JS

Javascript les bases

JS

Qu’est ce que Javascript ?

Introduction au javascript

Qu'est-ce que JavaScript ?

JavaScript a été initialement créé pour "rendre les pages web vivantes".

Les programmes dans ce langage sont appelés scripts. Ils peuvent être écrits directement dans une page HTML et exécutés automatiquement au chargement des pages.

Les scripts sont fournis et exécutés en texte brut. Ils n'ont pas besoin d'une préparation spéciale ou d'une compilation pour fonctionner.

De par cet aspect, JavaScript est très différent d'un autre langage appelé Java.

Pourquoi est-il appelé JavaScript ?

Quand JavaScript a été créé, il portait initialement un autre nom : "LiveScript". Mais à cette époque le langage Java était très populaire, il a donc été décidé que positionner un nouveau langage en tant que "petit frère" de Java pourrait aider. Mais au fur et à mesure de son évolution, JavaScript est devenu un langage totalement indépendant, avec ses propres spécifications appelées ECMAScript, aujourd'hui il n'a aucun rapport avec Java.

Introduction au javascript

Aujourd'hui, JavaScript peut s'exécuter non seulement dans le navigateur, mais également sur un serveur, ou encore sur n'importe quel appareil dans lequel existe un programme appelé le moteur JavaScript.
Le navigateur a un moteur intégré, parfois il peut être également appelé "la machine virtuelle JavaScript".

Différents moteurs ont différents "nom de code", par exemple : V8 – dans Chrome et Opera ou SpiderMonkey – dans Firefox. ... Il existe d'autres noms de code comme "Chakra" pour IE, "JavaScriptCore", "Nitro" et "SquirrelFish" pour Safari etc.
Les termes ci-dessus sont bons à retenir, car ils sont utilisés dans les articles destinés aux développeurs sur Internet. Nous les utiliserons aussi. Par exemple, si "une fonctionnalité X est prise en charge par V8", cela fonctionne probablement dans Chrome, Edge et Opera.

Comment fonctionnent les moteurs ? *Les moteurs sont compliqués. Mais le fonctionnement de base est facile à comprendre. Le moteur (intégré si c'est un navigateur) lit ("analyse") le script. Ensuite, il convertit ("compile") le script en langage machine. Enfin le code machine s'exécute, très rapidement.*
Le moteur applique des optimisations à chaque étape du processus. Il surveille même le script compilé en cours d'exécution, analyse les données qui le traversent et applique des optimisations au code machine en fonction de ces informations.

Introduction au javascript

Que peut faire JavaScript dans le navigateur ?

Le JavaScript moderne est un langage de programmation "sûr". Il ne fournit pas d'accès de bas niveau à la mémoire ou au processeur, parce qu'il a été initialement conçu pour les navigateurs qui n'en ont pas besoin.

Les fonctionnalités dépendent grandement de l'environnement qui exécute JavaScript. Par exemple, Node.js prend en charge les fonctions qui permettent à JavaScript de lire / écrire des fichiers arbitrairement, d'exécuter des requêtes réseau, etc. JavaScript intégré au navigateur peut faire tout ce qui concerne la manipulation des pages Web, l'interaction avec l'utilisateur et le serveur Web.

Par exemple, JavaScript dans le navigateur est capable de :

- Ajouter un nouveau code HTML à la page, modifier le contenu existant, modifier les styles.
- Réagir aux actions de l'utilisateur, s'exécuter sur des clics de souris, des mouvements de pointeur, des appuis sur des touches.
- Envoyer des requêtes sur le réseau à des serveurs distants, télécharger et envoyer des fichiers (AJAX).
- Obtenir et définir des cookies, poser des questions au visiteur, afficher des messages.
- Se souvenir des données du côté client ("stockage local").

Introduction au javascript

Qu'est-ce que JavaScript ne peut pas faire dans le navigateur ?

Les capacités de JavaScript dans le navigateur sont limitées pour la sécurité de l'utilisateur. L'objectif est d'empêcher une page Web malveillante d'accéder à des informations privées ou de nuire aux données de l'utilisateur.

Les exemples de telles restrictions sont :

- JavaScript sur une page Web ne peut pas lire/écrire des fichiers arbitrairement sur le disque dur, les copier ou exécuter des programmes. Il n'a pas d'accès direct aux fonctions du système d'exploitation.
  Les navigateurs modernes lui permettent de fonctionner avec des fichiers, mais l'accès est limité et n'est fourni que si l'utilisateur effectue certaines actions, comme «déposer» un fichier dans une fenêtre de navigateur ou le sélectionner via une balise <input>. Il existe des moyens d'interagir avec une webcam/microphone et d'autres périphériques, mais ils nécessitent une autorisation explicite de l'utilisateur. Ainsi, une page contenant du JavaScript ne permet pas d'activer une caméra Web, d'observer l'environnement et d'envoyer les informations à la NSA.
- Différents onglets / fenêtres ne se connaissent généralement pas. Parfois, ils se croisent, par exemple lorsqu'une fenêtre utilise JavaScript pour ouvrir l'autre. Mais même dans ce cas, le JavaScript d'une page ne peut pas accéder à l'autre si elle provient de sites différents (provenant d'un autre domaine, protocole ou port). C'est ce qu'on appelle la "politique de même origine" ("Same Origin Policy")...

Introduction au javascript

... Pour contourner cette sécurité, les deux pages doivent se mettre d'accord et contenir un code JavaScript spécial qui gère l'échange de données. Nous verrons cela plus loin dans ce cours. Cette limitation concerne également la sécurité de l'utilisateur. Une page de http://autresite.com qu'un utilisateur a ouvert ne doit pas pouvoir accéder à un autre onglet du navigateur avec l'URL http://gmail.com et y voler des informations.

- JavaScript peut facilement communiquer sur le net avec le serveur d'où provient la page. Mais sa capacité à recevoir des données d'autres sites / domaines est paralysée. Bien que possible, il nécessite un accord explicite (exprimé dans les en-têtes HTTP) du côté distant. Encore une fois, ce sont des limites de sécurité.

De telles limites n'existent pas si JavaScript est utilisé en dehors du navigateur, par exemple sur un serveur. Les navigateurs modernes permettent également l'installation de plug-ins / extensions susceptibles d'obtenir des autorisations étendues.

Introduction au javascript

Qu'est-ce qui rend JavaScript unique ?

Il y a au moins trois bonnes choses à propos de JavaScript :

- Intégration complète avec HTML / CSS.
- Les choses simples sont faites simplement.
- Pris en charge par tous les principaux navigateurs et activé par défaut.

JavaScript est la seule technologie de navigateur qui combine ces trois éléments.
C'est ce qui rend JavaScript unique. C'est pourquoi il l'outil le plus répandu pour créer des interfaces de navigateur. Cela dit, JavaScript permet également de créer des serveurs, des applications mobiles, etc.

Les langages "par dessus" JavaScript

La syntaxe de JavaScript ne convient pas aux besoins de tous. Différentes personnes veulent des fonctionnalités différentes.
Il faut s'y attendre, parce que besoins sont différents pour tous.
Donc, récemment, une pléthore de nouveaux langages sont apparus, qui sont transpilés (convertis) en JavaScript avant leur exécution dans le navigateur.

Introduction au javascript

Spécification

The ECMA-262 specification contient les informations les plus détaillées et formalisées sur JavaScript. C'est elle qui définit le langage. Une nouvelle version de la spécification est publiée chaque année. La dernière version en cours est disponible à cette adresse : *https://tc39.es/ecma262/*.

Pour en savoir plus sur les fonctionnalités à venir, y compris celles qui sont "presque standards" (appelées aussi "stage 3"), vous pouvez consulter les propositions à cette adresse : *https://github.com/tc39/proposals*.

De plus, si vous développez pour le navigateur, d'autres spécifications sont couvertes dans la seconde partie du tutoriel.

Manuels

La référence **MDN** (Mozilla) JavaScript est le principal manuel avec des exemples et d'autres informations. C'est une excellente source pour obtenir des informations détaillées sur les fonctions linguistiques, les méthodes, etc.

On peut la trouver à cette adresse : *https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference*.

Cependant, il est souvent préférable d'utiliser une recherche sur Internet.

Utilisez simplement "MDN [terme]" dans la requête, par exemple **https://google.com/search?q=MDN+parseInt** pour rechercher la fonction **parseInt**.

Introduction au javascript

Tableaux de compatibilité
JavaScript est un langage en développement, de nouvelles fonctionnalités sont ajoutées régulièrement.
Pour voir si elles sont supportées dans les moteurs, au sein des navigateurs et autres, voir :

- **http://caniuse.com** – tables de prise en charge par fonctionnalité, par exemple pour voir quels moteurs supportent les fonctions de cryptographie modernes : http://caniuse.com/#feat=cryptography.
- **https://kangax.github.io/compat-table** – un tableau avec les fonctionnalités linguistiques et les moteurs qui les prennent en charge ou non.

Toutes ces ressources sont utiles dans le quotidien des développeurs, parce qu'elles contiennent des informations précieuses sur les fonctionnalités du langage, leur support, etc.

Veuillez vous en souvenir (ou de cette page) pour les cas où vous avez besoin d'informations détaillées sur une fonctionnalité particulière.

Introduction au javascript

La console de développement

Le code est sujet aux erreurs. *Vous êtes susceptible de faire des erreurs...*
Vous allez absolument faire des erreurs, du moins si vous êtes un humain, pas un robot.
Mais dans le navigateur, un utilisateur ne voit pas les erreurs par défaut. Ainsi, si quelque chose ne va pas dans le script, nous ne verrons pas ce qui ne va pas et nous ne pourrons pas le réparer.

Pour voir les erreurs et obtenir beaucoup d'informations utiles sur les scripts, les navigateurs intègrent des "*outils de développement*".
Le plus souvent, les développeurs se tournent vers Chrome ou Firefox pour le développement, car ces navigateurs disposent des meilleurs outils de développement.
D'autres navigateurs fournissent également des outils de développement, parfois dotés de fonctions spéciales, mais jouent généralement le rôle de "rattrapage" pour Chrome ou Firefox.
Donc, la plupart des gens ont un navigateur "favori" et passent à d'autres si un problème est spécifique au navigateur. Les outils de développement sont très puissants ; ils possèdent de nombreuses fonctionnalités. Pour commencer, nous allons apprendre à les ouvrir, à examiner les erreurs et à exécuter des commandes JavaScript.

Introduction au javascript

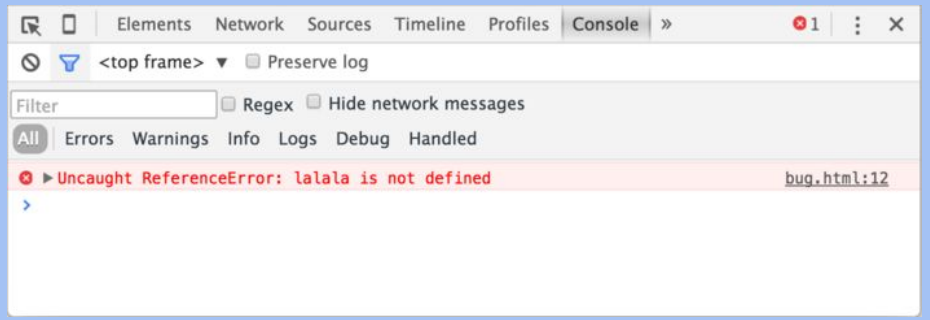
Nous allons créer une page bug.html avec le code ci-contre, et afficher la dans votre navigateur.

Il y a une erreur dans le code JavaScript. Elle est invisible à un visiteur habituel, alors ouvrons les outils de développement pour la voir.

Appuyez sur F12 ou, si vous êtes sur Mac, utilisez la combinaison Cmd+Opt+J.

Les outils de développement s'ouvriront sous l'onglet Console par défaut.

Cela ressemble un peu à ceci :




```
<!DOCTYPE HTML>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
Il y'a une erreur de script
sur cette page
<script>
lalala
</script>
</body>
</html>
```

Introduction au javascript

L'aspect exact des outils de développement dépend de votre version de Chrome. Cela change de temps en temps, mais devrait être similaire.

Ici, nous pouvons voir le message d'erreur de couleur rouge. Dans ce cas, le script contient une commande "*lalala*" inconnue.
Sur la droite, il y a un lien cliquable vers le code source **bug.html:12** avec le numéro de ligne où l'erreur s'est produite.
Sous le message d'erreur, il y a un symbole bleu >. Il marque une "ligne de commande" où l'on peut taper des commandes JavaScript et appuyer sur Enter pour les exécuter.

Nous pouvons maintenant voir les erreurs et c'est suffisant pour le début.
Nous reviendrons plus tard sur les outils de développement et approfondirons le débogage plus tard.

Multi-line input
*Habituellement, quand on met une ligne de code dans la console, puisque l'on appuie sur **Enter**, elle s'exécute.*
*Pour insérer plusieurs lignes, appuyez sur **Shift+Enter**.*
De cette façon, on peut saisir de longs fragments de code JavaScript.

