

Javascript



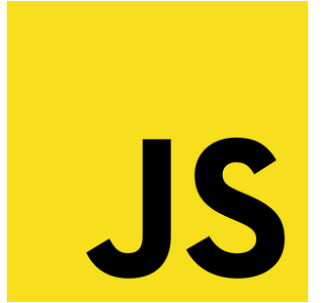
Le Javascript sert à contrôler le contenu dynamique d'un site pour qu'un utilisateur puisse interagir avec.

Contenu du cours



- Présentation du Javascript
- Types et structures de données
- Opérateurs de comparaison
- Flux de contrôle
- Fonctions
- Objets
- Collections
- Classes
- Annexe

Présentation de Javascript



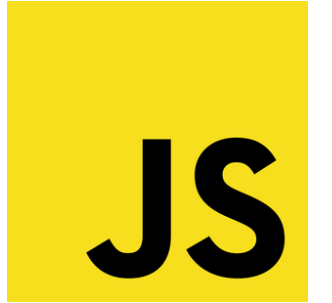
- **Les déclarations**

Une instruction est une unité d'instruction composée d'une ou de plusieurs lignes de code représentant une action. Par exemple, vous pouvez utiliser l'instruction suivante pour attribuer une valeur à une variable nommée **myVariable** :



```
let myVariable = 4;
```

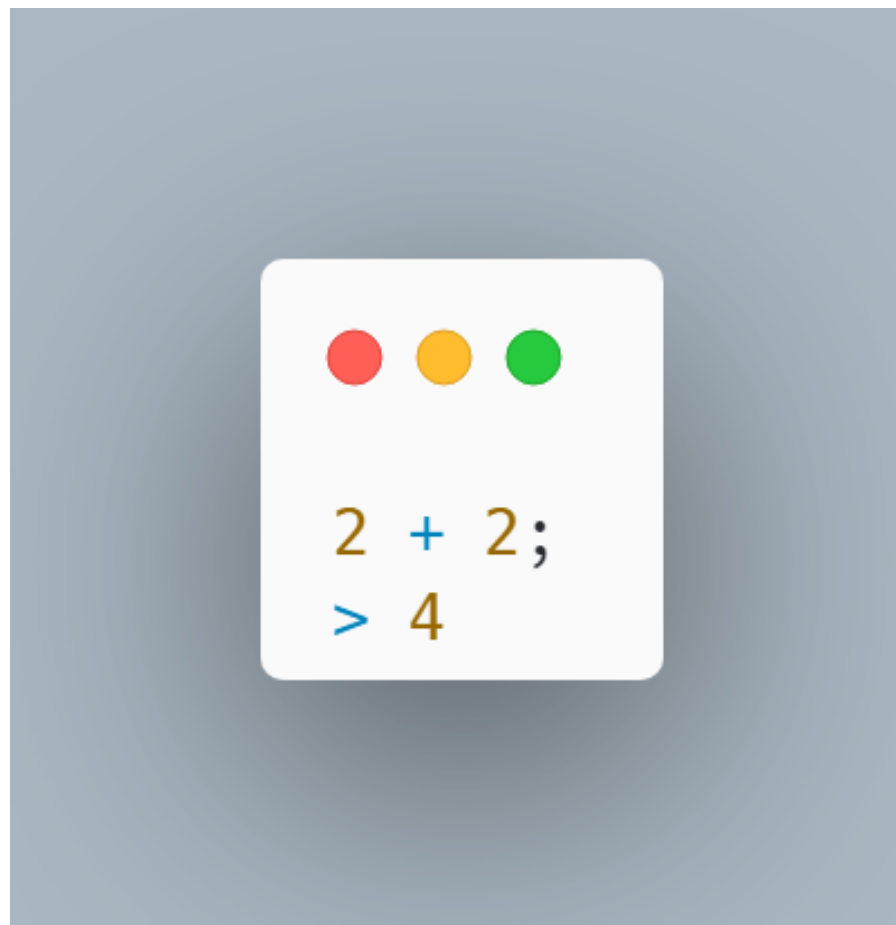
Présentation de Javascript



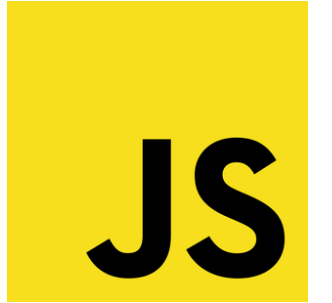
- **Les expressions**

Une expression est une unité de code qui génère une valeur et peut donc être utilisée partout où une valeur est attendue.

2 + 2 est une expression qui renvoie la valeur **4** :



Présentation de Javascript



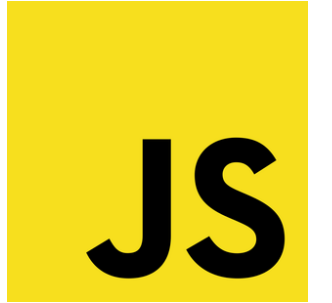
- **Sensibilité à la casse**

Contrairement au langage HTML et à la majorité des fichiers CSS, JavaScript lui-même est entièrement sensible à la casse. Cela signifie que vous devez toujours mettre tout en majuscules de manière cohérente, des propriétés et des méthodes intégrées au langage aux identifiants que vous définissez vous-même.

