LEARN JAVASCRIPT

BEGINNERS EDITION

A Complete Beginner's Guide to Learn JavaScript

Suman Kunwar

Table des matières

Dédicace

Cet ouvrage est dédié, avec respect et admiration, à l'esprit informatique et aux langages de programmation partout autour du monde.

> “L’art de la programmation peut être vu comme l’art d’organiser la complexité, de maîtriser la multitude et d’éviter le plus efficacement possible ce damné chaos qu'elle peut provoquer.”
>
> - Edsger Dijkstra

Copyright

ASIN: B0C53J11V7

Préface

"Apprendre JavaScript pour les débutants" est un livre qui propose une exploration progressive du langage JavaScript. Cette technologie, tant elle a influencé le paysage digital est un incontournable pour tous les développeurs web. En mettant l'accent sur les bases et l'aspect pratique, cette ressource s'adresse à tous ceux qui souhaitent apprendre le langage de programmation JavaScript.

Le livre débute par les aspects fondamentaux de JavaScript, pour progresser peu à peu vers des techniques plus avancées. Il aborde des sujets clés tels que les variables, les types de données, les structures de contrôle, les fonctions, la programmation orientée objet, les closures, les promesses et la syntaxe moderne ES6. Chaque chapitre s'appuie sur le précédent, fournissant une base solide aux apprenants et facilitant la compréhension des concepts les plus avancés.

Une caractéristique singulière d'"Apprendre JavaScript pour les débutants" est son aspect pratique. Le livre propose des exercices interactifs, des défis de codage et des problèmes issus du monde réel qui permettent aux lecteurs d'appliquer leurs connaissances et de développer des compétences essentielles. En s'appuyant sur des exemples concrets, les lecteurs acquièrent la confiance nécessaire pour résoudre les problèmes courants de développement web et libérer le potentiel de JavaScript en matière de solutions innovantes.

Les concepts avancés tels que les closures et la programmation asynchrone sont démystifiées grâce à des explications intuitives et des exemples pratiques. L'accent mis sur la clarté et la simplicité permet aux lecteurs de tous niveaux de saisir et de retenir efficacement les concepts clés du langage. Le livre est structuré en trois parties, les 14 premiers chapitres approfondissant les concepts de base. Les quatre chapitres suivants développent l'utilisation de JavaScript pour la programmation de navigateurs web, tandis que les deux derniers chapitres couvrent divers sujets et proposent des exercices. La section "Divers" explore des thèmes et des scénarii importants relatifs à la programmation JavaScript, suivis d'exercices pratiques.

En conclusion, "Apprendre JavaScript pour les débutants" est le compagnon indispensable pour ceux qui cherchent à maîtriser JavaScript et à progresser dans le développement web. Avec sa couverture complète, son approche pratique, ses explications claires, son orientation vers les applications concrètes et son engagement en faveur d'un apprentissage continu, ce livre constitue une ressource précieuse. En s'immergeant dans son contenu, les lecteurs acquerront les compétences et les connaissances nécessaires pour créer des applications web dynamiques et interactives, libérant ainsi tout leur potentiel en tant que développeurs JavaScript.

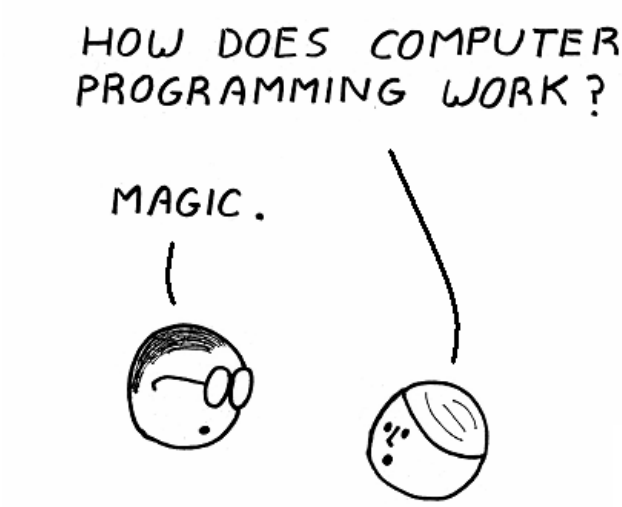
Note du traducteur: Dans les langages de programmation, la langue utilisée est l'anglais. D'ailleurs, il est conseillé aux aspirants développeurs de pratiquer l'anglais et de se forcer à le faire, car de nombreuses ressources sont disponibles uniquement dans cette langue. Ici, nous avons essayé de traduire tous les concepts au mieux tout en respectant l'esprit de la programmation. Cependant, les variables et les mots réservés ne font pas l'objet de traduction, car ils font partie de l'approche du code et permettent de le découvrir tel qu'il existe dans la réalité. D'ailleurs, nommer ses variables en français est une mauvaise habitude, et nous vous incitons à utiliser l'anglais partout dans votre code, même dans vos commentaires si vous êtes assez à l'aise. N'oubliez pas que votre code est fait pour être écrit une fois et lu cent fois. Afin de pouvoir lire facilement le code de n'importe quel programmeur dans le monde, et le partager à votre tour, utilisez l'anglais. Outre cet aspect, sachez que certains termes ne possèdent pas d'équivalents français qui aient du sens, ou pas suffisamment, dans le contexte de JavaScript. L'exemple typique est celui de la "closure". Si on pourrait traduire ce terme par "fermeture", en JavaScript, une "closure" est une "closure". Même en français. Dans la mesure du possible, nous essayerons d'alerter les lecteurs sur ce genre de cas. Bonne lecture !

