> Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

Types et grammaire

Ce chapitre décrit les bases de la grammaire et des types de données JavaScript.

Les bases du langage

JavaScript emprunte la plupart des éléments de sa syntaxe à Java, C et C++ mais sa syntaxe est également influencée par Awk, Perl et Python.

JavaScript est **sensible à la casse** et utilise l'ensemble de caractères **Unicode**. On pourrait donc tout à fait utiliser le mot früh comme nom de variable :

```
JS
var früh = "toto";
typeof Früh; // undefined car JavaScript est sensible à la casse
```

En JavaScript, les instructions sont appelées (statements) et sont séparées par des points-virgules.

Il n'est pas nécessaire d'inclure un point-virgule si l'on écrit une instruction sur une nouvelle ligne. Mais si vous voulez écrire plus d'une déclaration sur une seule ligne, alors elles doivent être séparées par un point-virgule. Ceci étant dit, la bonne pratique est d'inclure un point-virgule après chaque instruction. Les espaces, les tabulations et les caractères de nouvelles lignes sont considérés comme des blancs. Il existe aussi un ensemble de règles pour ajouter automatiquement des points-virgules à la fin des

instructions (ASI pour *Automatic Semicolon Insertion*). Cependant, il est conseillé de toujours ajouter des points-virgules à la fin des instructions afin d'éviter des effets de bord néfastes.

Le texte d'un code source JavaScript est analysé de gauche à droite et est converti en une série d'unités lexicales, de caractères de contrôle, de fins de lignes, de commentaires et de blancs. ECMAScript définit également certains mots-clés et littéraux. Pour plus d'informations, voir la page sur la grammaire lexicale de JavaScript dans la référence JavaScript.

Commentaires

La syntaxe utilisée pour **les commentaires** est la même que celle utilisée par le C++ et d'autres langages :

```
JS
// un commentaire sur une ligne

/* un commentaire plus
   long sur plusieurs lignes
 */

/* Par contre on ne peut pas /* imbriquer des commentaires */ SyntaxError */
```

> **Note :** Vous pourrez également rencontrer une troisième forme de commentaires au début de certains fichiers JavaScript comme `#!/usr/bin/env node`. Ce type de commentaire indique le chemin d'un interpréteur JavaScript spécifique pour exécuter le script. Pour plus de détails, voir la page sur les commentaires d'environnement.

Déclarations

Il existe trois types de déclarations de variable en JavaScript.

`var`

On déclare une variable, éventuellement en initialisant sa valeur.

`let`

On déclare une variable dont la portée est celle du bloc courant, éventuellement en initialisant sa valeur.

`const`

On déclare une constante nommée, dont la portée est celle du bloc courant, accessible en lecture seule.

Variables

Les variables sont utilisées comme des noms symboliques désignant les valeurs utilisées dans l'application. Les noms des variables sont appelés *identifiants*. Ces identifiants doivent respecter certaines règles.

Un identifiant JavaScript doit commencer par une lettre, un tiret bas (_) ou un symbole dollar ($). Les caractères qui suivent peuvent être des chiffres (0 à 9). À noter : puisque Javascript est sensible aux majuscules et minuscules: les lettres peuvent comprendre les caractères de « A » à « Z » (en majuscule) mais aussi les caractères de « a » à « z » (en minuscule).

On peut aussi utiliser la plupart lettres Unicode ou ISO 8859-1 (comme å et ü, pour plus de détails, voir ce billet de blog, en anglais) au sein des identifiants. Il est également possible d'utiliser les \uXXXX séquences d'échappement Unicode comme caractères dans les identifiants.

Voici des exemples d'identifiants valides : `Nombre_touches`, `temp99`, `$credit`, et `_nom`.