- **Espace blanc**

JavaScript n'est pas sensible aux espaces blancs. Cela signifie que l'interpréteur ignore la quantité et le type d'espaces blancs utilisés (tabulations ou espaces).

Types et structures de données

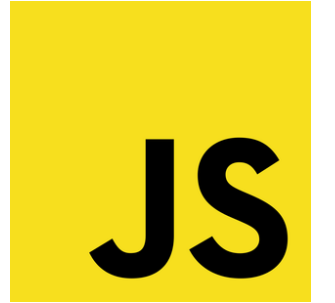


Les primitives sont les types de données les plus simples en JavaScript. Ils sont immuables, c'est-à-dire qu'ils ne peuvent pas être modifiés pour représenter d'autres valeurs de la même manière que les structures de données JavaScript plus complexes basées sur des objets.

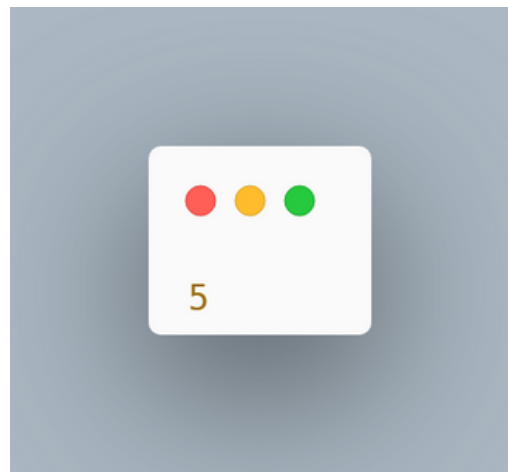
Il existe sept types de données primitifs :

- Nombres
- Chaînes de caractères
- Booléens
- **null**
- **undefined**
- BigInt
- Symbole

Nombres

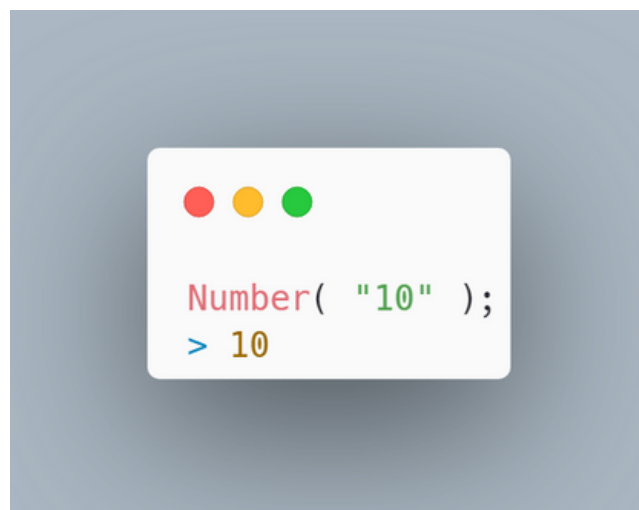


Une valeur numérique est composée d'une série de caractères numériques, comme dans l'exemple suivant :



- L'objet **Number**

Lorsqu'une valeur est transmise à la fonction **Number()**, cette valeur est convertie en équivalent numérique. Par exemple, une chaîne numérique donne un nombre primitif équivalent :



Nombres



- Les opérateurs numériques

Lorsque vous utilisez des opérateurs mathématiques standards avec des primitives numériques, **l'ordre des opérations** mathématiques s'applique: toutes les expressions entre parenthèses sont évaluées en premier, suivies des exposants, des multiplications, des divisions, des additions et des soustractions.

Opérateur	Nom	Description	Utilisation	Résultat
+	Ajout		2+2	4
-	Soustraction		4-2	2
*	Multiplication		2*5	10
/	Division		10/5	2
++	Incrément	Ajoute un à un nombre	2++	3
--	Diminuer	Soustrait un d'un nombre	3--	2
**	Exposant	Renvoie le résultat de l'élévation du premier opérande à la puissance du deuxième opérande.	2**4	16
%	Reste	Renvoie le reste restant lorsque le premier opérande est divisé par le deuxième opérande.	12%5	2

Nombres



Lorsque vous utilisez des opérateurs mathématiques standards avec des primitives numériques, **l'ordre des opérations** mathématiques s'applique: toutes les expressions entre parenthèses sont évaluées en premier, suivies des exposants, des multiplications, des divisions, des additions et des soustractions.