Chapitre 1

Introduction

De nos jours, les ordinateurs sont partout. Ils sont à même d'accomplir une grande variété de tâches rapidement et précisément. On les utilise dans de nombreux secteurs différents, comme le commerce, la santé, l'éducation ou le divertissement. De plus, ils sont devenus incontournables dans nos vies de tous les jours. Enfin, les ordinateurs sont également utilisés pour réaliser des calculs scientifiques et mathématiques complexes, pour stocker et interpréter de larges quantités de données et pour communiquer avec n'importe qui à travers le monde.

Programmer implique le fait de créer une suite d'instructions, appelées "programme", et de les faire appliquer par un ordinateur. Certaines fois, écrire un programme peut s'avérer être une entreprise fastidieuse et frustrante, car les ordinateurs réclament de la précision et des instructions spécifiques pour accomplir les tâches qui leur sont demandées.



Les langages de programmation sont des langages qui permettent d'échanger des instructions avec les ordinateurs. On les utilise dans la majorité des phases de programmation et ils sont établis sur la manière dont échangent les humains les uns avec les autres. Comme les langages que nous utilisons dans la vie de tous les jours, les langages de programmation utilisent les mots et les combinent dans des phrases pour exprimer un concept donné. Il est intéressant de noter que la manière la plus répandue de communiquer avec les machines est basée sur un langage similaire à celui utilisé par les humains dans la vie de tous les jours. Bien souvent, les mots-clés utilisés dans un langage de programmation tel que JavaScript sont en anglais et possèdent un "sens humain".

Par le passé, la manière privilégiée d'interagir avec les ordinateurs était basée sur des langages proches de la machine comme BASIC ou DOS. Aujourd'hui, cette manière a largement été supplantée par des interfaces visuelles, plus faciles à appréhender, mais qui offrent moins de flexibilité. Cependant, des langages comme *JavaScript* sont toujours utilisés et ont su s'adapter pour être toujours au premier plan dans les interfaces modernes et les navigateurs web courants.

JavaScript (*JS en abrégé*) est le langage de programmation qui est utilisé pour créer des interactions dynamiques sur les pages web, les jeux, les applications et même les serveurs. JavaScript a été créé chez Netscape, un navigateur web développé dans les années 1990. De nos jours, c'est un des langages informatiques le plus répandu et utilisé de par le monde.

À l'origine, JavaScript a été imaginé pour rendre les sites web "vivants". D'ailleurs, à cette époque, le langage n'était capable de s'exécuter qu'au sein d'un navigateur web. Actuellement, JavaScript peut être exécuté sur n'importe quel support qui est capable d'héberger son moteur de rendu. Les objets standards comme Array, Date, et Math sont disponibles en JavaScript. De la même manière, le langage supporte les opérateurs, les structures de contrôles et les déclarations. *JavaScript côté client* et *JavaScript côté serveur* sont deux manières d'appréhender le langage.

- *JavaScript côté client* permet de renforcer et de contrôler les pages web et le navigateur du client. Par exemple, il est possible de réagir aux événements utilisateur comme le clic de souris, la soumission d'un formulaire ou la navigation sur la page. En somme, JavaScript procure du contrôle et de l'interactivité.
- *JavaScript côté serveur* permet de créer et de paramétrer les serveurs, les bases de données et les systèmes de fichiers.