Déclaration de variables

Il est possible de déclarer des variables de plusieurs façons :

- En utilisant le mot-clé `var`, par exemple : `var x = 42`. Cette syntaxe peut être utilisée pour déclarer des variables locales ou globales selon le contexte d'exécution.
- En utilisant le mot-clé `const` ou le mot-clé `let`, par exemple avec `let y = 13`. Cette syntaxe peut être utilisée pour déclarer une variable dont la portée sera celle du bloc. Voir le paragraphe sur les portées des variables ci-après.

Il est également possible d'affecter une valeur à une variable sans utiliser de mot-clé (ex. `x = 42`). Cela créera une variable globale non-déclarée. Cette forme génèrera également un avertissement avec le mode strict. Attention, les variables globales non-déclarées peuvent mener à des comportements inattendus et sont considérées comme une mauvaise pratique.

Évaluation de variables

Une variable déclarée grâce à l'instruction `var` ou `let` sans valeur initiale définie vaudra `undefined`.

Tenter d'accéder à une variable qui n'a pas été déclarée ou tenter d'accéder à un identifiant déclaré avec `let` avant son initialisation provoquera l'envoi d'une exception `ReferenceError`.

JS

```
var a;
console.log("La valeur de a est " + a); // La valeur de a est undefined

console.log("La valeur de b est " + b); // La valeur de b est undefined
var b; // La déclaration de la variable est "remontée" (voir la section ci-après)

console.log("La valeur de x est " + x); // signale une exception ReferenceError
let x;
let y;
console.log("La valeur de y est " + y); // La valeur de y est undefined
```

Il est possible d'utiliser `undefined` pour déterminer si une variable possède une valeur. Dans l'exemple qui suit, la condition de l'instruction `if` sera validée car la variable n'a pas été initialisée (elle a simplement été déclarée) :

JS

```
var input;
if (input === undefined) {
  faireCeci();
} else {
  faireCela();
}
```

La valeur `undefined` se comporte comme le booléen `false` lorsqu'elle est utilisée dans un contexte booléen. Ainsi, le fragment de code qui suit exécutera la fonction `maFonction` puisque le premier élément de `monTableau` n'est pas défini :

```
JS
var monTableau = new Array();
if (!monTableau[0]) {
  maFunction();
}
```

La valeur `undefined` est convertie en `NaN` (pour *Not a Number* : « n'est pas un nombre ») lorsqu'elle est utilisée dans un contexte numérique.

```
JS
var a;
a + 2; // NaN
```

Une variable valant `null` sera toujours considérée comme 0 dans un contexte numérique et comme `false` dans un contexte booléen. Par exemple, on aura :

```
JS
var n = null;
console.log(n * 32); // Le log affichera 0
```

Les portées de variables

Lorsqu'une variable est déclarée avec `var` en dehors des fonctions, elle est appelée variable *globale* car elle est disponible pour tout le code contenu dans le document. Lorsqu'une variable est déclarée dans une fonction, elle est appelée variable *locale* car elle n'est disponible qu'au sein de cette fonction.

Avant ECMAScript 2015 (ES6), JavaScript ne définissait pas de portée pour une instruction de bloc ; les éléments du bloc seront locaux pour le code qui contient le bloc (que ce soit une fonction ou le contexte global). Ainsi, l'exemple qui suit affichera 5 car la portée de `x` est la fonction (ou le contexte global) dans lequel `x` est déclaré, pas le bloc (correspondant à l'instruction `if` dans ce cas) :

JS

```
if (true) {
  var x = 5;
}
console.log(x); // x vaut 5
```

La déclaration `let`, introduite avec ECMAScript 2015, ajoute un nouveau comportement :

JS

```
if (true) {
  let y = 5;
}
console.log(y); // ReferenceError: y is not defined
```

Remontée de variables (*hoisting*)

Une autre chose peut paraître étrange en JavaScript : il est possible, sans recevoir d'exception, de faire référence à une variable qui est déclarée plus tard. Ce concept est appelé « remontée » (*hoisting* en anglais) car, d'une certaine façon, les variables sont remontées en haut de la fonction ou de l'instruction. En revanche, les variables qui n'ont pas encore été initialisées renverront la valeur `undefined`. Ainsi, même si on déclare une variable et qu'on l'initialise après l'avoir utilisée ou y avoir fait référence, la valeur utilisée « la plus haute » sera toujours `undefined`.