Opérateur	Nom	Utilisation
<code>+=</code>	Attribution des ajouts	<code>myValue += 2</code>
<code>-=</code>	Attribution de soustraction	<code>myValue -= 2</code>
<code>*=</code>	Attribution de multiplications	<code>myValue *= 2</code>
<code>/=</code>	Affectation de la division	<code>myValue /= 2</code>
<code>**=</code>	Attribution d'exposant	<code>myValue **= 2</code>
<code>%=</code>	Affectation restante	<code>myValue %= 2</code>

Chaînes de caractères

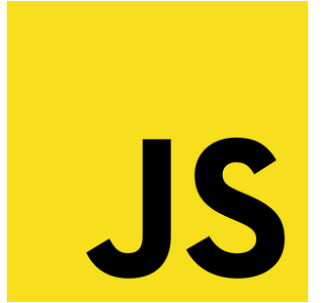


Tout ensemble de caractères (lettres, chiffres, symboles, etc.) entre un ensemble de guillemets doubles ("), de guillemets simples (') ou d'accents graves (`) est une chaîne primitive. Vous avez déjà vu quelques exemples de chaînes dans ce cours: les instances de `console.log` du module précédent contenaient des primitives de chaîne.



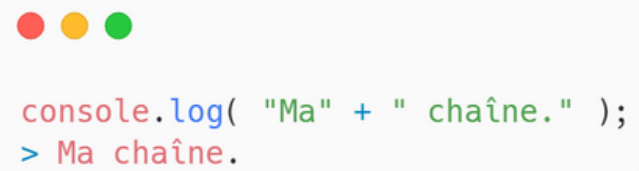
"Hello, World." est une primitive de chaîne. Vous obtenez le même résultat avec des guillemets simples ou des accents graves.

Chaînes de caractères



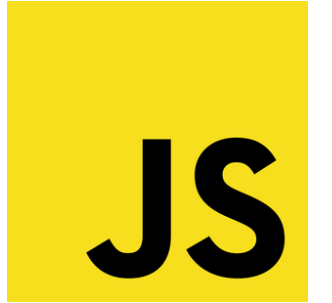
- **Concaténation**

Lorsqu'il est utilisé dans le contexte de chaînes plutôt que de nombres, un signe plus unique (+) agit comme un opérateur de concaténation, combinant plusieurs valeurs de chaîne en une seule chaîne :



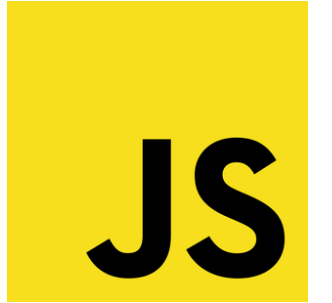
```
console.log( "Ma" + " chaîne." );  
> Ma chaîne.
```

Booléen



La primitive booléenne est un type de données logique comportant seulement deux valeurs: **true** et **false**.

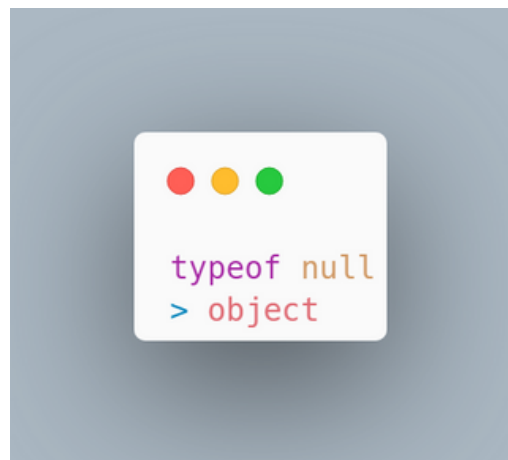
null et undefined



JavaScript propose plusieurs façons d'indiquer l'absence de valeur. Cette page décrit les deux méthodes les plus courantes : les types de données **null** et **undefined**.

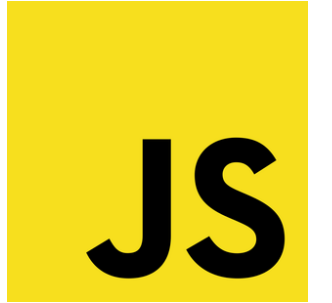
- **null**

Le mot clé **null** représente une absence de valeur définie intentionnellement. **null** est un élément primitif, bien que l'opérateur **typeof** indique que **null** est un objet. Il s'agit d'une erreur qui a été reportée à partir de la première version de JavaScript et qui a été laissée intentionnellement non corrigée pour éviter de perturber le comportement attendu sur le Web.



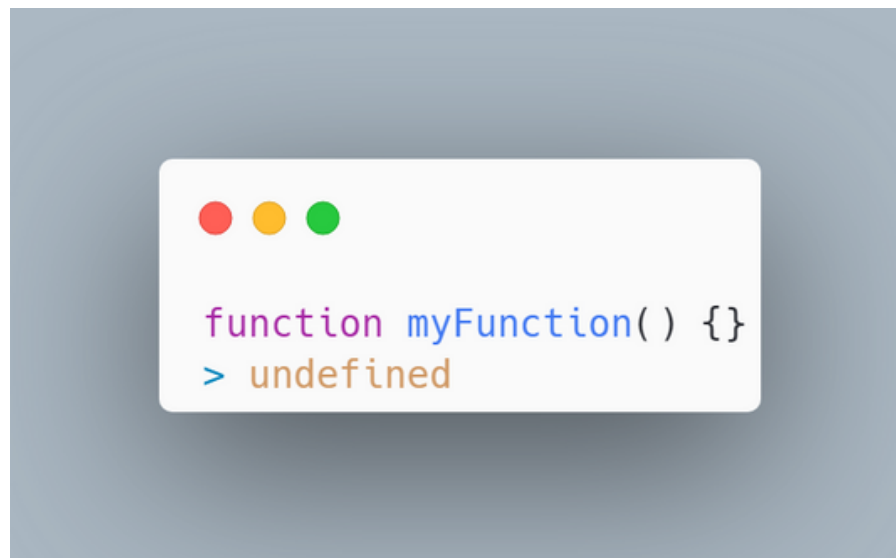
Vous pouvez définir une variable sur **null** pour indiquer qu'elle reflète soit une valeur qui lui est attribuée à un moment donné dans un script, soit une valeur explicitement absente. Vous pouvez également attribuer la valeur **null** à une référence existante pour effacer une valeur précédente.