JavaScript est un langage interprété. Cela signifie que lors de l'exécution du code écrit en JavaScript, un programme spécifique appelé interpréteur est chargé de traduire les directives en langage machine. Les navigateurs modernes utilisent la technologie Just In Time (JIT) pour compiler le langage à la volée en code source exécutable.

> "LiveScript" était le nom initial donné à JavaScript.

Le code, et que faire avec

Le code est l'ensemble des instructions écrites qui permettent de concevoir un programme. Dans cet ouvrage, de nombreux chapitres contiennent des exemples de code. Au cours de votre apprentissage, il est capital de lire et d'écrire à votre tour des instructions pour progresser. Vous ne pouvez pas vous contenter de lire rapidement les exemples - essayez de mettre en place une lecture active. Pour ce faire, entrez en profondeur dans les instructions et essayez de comprendre leur sens et leurs répercussions. Au début, vous rencontrerez probablement quelques difficultés à réaliser cette tâche, mais en faisant preuve de détermination, vous gagnerez en compréhension. Il en va de même pour les exercices - essayez de pratiquer et de résoudre ces derniers avant de consulter la solution. C'est la clé pour progresser dans votre acquisition des concepts du langage. Il peut aussi être utile d'écrire votre code dans un interpréteur JavaScript (en ligne ou directement dans votre navigateur) pour tester celui-ci. Progressivement, vous gagnerez en expérience au gré des exercises proposés.

Conventions typographiques

Ci-dessous, le texte écrit dans la police monospace représente les éléments d'un programme. Ceci peut être un code à part entière ou une partie d'un programme plus complet. Les programmes, comme celui écrit ci-dessous, le sont toujours de cette manière:

```
const numbers = [45, 4, 9, 16, 25];
let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

Parfois, la sortie attendue d'une suite d'instructions est écrite immédiatement après celles-ci. Elle est précédée de deux slashs indiquant le résultat *Result*, comme en suivant:

```
console.log(txt);

// Result: txt = '45491625'
```

Chapitre 2

Bases du langage

Dans ce premier chapitre, nous allons découvrir les bases de la programmation en JavaScript.

Programmer signifie écrire du code. À l'image d'un livre qui est fait de chapitres, de paragraphes, de phrases, de mots, de lettres et de ponctuation, un programme est un tout qui peut être réduit à sa plus petite portion. Pour l'heure, la partie la plus importante s'appelle la déclaration. Une déclaration peut s'apparenter à une phrase à l'intérieur d'un livre. Une déclaration possède une fin en soi et peut se suffire à elle-même. Sans le contexte fourni par les autres déclarations qui l'entoure, une seule déclaration n'a généralement pas beaucoup de sens.

Une déclaration s'entend dans la plupart des cas (et de manière générale) comme un *ligne de code*. C'est parce que les déclarations ont tendance à être écrites sur une seule ligne. De la même manière, les programmes se lisent du haut vers le bas, de la gauche vers la droite. Vous vous demandez peut-être ce qu'est le code (aussi appelé code source). Il s'agit d'un terme large qui peut désigner l'ensemble du programme ou la plus petite partie. Une ligne de code est donc simplement une ligne de votre programme.

En voici un exemple simple:

```
let hello = "Hello";
let world = "World";

// Message est égal à "Hello World"
let message = hello + " " + world;
```

Ce code peut être exécuté par un autre programme appelé *interprétreur* qui va se charger de lire le code et exécuter les différentes déclarations dans l'ordre attendu par la machine.

Commentaires

Les commentaires sont des déclarations qui ne sont pas exécutées par l'interpréteur. En effet, les commentaires sont faits pour faire apparaître des annotations à destination des autres programmeurs ou de courtes descriptions sur le fonctionnement du code. Les commentaires doivent fournir des explications supplémentaires de manière à mieux comprendre le code. D'autre part, les commentaires peuvent servir à "échapper" temporairement du code sans toucher au reste du programme.

En JavaScript, les commentaires peuvent être écrits de deux manières:

- *Commentaire monoligne*: Le commentaire commence avec deux slashs (`//`) et se prolonge jusqu'à la fin de la ligne courante. Tout ce qui suit le double slash sera ignoré par l'interpréteur JavaScript. Par exemple:

```
// Ceci est un commentaire monoligne, il sera ignoré par l'interpréteur
let a = "ceci est une variable définie dans une déclaration";
```

- *Commentaires multilignes*: Le commentaire commence avec un slash suivi directement par un astérisque (`/*`) et se termine par un astérisque suivi directement par un slash (`*/`). Tout ce qui est contenu entre ces deux paires de signes sera ignoré par l'interpréteur JavaScript. Par exemple:

```
/*
Ceci est un commentaire multilignes,
il sera ignoré par l'interpréteur
*/
let a = "ceci est une variable définie dans une déclaration";
```

Inclure des commentaires dans son code est essentiel pour assurer la qualité du code, favoriser la collaboration et simplifier les phases de debug. En apportant du contexte et des explications sur différentes parties du programme, les commentaires permettent de rendre le code compréhensible sur le long terme. De plus, commenter son code est considéré comme une pratique bénéfique.

Variables

Afin de bien comprendre la programmation, la première étape consiste à se rappeler l'algèbre. Si vous avez conservé quelques souvenirs de l'école, l'algèbre consiste à écrire les termes de la manière suivante.

```
3 + 5 = 8
```

Vous pouvez réaliser des calculs avec des chiffres avant d'introduire une inconnue, par exemple, x:

```
3 + x = 8
```

En déplaçant les termes qui entourent l'inconnue, vous pouvez déterminer la valeur de x:

```
x = 8 - 3
-> x = 5
```

Quand vous introduisez plus d'une inconnue, les termes deviennent plus flexibles - vous êtes en train d'utiliser des variables:

```
x + y = 8
```

En effet, vous pouvez faire varier les valeurs de x et de y tout en conservant la même égalité:

```
x = 4
y = 4
```

ou encore

```
x = 3
y = 5
```

L'approche est similaire pour les langages de programmation. En programmation, les variables sont des sortes de petites boîtes dont le contenu change. Les variables peuvent contenir toute sorte de valeur ou le résultat d'un calcul. Chaque variable possède un `nom` et une `valeur` séparés par un signe égal (=). Cependant, il est important de garder à l'esprit que le nom donné aux variables peut être conditionné par le langage dans lequel vous programmez. En effet, certains noms sont réservés et ne peuvent être utilisés pour nommer une variable.

Regardons comment ceci fonctionne en JavaScript. Le code suivant définit deux variables, calcule la somme des deux et assigne ce résultat au sein d'une troisième variable.

```
let x = 5;
let y = 6;
let result = x + y;
```

Il existe quelques lignes directrices impératives à respecter quand vous nommez des variables. Ce sont les suivantes:

- Le nom d'une variable ne peut commencer que par une lettre, un underscore (_) ou un signe dollar ($).
- Après le premier caractère, il est possible d'utiliser des lettres, des chiffres, des underscores ou des signes dollar.

- JavaScript fait la distinction entre les caractères en majuscules des caractères en minuscules. On dit que JavaScript est sensible à la casse. De ce fait, maVariable, MaVariable et MAVARIABLE sont trois variables distinctes.
- Pour rendre votre code facile à lire et à comprendre, il est recommandé d'utiliser des noms de variables qui possèdent un sens descriptif et qui reflète leur utilisation.

Exercise

Définir une variable `x` égale à 20.

```
let x =
```

ES6 Version

ECMAScript 2015 ou ES2015 aussi appelé ES6 est une mise à jour significative du langage JavaScript ayant eu lieu en 2009. En ES6, il existe trois manière de déclarer des variables.

```
var x = 5;
const y = "Test";
let z = true;
```

Les types de déclarations influent sur la portée de la variable. À la différence du mot clé `var`, qui définit une variable globale à laquelle il est possible d'accéder depuis n'importe où, `let` limite l'utilisation de la variable au bloc dans laquelle elle est déclarée. Par exemple:

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2; // on intervient sur la même variable
    console.log(x); //2
  }
  console.log(x); //2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;
    console.log(x); // 2
  }
  console.log(x); // 1
}
```

Les variables `const` sont dites immutables, ce qui signifie qu'on ne peut plus changer leur valeur après leur déclaration. On les appelle aussi les constantes.

```
const x = "hi!";
x = "bye"; // ceci provoquera une erreur puisqu'on tente d'affecter une nouvelle valeur à la variable
```

Types

Les ordinateurs sont sophistiqués et peuvent utiliser une ou plusieurs variables plus complexes qu'un simple nombre. C'est ainsi que les types entrent en jeu. Les variables se décomposent en différentes sortes. Les types supportés sont propres à chaque langages de programmation.