JS

```
/**
 * Exemple 1
 */
console.log(x === undefined); // donne "true"
var x = 3;

/**
 * Exemple 2
 */
// renverra undefined
var maVar = "ma valeur";

(function () {
  console.log(maVar); // undefined
```

```
  var maVar = "valeur locale";
})();
```

Les exemples précédents peuvent être reformulés plus explicitement ainsi :

JS

```
/**
 * Exemple 1
 */
var x;
console.log(x === undefined); // donne "true"
x = 3;

/**
 * Exemple 2
 */
var maVar = "ma valeur";

(function () {
  var maVar = "valeur locale";
})();
```

C'est pourquoi il est conseillé de placer les instructions `var` dès que possible dans le code. De plus, cette bonne pratique aide à rendre le code plus lisible.

Avec ECMAScript 2015, `let (const)` **remontera la variable en haut du bloc mais ne l'initialisera pas**. Aussi, si on fait référence à la variable dans le bloc avant la déclaration, on obtient une `ReferenceError` car la variable est dans une « zone morte temporelle ». entre le début du bloc et le traitement de la déclaration

JS

```
function faire_quelquechose() {
  console.log(toto); // ReferenceError
  let toto = 2;
}
```

Remontée de fonctions

En ce qui concerne les fonctions, seules les déclarations de fonctions sont remontées. Pour les expressions de fonctions, il n'y a pas de telle remontée car la variable associée n'a pas encore été affectée avec la valeur finale (comme vu avant) :

JS

```
/* Déclaration de fonction */
toto(); // "truc"
function toto() {
  console.log("truc");
}

/* Expression de fonction */
machin(); // erreur TypeError : machin n'est pas une fonction
var machin = function () {
  console.log("titi");
};
```

Les variables globales

Les variables globales sont en réalité des propriétés de l'*objet global*. Dans les pages web, l'objet global est `window`, et on peut donc accéder ou modifier la valeur de variables globales en utilisant la syntaxe suivante : `window.variable`.

Ainsi, il est possible d'accéder à des variables déclarées dans une fenêtre ou dans un cadre depuis une autre fenêtre ou depuis un autre cadre (*frame*) en spécifiant son nom. Si, par exemple, une variable appelée `numTéléphone` est déclarée dans un document `FRAMESET`, il est possible d'y faire référence, depuis un cadre fils, avec la syntaxe `parent.numTéléphone`.

Constantes

Il est possible de créer des constantes en lecture seule en utilisant le mot-clé `const`. La syntaxe d'un identifiant pour une constante est la même que pour les variables (elle doit débuter avec une lettre, un tiret du bas, un symbole dollar et peut contenir des caractères numériques, alphabétiques et des tirets bas voire des caractères Unicode).

JS

```
const préfixe = "212";
```

Une constante ne peut pas changer de valeur grâce à une affectation ou être re-déclarée pendant l'exécution du script.

Les règles de portée des constantes sont les mêmes que pour les variables, à l'exception du mot-clé `const` qui est obligatoire. S'il est oublié, l'identifiant sera considéré comme celui d'une variable.

Il est impossible de déclarer une constante avec le même nom qu'une autre variable ou fonction dans la même portée.

