map1.set('0', 'foo');
map1.set(1, 'bar');

const iterator2 = map1.values();
// On peut convertit l'objet Iterator en array
const values = Array.from(iterator2);
for (let i=0; i<values.length; i++) {
    console.log(values[i]);
}
```

Set : Exemple d'utilisation

Un objet **Set** permet de stocker un ensemble de valeurs uniques de n'importe quel type, qu'il s'agisse de valeurs primitives ou d'objets.

Au niveau Performance, Set dispose d'une méthode has() qui permet de vérifier si une valeur est contenue dans l'objet Set et qui utilise une approche qui est, en moyenne, plus rapide que de tester les éléments qui ont été précédemment ajoutés à Set. Cette méthode est, en moyenne, plus rapide que la méthode Array.prototype.includes() qui s'applique aux objets Array lorsque la longueur (length) du tableau est égale à celle de l'objet Set (size).

Nous allons voir rapidement le fonctionnement de l'objet Set. Pour connaitre toutes les méthodes et possibilités, voir la documentation : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Set.

```
const monSet = new Set();

// Ajouter des éléments uniques au set
monSet.add(1); // { 1 }
monSet.add(5); // { 1, 5 }
monSet.add(5); // { 1, 5 }
monSet.add("du texte"); // { 1, 5, 'du texte' }

const o = { a: 1, b: 2 };
monSet.add(o);
monSet.add({ a: 1, b: 2 });
// o fait référence à un objet différent
// il n'y a pas de problème pour cet ajout

monSet.has(1); // true
monSet.has(3); // false, 3 n'a pas été ajouté à l'ensemble
monSet.has(5); // true
monSet.has(Math.sqrt(25)); // true
monSet.has("Du Texte".toLowerCase()); // true
monSet.has(o); // true

monSet.size; // 5

monSet.delete(5); // retire 5 du set
monSet.has(5); // false, 5 a été retiré de l'ensemble

monSet.size; // 4, on a retiré une valeur de l'ensemble
console.log(monSet);
// affiche Set(4) [ 1, "du texte", {…}, {…} ] pour Firefox
// affiche Set(4) { 1, "du texte", {…}, {…} } pour Chrome
```

Les données structurées : JSON

JSON (*JavaScript Object Notation*) est un format d'échange de données léger, dérivé de JavaScript et utilisé par plusieurs langages de programmation. JSON permet ainsi de construire des structures de données universelles pouvant être échangées entre programmes.

Le JSON est énormément utilisé en JavaScript, mais aussi plus généralement dans le domaine du web. C'est en effet le format utilisé par les API Rest pour retourner les résultats de requêtes.

Le JSON est aussi le format idéal pour stocker des objets sérialisés. La sérialisation est le fait de transformer un objet en chaine de caractères dans le but de gérer la partie « persistance » (l'enregistrement des données) ou de le faire transiter par message (principe des API).

Un texte JSON comprend :

- 2 types composés :
  - Des objets JavaScript, ou ensemble de paires « clé/valeur »
  - Des listes ordonnées de valeurs (tableaux)
- 4 types scalaires :
  - Des booléens
  - Des nombres
  - Des chaines de caractères
  - La valeur « null »

Exemple de texte au format JSON

```
{
    "menu": {
        "id": "file",
        "value": "File",
        "popup": {
            "menuitem": [
                { "value": "New", "onclick": "CreateNewDoc()"
},
                { "value": "Open", "onclick": "OpenDoc()" },
                { "value": "Close", "onclick": "CloseDoc()" }
            ]
        }
    }
}
```

Evénements

Storages

Promesses et closures

Fonctions anonymes et fléchées

Manipulation du DOM

JWT

API Rest

TypeScript