null et undefined



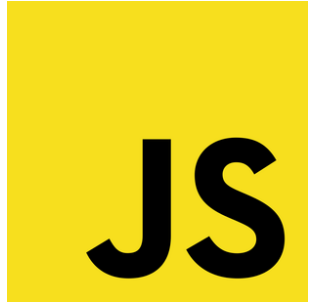
- **null**

undefined est une valeur primitive attribuée aux variables qui viennent d'être déclarées ou à la valeur résultante d'une opération qui ne renvoie pas de valeur significative. Cela peut se produire lorsque vous déclarez une fonction dans la console de développement d'un navigateur, par exemple:



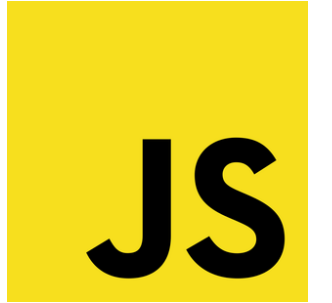
Une fonction renvoie explicitement **undefined** lorsque son instruction **return** ne renvoie aucune valeur.

BigInt



Les primitives BigInt sont un ajout relativement récent de JavaScript, permettant des opérations mathématiques sur des nombres non compris dans la plage autorisée par **Number**. Pour créer une valeur BigInt, ajoutez **n** à la fin d'un littéral de nombre, ou transmettez une valeur d'entier ou de chaîne numérique à la fonction **BigInt()**.

Symboles



Les symboles sont une primitive relativement nouvelle introduite dans ES6. Une primitive de symbole représente une valeur unique qui n'entre jamais en collision avec une autre valeur, y compris celles d'autres primitives de symbole. Deux primitives de chaîne composées de caractères identiques sont considérées comme strictement égaux:

Tableaux



Un tableau est un conteneur qui peut contenir zéro ou plusieurs valeurs de n'importe quel type de données, y compris des objets complexes ou d'autres tableaux. Les valeurs stockées dans un tableau sont parfois appelées "éléments" du tableau.



```
const a = [1, 2, 3];
```

```
a[0];
```

```
> 1
```

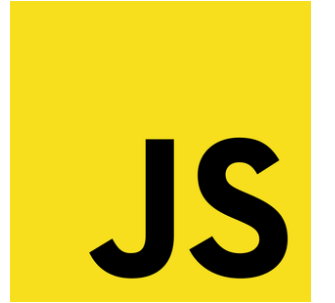
```
a[1];
```

```
> 2
```

```
a[2];
```

```
> 3
```

Fonctions



Une fonction JavaScript est un bloc de code conçu pour effectuer une tâche particulière.

Une fonction JavaScript est exécutée lorsque "quelque chose" l'invoque (l'appelle).



```
function myFunction() {  
  console.log( "Voici ma fonction." );  
};
```

```
myFunction();
```

```
> "Voici ma fonction."
```

Fonctions



Les **paramètres** de la définition de la fonction agissent comme des variables d'espace réservé pour les valeurs pouvant être transmises dans le corps de la fonction lorsque celle-ci est appelée. Lorsqu'une fonction est appelée, les valeurs entre parenthèses sont des "**arguments**"

La différence entre les **arguments** et les **paramètres** est la suivante : vous définissez les paramètres et c'est ce que vous voyez à l'intérieur de la fonction. Les arguments sont transmis par le programme lorsque vous appelez la fonction.



```
function test(color, age) {  
  // Faire quelque chose  
}  
  
test("vert", 24)  
test("noir", 0)
```

Fonctions



- Le mot clé “return”

Utilisez **return** pour spécifier la valeur que la fonction doit produire comme résultat final. Lorsque l'interpréteur atteint une instruction **return**, la fonction qui contient cette instruction se termine immédiatement et la valeur spécifiée est renvoyée au contexte dans lequel la fonction a été appelée:

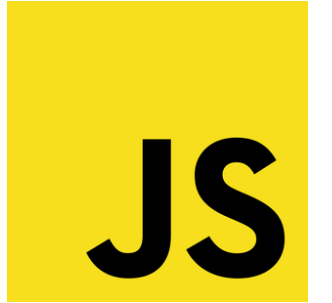
Dans votre programme, vous pouvez assigner la valeur de **retour** de la fonction à une variable, lorsque vous invoquez la fonction :

```
const myFunction = function() {  
  return 2 + 2;  
}  
  
myFunction();  
> 4
```