JS

```
// Renverra une erreur
function f() {}
const f = 5;

// Renverra également une erreur
function f() {
  const g = 5;
  var g;

  //instructions
}
```

Cependant, les propriétés des objets qui sont affectés comme constantes ne sont pas protégées, on pourra ainsi exécuter sans problème le code suivant :

JS

```
const MON_OBJET = { clé: "valeur" };
MON_OBJET.clé = "autreValeur";
```

De même, le contenu d'un tableau peut être modifié sans alerte :

JS

```
const MON_TABLEAU = ["HTML", "CSS"];
MON_TABLEAU.push("JavaScript");
console.log(MON_TABLEAU); // ["HTML", "CSS", "JavaScript"]
```

Structures de données et types

Types de données

La dernière version du standard ECMAScript définit sept types de données :

- Six types de données primitifs :
  - Type booléen : `true` et `false`.
  - Type nul (`null`), un mot-clé spécial pour indiquer une valeur nulle (au sens informatique). JavaScript étant sensible à la casse, `null` n'est pas `Null`, `NULL`, ou toute autre variante.
  - Un type pour les valeurs indéfinies (`undefined`).
  - Un type pour les nombres entiers ou décimaux. Par exemple : `42` ou `3.14159`.
  - Un type pour représenter les grands nombres entiers `BigInt`, par exemple `9007199254740992n`.
  - Un type pour les chaînes de caractères, une séquence de caractères qui représente une valeur textuelle. Par exemple : "Coucou"
  - Un type pour les symboles, apparus avec ECMAScript 2015 (ES6). Ce type est utilisé pour représenter des données immuables et uniques.
- et un type pour les objets (*Object*)

Bien que cette description couvre peu de types de données, ceux-ci vous permettent d'implémenter une grande variété de fonctions au sein de vos applications. Les objets et les fonctions sont parmi les briques fondamentales du langage. On peut considérer, à première vue, les objets comme des conteneurs de valeurs et de fonctions pour une application.

Conversion de types de données

JavaScript est un langage à typage dynamique. Cela signifie qu'il n'est pas nécessaire de spécifier le type de données d'une variable lors de sa déclaration. Les types de données sont convertis automatiquement durant l'exécution du script. Ainsi, il est possible de définir une variable de cette façon :

```
JS
```

```
var réponse = 42;
```

Et plus tard, d'affecter une chaîne de caractères à cette même variable :

```
JS
réponse = "Merci pour le dîner...";
```

JavaScript utilisant un typage dynamique, cette dernière instruction ne renverra pas d'erreur.

Lorsque des expressions impliquent des chaînes de caractères et des valeurs numériques ainsi que l'opérateur +, JavaScript convertit les nombres en chaînes de caractères :

```
JS
x = "La réponse est " + 42; // "La réponse est 42"
y = 42 + " est la réponse"; // "42 est la réponse"
```

Avec des instructions impliquant d'autres opérateurs, JavaScript ne convertit pas nécessairement les valeurs numériques en chaînes de caractères. Ainsi, on aura :

```
JS
"37" - 7; // 30
"37" + 7; // "377"
```

Conversion de chaînes de caractères en nombres

Si un nombre est représenté en mémoire par une chaîne de caractères, il y a des méthodes pour effectuer la bonne conversion :

- `parseInt()`
- `parseFloat()`

`parseInt` renverra uniquement des nombres entiers, étant ainsi inappropriée pour la manipulation de nombre décimaux. Une bonne pratique pour cette fonction est de toujours inclure l'argument qui indique dans quelle base numérique le résultat doit être renvoyé (base 2, base 10...).

JS

```
parseInt("101", 2); // 5
```

L'opérateur + unaire

Une autre méthode pour récupérer un nombre à partir d'une chaîne de caractères consiste à utiliser l'opérateur +.

JS

```
"1.1" + "1.1" = "1.11.1"
+"1.1" = 1.1 // fonctionne seulement avec le + unaire
```

Littéraux

Les littéraux sont utilisés pour représenter des valeurs en JavaScript. Ce sont des valeurs fixes, pas des variables, qui sont fournies *littéralement* au script. Cette section décrit les différents types de littéraux :

- Littéraux de tableaux
- Littéraux booléens
- Littéraux de nombres flottants
- Littéraux numériques
- Littéraux d'objets
- Littéraux d'expressions rationnelles
- Littéraux de chaînes de caractères

Les littéraux de tableaux

Un littéral de tableau est une liste de zéro ou plusieurs expressions, dont chacune représente l'élément d'un tableau, encadrées par des crochets (`[]`). Lorsqu'un tableau est créé à partir d'un littéral, il est initialisé avec les valeurs spécifiées qui sont ses éléments, sa longueur correspondant au nombre d'arguments donnés.