Les types les plus courants sont les suivants:

- **Nombres**: Les nombres peuvent être des entiers (par ex., `1`, `-5`, `100`) ou des flottants décimaux (par ex., `3.14`, `-2.5`, `0.01`). JavaScript ne considère pas deux types séparés pour différencier les entiers des décimaux; il les considère toujours comme des nombres.
- **Chaînes de caractères**: Les chaînes de caractères sont des séquences de caractères. On les appelle communément `strings`. Elles peuvent être représentées aussi bien grâce à des apostrophes simples (par ex., `'hello'`) ou des apostrophes doubles (par ex., `"world"`).
- **Booléens**: Les booléens représentent une valeur vraie ou fausse. Ils prennent uniquement les valeurs `true` ou `false` (sans apostrophes).
- **Null**: Le type null représente une valeur null, ce qui signifie "aucune valeur." Il s'écrit `null` (sans apostrophes).
- **Undefined**: Le type indéfini, ou `undefined`, représente une valeur qui n'a pas été définie. Si une variable a été déclarée, mais n'a pas reçu de valeur (on dit qu'elle n'a pas été assignée), elle sera du type `undefined`.
- **Objet**: Un objet peut être vu comme un ensemble de propriétés dont chacune possède un nom et une valeur. Vous pouvez créer un objet en utilisant les accolades ouvrantes et fermantes (`{}`) et en lui assignant chaque propriété au moyen d'une paire nom-valeur pour chacune d'entre elles.
- **Tableau**: Un tableau est un type spécial d'objet qui est fait pour stocker des valeurs. Vous pouvez créer des tableaux en utilisant les crochets (`[]`) et en lui assignant une liste de valeurs.
- **Fonction**: Une fonction est un bloc de code qui peut être utilisé au moyen d'un appel. Les fonctions acceptent des arguments (des entrées) et renvoient une valeur (sortie). Vous pouvez créer une fonction en utilisant le mot clé `function`.

JavaScript est un langage "*faiblement typé*", cela signifie que vous n'avez pas l'obligation d'indiquer le type de variable que vous allez déclarer. En utilisant simplement le mot clé `var` quand vous déclarez une variable, l'interpréteur se chargera de déterminer le type de donnée en fonction des conventions de déclaration.

Exercise

Déclarez trois variables et initialisez-les avec les valeurs suivantes: `age` doit être un nombre, `name` doit être une chaîne de caractères et `isMarried` un booléen.

```
let age =
let name =
let isMarried =
```

L'opérateur `typeof` est utilisé pour déterminer le type d'une variable.

```
typeof "John"; // Retourne "string"
typeof 3.14; // Retourne "number"
typeof NaN; // Retourne "number"
typeof false; // Retourne "boolean"
typeof [1, 2, 3, 4]; // Retourne "object"
typeof { name: "John", age: 34 }; // Retourne "object"
typeof new Date(); // Retourne "object"
typeof function () {}; // Retourne "function"
typeof myCar; // Retourne "undefined" *
typeof null; // Retourne "object
```

En se basant sur les valeurs qu'ils peuvent contenir, les types de données utilisés en JavaScript peuvent être classés en deux catégories distinctes.

Les types de données qui peuvent contenir des valeurs:

- `string`
- `number`
- `boolean`
- `object`
- `function`

`Object` , `Date` , `Array` , `String` , `Number` , et `Boolean` sont des types d'objets disponibles nativement en JavaScript.

Les types de données qui ne peuvent pas contenir de valeurs:

- `null`
- `undefined`

Une donnée primitive est une donnée simple sans propriété ni méthode, qui n'est pas un objet. Elles sont immutables, elles ne peuvent donc pas être altérées. Il existe 7 données primitives en JavaScript:

- string
- number
- bigint
- boolean
- undefined
- symbol
- null

Exercise

Déclarez une variable `person` et initialisez-là comme un objet avec les propriétés suivantes: `age` un nombre, `name` une chaîne de caractère et `isMarried` un booléen.

```
let person =
```

Égalité

Lorsque nous écrivons un programme, nous avons souvent besoin de déterminer l'égalité entre différentes variables. Ce test de comparaison est réalisé grâce à un opérateur d'égalité. L'opérateur d'égalité le plus courant est le double égal `==`. Cet opérateur permet de déterminer si deux variables sont égales l'une à l'autre, ce même si elles ne sont pas du même type.