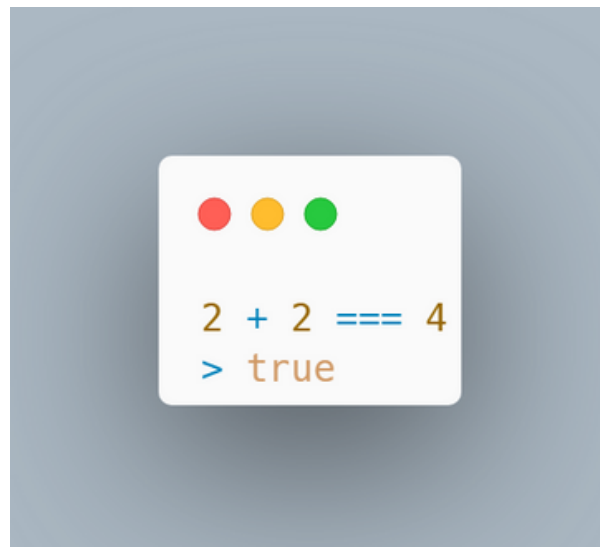
```
function myFunction() {  
  return  
}  
  
myFunction();  
> undefined
```

```
function test() {  
  // Faire quelque chose  
  return "Salut !"  
}  
  
const result = test()
```

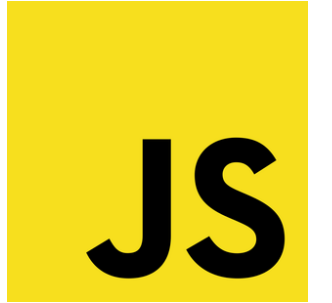
Opérateurs de comparaison



Les opérateurs de comparaison comparent les valeurs de deux opérandes et déterminent si l'instruction qu'ils forment est **true** ou **false**. L'exemple suivant utilise l'opérateur d'égalité stricte (**===**) pour comparer deux opérandes: l'expression **2 + 2** et la valeur **4**. Étant donné que le résultat de l'expression et la valeur numérique **4** sont identiques, cette expression renvoie **true** :



Opérateurs de comparaison



- **Coercition de type et égalité**

Deux des opérateurs de comparaison les plus fréquemment utilisés sont `==` pour une égalité libre et `===` pour une égalité stricte. `==` effectue une comparaison approximative entre deux valeurs en forçant les opérandes à correspondre aux types de données, si possible. Par exemple, `2 == "2"` renvoie **true**, même si la comparaison est effectuée entre une valeur numérique et une valeur de chaîne.



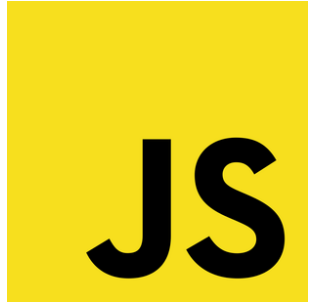
```
2 == 2
```

```
> true
```

```
2 == "2"
```

```
> true
```

Opérateurs de comparaison



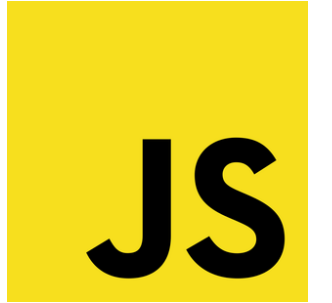
Il en va de même pour **!=**, qui ne renvoie **true** que si les opérandes comparés ne sont pas faiblement égaux.



```
2 != 3  
> true  
  
2 != "2"  
> false
```

A terminal window with a light gray background and rounded corners. At the top, there are three colored circles: red, yellow, and green. Below them, the text '2 != 3' is displayed in a monospace font, with '2' in blue, '!=' in red, and '3' in blue. The next line shows '> true' in a brownish-orange color. There is a blank line, followed by '2 != "2"' where '2' is blue, '!=' is red, and '"2"' is green. The final line shows '> false' in the same brownish-orange color.

Opérateurs de comparaison



Les comparaisons strictes utilisant `===` ou `!==` n'effectuent pas de coercition de type. Pour qu'une comparaison stricte soit évaluée sur **true**, les valeurs comparées doivent avoir le même type de données. De ce fait, `2 == "2"` renvoie **true**, mais `2 === "2"` renvoie **false**:



```
2 === 3  
> false
```

```
2 === "2"  
> false
```


Opérateurs de comparaison



Pour éliminer toute ambiguïté qui pourrait résulter de la coercition automatique, utilisez `===` dans la mesure du possible.