L'exemple suivant crée ainsi le tableau `cafés` avec trois éléments et une taille égale à 3 :

JS

```
var cafés = ["Brésilien", "Colombien", "Kona"];
```

> **Note :** Un littéral de tableau est du type d'un initialisateur d'objets. Voir l'utilisation d'initialisateurs d'objets.

Si un tableau est créé en utilisant un littéral dans un script du plus haut niveau, JavaScript interprète le tableau chaque fois qu'il évalue l'expression contenant le littéral. De plus, un littéral utilisé dans une fonction est créé chaque fois que la fonction est appelée.

Les littéraux de tableaux sont également des objets `Array`. Voir la page sur l'objet `Array` pour plus de détails.

Les virgules supplémentaires

Il n'est pas nécessaire de définir tous les éléments dans un littéral de tableau. Si vous utilisez deux virgules, l'une à la suite de l'autre, le tableau utilisera la valeur `undefined` pour les éléments non définis. L'exemple qui suit utilise le tableau poisson :

JS

```
var poisson = ["Clown", , "Chat"];
```

Ce tableau possède deux éléments ayant une valeur et un élément vide (`poisson[0]` vaut "Clown", `poisson[1]` vaut `undefined`, et `poisson[2]` vaut "Chat").

Si une virgule est ajoutée à la fin de la liste des éléments, elle est ignorée. Dans le prochain exemple, la longueur du tableau est égale à 3. Il n'y a pas d'élément `maListe[3]`. Les autres virgules indiquent un nouvel élément.

> **Note :** Avec d'anciennes versions de navigateurs, les virgules de fin peuvent causer des erreurs, il est fortement conseillé de les retirer.

JS

```
var maListe = ["maison", , "école"];
```

Dans l'exemple qui suit, la longueur du tableau est égale à 4 et `maListe[0]` et `maListe[2]` sont manquants.

JS

```
var maListe = [, "maison", , "école"];
```

Dans l'exemple qui suit, la longueur du tableau est égale à 4 et `maListe[1]` et `maListe[3]` sont manquants.

JS

```
var maListe = ["maison", , "école", ,];
```

Comprendre le fonctionnement des virgules supplémentaires est important. Cependant, lorsque vous écrivez du code, veillez, dès que c'est possible, à déclarer les éléments manquants avec `undefined` : cela améliorera la lisibilité de votre code et il sera ainsi plus facile à maintenir.

Les littéraux booléens

Le type booléen possède deux valeurs littérales : `true` et `false`.

Il ne faut pas confondre les valeurs `true` et `false` du type primitif booléen et les valeurs true et false de l'objet `Boolean`. L'objet `Boolean` permet de créer un objet autour du type de donnée booléen. Voir la page sur l'objet `Boolean` pour plus d'informations.

Les littéraux numériques

Les nombres `Number` et les grands entiers `BigInt` peuvent être exprimés en notation décimale (base 10), hexadécimale (base 16), octale (base 8) et binaire (base 2).

- Les littéraux représentant des entiers décimaux sont une suite de chiffres qui ne commence pas par un 0 (zéro)
- Un 0 (zéro) en préfixe indique que le littéral est en notation octale. Ces nombres ne peuvent être composés que des chiffres de 0 (zéro) à 7 (sept).
- Un préfixe 0x (ou 0X) indique une notation hexadécimale. Les nombres hexadécimaux peuvent être composés de chiffres (0-9) et des lettres A à F (minuscules et

majuscules) (la casse d'un caractère ne modifie pas sa valeur : `0xa = 0xA = 10` et `0xf = 0xF = 15`).