Introduction au javascript

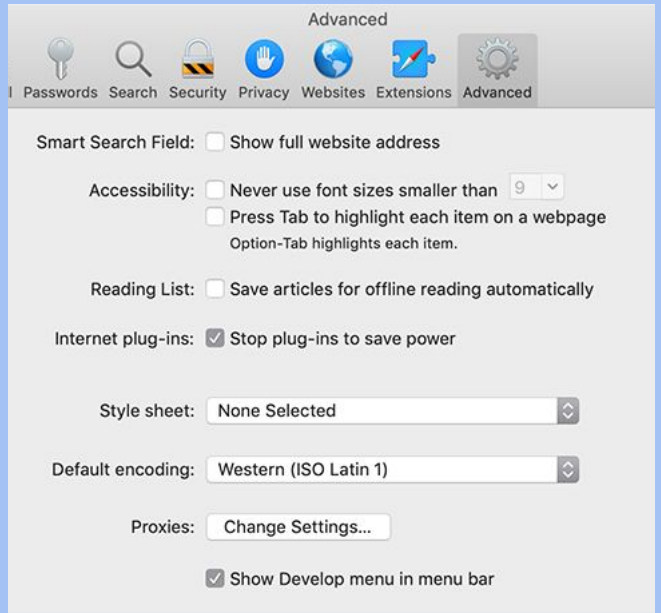
Firefox, Edge et autres

La plupart des autres navigateurs utilisent F12 pour ouvrir les outils de développement.
Leur apparence est assez similaire. Une fois que vous savez comment utiliser l'un d'entre eux (vous pouvez commencer avec Chrome), vous pouvez facilement passer à un autre.

Safari

Safari (navigateur Mac, non pris en charge par Windows / Linux) est un peu spécial ici. Nous devons d'abord activer le menu "Développement".

Ouvrez les préférences et accédez au volet "Avancé". Il y a une case à cocher en bas :



JS

Les fondamentaux du Javascript

JS

Les fondamentaux du javascript

Hello World !

Hello World

Hello, world!

Cette partie du tutoriel est à propos du coeur de JavaScript, le langage lui même.

Mais nous avons besoin d'un environnement de travail pour exécuter nos scripts et le navigateur est un bon choix.

Alors, voyons d'abord comment intégrer un script à une page Web.

La balise "script"

Les programmes JavaScript peuvent être insérés dans n'importe quelle partie d'un document HTML à l'aide de la balise **<script>**. Ajouter le code ci-contre dans votre fichier *bug.html*

La balise **<script>** contient du code JavaScript qui est automatiquement exécuté lorsque le navigateur rencontre la balise.

```
<!DOCTYPE HTML>
<html>
<head>
 <meta charset="utf-8">
 <title>Script</title>
</head>
<body>
<p>Texte avant le script</p>
<script>
alert( 'Hello, world!' );
</script>
<p>Texte après le script</p>
</body>
</html>
```

Hello World

Le balisage moderne

La balise **<script>** a quelques attributs qui sont rarement utilisés de nos jours, mais nous pouvons les trouver dans l'ancien code :

L'attribut **type** : **<script type=...>**
L'ancien standard HTML 4, nécessitait pour chaque script d'avoir un type.
Habituellement c'était *type="text/javascript"*. Dorénavant ce n'est plus nécessaire. De plus, le standard HTML moderne a totalement changé la signification de cet attribut. Maintenant, il peut être utilisé pour les modules JavaScript. Mais c'est un sujet avancé, nous parlerons de modules plus tard.

L'attribut **language** : **<script language=...>**
Cet attribut était destiné à afficher le langage du script. Pour l'instant, cet attribut n'a aucun sens, le langage est le JavaScript par défaut. Pas besoin de l'utiliser.

Hello World

Commentaires avant et après les scripts.

Dans des livres et des guides vraiment anciens, on peut trouver des commentaires dans <script>, comme ceci :

```
<script type="text/javascript"><!--
...
//--></script>
```

Cette astuce n'est plus utilisée dans le JavaScript moderne. Ces commentaires ont été utilisés pour masquer le code JavaScript des anciens navigateurs qui ne savaient pas comment traiter une balise <script>. Comme les navigateurs nés au cours des 15 dernières années n'ont pas ce problème, ce type de commentaire peut vous aider à identifier un code très ancien.

Hello World

Scripts externes

Si nous avons beaucoup de code JavaScript, nous pouvons le placer dans un fichier séparé.
Le fichier de script est attaché au fichier HTML avec l'attribut **src** :

```
<script src="/path/to/script.js"></script>
```

Ici, **/path/to/script.js** est un chemin absolu du script depuis la racine du site. On peut également fournir un chemin relatif à partir de la page en cours. Par exemple **src="script.js"** signifierait un fichier "**script.js**" dans le dossier courant.
Nous pouvons également donner une URL complète, par exemple :

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>
```

Pour joindre plusieurs scripts, utilisez plusieurs tags :

```
<script src="js/scriptA.js"></script>
<script src="js/scriptB.js"></script>
```

Hello World

Veuillez noter :
En règle générale, seuls les scripts les plus simples sont mis en HTML. Les plus complexes résident dans des fichiers séparés.
L'avantage d'un fichier séparé est que le navigateur le télécharge puis le stocke dans son cache.
Après cela, les autres pages qui veulent le même script le récupéreront à partir du cache au lieu de le télécharger à nouveau. Le fichier n'est donc téléchargé qu'une seule fois.
Cela économise du trafic et rend les pages plus rapides.

Attention : *Si **src** est défini, le contenu de la balise script est ignoré.*
*Une seule balise **<script>** ne peut pas avoir à la fois l'attribut **src** et le code à l'intérieur.*

Ceci ne fonctionnera pas :

```
<script src="script.js">
alert( 'Hello, world!' ); // le contenu est ignoré, parce
que src est défini
</script>
```

Hello World

Nous devons choisir : soit un script ***<script src = "...">*** *externe, soit un* ***<script>*** *régulier avec du code. L'exemple ci-dessus peut être divisé en deux scripts pour fonctionner :*

```
<script src="script.js"></script>
<script>
alert( 'Hello, world!' );
</script>
```

JS

Les fondamentaux du javascript

Structure du code

Structure du code

Structure du code

La première chose à apprendre est de construire des blocs de code.

Instructions

Les instructions sont des constructions de syntaxe et des commandes qui effectuent des actions.
Nous avons déjà vu une instruction **alert("Hello World!")** qui affiche le message.
Nous pouvons avoir autant d'instructions dans le code que nous le souhaitons. Une autre instruction peut être séparée par un point-virgule.
Par exemple, ici nous divisons le message en deux:

```
alert( 'Hello' );alert( 'World!' );
```

Chaque instruction est généralement écrite sur une ligne distincte – le code devient donc plus lisible :

```
alert( 'Hello' );
alert( 'World!' );
```

Structure du code

Les points-virgules

Un point-virgule peut être omis dans la plupart des cas lorsqu'une rupture de ligne existe.

Cela fonctionnerait aussi :

```
alert( 'Hello' )
alert( 'World!' )
```

Ici, JavaScript interprète le saut de ligne comme un point-virgule "implicite". Cela s'appelle également une insertion automatique de point-virgule.

Dans la plupart des cas, une nouvelle ligne implique un point-virgule. Mais "dans la plupart des cas" ne signifie pas "toujours"!

Il y a des cas où une nouvelle ligne ne signifie pas un point-virgule, par exemple :

```
alert(3 +
1
+ 2);
```

Le code génère 6, car JavaScript n'insère pas de point-virgule ici. Il est intuitivement évident que si la ligne se termine par un plus "+", alors c'est une "expression incomplète", donc un point-virgule serait incorrect. Et dans ce cas, cela fonctionne comme prévu.

Structure du code

Mais il existe des situations où JavaScript "échoue" à prendre un point-virgule là où il est vraiment nécessaire. Les erreurs qui surviennent dans de tels cas sont assez difficiles à trouver et à corriger.

```
alert("Hello");
[1, 2].forEach(alert);
```

Pas besoin de penser à la signification des crochets [] et forEach pour le moment. Nous les étudierons plus tard, pour l'instant, retenons simplement le résultat de l'exécution du code : il affiche Hello, puis 1, puis 2.

Supprimons maintenant le point-virgule après l'alert :

```
alert("Hello")
[1, 2].forEach(alert);
```

La différence par rapport au code ci-dessus n'est qu'un caractère : le point-virgule à la fin de la première ligne a disparu. Si nous exécutons ce code, seul le premier Hello s'affiche (et il y a une erreur, vous devrez peut-être ouvrir la console pour le voir). Il n'y a plus de chiffres.

Structure du code

C'est parce que JavaScript ne suppose pas de point-virgule avant les crochets [...]. Ainsi, le code du dernier exemple est traité comme une seule instruction.

Voici comment le moteur le voit :

```
alert("Hello")[1, 2].forEach(alert);
```

Ça a l'air bizarre, non ? Une telle fusion dans ce cas est tout simplement une erreur. Nous devons mettre un point-virgule après l'alert pour que le code fonctionne correctement.

Cela peut arriver dans d'autres situations aussi.

Il est recommandé de mettre les points-virgules entre les instructions, même si elles sont séparées par des nouvelles lignes. Cette règle est largement adoptée par la communauté. Notons encore une fois – il est possible de laisser de côté les points-virgules la plupart du temps. Mais il est plus sûr – surtout pour un débutant – de les utiliser.

Structure du code

Les Commentaires

Au fil du temps, le programme devient de plus en plus complexe. Il devient nécessaire d'ajouter des commentaires qui décrivent ce qui se passe et pourquoi.

Les commentaires peuvent être placés à n'importe quel endroit du script. Ils n'affectent pas l'exécution, car le moteur les ignore simplement.

Les commentaires sur une ligne commencent par deux barres obliques **//**.

Le reste de la ligne est un commentaire. Il peut occuper une ligne complète ou suivre une déclaration.

Comme ici :

```
// Ce commentaire occupe une ligne à part
alert("Hello")
[1, 2].forEach(alert); // Ce commentaire suit l'instruction
```

Structure du code

Les commentaires multilignes commencent par une barre oblique et un astérisque **/*** et se termine par un astérisque et une barre oblique ***/**.

```
/* Un exemple avec deux messages.
C'est un commentaire multiligne.
*/
alert("Hello")
alert("World");
```

Comme ceci :

Le contenu des commentaires est ignoré, donc si nous mettons du code à l'intérieur /* ... */ il ne s'exécutera pas.

Parfois, il est utile de désactiver temporairement une partie du code :

```
/* Commenter le code.
alert("Hello")
*/
alert("World");
```

Structure du code

Veuillez noter :
Dans la plupart des éditeurs, une ligne de code peut être commentée par le raccourci Ctrl+/ pour un commentaire sur une seule ligne et quelque chose comme Ctrl+Shift+/ – pour les commentaires multilignes (sélectionnez un morceau de code et appuyez sur la combinaison de touches). Pour Mac essayez Cmd au lieu de Ctrl et Option au lieu de Shift.

Attention : *Les commentaires imbriqués ne sont pas supportés !*
Il peut ne pas y avoir ***/*...*/ à l'intérieur d'un autre /*...*/.***
Le code ci-contre se terminera avec une erreur !

```
/*
  /* commentaire imbriqué ?!? */
*/
alert( 'World' );
```

N'hésitez pas à commenter votre code.
Les commentaires augmentent la taille globale du code, mais ce n'est pas un problème du tout. De nombreux outils permettent de réduire le code avant de le publier sur le serveur de production. Ils suppriment les commentaires, ils n'apparaissent donc pas dans les scripts de travail. Les commentaires n'ont donc aucun effet négatif sur la production.

JS

Les fondamentaux du javascript

Le mode moderne, "use strict"

Le mode moderne, "use strict"

Le mode moderne, "use strict"
JavaScript a longtemps évolué sans problèmes de compatibilité. De nouvelles fonctionnalités ont été ajoutées au langage, mais les anciennes fonctionnalités n'ont pas été modifiées.
Cela a l'avantage de ne jamais casser le code existant. Mais l'inconvénient est que toute erreur ou décision imparfaite prise par les créateurs de JavaScript est restée bloquée dans le langage pour toujours.
Il en a été ainsi jusqu'en 2009 lorsque ECMAScript 5 (ES5) est apparu.
Il a ajouté de nouvelles fonctionnalités au langage et modifié certaines des fonctionnalités existantes. Pour conserver l'ancien code, la plupart des modifications sont désactivées par défaut.
Vous devez les activer explicitement avec une directive spéciale : "**use strict**".

"use strict"
La directive ressemble à une chaîne de caractères : **"use strict"** ou **'use strict'.**
Lorsqu'il se trouve en haut du script, l'ensemble du script fonctionne de manière "*moderne*".
Par exemple

```
"use strict";
// ce code fonctionne de manière
moderne
...
```

Le mode moderne, "use strict"

Assurez-vous que “use strict” est tout en haut

Assurez-vous que "use strict" est en haut du script, sinon le mode strict peut ne pas être activé.
Seuls les commentaires peuvent apparaître avant "use strict"
Il n'y a pas de mode strict ici :

```
alert("some code");
// "use strict" ci-dessous est ignoré, il doit être en haut

"use strict";
// le mode strict n'est pas activé
```

Il n'y a aucun moyen d'annuler use strict

Il n'y a pas de directive "no use strict" ou similaire, qui réactiverait l'ancien comportement.

Une fois que nous entrons dans le mode strict, il n'y a plus de retour possible.

Le mode moderne, "use strict"

Console du Navigateur
A l'avenir, lorsque vous utiliserez une console de navigation pour tester des fonctionnalités, veuillez noter *qu'elle n'utilise pas use strict par défaut.*

Parfois, lorsque **use strict** fait une différence, vous obtiendrez des résultats incorrects.

Alors, comment utiliser use strict dans la console ?

D'abord, vous pouvez essayer d'appuyer sur Shift+Enter pour saisir plusieurs lignes et mettre use strict en haut comme cela :

```
"use strict"; <Shift+Enter pour une nouvelle ligne>
//  ...votre code
<Appuyer sur la touche Entrée pour lancer la commande >
```

Le mode moderne, "use strict"

Cela fonctionne dans la plupart des navigateurs, de façon certaine avec Firefox et Chrome.

Si ce n'est pas le cas, comme par exemple dans un ancien navigateur, le moyen le plus fiable d'assurer use strict serait d'encapsuler le code dans la console comme ceci :

