Javascript

Opérateurs

Opérateurs d'affectation

Nom	Opérateur	Signification
affectation	`x = y`	`x = y`
affectation après addition	`x += y`	`x = x + y`
affectation après soustraction	`x -= y`	`x = x - y`
affectation après multiplication	`x *= y`	`x = x * y`
affectation après division	`x /= y`	`x = x / y`
affectation du reste	`x %= y`	`x = x % y`

Opérateurs de comparaison

Nom	Op	Description
Égalité simple	==	Renvoie `true` si les opérandes sont égaux après conversion en valeurs de mêmes types.
Inégalité simple	!=	Renvoie `true` si les opérandes sont différents.
Égalité stricte	===	Renvoie `true` si les opérandes sont égaux et de même type.
Inégalité stricte	!==	Renvoie `true` si les opérandes ne sont pas égaux et/ou ne sont pas de même type.
Supériorité stricte	>	Renvoie `true` si l'opérande gauche est supérieur (strictement) à l'opérande droit.
Supériorité ou égalité	>=	Renvoie `true` si l'opérande gauche est supérieur ou égal à l'opérande droit.
Infériorité stricte	<	Renvoie `true` si l'opérande gauche est inférieur (strictement) à l'opérande droit.
Infériorité ou égalité	<=	Renvoie `true` si l'opérande gauche est inférieur ou égal à l'opérande droit.

Egalité simple / stricte

x	y	==	===
undefined	undefined	true	true
null	null	true	true
true	true	true	true
false	false	true	true
"toto"	"toto"	true	true
{ toto: "truc" }	x	true	true
0	0	true	true
+0	-0	true	true
0	false	true	false
""	false	true	false
""	0	true	false
"0"	0	true	false
"17"	17	true	false
[1,2]	"1,2"	true	false
new String("toto")	"toto"	true	false
null	undefined	true	false
null	false	false	false
undefined	false	false	false
{ toto: "truc" }	{ toto: "truc" }	false	false
new String("toto")	new String("toto")	false	false
0	null	false	false
0	NaN	false	false
"toto"	NaN	false	false
NaN	NaN	false	false

Algorithme d'égalité stricte

1. Si Type(*x*) est différent de Type(*y*), renvoie **false**.
2. Si Type(*x*) est **undefined**, renvoie **true**.
3. Si Type(*y*) est **null**, renvoie **true**.
4. Si Type(*x*) est number, alors
    1. Si *x* est **NaN**, renvoie **false**.
    2. Si *y* est **NaN**, renvoie **false**.
    3. Si *x* est le même nombre que *y*, renvoie **true**.
    4. Si *x* est **+0** et *y* est **−0**, renvoie **true**.
    5. Si *x* est **−0** et *y* est **+0**, renvoie **true**.
    6. Renvoie **false**.
5. Si Type(*x*) est **string**, alors renvoie **true** si *x* et *y* sont exactement la même séquence de caractères (même longueur et mêmes caractères dans les positions correspondantes). Sinon, renvoie **false**.
6. Si Type(*x*) est **boolean**, renvoie **true** si *x* et *y* sont tous les deux **true** ou tous les deux **false**. Sinon, renvoie **false**.
7. Renvoie **true** si *x* et *y* font référence au même objet. Sinon, renvoie **false**.

Algorithme d'égalité simple

1. Si Type(*x*) est le même que Type(*y*), applique l'algorithme d'égalité stricte.
2. Si *x* est **null** et *y* est **undefined**, renvoie **true**.
3. Si *x* est **undefined** et *y* est **null**, renvoie **true**.
4. Si Type(*x*) est **number** et Type(*y*) est **string**, renvoie le résultat de la comparaison *x* == ToNumber(*y*).
5. Si Type(*x*) est **string** et Type(*y*) est **number**, renvoie le résultat de la comparaison ToNumber(*x*) == *y*.
6. Si Type(*x*) est **boolean**, renvoie le résultat de la comparaison ToNumber(*x*) == *y*.
7. Si Type(*y*) est **boolean**, renvoie le résultat de la comparaison *x* == ToNumber(*y*).
8. Si Type(*x*) est soit **string** soit **number** et Type(*y*) est **object**, renvoie le résultat de la comparaison *x* == ToPrimitive(*y*).
9. Si Type(*x*) est **object** et Type(*y*) est soit **string** soit **number**, renvoie le résultat de la comparaison ToPrimitive(*x*) == *y*.
10. Renvoie **false**.

Egalité simple / stricte

Pour éviter toute ambigüité, il est conseillé de **toujours** utiliser l'égalité stricte.