- Un préfixe 0b (ou 0B) indique une notation binaire. Les nombres binaires peuvent être composés de 0 ou de 1 uniquement.

Voici des exemples pour ces littéraux :

```
0, 117, -345, 123456789123456789n (notation décimale, base 10)
015, 0001, -077, 0o7777777777777n (notation octale, base 8)
0x1123, 0x00111, -0xF1A7, 0x123456789ABCDEFn (notation hexadécimale, base 16)
0b11, 0B0011, -0b11, 0b11101001010101010101n (notation binaire, base 2)
```

Pour plus d'informations, voir les littéraux numériques dans la grammaire lexicale de JavaScript.

Les littéraux de nombres décimaux

Un littéral de nombre décimal peut être composé de ces différentes parties :

- Un entier, pouvant être signé (précédé d'un « + » ou d'un « - »),
- Un point, séparateur décimal (« . »),
- La partie décimale (un autre nombre)
- Un exposant.

L'exposant est la partie du nombre décimal commençant par un « e » ou un « E », suivie d'un entier pouvant être signé (précédé d'un « + » ou d'un « - »). Un littéral de nombre décimal doit comporter au moins un chiffre et soit un point (séparateur décimal) soit un « e » ou un « E ».

Des exemples sont : 3.1415, -3.1E12, .1e12, et 2E-12.

On peut raccourcir cette syntaxe en :

```
[(+|-)][chiffres].[chiffres][(E|e)[(+|-)]chiffres]
```

Par exemple :

```
3.14
2345.789
.3333333333333333
```

Les littéraux d'objets

Un littéral d'objet - ou 'objet littéral' - est une liste de zéro ou plusieurs propriétés définies par des paires de noms/valeurs. Ces paires sont délimitées par des accolades (`{}`). N'utilisez pas un tel littéral en début d'instruction. En effet, l'accolade ouvrante sera mal interprétée (début de bloc) et causera une erreur ou un comportement incohérent.

L'exemple qui suit montre l'utilisation d'un littéral d'objet. Le premier élément de l'objet `voiture` définit une propriété `maVoiture`, le deuxième élément : la propriété `getVoiture` invoque une fonction `(carTypes("Honda"))`, le troisième élément, la propriété `special` utilise une variable existante (`soldes`).

JS

```
var soldes = "Toyota";

function carTypes(nom) {
  return nom === "Honda" ? nom : "Désolé, nous ne vendons pas de " + nom + ".";
}

var voiture = {
  maVoiture: "Saturn",
  getVoiture: carTypes("Honda"),
  spécial: soldes,
};

console.log(voiture.maVoiture); // Saturn
console.log(voiture.getVoiture); // Honda
console.log(voiture.spécial); // Toyota
```

Il est également possible d'utiliser un littéral numérique ou un littéral de chaîne de caractères pour désigner le nom d'une propriété ou pour imbriquer un objet dans un autre. L'exemple qui suit illustre cette possibilité :

JS

```
var voiture = { plusieursVoitures: { a: "Saab", b: "Jeep" }, 7: "Mazda" };

console.log(voiture.plusieursVoitures.b); // Jeep
console.log(voiture[7]); // Mazda
```

Les noms des propriétés d'objets peuvent être n'importe quelle chaîne de caractères, y compris la chaîne vide. Si le nom de la propriété n'est pas un identifiant valide, il faudra qu'il soit placé entre guillemets. Les noms de propriétés qui ne sont pas des identifiants valides ne peuvent pas être utilisés pour accéder à la valeur en utilisant la notation pointée (objet.propriété). En revanche, il est possible d'y accéder avec la notation utilisant les crochets (" `[]` ") comme pour les tableaux.