```
(function() {
  "use strict";
  // ... votre code ici ...
})()
```

Pourquoi activer "use strict" ?

La question peut sembler évidente, mais ce n'est pas le cas.
On pourrait recommander de démarrer les scripts avec "use strict" ... Mais vous savez ce qui est cool ?
Le JavaScript moderne prend en charge les "classes" et les "modules" – des structures de langage avancées (nous y arriverons sûrement), qui activent automatiquement use strict.

Le mode moderne, "use strict"

Nous n'avons donc pas besoin d'ajouter la directive "use strict" si nous les utilisons.

Donc, pour l'instant "use strict"; est un invité bienvenu en haut de vos scripts.

Plus tard, lorsque votre code est entièrement dans des classes et des modules, vous pouvez l'omettre.

A partir de maintenant, nous devons connaître use strict en général.

Dans les cours suivants, au fur et à mesure que nous apprendrons les fonctionnalités du langage, nous verrons les différences entre les modes strict et les anciens modes.

JS

Les fondamentaux du javascript

Les variables

Les variables

La plupart du temps, une application JavaScript doit utiliser des informations. Voici 2 exemples :

- Une boutique en ligne – les informations peuvent inclure des articles vendus et un panier d'achat.
- Une application de chat – les informations peuvent inclure des utilisateurs, des messages et bien plus encore. Les variables sont utilisées pour stocker ces informations.

Une variable

Une variable est un "*stockage nommé*" pour les données. Nous pouvons utiliser des variables pour stocker n'importe quelles données.

Pour créer une variable en JavaScript, nous devons utiliser le mot-clé **let**.

L'instruction ci-contre crée (autrement dit: **déclare**) une variable avec le nom "*message*" :

```
let message;
```

Maintenant, nous pouvons y mettre des données en utilisant l'opérateur d'affectation **=** :

```
let message;
message = 'Hello'; // stocke la chaîne de caractères 'Hello' dans la variable nommée message
```

Les variables

La chaîne de caractères est maintenant enregistrée dans la zone de mémoire associée à la variable. Nous pouvons y accéder en utilisant le nom de la variable :

```
let message;
message = 'Hello'; // stocke la chaîne de caractères 'Hello' dans la variable nommée message
alert( message );    // affiche le contenu de la variable, “Hello”
```

Pour être concis, nous pouvons fusionner la déclaration et l'affectation de variables en une seule ligne :

```
let message = 'Hello'; // Définie la variable et affecte la valeur “Hello”
alert( message );    // affiche le contenu de la variable, “Hello”
```

Nous pouvons également déclarer plusieurs variables sur une seule ligne :

```
let user= 'John', age = 25, message = 'Hello';
```

Les variables

Cela peut sembler plus court, mais ce n'est pas recommandé. Pour une meilleure lisibilité, veuillez utiliser une seule ligne par variable.

La variante multiligne est un peu plus longue, mais plus facile à lire :

```
let user= 'John';
let age = 25;
let message = 'Hello';
```

Certaines personnes définissent également plusieurs variables dans ce style multiligne :

```
let user= 'John',
    age = 25,
    message = 'Hello';
```

... Ou même dans le style "virgule première" :

```
let user= 'John'
   , age = 25
   , message = 'Hello';
```

Les variables

Techniquement, toutes ces variantes font la même chose. C'est donc une question de goût personnel et d'esthétique.

var au lieu de let

*Dans les anciens scripts, vous pouvez également trouver un autre mot-clé : **var** au lieu de **let** :*

```
var message = 'Hello';
```

*Le mot-clé **var** est presque identique à let. Il déclare également une variable, mais d'une manière légèrement différente, à la mode "old school".*

*Il y a des différences subtiles entre **let** et **var**, mais elles n'ont pas encore d'importance pour nous. Nous les couvrirons en détails plus tard,*

Les variables

Une analogie avec la vie réelle

Nous pouvons facilement saisir le concept d'une "**variable**" si nous l'imaginons comme une "**boîte**" pour les données, avec un **autocollant** portant un nom unique.

Par exemple, la variable message peut être imaginée comme une y *boîte étiquetée "message"* avec la valeur "*Hello!*" à l'intérieur :

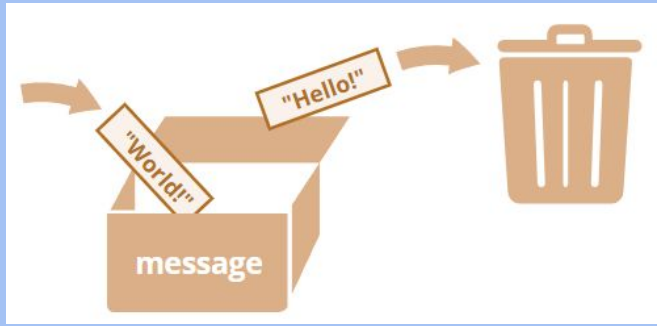
Nous pouvons mettre n'importe quelle valeur dans la boîte.

On peut aussi le changer autant de fois qu'on veut :

```
let message;
let message = 'Hello';
let message = 'World';
alert( message );
```

Les variables

Lorsque la valeur est modifiée, les anciennes données sont supprimées de la variable :



Nous pouvons également déclarer deux variables et copier des données de l'une à l'autre.

```
let hello = 'Hello World !';

let message;

// Copie le contenu de la variable, “Hello” dans la variable message
message = hello ;

// Les deux variables ont la même valeur
alert( message ); // 'Hello World !'
alert( hello );   // 'Hello World !'
```

Les variables

Déclarer deux fois déclenche une erreur

Une variable ne doit être déclarée qu'une seule fois.
Une déclaration répétée de la même variable est une erreur :

```
let message = 'A';
// répéter 'let' conduit à une erreur
let message = 'B'; // SyntaxError: 'message' has already been declared
```

Nom de variable

Il existe deux limitations pour un nom de variable en JavaScript :

- *Le nom ne doit contenir que des lettres, des chiffres, des symboles $ et _.*
- *Le premier caractère ne doit pas être un chiffre.*

Noms valides, par exemple :

```
let userName;
let test123
```

Les variables

Lorsque le nom contient plusieurs mots, l'écriture *camelCase* est couramment utilisé.
C'est-à-dire que les mots se succèdent, chaque mot à l'exception du premier commence par une majuscule : ***monTresLongNom***.

Ce qui est intéressant – le signe dollar '$' et l'underscore '_' peuvent également être utilisé dans les noms.
Ce sont des symboles réguliers, tout comme les lettres, sans aucune signification particulière.

Ces noms sont valides :

```
let $ = 1;
let _ = 2;

alert( $ + _ ); // 3
```

Exemples de noms de variables **incorrects** :

```
let 1a = // ne peut pas commencer avec un chiffre
let nom-no; // un trait d'union '-' n'est pas autorisé dans le nom
```

Les variables

La casse est importante

*Des variables nommées **apple** et **APPLE** sont deux variables différentes.*

Les lettres non latines sont autorisées mais non recommandées

Il est possible d'utiliser n'importe quelle langue, y compris les lettres cyrilliques, les logogrammes chinois, etc., comme ceci :

```
let имя = '...';
let 我 = '...';
```

Techniquement, il n'y a pas d'erreur ici, ces noms sont autorisés, mais il existe une convention internationale d'utiliser l'anglais dans les noms de variables. Même si nous écrivons un petit script, sa vie peut être longue. Les personnes d'autres pays peuvent avoir besoin de les lire quelque temps.

Noms réservés

Il existe une liste de mots réservés, qui ne peuvent pas être utilisés comme noms de variables, car ils sont utilisés par le langage lui-même.

Les variables

*Par exemple, les mots **let**, **class**, **return**, **function** sont réservés.*

Le code ci-contre donne une erreur de syntaxe :

```
let let = 5;
let return = 5;
```

Les Constantes

Pour déclarer une constante (non changeante), on peut utiliser **const** plutôt que let :

```
const myBirthday = '18.04.1982';
```

Les variables déclarées à l'aide de **const** sont appelées "constantes". Elles ne peuvent pas être réassignées. Une tentative de le faire provoquerait une erreur :

```
const myBirthday = '18.04.1982';
myBirthday = '10.01.1968';
```

Lorsqu'un programmeur est certain que la variable ne doit jamais changer, il peut utiliser const pour le garantir et également le montrer clairement à tout le monde.

Les variables

Les constantes en majuscules

Il existe une pratique répandue d'utiliser des constantes comme alias pour des valeurs difficiles à mémoriser, qui sont connues avant leur exécution.
Ces constantes sont nommées en utilisant des majuscules et des underscores.
Par exemple, créons des constantes pour les couleurs en hexadécimal :

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";
// ... quand il faut choisir une couleur
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Bénéfices:

- COLOR_ORANGE est beaucoup plus facile à retenir que "#FF7F00".
- Il est beaucoup plus facile de mal saisir "#FF7F00" que COLOR_ORANGE.
- En lisant le code, COLOR_ORANGE est beaucoup plus significatif que #FF7F00.

Quand devrions-nous utiliser les majuscules pour une constante et quand devrions-nous les nommer normalement ? Soyons clairs.

Les variables

Être une "constante" signifie simplement que la valeur ne change jamais. Mais il existe des constantes connues avant l'exécution (comme une valeur hexadécimale pour le rouge), et il y a celles qui sont calculées en temps réel, pendant l'exécution, mais ne changent pas après l'affectation.
Par exemple :

```
const pageLoadTime = /* temps pris par une page Web pour charger */;
```

La valeur de pageLoadTime n'est pas connue avant le chargement de la page, elle est donc nommée normalement. Mais cela reste une constante, car elle ne change pas après l'affectation.
En d'autres termes, les constantes nommées en majuscules ne sont utilisées que comme alias pour les valeurs "codées en dur".

Nommez les choses correctement
En parlant de variables, il y a une autre chose extrêmement importante.
Un nom de variable doit avoir une signification claire et évidente, décrivant les données qu'elle stocke.
Le nommage de variables est l'une des compétences les plus importantes et les plus complexes de la programmation. Un rapide coup d'œil sur les noms de variables peut révéler quel code est écrit par un débutant et par un développeur expérimenté.

Les variables

Dans un projet réel, la majeure partie du temps est consacrée à la modification et à l'extension de la base de code existant, plutôt que d'écrire quelque chose de complètement séparé de zéro.
Et lorsque nous revenons au code après un certain temps, il est beaucoup plus facile de trouver des informations bien étiquetées. Ou, en d'autres termes, lorsque les variables ont de bons noms.
Veuillez prendre le temps de réfléchir à un nom pertinent pour une variable avant de la déclarer. Cela vous facilitera énormément la vie.

Voici quelques règles à suivre :

- Utilisez des noms lisibles par des humains comme userName ou shoppingCart.
- Restez à l'écart des abréviations ou des noms courts tels que a, b, c, à moins que vous ne sachiez vraiment ce que vous faites.
- Faire en sorte que le nom soit le plus descriptif et concis possible. Des exemples de noms incorrects sont data et value. Un tel nom ne dit rien. C'est tout à fait acceptable de les utiliser si le contexte dans lequel les données ou les valeurs sont impliquées est particulièrement évident.
- S'accorder avec son équipe (et soi-même) sur les termes utilisés. Si un visiteur du site est appelé un "utilisateur", nous devrions nommer les variables connexes comme currentUser ou newUser, mais non currentVisitor ou encore newManInTown.

Les variables

Résumé

Nous pouvons déclarer des variables pour stocker des données. Cela peut être fait en utilisant var ou let ou const.

- **let** – est une déclaration de variable moderne.
- **var** – est une déclaration de variable old-school. Normalement, nous ne l'utilisons pas du tout, mais nous couvrirons les différences subtiles par rapport à let dans le chapitre L'ancien "var", juste au cas où vous en auriez besoin.
- **const** – est équivalent à let, mais la valeur de la variable ne peut pas être modifiée.

Les variables doivent être nommées d'une manière qui nous permet de comprendre facilement ce qui est à l'intérieur.

JS

Exercices

Les variables

Créer un fichier vars.html avec un élément script placé à la fin, avant "/body".
Ecriver le code nécessaire pour répondre aux énoncés suivants :

Travailler avec des variables

- Déclarez deux variables : **admin** et **name**.
- Assignez la valeur "*Jean*" à **name**.
- Copiez la valeur de name à **admin**.
- Afficher la valeur de **admin** en utilisant la fonction **alert** cela (devrait afficher "Jean").

Assigner le bon nom

- Créez la variable qui continedra le nom de notre planète. Comment nommeriez-vous une telle variable ?
- Créez la variable pour stocker le nom du visiteur actuel, votre nom par exemple.
  Comment nommeriez-vous cette variable ?

Les variables

Correction : Travailler avec des variables

```
let admin, name // on peut déclarer deux variables à la fois

name= "Jean";

admin = name;

alert(admin); "Jean";
```

Correction : Assigner le bon nom

```
let notrePlanete = "Terre";
```

Notez que nous pourrions utiliser un nom planète plus court, mais la planète à laquelle il fait référence pourrait ne pas être évidente. C'est bien d'être plus verbeux. Au moins jusqu'à ce que la variable "*nesoitPasTropLongue*".

```
let currentUserName = "Jean";
```

Encore une fois, nous pourrions réduire cela à **userName** si nous savons avec certitude que c'est bien l'utilisateur actuel.
Les éditeurs modernes et la saisie semi-automatique facilitent l'écriture de noms de variables longs. Ne vous en privez pas. Un nom avec 3 mots est bien.

JS

Les fondamentaux du javascript

Les types de données

Les types de données

Une valeur en JavaScript est toujours d'un certain type.
Par exemple, une chaîne de caractères ou un nombre.
Il existe huit types de données de base en JavaScript.

Nous pouvons mettre n'importe quel type dans une variable.
Par exemple, une variable peut à un moment être une chaîne de caractères puis stocker un nombre.