Par exemple, considérons:

```
let foo = 42;
let bar = 42;
let baz = "42";
let qux = "life";
```

Comme on peut s'y attendre, `foo == bar` sera évalué à `true` tandis que `baz == qux` sera évalué `false`. Cependant, `foo == baz` sera également évalué à `true` en dépit du fait que `foo` et `baz` sont de types différents. En coulisses, l'opérateur d'égalité `==` essaye de forcer les opérandes à être de même type pour pouvoir effectuer la comparaison. C'est la différence significative qui existe avec l'opérateur d'égalité `===`.

L'opérateur d'égalité triple égal `===` évalue l'égalité en se basant à la fois sur la valeur des variables *et* sur leur type. Si l'on reprend l'exemple précédent, cela revient à dire que `foo === bar` sera toujours évalué à `true`, mais que `foo === baz` sera à présent évalué `false`. En effet, bien que les valeurs semblent être égales, le type des variables diffère, l'égalité n'est pas retenue par l'opérateur triple égal. Quant à lui, `baz === qux` sera toujours `false`.

Exercise

Utilisez les opérateurs d'égalité `==` et `===` pour comparer les valeurs de `str1` et de `str2`.

```
let str1 = "hello";
let str2 = "HELLO";
let bool1 = true;
let bool2 = 1;
// comparez en utilisant ==
let stringResult1 =
let boolResult1 =
// comparez en utilisant ===
let stringResult1 =
let boolResult2 =
```

Chapitre 3

Nombres

JavaScript n'admet qu'**un seul type de nombre** – les flottants stockés sur 64 bits. Pour ceux et celles qui connaissent Java, il s'agit des nombres `double`. À la différence de la plupart des autres langages de programmation, JavaScript ne différencie pas les nombres en mettant à disposition un type spécial pour les entiers par exemple. Aussi, en JavaScript, 1 et 1.0 représentent exactement la même valeur (attention, on utilise le point numérique `.` pour indiquer un décimal). Créer un nombre en JavaScript est simplissime, il suffit de rentrer une valeur numérique à la suite du mot clé `var`, du nom de sa variable et de l'opérateur d'affectation `=`.

Les nombres peuvent être créés à partir de valeurs définies:

```
// Ceci est un nombre décimal:
let a = 1.2;

// Ceci est un nombre entier:
let b = 10;
```

Par ailleurs, il est possible d'affecter un nombre par l'intermédiaire d'une autre variable:

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

La précision des entiers est le plus souvent déterminée sur 15 chiffres et peut monter à 17 chiffres au maximum.

```
let x = 999999999999999; // x sera 999999999999999
let y = 9999999999999999; // y sera 10000000000000000
```

Les constantes numériques peuvent être passées en hexadécimal si elles sont préfixées par `0x`.

```
let z = 0xff; // 255
```

Math

L'objet `Math` permet d'effectuer des opérations mathématiques en JavaScript. C'est un objet statique qui ne possède pas de constructeur. Cela signifie qu'il est possible d'utiliser les méthodes de l'objet Math sans l'instancier au préalable. Pour accéder à ses propriétés on utilisera *Math.propriété*. Dans cette notation, le point numérique `.`, est appelé un accesseur de propriété. Quelques propriétés de l'objet Math sont définies ci-après:

```
Math.E; // retourne le nombre d'Euler
Math.PI; // retourne le nombre PI
Math.SQRT2; // retourne la racine carrée de 2
Math.SQRT1_2; // retourne la racine carrée de 1/2
Math.LN2; // retourne le logarithme naturel de 2
Math.LN10; // retourne le logarithme naturel de 10
Math.LOG2E; // retourne la valeur du logarithme en base 2 de E
Math.LOG10E; // retourne la valeur du logarithme en base 10 de E
```

Examples of some of the math methods are:

```
Math.pow(8, 2); // 64
Math.round(4.6); // 5
Math.ceil(4.9); // 5
Math.floor(4.9); // 4
Math.trunc(4.9); // 4
Math.sign(-4); // -1
Math.sqrt(64); // 8
Math.abs(-4.7); // 4.7
Math.sin((90 * Math.PI) / 180); // 1 (le sinus de 90 degrés)
Math.cos((0 * Math.PI) / 180); // 1 (le cosinus de 0 degré)
Math.min(0, 150, 30, 20, -8, -200); // -200
Math.max(0, 150, 30, 20, -8, -200); // 150
Math.random(); // 0.44763808380924375
Math.log(2); // 0.6931471805599453
Math.log2(8); // 3
Math.log10(1000); // 3
```

Pour accéder aux méthodes de Math, on appelle celles-ci directement en précisant les arguments attendus si ces derniers sont nécessaires.