JS

```
var nomsBizarres = {
  "": "Chaîne vide",
  "!": "Bang !"
}
console.log(nomsBizarres.""); // SyntaxError: Unexpected string
console.log(nomsBizarres[""]); // Chaîne vide
console.log(nomsBizarres.!); // SyntaxError: Unexpected token !
console.log(nomsBizarres["!"]); // Bang !
```

Augmentation des littéraux d'objets avec ES2015/ES6

Avec ES2015, les littéraux d'objets permettent de définir le prototype lors de la construction de l'objet, permettent d'utiliser les affectations en notation raccourcie : `toto: toto` , de définir des méthodes, d'appeler les méthodes de l'objet parent avec `super` et d'utiliser des noms de propriétés calculées.

JS

```
var obj = {
  // __proto__
  __proto__: lePrototypeDeLObjet,
  // Notation raccourcie pour ‘handler: handler’
  handler,
  // Méthodes
  toString() {
    // Appelle les méthodes de l'objet parent
    return "d " + super.toString();
```

```
  },
  // Noms de propriétés calculés dynamiquement
  ["prop_" + (() => 42)()]: 42,
};
```

Attention :

JS

```
var toto = { a: "alpha", 2: "deux" };
console.log(toto.a); // alpha
console.log(toto[2]); // deux
//console.log(toto.2);  // Erreur: parenthèse ) manquante après la liste d'argument
//console.log(toto[a]); // Erreur: a n'est pas défini
console.log(toto["a"]); // alpha
console.log(toto["2"]); // deux
```

Les littéraux d'expressions rationnelles

Un littéral d'expression rationnelle est un motif encadré par deux barres obliques. Par exemple :

JS

```
var re = /ab+c/;
```

Les littéraux de chaînes de caractères

Un littéral de chaîne de caractères consiste en zéro ou plusieurs caractères encadrés par des guillemets droits doubles (`"`) ou des guillemets droits simples (`'`). Une chaîne de caractères doit être encadrée par des symboles du même type (guillemets droits doubles ou guillemets droits simples) :

- `"toto"`
- `'truc'`
- `"1234"`
- `"Une ligne \n une autre ligne"`
- `"Aujourd'hui j'ai mangé une pomme"`

Il est possible d'utiliser les méthodes de `String` sur un tel littéral. JavaScript convertira automatiquement le littéral en un objet `String`, appellera la méthode puis détruira l'objet `String`. On peut également utiliser la propriété `String.length` sur un littéral de chaîne de caractère :

JS

```
console.log("j'ai mangé une pomme".length);
// Affichera le nombre de caractères (y compris les blancs).
// Dans ce cas, 20.
```

Il est préférable d'utiliser des littéraux de chaînes de caractères s'il n'est pas spécifiquement nécessaire d'utiliser un objet `String`. Voir la page sur l'objet `String` pour plus de détails sur les objets `String`.

Avec ECMAScript 2015, on peut également utiliser des littéraux sous forme de *gabarits* (*templates*) en utilisant le caractère accent grave (`) comme séparateur. Les gabarits de chaînes de caractères sont semblables aux fonctionnalités d'interpolation existantes en Python, Perl, etc. Ces gabarits permettent d'utiliser des balises afin d'adapter la construction de chaînes.

JS

```
// Littéral simple pour une chaîne
`Un saut de ligne '\n' en JavaScript.` // On peut écrire une chaîne sur plusieurs
// lignes
`Dans les gabarits, on peut écrire
  sur plusieurs lignes. `;

// Interpolation de chaîne
var nom = "Robert",
  jour = "aujourd'hui";
`Bonjour ${nom}, comment allez-vous ${jour} ?`;

// On peut construire un préfixe HTTP
// afin de construire plus facilement
// des requêtes via des substitutions
POST`http://toto.org/truc?a=${a}&b=${b}
     Content-Type: application/json
     X-Credentials: ${credentials}
     { "toto": ${toto},
       "truc": ${truc}}`(myOnReadyStateChangeHandler);
```

Utilisation des caractères spéciaux

En plus des caractères « classiques », il est possible d'insérer des caractères spéciaux dans les chaînes de caractères. Voici un exemple :