Opérateur	Description	Utilisation	Résultat
<code>===</code>	Strictement égal	<code>2 === 2</code>	<code>true</code>
<code>!==</code>	Pas strictement égal	<code>2 !== "2"</code>	<code>true</code>
<code>==</code>	Égal à (ou "grossièrement égal")	<code>2 == "2"</code>	<code>true</code>
<code>!=</code>	Not Equal (Non égal à)	<code>2 != "3"</code>	<code>true</code>
<code>></code>	Supérieur à	<code>3 > 2</code>	<code>true</code>
<code>>=</code>	Supérieur ou égal à	<code>2 >= 2</code>	<code>true</code>
<code><</code>	Moins de	<code>2 < 3</code>	<code>true</code>
<code><=</code>	Inférieur ou égal à	<code>2 <= 3</code>	<code>true</code>

Opérateurs de comparaison



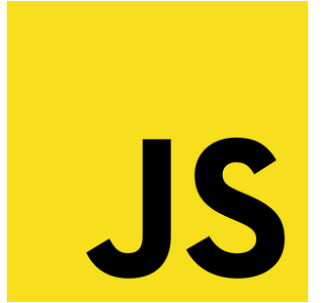
- **Vrai et faux**

Toutes les valeurs en JavaScript sont implicitement **true** ou **false** et peuvent être forcées sur la valeur booléenne correspondante, par exemple à l'aide du comparateur "grossièrement égal". Un ensemble limité de valeurs est converti (par coercion) en **false** :

- **0**
- **null**
- **undefined**
- **NaN**
- Une chaîne vide ("")

Toutes les autres valeurs sont converties de manière forcée en **true**, y compris toute chaîne contenant un ou plusieurs caractères et tous les nombres non nuls. Ce sont des valeurs communément appelées « vérité » (truthy) et « fausse » (falsy).

Opérateurs de comparaison



- **Opérateurs logiques**

Utilisez les opérateurs logiques AND (&&), OR (||) et NOT (!) pour contrôler le flux d'un script en fonction de l'évaluation d'au moins deux instructions conditionnelles :

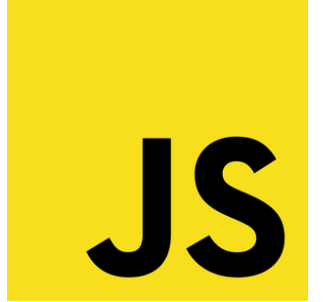


```
2 === 3 || 5 === 5;  
> true
```


```
2 === 2 && 2 === "2"  
> false
```

```
2 === 2 && !"Ma chaîne."  
> false
```

Opérateurs de comparaison

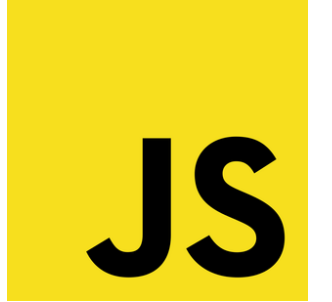


Une expression logique NOT (!) annule la valeur véridique ou fausse d'un opérande, en évaluant **true** si l'opérande renvoie false et false si l'opérande renvoie **true** :



```
true  
> true  
  
!true  
> false  
  
!false  
> true
```

Opérateurs de comparaison



L'utilisation de l'opérateur logique NOT (!) devant un autre type de données, comme un nombre ou une chaîne, convertit cette valeur en valeur booléenne et inverse la valeur véridique ou fausse du résultat.



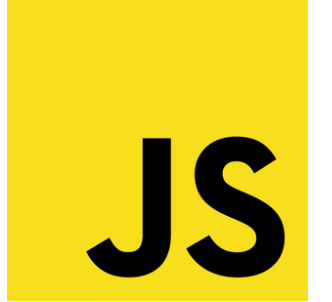
```
"string"  
> "string"
```

```
!"string"  
> false
```

```
0  
> 0
```

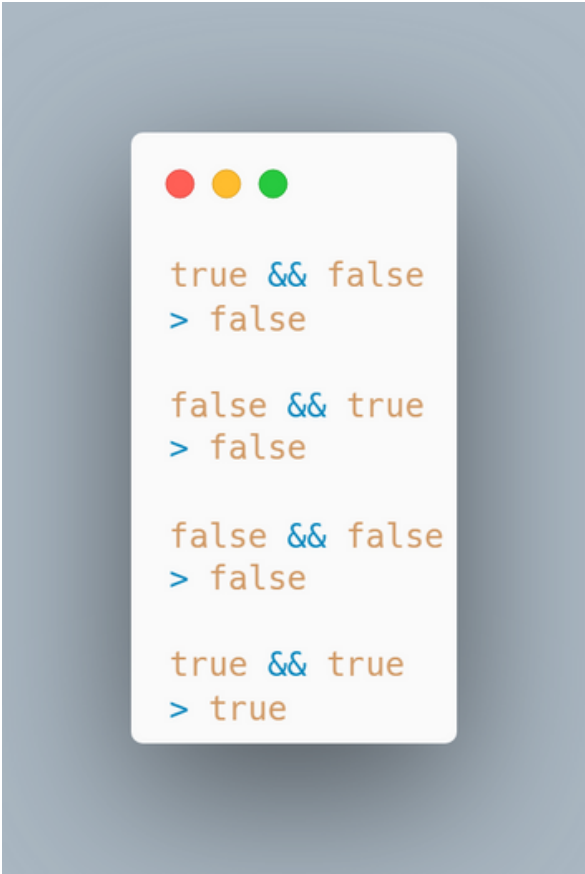
```
!0  
> true
```

Opérateurs de comparaison



Les opérateurs logiques AND et OR n'effectuent pas de coercion seuls. Elles renvoient la valeur de l'un des deux opérandes évalués, l'opérande choisi étant déterminé par cette évaluation.