```
// pas d'erreur
let message = "hello";
message = 123456;
```

Les langages de programmation qui permettent de telles choses sont appelés "typés dynamiquement", ce qui signifie qu'il existe des types de données, mais que les variables ne sont liées à aucun d'entre eux.

Les types de données

Number

```
let n = 123;
n = 12.345;
```

Le type number sert à la fois à des nombres entiers et à des nombres à virgule flottante.
Il existe de nombreuses opérations pour les nombres, par ex. multiplication *, division /, addition +, soustraction - et ainsi de suite.

Outre les nombres réguliers, il existe des "valeurs numériques spéciales" qui appartiennent également à ce type: **Infinity**, **-Infinity** et **NaN**.

- **Infinity** représente l'Infini ∞ mathématique. C'est une valeur spéciale qui est plus grande que n'importe quel nombre.

  Nous pouvons l'obtenir à la suite d'une division par zéro :

```
alert( 1 / 0 ); // Infinity
```

Les types de données

Ou mentionnez-le simplement dans le code directement :

```
alert( Infinity ); // Infinity
```

- **NaN** (Not a Number), représente une erreur de calcul. C'est le résultat d'une opération mathématique incorrecte ou non définie, par exemple :

```
alert( "pas un nombre" / 0 ); // NaN, une telle division est erronée
```

Toute opération sur **NaN** donne **NaN** :

```
alert( NaN + 1 ); // NaN
alert( NaN * 3 ); // NaN
alert( "pas un nombre" / 2 - 1 ); // NaN
```

Donc, s'il y a NaN quelque part dans une expression mathématique, il se propage à l'ensemble du résultat (il n'y a qu'une seule exception : *NaN ** 0 vaut 1*).

Les types de données

BigInt

En JavaScript, le type "number" ne peut pas représenter des valeurs entières supérieures à $(2^{53}-1)$ (c'est 9007199254740991), ou moins que $-(2^{53}-1)$ pour les chiffres négatifs.
C'est une limitation technique causée par leur représentation interne.

Pour être vraiment précis, le type "number" peut stocker des entiers plus grands (jusqu'à 1.7976931348623157 * 10308), mais en dehors de la plage d'entiers sûrs $\pm(2^{53}-1)$ il y aura une erreur de précision, car tous les chiffres ne rentrent pas dans le stockage 64 bits fixe.
Ainsi, une valeur "approximative" peut être stockée.

Par exemple, ces deux nombres (juste au-dessus de la plage de sécurité) sont identiques :

```
console.log(9007199254740991 + 1); // 9007199254740992
console.log(9007199254740991 + 2); // 9007199254740992
```

Les types de données

Ainsi, tous les entiers impairs supérieurs à $(2^{53}-1)$ ne peuvent pas du tout être stockés dans le type "number".

Dans la plupart des cas, la plage $\pm(2^{53}-1)$ est tout à fait suffisante, mais parfois nous avons besoin de toute la plage de très grands nombres entiers, par ex. pour la cryptographie ou les horodatages de précision à la microseconde.

BigInt a récemment été ajouté au langage pour représenter des entiers de longueur arbitraire.

Une valeur BigInt est créé en ajoutant n à la fin d'un entier :

```
// le "n" à la fin signifie que c'est un BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Les types de données

String

Une chaîne de caractères en JavaScript doit être entre guillemets.

```
let str = "Hello";
let str2 = 'Single quotes are ok too';
let phrase = `can embed another ${str}`;
```

En JavaScript, il existe 3 types de guillemets.

- **Double quotes**: "Hello".
- **Single quotes**: 'Hello'.
- **Backticks**: `Hello`.

Les guillemets simples et doubles sont des guillemets "simples".
Il n'y a pratiquement pas de différence entre eux en JavaScript.
Les *backticks* sont des guillemets "à fonctionnalité étendue". Ils nous permettent d'intégrer des variables et des expressions dans une chaîne en les encapsulant dans *${...}*, par exemple :

Les types de données

```
let name = "Jean";
alert( `Bonjour ${name}` ); // Bonjour Jean
alert( `Le résultat est ${1+2}` ); // Le résultat est 3
```

L'expression à l'intérieur de ${...} est évaluée et le résultat devient une partie de la chaîne. On peut y mettre n'importe quoi : une variable comme name ou une expression arithmétique comme 1 + 2 ou quelque chose de plus complexe.

Veuillez noter que cela ne peut être fait que dans les backticks. Les autres guillemets ne permettent pas une telle intégration !

```
alert( "Le résultat est ${1+2}" ); // le résultat est ${1 + 2}
(les doubles quotes ne font rien)
```

Les types de données

Boolean

Le type booléen n'a que deux valeurs: **true** et **false**.

Ce type est couramment utilisé pour stocker des valeurs **oui / non**: **true** signifie "**oui**, **correct**" et **false** signifie "**non**, **incorrect**".

Par exemple :

```
let nameFieldChecked = true; // oui, le champ de nom est coché
let ageFieldChecked = true; // non, le champ d'âge n'est pas coché
```

Les valeurs booléennes résultent également de comparaisons :

```
let isGreater= 4 > 1;
alert( isGreater ); // true (le résultat de la comparaison est "oui")
```

Les types de données

La valeur “null”

La valeur spéciale **null** n’appartient à aucun type de ceux décrits ci-dessus.
Il forme un type bien distinct qui ne contient que la valeur null :

```
let age = null;
```

En JavaScript, null n’est pas une “référence à un objet non existant” ou un “pointeur nul” comme dans d’autres langages. C’est juste une valeur spéciale qui a le sens de “rien”, “vide” ou “valeur inconnue”.
Le code ci-dessus indique que l’age est inconnu.

La valeur “undefined”

La valeur spéciale **undefined** se distingue des autres. C’est un type à part entière, comme null.
La signification de **La valeur “null”**est “la valeur n’est pas attribuée”.
Si une variable est déclarée mais non affectée, alors sa valeur est exactement undefined :

```
let age;
alert( age); // "undefined"
```

Techniquement, il est possible d’affecter explicitement **undefined** à une variable :

Les types de données

La valeur “null”

La valeur spéciale **null** n’appartient à aucun type de ceux décrits ci-dessus.
Il forme un type bien distinct qui ne contient que la valeur null :

```
let age = null;
```

En JavaScript, null n’est pas une “référence à un objet non existant” ou un “pointeur nul” comme dans d’autres langages.
C’est juste une valeur spéciale qui a le sens de “rien”, “vide” ou “valeur inconnue”.
Le code ci-dessus indique que l’age est inconnu.

La valeur “undefined”

La valeur spéciale **undefined** se distingue des autres. C’est un type à part entière, comme null.
La signification de **La valeur “null”**est “la valeur n’est pas attribuée”.
Si une variable est déclarée mais non affectée, alors sa valeur est exactement **undefined** :

```
let age;
alert( age ); // "undefined"
```

Techniquement, il est possible d’affecter explicitement **undefined** à une variable, mais il n’est pas recommandé de faire cela. Normalement, nous utilisons null pour assigner une valeur “vide” ou “inconnue” à une variable, tandis que undefined est réservé comme valeur initiale par défaut pour les éléments non attribués.

Les types de données

Objects

Le type **object** est spécial.
Tous les autres types sont appelés "**primitifs**", car leurs valeurs ne peuvent contenir qu'une seule chose (que ce soit une chaîne de caractères, un nombre ou autre).
A contrario, les **"objets"** servent à stocker des collections de données et des entités plus complexes.

Étant aussi important, les objets méritent un traitement spécial. Nous les traiterons plus tard dans un *chapitre Objets*, après en savoir plus sur les primitifs.

L'opérateur typeof

L'opérateur typeof renvoie le type de l'opérande. Il est utile lorsqu'on souhaite traiter différemment les valeurs de différents types ou de faire une vérification rapide.

Les types de données

L'appel **typeof *x*** renvoie une chaîne de caractères avec le nom du type :
Les trois dernières lignes peuvent nécessiter des explications supplémentaires :

1. **Math** est un objet interne au langage qui fournit des opérations mathématiques. Nous allons l'apprendre dans le chapitre Nombres. Ici, il sert uniquement comme exemple d'un objet.
2. Le résultat de **typeof null** est "**object**". C'est une erreur officiellement reconnue dans typeof, datant des premiers jours de JavaScript et conservée pour compatibilité. Bien sûr, null n'est pas un objet. C'est une valeur spéciale avec un type distinct qui lui est propre. Le comportement de typeof est incorrect ici.
3. Le résultat de **typeof alert** est "**function**", car alert est une fonction. Nous étudierons les fonctions dans les chapitres suivants, et nous verrons qu'il n'y a pas de type "fonction" en JavaScript. Les fonctions appartiennent au type object mais typeof les traite différemment en retournant "function". Cela vient également des débuts de JavaScript. Techniquement ce n'est pas tout à fait correct, mais très pratique à l'usage.

```
typeof undefined // "undefined"
typeof 0         // "number"
typeof 10n       // "bigint"
typeof true      // "boolean"
typeof "foo"     // "string"
typeof Math      // "object" (1)
typeof null      // "object" (2)
typeof alert     // "function" (2)
```

La syntaxe typeof(x)

Vous pouvez également rencontrer une autre syntaxe : typeof(x).
C'est la même chose que typeof x.

Les types de données

Résumé

Il existe 8 types de données de base en JavaScript.

Sept types de données primitifs :

- **number** pour les nombres de toute nature : entier ou virgule flottante, les nombres entiers sont limités à $\pm(2^{53}-1)$.
- **bigint** pour des nombres entiers de longueur arbitraire.
- **string** pour les chaînes de caractères. Une chaîne de caractères peut avoir zéro ou plusieurs caractères, il n'y a pas de type à caractère unique distinct.
- boolean pour true/false (vrai/faux).
- **null** pour les valeurs inconnues – un type autonome qui a une seule valeur null.
- **undefined** pour les valeurs non attribuées – un type autonome avec une valeur unique undefined.

Et un type de données non primitif :

- **object** pour des structures de données plus complexes.

L'opérateur **typeof** nous permet de voir quel type est stocké dans la variable.

- Généralement utilisé sous cette forme typeof x, mais typeof(x) est également possible.
- Renvoie une chaîne de caractères avec le nom du type, comme "string".
- Pour null il renvoie "object" – C'est une erreur dans le langage, ce n'est pas un objet en fait.

JS

Les fondamentaux du javascript

Interaction: alert, prompt, confirm

Interaction: alert, prompt, confirm

Comme nous allons utiliser le navigateur comme environnement de démonstration, voyons quelques fonctions pour interagir avec l'utilisateur : alert, prompt et confirm.

```
alert( "Hello" );
```

alert

Celui-ci, nous l'avons déjà vu. Il affiche un message et attend que l'utilisateur appuie sur "OK".

La mini-fenêtre avec le message s'appelle une fenêtre modale. Le mot "modal" signifie que le visiteur ne peut pas interagir avec le reste de la page, appuyer sur d'autres boutons, etc., tant qu'il n'a pas traité la fenêtre. Dans ce cas – jusqu'à ce qu'ils appuient sur "OK".

```
result = prompt( title, [default]);
```

prompt

La fonction prompt accepte deux arguments :

Elle affiche une fenêtre modale avec un message texte, un champ de saisie pour le visiteur et les boutons OK/Annuler.

- **title** : Le texte à afficher au visiteur.
- **default** : Un deuxième paramètre facultatif, la valeur initiale du champ d'entrée.

Les crochets dans la syntaxe [...]

*Les crochets autour de **default** dans la syntaxe ci-dessus indiquent que le paramètre est facultatif, non requis.*

Interaction: alert, prompt, confirm

Le visiteur peut taper quelque chose dans le champ de saisie d'invite et appuyer sur OK. Ensuite, nous obtenons ce texte dans le result. Ou ils peuvent annuler l'entrée en appuyant sur Annuler ou en appuyant sur la touche Esc, puis nous obtenons null comme resultat.
L'appel à prompt renvoie le texte du champ de saisie ou null si l'entrée a été annulée.

```
let age = prompt( "Quel age as tu ?", 100);
alert( `Vous avez ${age} ans !` ); // Vous avez 100 ans !
```

confirm
La syntaxe

```
result = confirm( question );
```

La fonction confirm affiche une fenêtre modale avec une question et deux boutons : OK et Annuler.
Le résultat est true si vous appuyez sur OK et false dans le cas contraire.

```
let estChef = confirm( "Es tu le chef ?" );
alert( estChef ); // true si OK est pressé
```

JS

Exercices

Interaction: alert, prompt, confirm

Une page

Créez une page Web qui demande votre nom et l'affiche.

Les variables

Correction : Une page

```
let name = prompt( "Quel es ton nom ?" );
alert( name );
```

La page complète

```
<!DOCTYPE html>
<html>
<body>
  <script>
    'use strict';
    let name = prompt( "Quel es ton nom ?" );
    alert( name );
  </script>
</body>
</html>
```

JS

Les fondamentaux du javascript

Les conversions de types

Les conversions de types

La plupart du temps, les opérateurs et les fonctions convertissent automatiquement les valeurs qui leur sont attribuées dans le bon type.
Par exemple, **alert** convertit automatiquement toute valeur en chaîne de caractères pour l'afficher.
Les **opérations mathématiques** convertissent les valeurs en nombres.
Il y a aussi des cas où nous devons convertir explicitement une valeur pour corriger les choses.

Conversion de chaîne de caractère - string -
La conversion String se produit lorsque nous avons besoin de la forme chaîne de caractères d'une valeur.
Par exemple, **alert(value)** le fait pour afficher la valeur.
Nous pouvons également utiliser un appel de fonction **String**(*value*) pour ça :

```
let value = true;
alert( typeof value );   // boolean
value = String( value ); // maintenant la valeur est une chaîne de caractères "true"
alert( typeof value );   // string
```

La conversion String est assez évidente. Un **false** devient "**false**", **null** devient "**null**" etc.