JS

```
"une ligne \n une autre ligne";
```

Voici un tableau listant les caractères spéciaux qu'il est possible d'utiliser dans les chaînes de caractères JavaScript :

Caractère	Signification
`\0`	Octet null
`\b`	Retour arrière
`\f`	Saut de page
`\n`	Nouvelle ligne
`\r`	Retour chariot
`\t`	Tabulation
`\v`	Tabulation verticale
`\'`	Apostrophe ou guillemet droit simple
`\"`	Guillemet droit double
`\\`	Barre oblique inversée
`\XXX`	Le caractère dont l'encodage Latin-1 est spécifié grâce à, au plus, 3 chiffres octaux *XXX* entre 0 et 377. \251, par exemple représente le caractère copyright.
`\xXX`	Le caractère dont l'encodage Latin-1 est spécifié par deux chiffres hexadécimaux entre 00 et FF. Ainsi, \xA9 correspond à la séquence hexadécimale pour le caractère copyright.

Caractère	Signification
`\uXXXX`	Le caractère Unicode spécifié par quatre chiffres hexadécimaux *XXXX*. Ainsi, \u00A9 correspondra à la séquence Unicode du symbole copyright. Voir Les caractères d'échappement Unicode.
`\u{XXXXX}`	Échappement de codes Unicode. Par exemple, \u{2F804} est équivalent à la combinaison d'échappements « simples » \uD87E\uDC04.

Les caractères d'échappement

Pour les caractères qui ne font pas partie du tableau précédent, les barres obliques inversées (*backslash*) les précédant sont ignorées. Cependant, cet usage est obsolète et devrait être évité.

En précédant d'une barre oblique inversée les guillemets droits doubles, on *échappe* ces caractères. Voici un exemple :

JS

```
var citation = "Il lit \"Bug Jargal\" de V. Hugo.";
console.log(citation);
```

Le résultat serait alors

```
Il lit "Bug Jargal" de V. Hugo.
```

Pour inclure une barre oblique inversée dans une chaîne de caractères, il faut aussi l'échapper. Par exemple, pour stocker le chemin `c:\temp` dans une chaîne de caractères, on utilisera le code suivant :

JS

```
var chemin = "c:\\temp";
```

Il est également possible d'échapper des sauts de lignes de la même façon. La barre oblique inversée et le saut de ligne seront alors ignorés dans la valeur de la chaîne de caractères.

```
JS
var str =
  "cette chaîne \
est cassée \
sur plusieurs \
lignes.";
console.log(str); // cette chaîne est cassée sur plusieurs lignes.
```

Avant ECMAScript 2015 (ES6), JavaScript ne disposait pas d'une syntaxe permettant de traiter les chaînes de caractères comme des contenus de fichier, il est possible d'ajouter un caractère de saut de ligne échappé et un saut de ligne en fin de ligne en utilisant cette façon :

```
JS
var poème =
  "Les roses sont rouges,\n\
Les violettes sont bleues.\n\
Le miel est sucré,\n\
Et moi je suis.";
```

Grâce à ES6, on peut utiliser des littéraux de gabarits qui offrent de nouvelles fonctionnalités dont une qui permet d'avoir des chaînes de caractères écrites sur plusieurs lignes :

```
JS
var poème = `Les roses sont rouges,
Les violettes sont bleues,
Le miel est sucré,
Et moi je suis.`;
```

En savoir plus

Ce chapitre est centré sur les bases de la syntaxe, les déclarations et les types utilisés en JavaScript. Pour en savoir plus sur les différents composants du langage, voir les chapitres suivants du guide:

- Contrôle du flux et gestion des erreurs
- Boucles et itération

- Fonctions
- Expressions et opérateurs

Dans le chapitre suivant, on abordera les structures conditionnelles, permettant de diriger le flux d'instructions et la gestion des erreurs.

This page was last modified on 3 août 2023 by MDN contributors.