L'opérateur logique AND (**&&**) ne renvoie le premier de ses deux opérandes que si son résultat est **false**, et le deuxième dans le cas contraire. Dans les comparaisons qui évaluent des valeurs booléennes, elle ne renvoie **true** que si les opérandes des deux côtés de l'opérateur logique AND ont la valeur **true**. Si l'un ou l'autre des côtés renvoie false, il renvoie **false**.

A terminal window with a light gray background and a white title bar containing three colored dots (red, yellow, green). It displays four lines of JavaScript code and their results.

```
true && false  
> false  
  
false && true  
> false  
  
false && false  
> false  
  
true && true  
> true
```

Opérateurs de comparaison



L'opérateur logique OU (||) ne renvoie le premier de ses deux opérandes que si son opérande renvoie **true**, et le deuxième dans le cas contraire. Dans les comparaisons qui renvoient des valeurs booléennes, cela signifie qu'elle renvoie **true** si l'un des opérandes renvoie **true**. Si aucun côté n'évalue la valeur **true**, il renvoie **false**:



```
true || false  
> true
```

```
false || true  
> true
```

```
true || true  
> true
```

```
false || false  
> false
```

Contrôle de flux



Le *flux de contrôle* correspond à l'ordre dans lequel l'interpréteur JavaScript exécute les instructions. Si un script n'inclut pas d'instructions qui modifient son flux, il est exécuté du début à la fin, ligne par ligne. Les *structures de contrôle* permettent de déterminer si un ensemble d'instructions est exécuté ou non en fonction d'un ensemble défini de critères, d'exécuter un ensemble d'instructions de manière répétée ou d'interrompre une séquence d'instructions.

Instructions conditionnelles

Les instructions conditionnelles déterminent si le code doit être exécuté en fonction d'une ou de plusieurs conditions. Une instruction conditionnelle exécute le code qu'elle contient si la condition (ou l'ensemble de conditions) associée renvoie la valeur **true**. Sinon, le code est ignoré.

Contrôle de flux



- **if...else**

Une instruction **if** évalue une condition à l'intérieur des parenthèses correspondantes qui suivent. Si la condition entre parenthèses renvoie la valeur **true**, l'instruction ou l'instruction de bloc qui suit les parenthèses correspondantes est exécutée :



```
if (true) console.log("True.");  
> "True."  
  
if (true) {  
    const myString = "True.";  
    console.log(myString);  
}  
> "True."
```

Contrôle de flux



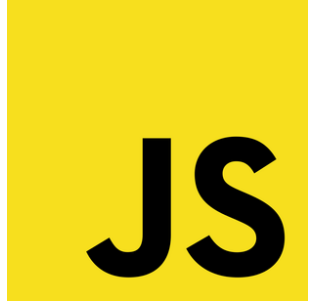
Si la condition entre parenthèses renvoie la valeur **false**, l'instruction qui la suit est ignorée:

```
if (false) console.log("True.");
```

Un mot clé **else** qui suit immédiatement une instruction **if** et son instruction exécutée de manière conditionnelle spécifie l'instruction à exécuter si la condition **if** renvoie la valeur **false** :

```
if (false) {  
  console.log("True.")  
} else {  
  console.log("False");  
}  
> "False."
```

Contrôle de flux

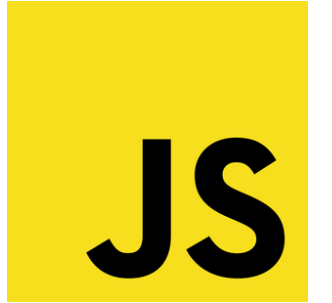


Pour enchaîner plusieurs instructions **if**, vous pouvez créer l'instruction exécutée de manière conditionnelle après **else**, une autre instruction **if** :

```
const myCondition = 2;

if (myCondition === 5) {
  console.log( "Five." );
} else if (myCondition === 2) {
  console.log( "Two." );
}
```