Les conversions de types

Conversion de nombre - number -

La conversion numérique dans les fonctions et expressions mathématiques s'effectue automatiquement.
Par exemple, lorsque la division / est appliqué à des "non nombre" :

```
let value = true;
alert( "6"/"2" );   // 3, les chaînes de caractères sont converties en nombres
```

Nous pouvons utiliser une fonction **Number**(*value*) pour convertir explicitement une valeur :

```
let str = "123";
alert( typeof str );     // string
let num = Number( str ); // devient un nombre 123
alert( typeof num );    // number
```

Une conversion explicite est généralement requise lorsque nous lisons une valeur à partir d'une source basée sur des chaînes de caractères, par exemple un champ texte, mais qu'un nombre doit être entré.
Si la chaîne de caractères n'est pas un nombre valide, le résultat de cette conversion est **NaN**, par exemple :

Les conversions de types

Conversion de nombre - number -

La conversion numérique dans les fonctions et expressions mathématiques s'effectue automatiquement.
Par exemple, lorsque la division / est appliqué à des "non nombre" :

```
let age = Number( "une chaîne de caractères au lieu d'un nombre" );
alert( age );   // NaN, la conversion a échoué
```

Règles de conversion numériques :

Valeur	**Devient** ...
undefined	*NaN*
null	*0*
true et false	*1 et 0*
string	*Les espaces blancs du début et de la fin sont supprimés. Ensuite, si la chaîne restante est vide, le résultat est 0. Sinon, le nombre est «lu» dans la chaîne. Une erreur donne NaN.*

Exemples

```
alert( Number( "   123   ") ); // 123
alert( Number( "123z") ); // NaN
alert( Number( true ) );  // 1
alert( Number( true ) );  // 0
```

Les conversions de types

Conversion de nombre - number -

La conversion numérique dans les fonctions et expressions mathématiques s'effectue automatiquement.
Par exemple, lorsque la division / est appliqué à des "non nombre" :

```
let value = true;
alert( "6"/"2" );   // 3, les chaînes de caractères sont converties en nombres
```

Nous pouvons utiliser une fonction **Number**(*value*) pour convertir explicitement une valeur :

```
let str = "123";
alert( typeof str );     // string
let num = Number( str ); // devient un nombre 123
alert( typeof num );     // number
```

Une conversion explicite est généralement requise lorsque nous lisons une valeur à partir d'une source basée sur des chaînes de caractères, par exemple un champ texte, mais qu'un nombre doit être entré.
Si la chaîne de caractères n'est pas un nombre valide, le résultat de cette conversion est **NaN**, par exemple :

JS

Les fondamentaux du javascript

Opérateurs de base, mathématiques

Opérateurs de base, mathématiques

De nombreux opérateurs nous sont connus de l'école. Ce sont les additions +, multiplications *, soustractions - et ainsi de suite.
Dans ce chapitre, nous nous concentrons sur les aspects qui ne sont pas couverts par l'arithmétique scolaire.

Termes: "unaire", "binaire", "opérande"
Avant de continuer, saisissons la terminologie commune.

- Un opérande est ce à quoi les opérateurs sont appliqués. Par exemple, dans la multiplication 5 * 2, il y a deux opérandes : l'opérande gauche est 5 et l'opérande droit est 2. Parfois, certains disent "arguments" au lieu de "opérandes".
- Un opérateur est unaire s'il a un seul opérande. Par exemple, la négation unaire - inverse le signe du nombre :

```
let x = 1;
x = -x;
alert( x ); // -1, le moins unaire a été appliqué
```

- Un opérateur est binaire s'il a deux opérandes. La même négation existe également dans la forme binaire :

Opérateurs de base, mathématiques

- Un opérateur est binaire s'il a deux opérandes. La même négation existe également dans la forme binaire :

```
let x = 1,  y = 3;
x = -x;
alert( y - x ); // 2, le moins binaire soustrait des valeurs
```

D'un point de vue formel, dans les exemples ci-dessus, nous avons deux opérateurs différents qui partagent le même symbole : l'opérateur de négation, un opérateur unaire qui inverse le signe, et l'opérateur de soustraction, un opérateur binaire qui soustrait un nombre d'un autre.

Opérations mathématiques

Les opérations mathématiques suivantes sont supportées :

- Addition +,
- Soustraction -,
- Multiplication *,
- Division /,
- Reste %,
- Exponentiation **.

Les quatre premières sont assez simples, tandis que % et ** nécessitent quelques explications.

Opérateurs de base, mathématiques

Reste % (Modulo)

L'opérateur reste %, malgré son apparence, n'est pas lié aux pourcentages.
Le résultat de a % b est le reste de la division entière de a par b. Par exemple :

```
alert( 5 % 2 ); // 1, le reste de 5 divisé par 2
alert( 8 % 3 ); // 2, le reste de 8 divisé par 3
alert( 8 % 4 ); // 0, le reste de 8 divisé par 4
```

Exponentiation **

L'opérateur d'exponentiation a ** b multiplie a par lui-même b fois.
En mathématiques à l'école, nous écrivons cela a^b. Par exemple :

```
alert( 2 ** 2 ); // 2² = 4
alert( 2 ** 3 ); // 2³ = 8
alert( 2 ** 4 ); // 2⁴ = 16
```

Tout comme en mathématiques, l'opérateur d'exponentiation est également défini pour les nombres non entiers.
Par exemple, une racine carrée est une exponentiation de ½ :

```
alert( 2 ** ( 1 / 2 ); // 1.41...
(la puissance de ½ équivaut à une racine carrée)
alert( 2 ** ( 1 / 3 ); // 1.25...
(la puissance de 1/3 équivaut à une racine cubique)
```

Opérateurs de base, mathématiques

Concaténation de chaînes de caractères, binaire +

Découvrons les fonctionnalités des opérateurs JavaScript qui vont au-delà de l'arithmétique scolaire. Habituellement, l'opérateur + additionne des chiffres. Mais si le binaire + est appliqué aux chaînes de caractères, il les fusionne (**concatène**) :

```
let s = "my" + "string";
alert( s ); // mystring
```

Notez que si l'un des opérandes est une chaîne de caractères, l'autre est automatiquement converti en chaîne de caractères. Par exemple :

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Peu importe que le premier opérande soit une chaîne de caractères ou le second. La règle est simple : si l'un des opérandes est une chaîne de caractères, convertissez l'autre également en une chaîne de caractères.

Cependant, notez que les opérations se déroulent de gauche à droite. S'il y a deux nombres suivis d'une chaîne, les nombres seront ajoutés avant d'être convertis en chaîne :

```
alert( 2 + 2 + '1' ); "41" et non "221"
```

Opérateurs de base, mathématiques

Ici, les opérateurs travaillent les uns après les autres. Le premier + additionne deux nombres, donc il renvoie 4, puis le + suivant ajoute la chaîne de caractères 1, donc c'est comme 4 + '1' = 41.

```
alert( '1' + 2 + 2 ); "14" et non "122"
```

Ici, le premier opérande est une chaîne de caractères, le compilateur traite également les deux autres opérandes comme des chaînes de caractères. Le 2 est concaténé à '1', donc c'est comme '1'+ 2 = "12" et "12" + 2 = "122".
Le binaire + est le seul opérateur qui prend en charge les chaînes de caractères de cette manière.
D'autres opérateurs arithmétiques ne fonctionnent qu'avec des nombres et convertissent toujours leurs opérandes en nombres.
Voici l'exemple pour la soustraction et la division :

```
alert( 6 - '2' ); // 4, convertit '2' en nombre
alert( '6' / '2' ); // 3, convertit les deux opérandes en nombres
```

Opérateurs de base, mathématiques

Conversion numérique, unaire +

Le plus + existe sous deux formes. La forme binaire que nous avons utilisée ci-dessus et la forme unaire. L'unaire plus ou, en d'autres termes, l'opérateur plus + appliqué à une seule valeur, ne fait rien avec les nombres, mais si l'opérande n'est pas un nombre, alors il est converti en nombre.

Par exemple :

```
// Aucun effet sur les nombres
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Convertit les non-nombres
alert( +true ); // 1
alert( +"" );   // 0
```

Opérateurs de base, mathématiques

En fait, il fait la même chose que **Number(...),** mais il est plus court.
La nécessité de convertir des chaînes de caractères en nombres est très fréquente. Par exemple, si nous obtenons des valeurs à partir de champs de formulaire HTML, il s'agit généralement de chaînes de caractères.
Et si on veut les additionner ?
Le binaire plus les ajouterait comme des chaînes de caractères :

```
let apples = "2";
let oranges = "3";
alert( apples + oranges ); // "23", le binaire plus concatène les chaînes de caractères
```

Si nous voulons les traiter comme des nombres, nous devons d'abord les convertir et ensuite seulement nous pouvons les additionner :

```
let apples = "2";
let oranges = "3";
// les deux valeurs converties en nombres avant le binaire plus
alert( +apples + +oranges ); // 5
// c'est équivalent à cette variante plus longue
// alert( Number(apples) + Number(oranges) ); // 5
```

Opérateurs de base, mathématiques

Du point de vue du mathématicien, l'abondance des + peut sembler étrange. Mais du point de vue du programmeur, il n'y a rien de spécial : les plus unaires sont appliqués en premier, ils convertissent les chaînes de caractères en nombres, puis le plus binaire les additionne.
Pourquoi les plus unaires sont-ils appliqués aux valeurs avant les binaires ? Comme nous allons le voir, c'est à cause de leur précédence supérieure.

Précédence des opérateurs
Si une expression à plusieurs opérateurs, l'ordre d'exécution est défini par leur priorité ou, en d'autres termes, il existe un ordre de priorité implicite entre les opérateurs.
De l'école, nous savons tous que la multiplication dans l'expression 1 + 2 * 2 devrait être calculée avant l'addition. C'est exactement cela la précédence. La multiplication est dite avoir une précédence supérieure à l'addition.
Les parenthèses outrepassent toute priorité, donc si nous ne sommes pas satisfaits de l'ordre par défaut, nous pouvons les utiliser, comme : (1 + 2) * 2.
Il y a beaucoup d'opérateurs en JavaScript. Chaque opérateur a un numéro correspondant à sa priorité de précédence. Celui qui est plus haut sur le tableau s'exécute en premier. Si la priorité est la même, l'ordre d'exécution est de gauche à droite.

Opérateurs de base, mathématiques

Un extrait du tableau de précédence (vous n'avez pas besoin de vous en souvenir, mais notez que les opérateurs unaires ont une priorité plus élevée que les binaires correspondants)

Comme on peut le voir, le "plus unaire" a une priorité de 14, ce qui est supérieur à 11 pour "l'addition" (plus binaire).

C'est pourquoi, dans l'expression "+apples + +oranges", les plus unaires fonctionnent avant l'addition.

Précédence	Nom	Symbole
...	...	...
14	plus	unaire +
14	négation	unaire -
13	exponentiation	**
12	multiplication	*
12	division	/
11	addition	+
11	soustraction	-
...	...	...
2	affectation	=
...	...	...

Opérateurs de base, mathématiques

Affectation

Notons qu'une affectation = est aussi un opérateur. Il est répertorié dans le tableau des précédences avec la très faible priorité de 2.

C'est pourquoi lorsque nous assignons une variable, comme x = 2 * 2 + 1, les calculs sont effectués en premier, puis le = est évalué, stockant le résultat dans x.

```
let x = 2 * 2 + 1;
alert( x ); // 5
```

Assignment = retourne une valeur

Le fait que = soit un opérateur, pas une construction de langage "magique" a une implication intéressante.
Tous les opérateurs en JavaScript renvoient une valeur. C'est évident pour + et -, mais aussi vrai pour =.
L'appel x = valeur écrit la valeur dans x puis la renvoie.
Voici un exemple qui utilise une affectation dans le cadre d'une expression plus complexe :

Opérateurs de base, mathématiques

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

Dans l'exemple ci-dessus, le résultat de l'expression (a = b + 1) est la valeur qui a été affectée à a (c'est-à-dire 3). Il est ensuite utilisé pour d'autres évaluations.

Drôle de code, n'est-ce pas? Nous devons comprendre comment cela fonctionne, car parfois nous le voyons dans les bibliothèques JavaScript.

Cependant, n'écrivez pas le code comme ça. De telles astuces ne rendent certainement pas le code plus clair ou lisible.

Opérateurs de base, mathématiques

Affectations chaînées

Une autre caractéristique intéressante est la possibilité de chaîner des affectations :

```
let a , b , c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Les affectations en chaîne sont évaluées de droite à gauche.
D'abord, l'expression la plus à droite 2 + 2 est évaluée puis assignée aux variables de gauche : c, b et a. À la fin, toutes les variables partagent une seule valeur.

Encore une fois, pour des raisons de lisibilité, il est préférable de diviser ce code en quelques lignes :

C'est plus facile à lire, en particulier lors de la numérisation rapide du code.

```
c = 2 + 2;
b = c
a = b
```

Opérateurs de base, mathématiques

Modification sur place
Nous avons souvent besoin d'appliquer un opérateur à une variable et d'y stocker le nouveau résultat.