Il existe une exception, le test d'égalité avec la valeur `null`, vrai lorsque la valeur testée est `null` ou `undefined`.

```
function aucunArgument(arg) {
  return arg == null;
}

aucunArgument(); // true
aucunArgument(null); // true
aucunArgument(undefined); // true

aucunArgument(0); // false
aucunArgument(""); // false
```

Opérateurs arithmétiques

Nom	Op	Description	Exemple
Reste	%	Opérateur binaire. Renvoie le reste entier de la division entre les deux opérandes.	12 % 5 renvoie 2.
Incrément	++	Opérateur unaire. Ajoute un à son opérande. S'il est utilisé en préfixe (++x), il renvoie la valeur de l'opérande après avoir ajouté un, s'il est utilisé comme opérateur de suffixe (x++), il renvoie la valeur de l'opérande avant d'ajouter un.	Si x vaut 3, ++x incrémente x à 4 et renvoie 4, x++ renvoie 3 et seulement ensuite ajoute un à x.
Décrément	--	Opérateur unaire. Il soustrait un à son opérande. Il fonctionne de manière analogue à l'opérateur d'incrément.	Si x vaut 3, --x décrémente x à 2 puis renvoie2, x-- renvoie 3 puis décrémente la valeur de x.
Négation unaire	-	Opérateur unaire. Renvoie la valeur opposée de l'opérande.	Si x vaut 3, alors -x renvoie -3.
Plus unaire	+	Opérateur unaire. Si l'opérande n'est pas un nombre, il tente de le convertir en une valeur numérique.	`+"3"` renvoie `3`. `+true` renvoie 1.
Exponentiel	**	Opérateur d'exponentiation. Fournit le résultat obtenu lorsqu'on élève le premier opérande à la puissance indiquée par le second.	`3**4` renvoie 81.

Opérateurs logiques

Nom	Op	Usage	Description
ET logique	`&&`	`expr1 && expr2`	Renvoie `expr2` si `expr1` peut être converti à true, sinon renvoie `expr1`.
OU logique	`||`	`expr1 || expr2`	Renvoie `expr1` s'il peut être converti à `true`, sinon renvoie `expr2`.
NON logique	`!`	`!expr`	Renvoie `true` si son unique opérande peut être converti à `false`, sinon renvoie `false`.
Opérateur de coalescence des nuls	`??`	`expr1 ?? expr2`	Renvoie `expr1` s'il est différent de `null` et `undefined`, sinon renvoie `expr2`.

Opérateurs Logiques

ET logique

Utile pour accéder à la propriété d'un objet potentiellement inexistant.

```
let age = toto && toto.age; // undefined si toto n'existe pas
```

Opérateurs Logiques

OU logique

Utile pour définir une valeur par défaut

```
let age = toto.age || 25; // 25 si toto n'a pas de propriété age
```

Opérateurs Logiques

Opérateur de coalescence des nuls

```
let toto = { age : 0 }

// Attention si la première expression est une valeur de type fausse !
let age = toto.age || 25; // 25

let age = toto.age ?? 25; // 0
// Renvoie le 2ème opérande seulement si le premier est null ou undefined
```

Chaînage optionnel

Il permet d'éviter l'emploi du ET logique

```
let age = toto?.age;
// renvoie undefined si une référence est null ou undefined
```

On peut l'utiliser avec la notation crochet et les fonctions.

```
let age = toto?.["age"];
// renvoie undefined si toto n'existe pas

toto?.renaître?.()
// renvoie undefined si toto n'existe pas ou s'il ne possède pas de méthode renaître.
```

Opérateurs relationnels

`in` : renvoie true si la propriété donnée appartient à l'objet

```
let toto = { nom : "Toto", age : 25, activites : null };
"nom" in toto; //true
"Toto" in toto; //false
"activites" in toto; //true
```

`instanceof` : renvoie true si l'objet donné est instance du constructeur spécifié

```
let date = new Date();
date instanceof Date; //true
date instanceof Object; //true
date instanceof Array; //false
```

Opérateur typeof

Renvoie une chaîne qui indique le type de son opérande.

Type	**Résultat**
indéfini	`"undefined"`
nul	`"object"`
booléen	`"boolean"`
nombre	`"number"`
chaîne de caractère	`"string"`
fonction	`"function"`
tableau	`"object"`
tout autre objet	`"object"`

Opérateur typeof

```
typeof null == "object";
```

Ceci est un bug de conception de javascript, car `null` est une valeur primitive.

Un correctif fut proposé pour ECMAScript mais il fut refusé, car trop de sites auraient été impactés.

Opérateur typeof

```
typeof [] == "object";
```

Ceci n'est pas un bug de conception, les tableaux sont bien des objets.

Mais ce n'est pas cohérent avec les fonctions qui sont aussi des objets mais renvoient `"function"`.

Pour tester si une variable est un tableau, il faut utiliser la méthode statique `isArray` du constructeur `Array` :

```
let toto = [];

Array.isArray(toto); //true
```

Précédence des opérateurs

Type d'opérateur	Opérateurs individuels
membre	`. []`
appel/création d'instance	`() new`
négation/incrémentation	`! ~ - + ++ -- typeof void delete`
multiplication/division	`* / %`
addition/soustraction	`+ -`
décalage binaire	`<< >> >>>`
relationnel	`< <= > >= in instanceof`
égalité	`== != === !==`
ET binaire	`&`
OU exclusif binaire	`^`
OU binaire	`|`
ET logique	`&&`
OU logique	`||`
conditionnel	`?:`
assignation	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`
virgule	`,`

Opérateur de groupement

L'opérateur de groupement consiste en une paire de parenthèses encadrant une expression et permettant de surcharger la précédence normale des opérateurs.

```
let a = 1, b = 2, c = 3;

a + b * c; //7
(a + b) * c;   // 9

typeof a == b; //false
typeof (a == b); //boolean
```