Contrôle de flux



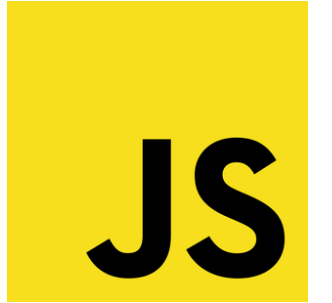
- **switch...case**

Une instruction **if...else** est très utile lorsque vous n'avez que quelques options à choisir. Cependant, lorsqu'elles sont trop nombreuses, elle peut s'avérer superflue. Votre code sera alors trop complexe. Dans ce cas, il est préférable d'utiliser un **switch**. En fonction du résultat de l'expression, JavaScript déclenchera un cas spécifique que vous aurez défini. Vous devez ajouter une instruction **break** à la fin de chaque cas, sinon JavaScript exécutera également le code dans le cas suivant (c'est parfois utile, mais attention aux bugs). Vous pouvez fournir un cas spécial par défaut, qui est appelé lorsqu'aucun cas ne traite le résultat de l'expression :

```
const a = 2

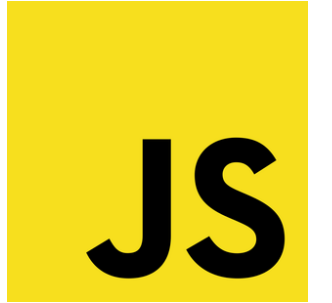
switch(a) {
  case 1:
    console.log("La valeur est un.")
    break
  case 2:
    console.log("La valeur est deux.")
    break
  case 3:
    console.log("La valeur est trois.")
    break
  default:
    console.log("La valeur est quelque chose d'inattendu.")
    break
}
```

Boucles et itérations



En utilisant une boucle, nous pouvons exécuter un morceau de code plusieurs fois pour effectuer toutes sortes de tâches.

Boucles et itérations



- **while**

Une boucle **while** est créée à l'aide du mot clé **while**, suivi d'une paire de parenthèses correspondantes contenant une condition à évaluer. Si la condition spécifiée renvoie la valeur **true**, l'instruction qui suit ces parenthèses est exécutée. Sinon, la boucle ne s'exécute jamais. Après chaque itération, la condition est réévaluée et si elle est toujours **true**, la boucle se répète. Vous pouvez passer à l'itération suivante en utilisant l'instruction **continue** :

```
let iterationCount = 0;

while(iterationCount < 5) {
  iterationCount++;

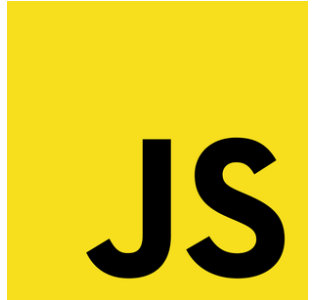
  if(iterationCount === 3) {
    continue;
  }

  console.log(`Loop ${ iterationCount }.`);
}

console.log("Boucle terminée.");

> "Boucle 1."
> "Boucle 2."
> "Boucle 4."
> "Boucle 5."
> "Boucle terminée."
```

Boucles et itérations



- **do...while**

do...while est une variante de la boucle while dans laquelle l'évaluation conditionnelle se produit à la fin de chaque itération de la boucle. Cela signifie que le corps de la boucle est toujours exécuté au moins une fois. Comme pour une boucle **while**, le cas d'utilisation le plus courant pour **do...while** est une boucle de longueur indéterminée:

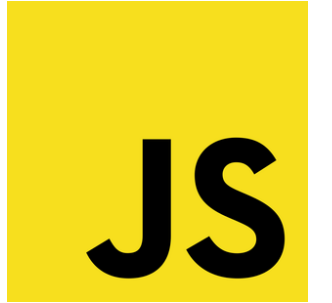
```
let randomNum;

do {
  randomNum = (() => Math.floor( Math.random() * 10 ))();
  console.log(`Est-ce le nombre ${randomNum} ?`);
} while (randomNum !== 3);

console.log(`Oui, ${randomNum} est le nombre correcte.`);

> "Est-ce le nombre 9?"
> "Est-ce le nombre 2?"
> "Est-ce le nombre 8?"
> "Est-ce le nombre 2?"
> "Est-ce le nombre 3?"
> "Oui, 3 est le nombre correcte."
```

Boucles et itérations



- **for**

Utilisez des boucles **for** pour itérer une quantité connue.

Pour créer une boucle **for**, utilisez le mot clé **for**, suivi d'une paire de parenthèses qui accepte les trois expressions suivantes dans l'ordre, séparées par un point-virgule :

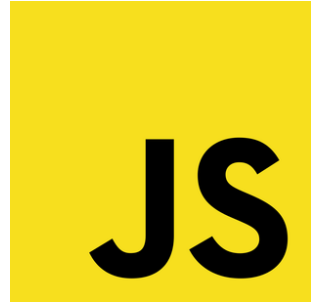
1. Expression à évaluer lorsque la boucle commence
2. Une condition qui détermine si la boucle doit continuer
3. Expression à exécuter à la fin de chaque boucle

Après ces parenthèses, ajoutez l'instruction à exécuter pendant la boucle.



```
for( let i = 0; i < 3; i++ ) {  
  console.log( "Cette boucle sera exécutée trois fois.")  
}
```


Les Objets



En JavaScript, nous utilisons des objets pour tout.

Ils constituent l'une des structures de données intégrées fondamentales, avec les tableaux.

Alors que les tableaux stockent les données dans des "emplacements" commençant à 0 et allant jusqu'à 1, 2, 3..., les objets stockent les données dans des propriétés.

```
const car = {  
  model: 'Fiesta',  
  color: 'green'  
}  
  
car.model = 'Fiesta'  
  
console.log(car.model)  
> 'Fiesta'
```