```
let n = 2;
n = n + 5;
n = n * 2;
```

Cette notation peut être raccourcie en utilisant les opérateurs **+=** et ***=** :

```
let n = 2;
n += 5; // 7
n *= 2; // 14
alert( n ); // 14
```

Il existe des opérateurs de "modification et assignation" courts pour tous les opérateurs arithmétiques et binaires : /=, -= etc.

Ces opérateurs ont la même précédence qu'une affectation normale. Ils s'exécutent donc après la plupart des autres calculs :

```
let n = 2;
n *= 3 + 5; // la partie de droite est évaluée en premier (identique à n *= 8)
alert( n ); // 16
```

Opérateurs de base, mathématiques

Incrémentation / décrémentation

L'augmentation ou la diminution d'un nombre par 1 compte parmi les opérations numériques les plus courantes.
Il y a donc des opérateurs spéciaux pour cela :

- Incrémentation **++** augmente une variable de 1 :
- Décrémentation -- diminue une variable de 1 :

```
let n = 2;
n++; // identique à n = n + 1
alert( n ); // 3
```

```
let n = 2;
n--; // identique à n = n - 1
alert( n ); // 1
```

Les opérateurs ++ et -- peuvent être placés à la fois après et avant la variable.

Lorsque l'opérateur va après la variable, cela s'appelle une "forme postfixe" : counter++.
La "forme préfixe" est celle où l'opérateur se place devant la variable : ++counter.
Ces deux opérateurs font la même chose : augmenter le counter de 1.

Y a-t-il une différence ? Oui, mais nous ne pouvons le voir que si nous utilisons la valeur renvoyée de ++/--.

Opérateurs de base, mathématiques

Soyons clairs. Comme nous le savons, tous les opérateurs renvoient une valeur. L'incrémentation / décrémentation n'est pas une exception ici. La forme préfixe renvoie la nouvelle valeur, tandis que la forme postfixe renvoie l'ancienne valeur (avant l'incrémentation / décrémentation).

Pour voir la différence, voici un exemple :

```
let counter = 1;
let a = ++counter;
alert( a ); // 2
```

Maintenant, utilisons la forme postfixe :
La forme postfixe counter++ incrémente également counter, mais renvoie l'ancienne valeur (avant l'incrémentation).
Donc, l'alert montre 1.

```
let counter = 1;
let a = counter++;
alert( a ); // 1
```

Opérateurs de base, mathématiques

Opérateurs binaires

Les opérateurs binaires traitent les arguments comme des nombres entiers de 32 bits et travaillent au niveau de leur représentation binaire.

Ces opérateurs ne sont pas spécifiques à JavaScript. Ils sont pris en charge dans la plupart des langages de programmation.

La liste des opérateurs :

- AND (&)
- OR (|)
- XOR (^)
- NOT (~)
- LEFT SHIFT (<<)
- RIGHT SHIFT (>>)
- ZERO-FILL RIGHT SHIFT (>>>)

Ces opérateurs sont très rarement utilisés, lorsque nous devons jouer avec des nombres au niveau le plus bas (bit à bit). Nous n'aurons pas besoin de ces opérateurs de si tôt, car le développement Web les utilise peu, mais dans certains domaines particuliers, comme la cryptographie, ils sont utiles. Vous pouvez lire le chapitre Opérateurs binaires sur MDN en cas de besoin.

JS

Exercices

Opérateurs de base, mathématiques

Les formes postfixes et préfixes

Quelles sont les valeurs finales de toutes les variables a, b, c et d après le code ci-dessous ?

```
let a = 1, b = 1;
let c = ++a; // ?
let d = b++; // ?
```

Opérateurs de base, mathématiques

La réponse est :

- a = 2
- b = 2
- c = 2
- d = 1

```
let a = 1, b = 1;

alert( ++a ); // 2, la forme préfixe renvoie la nouvelle valeur
alert( b++ ); // 1, la forme postfixe renvoie l'ancienne valeur

alert( a ); // 2, incrémenté une fois
alert( b ); // 2, incrémenté une fois
```

Opérateurs de base, mathématiques

Résultat d'affectation

Quelles sont les valeurs de a et x après le code ci-dessous ?

```
let a = 1;
let x = 1 + ( a *= 2 );
```

Opérateurs de base, mathématiques

Résultat d'affectation

La réponse est :

- **a = 4 (multiplié par 2)**
- **x = 5 (calculé comme 1 + 4)**

Opérateurs de base, mathématiques

Les conversions de type

Quels sont les résultats de ces expressions ?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
"  -9  " + 5
"  -9  " - 5
null + 1
undefined + 1
" \t \n" - 2
```

Opérateurs de base, mathématiques

La réponse

```
"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
"   -9  " + 5 = "   -9  5" // (3)
"   -9  " - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
" \t \n" - 2 = -2 // (7)
```

1. L'addition avec une chaîne de caractères "" + 1 converti 1 vers une chaîne de caractères : "" + 1 = "1", ensuite nous avons "1" + 0, la même règle est appliquée.
2. La soustraction - (comme la plupart des opérations mathématiques) ne fonctionne qu'avec des nombres, il convertit une chaîne de caractères vide "" vers 0.
3. L'addition avec un string ajoute le number 5 au string.
4. La soustraction est toujours convertie en nombres, donc elle fait de " -9 " un number -9 (en ignorant les espaces qui l'entourent).
5. null devient 0 après la conversion numérique.
6. undefined devient NaN après la conversion numérique.
7. Les caractères d'espacement sont coupés au début et à la fin de la chaîne de caractères lorsque celle-ci est convertie en nombre. Ici toute la chaîne se compose d'espaces, tels que \t, \n et d'un espace "normal" entre eux. Ainsi, de manière similaire à une chaîne de caractères vide, elle devient 0.

Opérateurs de base, mathématiques

Corrigez l'addition

Voici un code qui demande à l'utilisateur deux nombres et affiche leur somme.

```
let a = prompt("First number?", 1);
let b = prompt("Second number?", 2);

alert( a + b ); // 12
```

Cela ne fonctionne pas correctement. La sortie dans l'exemple ci-dessous est 12 (pour les valeurs d'invite par défaut).

Pourquoi ? Réparez-le. Le résultat doit être 3.

Opérateurs de base, mathématiques

La solution

La raison en est que le prompt renvoie l'entrée utilisateur sous forme de chaîne de caractères.
Les variables ont donc respectivement les valeurs "1" et "2".

```
let a = "1"; // prompt("First number?", 1);
let b = "2"; // prompt("Second number?", 2);

alert( a + b ); // 12
```

Ce que nous devons faire est de convertir les chaînes de caractères en nombres avant +. Par exemple, en utilisant Number() ou en les préfixant avec +.

Par exemple, juste avant prompt :

```
let a = +prompt("First number?", 1);
let b = +prompt("Second number?", 2);

alert( a + b ); // 3
```

JS

Les fondamentaux du javascript

Comparaisons

Comparaisons

Il y a de nombreux opérateurs de comparaison que nous connaissons des mathématiques :

- Plus grand/petit que : a > b, a < b.
- Plus grand/petit ou égal à : a >= b, a <= b.
- Égalité : a == b (veuillez noter le signe de la double égalité == signifie un test d'égalité. Un seul symbole a = b signifierait une affectation).
- Pas égal : en maths la notation est ≠, mais en JavaScript elle est écrite comme une assignation avec un signe d'exclamation : a != b.

Booléen est le résultat

Tout comme tous les autres opérateurs, une comparaison renvoie une valeur de type booléenne.

- true – signifie "oui", "correct" ou "vrai".
- false – signifie "non", "incorrect" ou "faux".

Par exemple :

```
alert( 2 > 1 );  // true (correct)
alert( 2 == 1 ); // false (faux)
alert( 2 != 1 ); // true (correct)
```

Comparaisons

Un résultat de comparaison peut être affecté à une variable, comme toute valeur :

```
let result = 5 > 4; // attribue le résultat de la comparaison
alert( result );  // true (correct)
```

Comparaison de chaînes de caractères

Pour voir quelle chaîne de caractères est plus grande que l'autre, on utilise l'ordre dit "dictionnaire" ou "lexicographique".

En d'autres termes, les chaînes de caractères sont comparées lettre par lettre.

Par exemple :

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
alert( '9' > '10' ); // true
```

Comparaisons

L'algorithme pour comparer deux chaînes de caractères est simple :

1. Compare les premiers caractères des deux chaînes de caractères.
2. Si le premier est supérieur (ou inférieur), la première chaîne de caractères est supérieure (ou inférieure) à la seconde. Nous en avons fini.
3. Sinon, si les premiers caractères sont égaux, comparez les deuxièmes caractères de la même manière.
4. Répéter jusqu'à la fin d'une chaîne de caractères.
5. Si les deux chaînes de caractères se sont terminées simultanément, alors elles sont égales. Sinon, la chaîne la plus longue est plus grande.

Dans l'exemple ci-dessus, la comparaison 'Z' > 'A' obtient le résultat à la première étape.

La deuxième comparaison 'Glow' et 'Glee' nécessite plus d'étapes car les chaînes de caractères sont comparées caractère par caractère :

1. G est identique à G.
2. l est identique à l.
3. o est plus grand que e. On stop ici. La première chaîne de caractères est plus grande.

Comparaisons

Pas vraiment un dictionnaire, mais plutôt l'ordre Unicode

L'algorithme de comparaison ci-dessus est à peu près équivalent à celui utilisé dans les dictionnaires ou les annuaires téléphoniques. Mais ce n'est pas exactement la même chose.

Par exemple, cette notion est importante à comprendre. Une lettre majuscule "A" n'est pas égale à la lettre minuscule "a". Lequel est le plus grand ? En fait, le minuscule "a" l'est. Pourquoi ? Parce que le caractère minuscule a un index plus grand dans la table de codage interne (Unicode). Nous reviendrons sur les détails spécifiques et leurs conséquences dans le chapitre Strings.

Comparaison de différents types

Lorsque les valeurs comparées appartiennent à différents types, elles sont converties en nombres.

Par exemple :

```
alert( '2' > 1 );   // true, la chaîne '2' devient le nombre 2
alert( '01' == 1 ); // true, chaîne '01' devient le nombre 1
```

Comparaisons

Pour les valeurs booléennes, true devient 1 et false devient 0.

Par exemple :

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Une conséquence amusante

Il est possible que dans le même temps :

- *Deux valeurs sont égales.*
- *L'un d'eux est vrai comme booléen et l'autre est faux comme booléen.*

Par exemple :

```
let a = 0;
alert( Boolean(a) ); // false
let b = "0";
alert( Boolean(b) ); // true
alert( a == b ); // true!
```

Du point de vue de JavaScript, c'est tout à fait normal. Un contrôle d'égalité convertit en utilisant la conversion numérique (par conséquent, "0" devient 0), tandis que la conversion booléenne utilise un autre ensemble de règles.

Comparaisons

Égalité stricte

Une vérification d'égalité régulière == a un problème. Elle ne peut pas faire la différence entre 0 et false :

```
alert( false == 0 ); // true
```

La même chose avec une chaîne de caractères vide :

```
alert( false == "" ); // true
```

C'est parce que les opérandes de différents types sont convertis en un nombre par l'opérateur d'égalité ==. Une chaîne de caractères vide, tout comme false, devient un zéro.

Que faire si nous voulons différencier 0 de faux ?

Un opérateur d'égalité stricte === vérifie l'égalité sans conversion de type.

En d'autres termes, si a et b sont de types différents, alors a === b renvoie immédiatement false sans tenter de les convertir.

Comparaisons

Essayons :

```
alert( false === 0 ); // false
```

Il existe également un opérateur de "strict non-égalité" **!==,** par analogie à la non-égalité **!=**.

L'opérateur de vérification de l'égalité stricte est un peu plus long à écrire, mais rend évident ce qui se passe et laisse moins d'espace pour les erreurs.

Comparaison avec null et undefined

Il existe un comportement non intuitif lorsque null ou undefined sont comparés à d'autres valeurs.

Pour un contrôle de strict égalité ===

Ces valeurs sont différentes car chacune d'entre elles appartient à un type distinct.

```
alert( null === undefined ); // false
```

Pour un contrôle d'égalité non strict ==

Il y a une règle spéciale. Ces deux là forment "un beau couple" : ils sont égaux (au sens de ==), mais pas à d'autres valeurs.