Méthode	Description
`abs(x)`	Retourne la valeur absolue de `x`
`acos(x)`	Retourne l'arc cosinus de `x` , en radians
`acosh(x)`	Retourne l'arc cosinus hyperbolique de `x`
`asin(x)`	Retourne l'arc sinus de `x` , en radians
`asinh(x)`	Retourne l'arc sinus hyperbolique de `x`
`atan(x)`	Retourne l'arc tangente de `x` comme une valeur numérique comprise entre `-PI/2` et `PI/2` radians
`atan2(y,x)`	Retourne l'arc tangente du quotient de ses arguments
`atanh(x)`	Retourne l'arc tangent hyperbolique de `x`
`crbt(x)`	Retourne la racine cubique de `x`
`ceil(x)`	Retourne l'arrondi vers le haut à l'entier le plus proche de `x`
`cos(x)`	Retourne le cosinus de `x` , en radians
`cosh(x)`	Retourne le cosinus hyperbolique de `x`
`exp(x)`	Retourne la valeur exponentielle de `x`
`floor(x)`	Retourne l'arrondi vers le bas à l'entier le plus proche de `x`
`log(x)`	Retourne le logarithme naturel de `x`
`max(x,y,z,...n)`	Retourne nombre avec la valeur la plus haute
`min(x,y,z,...n)`	Retourne nombre avec la valeur la plus basse
`pow(x,y)`	Retourne la valeur de `x` élevé à la puissance `y`
`random()`	Retourne un nombre pseudo-aléatoire compris entre 0 et 1
`round(x)`	Retourne la valeur de x arrondi à l'entier le plus proche
`sign(x)`	Retourne si x est négatif, `null` ou positif (-1,0,1)
`sin(x)`	Retourne le sinus de x, en radians
`sinh(x)`	Retourne le sinus hyperbolique de x
`sqrt(x)`	Retourne la racine carrée de x
`tan(x)`	Retourne la tangente de x, en radians
`tanh(x)`	Retourne la tangente hyperbolique de x
`trunc(x)`	Retourne la partie entière d'un nombre (x)

Opérateurs de base

Les opérations mathématiques sur les nombres peuvent être réalisées en utilisant certains opérateurs de base comme:

- **Opérateur d'addition (`+`)**: L'opérateur d'addition ajoute deux nombres l'un à l'autre. Par exemple:

```
console.log(1 + 2); // 3
console.log(1 + -2); // -1
```

- **Opérateur de soustraction (`-`)**: L'opérateur de soustraction soustrait deux nombres l'un à l'autre. Par exemple:

```
console.log(3 - 2); // 1
console.log(3 - -2); // 5
```

- **Opérateur de multiplication (`*`)**: L'opérateur de multiplication multiplie deux nombres l'un à l'autre. Par exemple:

```
console.log(2 * 3); // 6
console.log(2 * -3); // -6
```

- **Opérateur de division (`/`)**: L'opérateur de division divise deux nombres l'un à l'autre et retourne la partie entière (le quotient) de la division euclidienne. Par exemple:

```
console.log(6 / 2); // 3
console.log(6 / -2); // -3
```

- **Modulo (`%`)**: Le modulo retourne le reste d'une division euclidienne. Par exemple:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

L'interpréteur JavaScript fonctionne de la gauche vers la droite. Il est donc impératif d'utiliser les parenthèses pour séparer et regrouper les expressions: `c = (a / b) + d`

> En JavaScript on utilise aussi bien l'opérateur d'addition `+` pour réaliser des calculs que des concaténations. L'usage est implicite en fonction du type: les nombres seront additionnés, alors que les chaînes de caractères seront concaténées.

Le terme `NaN` est un mot clé réservé par le langage qui indique qu'un nombre n'est pas "légal". Ceci arrive quand on tente d'effectuer une opération arithmétique quand les opérandes ne sont pas tous du type numérique. Le résultat renvoyé sera `NaN` , ce qui signifie Not a Number.

```
let x = 100 / "10";
```

La fonction `parseInt` analyse une valeur passée en paramètre et renvoie le premier entier rencontré exprimé dans une base décimale donnée. Attention, hormis si on indique explicitement la base en second paramètre pour convertir les nombres représentés par des chaînes de caractères, `parseInt` renverra `NaN` quand elle ne pourra pas convertir les caractères en entier.

```
parseInt("10"); // 10
parseInt("10.00"); // 10
parseInt("10.33"); // 10
parseInt("34 45 66"); // 34
parseInt(" 60 "); // 60
parseInt("ff", 16); // 255
parseInt("40 years"); //40
parseInt("He was 40"); //NaN
```