```
alert( null === undefined ); // true
```

Comparaisons

Pour les maths et autres comparaisons <, >, <=, >=
Les valeurs null/undefined sont converties en un nombre : null devient 0, alors qu'undefined devient NaN.

Voyons maintenant des choses amusantes qui se produisent lorsque nous appliquons ces règles. Et, ce qui est plus important, comment ne pas tomber dans un piège avec ces caractéristiques.

L'étrange résultat : null vs 0
Comparons null avec un zéro :

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

Ouais, mathématiquement c'est étrange. Le dernier résultat indique que "null est supérieur ou égal à zéro".
Alors que l'une des comparaisons au dessus devrait être correcte, mais les deux sont fausses.
La raison est qu'une vérification d'égalité (==) et les comparaisons (<, >, <=, >=) fonctionnent différemment. Les comparaisons convertissent null en un nombre, donc le traitent comme 0.
C'est pourquoi (3) null >= 0 est vrai et (1) null > 0 est faux.
D'un autre côté, la vérification de l'égalité == pour undefined et null est définie de telle sorte que, sans aucune conversion, ils sont égaux et ne correspondent à rien d'autre. C'est pourquoi (2) null == 0 est faux.

Comparaisons

Un undefined incomparable

La valeur undefined ne doit pas du tout participer aux comparaisons :

```
alert( undefined > 0 );  // (1) false
alert( undefined < 0 );  // (2) false
alert( undefined == 0 ); // (3) false
```

Pourquoi est-ce qu'il n'aime pas le zéro ? Toujours faux!
Nous avons ces résultats parce que :

- Les comparaisons (1) et (2) renvoient false car undefined est converti en NaN et NaN est une valeur numérique spéciale qui renvoie false pour toutes les comparaisons.
- Le contrôle d'égalité (3) renvoie false, car undefined est uniquement égal à null et à aucune autre valeur.

Éviter les problèmes

Pourquoi avons-nous observé ces exemples? Devrions-nous nous souvenir de ces particularités tout le temps ? Eh bien pas vraiment. En fait, ces notions délicates deviennent progressivement familières au fil du temps, mais il existe un moyen solide d'éviter tout problème avec elles.

Comparaisons

Il suffit de traiter toute comparaison avec null/undefined (à l'exception de la stricte égalité ===) avec un soin exceptionnel.

N'utilisez pas de comparaisons =>, >, <, <= avec une variable qui peut être null/undefined, sauf si vous êtes vraiment sûr de ce que vous faites. Si une variable peut avoir de telles valeurs, vérifiez-les séparément.

Résumé

- Les opérateurs de comparaison renvoient une valeur logique.
- Les chaînes de caractères sont comparées lettre par lettre dans l'ordre "dictionnaire".
- Lorsque des valeurs de différents types sont comparées, elles sont converties en nombres (à l'exclusion d'un contrôle d'égalité strict).
- Les valeurs null et undefined sont égales (==) et ne correspondent à aucune autre valeur.
- Soyez prudent lorsque vous utilisez des comparaisons telles que > ou < avec des variables pouvant parfois être null/undefined. Faire une vérification séparée pour null/undefined est une bonne idée.

JS

Exercices

Comparaisons

Quel sera le résultat pour les expressions suivantes :

```
5 > 4
"apple" > "pineapple"
"2" > "12"
undefined == null
undefined === null
null == "\n0\n"
null === +"\n0\n"
```

Comparaisons

Quel sera le résultat pour les expressions suivantes :

```
1 - 5 > 4                   => true
2 - "apple" > "pineapple"   => false
3 - "2" > "12"              => true
4 - undefined == null       => true
5 - undefined === null      => false
6 - null == "\n0\n"         => false
7 - null === +"\n0\n"       => false
```

1. Évidemment, c'est vrai.
2. Comparaison du dictionnaire, donc fausse. "a" est plus petit que "p".
3. Encore une fois, comparaison du dictionnaire, le premier caractère de "2" est plus grand que le premier caractère de "1".
4. Les valeurs null et undefined sont exclusivement égale entre elles.
5. L'égalité stricte est stricte. Des types différents des deux côtés conduisent à false.
6. Similaire à (4), null n'est égale qu'à undefined.
7. Egalité stricte de différents types.

JS

Les fondamentaux du javascript

Branche conditionnelle : if, '?'

Comparaisons

Branche conditionnelle : if, '?'

Parfois, nous devons effectuer différentes actions en fonction d'une condition.

Pour ce faire, nous pouvons utiliser l'instruction if et l'opérateur conditionnel ?, également appelé opérateur "point d'interrogation".

L'instruction "if"

L'instruction if(...) évalue une condition entre parenthèses et, si le résultat est true, exécute un bloc de code.

Par exemple :

```
let year = prompt('En quelle année la spécification ECMAScript-2015 à été publiée ?' '');
if (year == 2015) alert( 'Vous avez raison !' );
```

Dans l'exemple ci-dessus, la condition est une vérification d'égalité simple : year == 2015, mais elle peut être beaucoup plus complexe. S'il y a plus d'une instruction à exécuter, nous devons envelopper notre bloc de code entre accolades.

Il est recommandé d'entourer votre bloc de code avec des accolades {} à chaque fois avec if, même s'il n'y a qu'une seule instruction.

Cela améliore la lisibilité.

```
if (year == 2015){
  alert( 'Vous avez raison !' )
  alert( 'Vous êtes fort !' )
};
```

Comparaisons

Conversion booléenne

L'instruction if (...) évalue l'expression entre parenthèses et la convertit en type booléen.

Rappelons les règles de conversion du chapitre Les conversions de types:

- Un nombre 0, une chaîne de caractères vide "", null, undefined et NaN deviennent false. À cause de cela, on dit de ces valeurs qu'elles sont "falsy".
- Les autres valeurs deviennent true, on dit qu'elles sont "truthy".

Donc, le code sous cette condition ne sera jamais exécuté :

```
if ( 0 ){// 0 est falsy
  ...
}
```

... Et à l'intérieur de cette condition – il fonctionne toujours :

```
if ( 1 ){// 1 est truthy
  ...
}
```

Comparaisons

Nous pouvons également transmettre une valeur booléenne pré-évaluée à if, comme ici :

```
let cond = (year == 2015); // l'égalité évalue à vrai ou faux
if ( cond ){
  ...
}
```

La clause "else"

L'instruction if peut contenir un bloc optionnel else. Il s'exécute lorsque la condition est fausse.

Par exemple :

```
let year = prompt('En quelle année la spécification ECMAScript-2015 à été publiée ?' '');
if (year == 2015){
  alert( 'Vous avez raison !' );
} else {
  alert( 'Vous avez tort !' );
}
```

Comparaisons

Plusieurs conditions : “else if”

Parfois, nous aimerions tester plusieurs variantes d'une condition. Il y a une clause else if pour cela.
Par exemple :

```
let year = prompt('En quelle année la spécification ECMAScript-2015 à été publiée ?' '');
if (year < 2015){
  alert( 'Trop tôt !' );
} else if (year > 2015){
  alert( 'Trop tard !' );
} else {
  alert( 'Vous avez raison !' );
}
```

Dans le code ci-dessus, JavaScript vérifie year < 2015. S'il est falsy, il passe à l'année de condition suivante year > 2015 et c'est toujours false il passe à la dernière instruction et affiche la dernière alerte.
Il peut y avoir plus de blocks else if.
Le dernier else est optionnel.

Comparaisons

Opérateur ternaire '?'

Parfois, nous devons attribuer une variable en fonction d'une condition. Par exemple :

```
let accessAllowed;
let age = prompt('How old are you?', '');

if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
```

L'opérateur dit "ternaire" ou "point d'interrogation" nous permet de le faire plus rapidement et plus simplement. L'opérateur est représenté par un point d'interrogation ?. Appelé aussi "ternaire" parce que l'opérateur a trois opérandes. C'est en fait le seul et unique opérateur en JavaScript qui en a autant.

Comparaisons

La syntaxe est :

```
let result = condition ? value1 : value2;
```

La condition est évaluée, si elle est vraie, alors value1 est retournée, sinon – value2. Par exemple :

```
let accessAllowed = (age > 18) ? true : false
```

Techniquement, nous pouvons omettre les parenthèses autour de age > 18. L'opérateur point d'interrogation a une faible précédence, il s'exécute donc après la comparaison >. Cet exemple fera la même chose que le précédent :

```
// l'opérateur de comparaison "age > 18" s'exécute en premier
// (pas besoin de l'envelopper entre parenthèses)
let accessAllowed = (age > 18) ? true : false
```

Mais les parenthèses rendent le code plus lisible, il est donc recommandé de les utiliser.

Comparaisons

Veuillez noter

Dans l'exemple ci-dessus, il est possible d'éviter l'opérateur ternaire, parce que la comparaison elle-même renvoie un true/false:

```
let accessAllowed = age > 18
```

Multiple '?'

Une séquence d'opérateurs point d'interrogation ? permettent de renvoyer une valeur qui dépend de plusieurs conditions. Par exemple :

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hello, bébé!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Félicitations !' : 'Le bel âge!';
alert( message );
```

Comparaisons

Il peut être difficile au début de comprendre ce qui se passe. Mais après un examen plus approfondi, nous constatons que ce n'est qu'une séquence de tests ordinaire.

Le premier point d'interrogation vérifie si age < 3.

1. Si vrai – retourne 'Hello, bébé !', Sinon, il continue avec l'expression après les deux points ":" suivants et vérifie si age < 18.
2. Si vrai – retourne 'Hello!', Sinon, il continue avec l'expression après les deux points ":" suivants et vérifie si age < 100.
3. Si vrai – retourne Félicitations !', Sinon, l'expression continue après les derniers deux-points et retourne 'Le bel âge !'.

La même logique utilisant if..else :

```
if ( age < 3 ){
   message = 'Hello, bébé !';
} else if (age < 18){
   message = 'Hello!';
} else if (age < 100) {
  message = 'Félicitations !';
} else {
  message = 'Le bel âge!';
}
```

Comparaisons

Utilisation non traditionnelle de '?'

Parfois, le point d'interrogation ? est utilisé en remplacement de if :

```
let company = prompt('Qui à créer Javascript ?','');
(company == 'Netscape') ? alert( 'Vrai!') : alert( 'Faux!' );
```

Selon si la condition company == 'Netscape' est vraie ou non, la première ou la deuxième partie après ? s'exécute et affiche l'alerte correspondante.

Nous n'attribuons pas de résultat à une variable ici. L'idée est d'exécuter un code différent en fonction de la condition.

Il n'est pas recommandé d'utiliser l'opérateur ternaire de cette manière.

La notation semble être plus courte qu'un if, ce qui plaît à certains programmeurs. Mais c'est moins lisible.

Voici le même code avec if pour comparaison :

```
let company = prompt('Qui à créer Javascript ?','');
if (company == 'Netscape'){
  alert( 'Vrai!')
} else {
  alert( 'Faux!' );
}
```

JS

Exercices

Comparaisons

if (une chaîne de caractères avec zéro)

Est-ce que alert sera affiché ?

```
if ( '0' ){
  alert( 'Bonjour !')
}
```

Comparaisons

if (une chaîne de caractères avec zéro)
Est-ce que alert sera affiché ?

```
if ( '0' ){
  alert( 'Bonjour !')
}
```

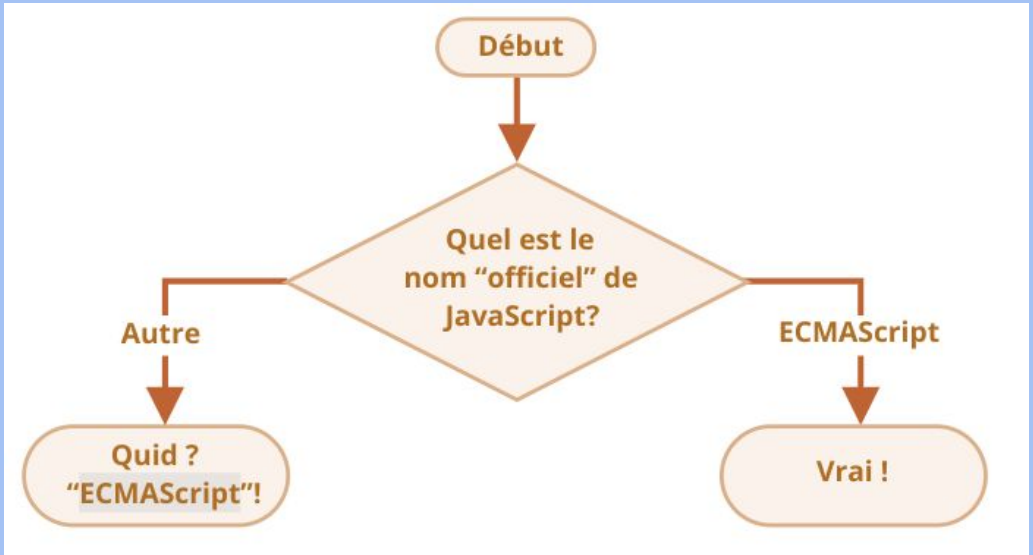
Oui, il sera affiché.

Toute chaîne de caractères à l'exception d'une chaîne vide (et "0" n'est pas vide) devient vraie dans le contexte logique.

Comparaisons

Le nom de javascript

En utilisant la construction if..else, écrivez le code qui demande : 'Quel est le nom "officiel" de JavaScript?'
Si le visiteur entre "ECMAScript", alors éditez une sortie "Bonne réponse !", Sinon – retourne "Ne sait pas ? ECMAScript!"



Comparaisons

Le nom de javascript

En utilisant la construction if..else, écrivez le code qui demande : 'Quel est le nom "officiel" de JavaScript?'
Si le visiteur entre "ECMAScript", alors éditez une sortie "Bonne réponse !", Sinon – retourne "Ne sait pas ? ECMAScript!"

```
<!DOCTYPE html>
<html>
<body>
  <script>
    'use strict';
    let value = prompt('Quel est le nom "officiel" de JavaScript?', '');
    if (value == 'ECMAScript') {
      alert('Right!');
    } else {
      alert("You don't know? ECMAScript!");
    }
  </script>
</body>
</html>
```

Comparaisons

Afficher le signe

En utilisant if..else, écrivez le code qui obtient un numéro via le prompt, puis l'affiche en alert :

- 1, si la valeur est supérieure à zéro,
- -1, si inférieur à zéro,
- 0, si est égal à zéro.

Dans cet exercice, nous supposons que l'entrée est toujours un nombre.

Comparaisons

Afficher le signe

En utilisant if..else, écrivez le code qui obtient un numéro via le prompt, puis l'affiche en alert :

- 1, si la valeur est supérieure à zéro,
- -1, si inférieur à zéro,
- 0, si est égal à zéro.

Dans cet exercice, nous supposons que l'entrée est toujours un nombre.

```
let value = prompt('Saisir un nombre',0);
if (value > 0){
  alert( 1 )
} else if(value < 0) {
  alert( -1 );
} else {
  alert( 0 );
}
```

Comparaisons

Réécrire 'if' en '?'

Réécrivez ce if en utilisant l'opérateur conditionnel '?' :

```
let result;
if ( a + b < 4 ){
  result = 'Plus petit'
} else {
  result = 'Plus grand'
}
```

Comparaisons

Réécrire 'if' en '?'

Réécrivez ce if en utilisant l'opérateur conditionnel '?' :

```
let result = ( a + b < 4 ) ? 'Plus petit' : 'Plus grand';
```

Comparaisons

Réécrire 'if..else' en '?'

Réécrire ce if..else en utilisant plusieurs opérateurs ternaires '?'.
Pour plus de lisibilité, il est recommandé de diviser le code en plusieurs lignes.

```
let message;
if ( login == 'Employé' ){
  message = 'Hello';
} else if (login == 'Directeur') {
  message = 'Félicitation';
} else if (login == '') {
  message = 'Pas de login';
} else {
  message = '';
}
```

Comparaisons

Réécrire 'if..else' en '?'

Réécrire ce if..else en utilisant plusieurs opérateurs ternaires '?'.
Pour plus de lisibilité, il est recommandé de diviser le code en plusieurs lignes.

```
let message = (login == 'Employee') ? 'Hello' :
  (login == 'Director') ? 'Greetings' :
  (login == '') ? 'No login' :
  '';
```

JS

Les fondamentaux du javascript

Opérateurs logiques

Opérateurs logiques

Il y a trois opérateurs logiques en JavaScript : || (OR), && (AND), ! (NOT), ?? (Coalescence des nulles).
Nous couvrons ici les trois premiers, l'opérateur ?? sera dans une prochaine leçon.
Bien qu'ils soient appelés "logiques", ils peuvent être appliqués à des valeurs de tout type, pas seulement booléennes. Le résultat peut également être de tout type.

|| (OR)

L'opérateur "OR" est représenté avec deux symboles de ligne verticale :

```
let resut = a || b;
```

En programmation classique, le OU logique est destiné à manipuler uniquement les valeurs booléennes.
Si l'un de ses arguments est true, alors il renvoie true, sinon il renvoie false.
En JavaScript, l'opérateur est un peu plus compliqué et puissant. Mais voyons d'abord ce qui se passe avec les valeurs booléennes.
Il existe quatre combinaisons logiques possibles :
Comme on peut le voir, le résultat est toujours true sauf pour le cas où les deux opérandes sont false.
Si un opérande n'est pas booléen, il est converti en booléen pour l'évaluation.

```
alert( true || true ) // true
alert( false || true ) // true
alert( true || false ) // true
alert( false || false ) // false
```

Opérateurs logiques

Par exemple, un nombre 1 est traité comme true, un nombre 0 – comme false :

```
if( 1 || 0 ) { // fonctionne comme ( true || false )
   alert( 'truthy!' )
}
```

La plupart du temps, OR || est utilisé dans une instruction if pour tester si l'une des conditions données est correcte. Par exemple :

```
let hour = 9;
if( hour < 10 || hour > 18 ) {
   alert( 'Le bureau est fermé !' )
}
```

Nous pouvons passer plus de conditions :

```
let hour = 9;
let isWeekend = true;
if( hour < 10 || hour > 18 || isWeekend ) {
   alert( 'Le bureau est fermé !' )
}
```

Opérateurs logiques

OR "||" cherche la première valeur vraie

La logique décrite ci-dessus est quelque peu classique. Maintenant, apportons les fonctionnalités "supplémentaires" de JavaScript.

L'algorithme étendu fonctionne comme suit.

En cas de multiples valeurs liées par OR :

```
let resut = value1 || value2 || value3;
```

L'opérateur OR || fait ce qui suit :
Évaluez les opérandes de gauche à droite.
Pour chaque opérande, il le convertit en booléen. Si le résultat est true, arrêtez et retournez la valeur d'origine de cet opérande.
Si tous les autres opérandes ont été évalués (c'est-à-dire que tous étaient false), renvoyez le dernier opérande.
Une valeur est renvoyée sous sa forme d'origine, sans conversion.
En d'autres termes, une chaîne de OR || renvoie la première valeur true ou la dernière valeur si aucune valeur true n'a été trouvée.

Opérateurs logiques

Par exemple

```
alert( true || true ) // true
alert( 1 || 0 ); // 1 (1 est vrai)

alert( null || 1 ); // 1 (1 est la première valeur vraie)
alert( null || 0 || 1 ); // 1 (la première valeur vraie)

alert( undefined || null || 0 ); // 0 (tous faux, renvoie la dernière valeur)
```

Cela conduit à des usages intéressants par rapport à un "OR pur, classique, booléen uniquement".

1 Obtenir la première valeur vraie dans la liste des variables ou des expressions.

Par exemple, nous avons les variables firstName, lastName et nickName, toutes optionnelles (c'est-à-dire peut être indéfini ou avoir des valeurs fausses).

Utilisons OR || pour choisir celui qui contient les données et l'afficher (ou Anonymous si rien n'est défini) :

Opérateurs logiques

Par exemple

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

Si toutes les variables étaient fausses, ce serait "Anonymous" qui apparaîtrait.

2 Évaluation des courts-circuits.

Une autre caractéristique de l'opérateur OR || est l'évaluation dite de "court-circuit".

Cela signifie que || traite ses arguments jusqu'à ce que la première valeur de vérité soit atteinte, puis la valeur est renvoyée immédiatement, sans même toucher l'autre argument.

L'importance de cette fonctionnalité devient évidente si un opérande n'est pas seulement une valeur, mais une expression avec un effet secondaire, comme une affectation de variable ou un appel de fonction.

Opérateurs logiques

Dans l'exemple ci-dessous, seul le deuxième message est imprimé :

```
true || alert("not printed");
false || alert("printed");
```

Dans la première ligne, l'opérateur OR || arrête l'évaluation immédiatement après avoir vu true, de sorte que alert n'est pas exécuté.

Parfois, les gens utilisent cette fonctionnalité pour exécuter des commandes uniquement si la condition sur la partie gauche est fausse.

&& (AND)

L'opérateur AND est représenté avec deux esperluettes && :

```
result = a && b;
```

En programmation classique, AND retourne true si les deux opérandes sont true et false dans les autres cas :

Opérateurs logiques

```
alert( true && true ) // true
alert( false && true  ); // false
alert( true  && false ); // false
alert( false && false ); // false
```

Un exemple avec if:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( Il est 12:30' );
}
```

Tout comme pour OR, toute valeur est autorisée en tant qu'opérande de AND :

```
if (1 && 0) { // évalué comme true && false
  alert( "Ne marchera pas, car le résultat est faux" );
}
```

Opérateurs logiques

En cas de multiples valeurs liées par AND :

```
result = value1 && value2 && value3;
```

L'opérateur AND && effectue les opérations suivantes :

- Évalue les opérandes de gauche à droite.
- Pour chaque opérande, il le convertit en booléen. Si le résultat est false, arrêtez et retournez la valeur d'origine de cet opérande.
- Si tous les autres opérandes ont été évalués (c'est-à-dire tous étaient vrais), retournez le dernier opérande.

En d'autres termes, une chaîne de AND && renvoie la première valeur false ou la dernière valeur si aucune valeur false n'a été trouvée.

Les règles ci-dessus sont similaires à OR. La différence est que AND retourne la première valeur false tandis que OR renvoie la première valeur true.

Opérateurs logiques

Exemple :

```
// si le premier opérande est vrai,
// AND retourne le second opérande :
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5

// si le premier opérande est faux,
// AND le retourne. Le deuxième opérande est ignoré
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

Nous pouvons également transmettre plusieurs valeurs à la suite sur une même ligne. Voyez comment le premier faux est retourné :

```
alert( 1 && 2 && null && 3 ); // null
```

Lorsque toutes les valeurs sont vraies, la dernière valeur est renvoyée :

```
alert( 1 && 2 && 3 ); // 3 la dernière
```

Opérateurs logiques

La précédence de AND && est supérieure à OR ||

La priorité de l'opérateur AND && est supérieure à OR ||.

Donc, le code a && b || c && d est essentiellement le même que si && était entre parenthèses: (a && b) || (c && d).

! (NOT)

L'opérateur booléen NOT est représenté par un point d'exclamation !.

La syntaxe est assez simple :

```
result = !value1
```

L'opérateur accepte un seul argument et effectue les opérations suivantes :

1. Convertit l'opérande en type booléen : true/false.
2. Renvoie la valeur inverse.

Par exemple :

```
alert( !true ); // false
alert( !0 ); // true
```

Opérateurs logiques

Un double NOT !! est parfois utilisé pour convertir une valeur en type booléen :

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
```

C'est-à-dire que le premier NOT convertit la valeur en booléen et retourne l'inverse, et que le second NOT l'inverse encore. À la fin, nous avons une conversion valeur à booléen simple.

Il existe un moyen un peu plus verbeux de faire la même chose – une fonction Boolean intégrée :

```
alert( Boolean("non-empty string") ); // true
alert( Boolean(!!null) ); // false
```

La précédence de NOT ! est la plus élevée de tous les opérateurs binaire, il est donc toujours exécuté en premier, avant les autres.

JS

Exercices

Opérateurs logiques

Quel est le résultat de OR ?
Qu'est-ce que le code ci-dessous va sortir ?

```
alert( null || 2 || undefined );
```

La réponse est **2**, c'est la première valeur vraie.

Quel est le résultat de AND ?
Qu'est-ce que le code ci-dessous va sortir ?

```
alert( 1 && null && 2 );
```

La réponse est **null**, c'est la première valeur fausse.

Quel est le résultat de OR AND OR ?
Qu'est-ce que le code ci-dessous va sortir ?

```
alert( null || 2 && 3 || 4 );
```

La réponse est **3**,
La priorité de AND && est supérieure à OR ||, elle s'exécute donc en premier.
Le résultat de 2 && 3 = 3, donc l'expression devient : **null || 3 || 4**
Maintenant, le résultat est la première valeur vraie : **3**.

Opérateurs logiques

Vérifiez la plage entre

Ecrivez une condition "if" pour vérifier que l'age est compris entre 14 et 90 ans inclus.
"Inclus" signifie que l'age peut atteindre les 14 ou 90 ans.

```
if (age >= 14 && age <= 90)
```

Vérifiez à l'extérieur de la plage

Ecrivez une condition if pour vérifier que l'age n'est PAS compris entre 14 et 90 ans inclus.
Créez deux variantes: la première utilisant NOT !, La seconde – sans ce dernier.

La première variante :

```
if (!(age >= 14 && age <= 90))
```

La seconde variante :

```
if (age < 14 || age > 90)
```

Opérateurs logiques

Vérifier la connexion

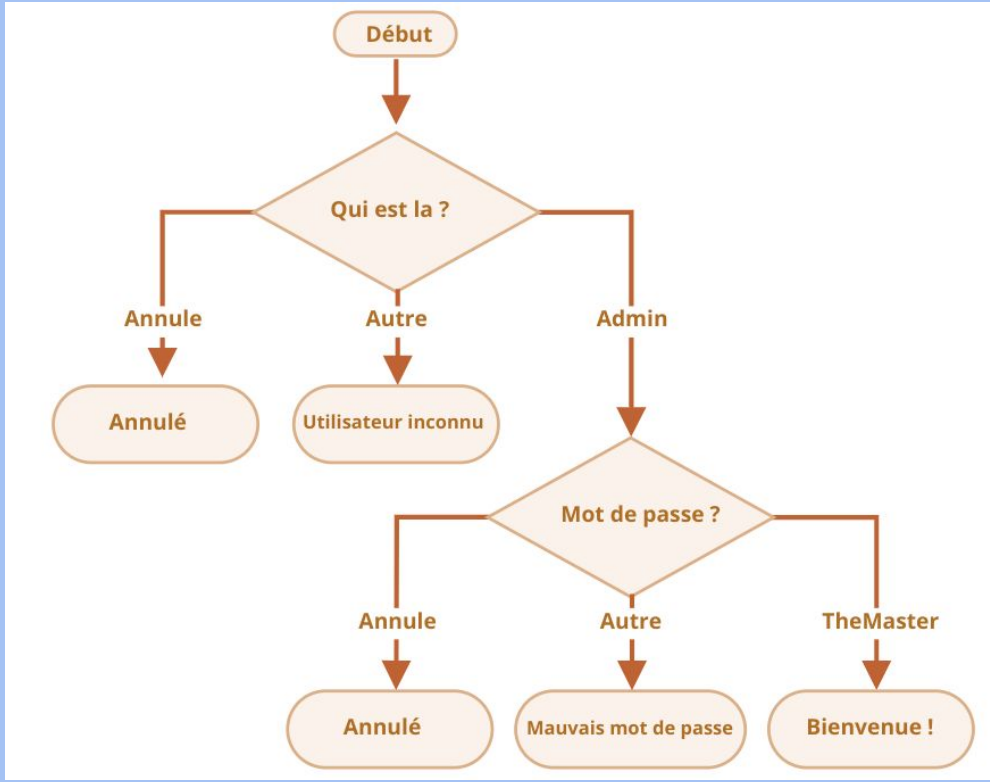
Écrivez le code qui demande une connexion avec prompt.
Si le visiteur entre "Admin", puis prompt pour un mot de passe, si l'entrée est une ligne vide ou Esc - affichez "Annulé", s'il s'agit d'une autre chaîne de caractères - alors affichez "Utilisateur inconnu".

Le mot de passe est vérifié comme suit :

- S'il est égal à "TheMaster", alors affichez "Bienvenu!",
- Une autre chaîne de caractères - affichez "Mot de passe invalide",
- Pour une chaîne de caractères vide ou une entrée annulée, affichez "Annulé".

Veuillez utiliser des blocs if imbriqués. Attention à la lisibilité globale du code.
Astuce: passer une entrée vide à un prompt renvoie une chaîne de caractères vide ''. En pressant ESC lors d'un prompt cela retourne null.

Opérateurs logiques



Opérateurs logiques

```
let userName = prompt("Who's there?", '');

if (userName === 'Admin') {
  let pass = prompt('Password?', '');
  if (pass === 'TheMaster') {
    alert( 'Welcome!' );
  } else if (pass === '' || pass === null) {
    alert( 'Canceled' );
  } else {
    alert( 'Wrong password' );
  }
} else if (userName === '' || userName === null) {
  alert( 'Canceled' );
} else {
  alert( "I don't know you" );
}
```