En JavaScript, si réalise une opération qui retourne un nombre plus grand que les capacités du langage, celui-ci retournera `Infinity`.

```
let x = 2 / 0; // Infinity
let y = -2 / 0; // -Infinity
```

Exercise

Utilisez les opérateurs mathématiques +, -, *, /, et % pour réaliser les opérations suivantes entre `num1` et `num2`.

```
let num1 = 10;
let num2 = 5;

// Ajouter num1 et num2.
let addResult =
// Soustraire num2 à num1.
let subtractResult =
// Multiplier num1 et num2.
let multiplyResult =
// Diviser num1 par num2.
let divideResult =
// Trouvez le reste de la division de num1 par num2.
let reminderResult =
```

Opérateurs avancés

Quand les opérateurs se succèdent sans parenthèses, l'ordre dans lequel il s'appliquent est déterminé par la *précédence* des opérateurs. Ainsi, la multiplication `(*)` et la division `(/)` ont une précédence supérieure que l'addition `(+)` et la soustraction `(-)`.

```
// la multiplication est réalisée en premier lieu, suivie par l'addition
let x = 100 + 50 * 3; // 250
// en présence de parenthèses, c'est l'opération contenue entre ces dernières qui est exécutée en premier lieu
let y = (100 + 50) * 3; // 450
// des opérations ayant une précédence équivalente sont exécutées de gauche à droite
let z = (100 / 50) * 3; // 6
```

Quelques opérateurs mathématiques avancés peuvent être utilisés lors de l'écriture d'un programme. Voici une liste de certains des principaux opérateurs mathématiques avancés:

- **Modulo (`%`)**: Le modulo operator retourne le reste d'une division euclidienne. Par exemple:

```
console.log(10 % 3); // 1
console.log(11 % 3); // 2
console.log(12 % 3); // 0
```

- **Opérateur puissance (* *)**: L'opérateur puissance permet de calculer la valeur d'un nombre élevé à la puissance du second. Il s'agit d'un nouvel opérateur et il se peut qu'il ne soit pas supporté par tous les navigateurs, vous pourriez avoir besoin de le substituer par la méthode `Math.pow`, qui produit exactement le même effet. Par exemple:

```
console.log(2 ** 3); // 8
console.log(3 ** 2); // 9
console.log(4 ** 3); // 64
```

- **Opérateur d'incrémentation (`++`)**: L'opérateur d'incrémentation augmente un nombre d'une quantité de 1. On peut l'utiliser avant la variable (comme un préfixe, dans ce-cas là l'incrémentation sera faite en premier lieu) ou comme un suffixe.

```
let x = 1;
x++; // x vaut à présent 2
++x; // x vaut à présent 3
```

- **Opérateur de décrémentation (`--`)**: L'opérateur de décrémentation diminue un nombre d'une quantité de 1. On peut l'utiliser avant la variable (comme un préfixe, dans ce cas-là la décrémentation sera faite en premier lieu) ou comme un suffixe. Par exemple:

```
let y = 3;
y--; // y vaut à présent 2
--y; // y vaut à présent 1
```

- **Objet math**: L'objet `Math` est un objet natif en JavaScript. Celui-ci fournit des fonctions mathématiques et des constantes. Vous pouvez utiliser les méthodes de l'objet `Math` pour réaliser des opérations avancées, comme trouver la racine carrée d'un nombre, calculer le sinus d'un angle ou encore générer un nombre pseudo-aléatoire. Par exemple:

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sin(0)); // 0
console.log(Math.random()); // un nombre pseudo-aléatoire compris entre 0 et 1
```

Il s'agit là de quelques exemples succincts des opérateurs mathématiques avancés et des fonctions disponibles en JavaScript. En réalité, il en existe bien plus et ceux-ci peuvent être utilisés pour réaliser des opérations avancées lors de l'écriture d'un programme.

Exercise

Utilisez les opérateurs avancés suivants pour réaliser les opérations sur `num1` et `num2`.

```
let num1 = 10;
let num2 = 5;

// ++ opérateur d'incrémentation pour incrémenter la valeur de num1.
const result1 =
// -- opérateur de décrémentation pour décrémenter la valeur de num2.
const result2 =
// += opérateur pour additionner num2 à num1 et stocker le résultat dans num1.
const result3 =
// -= opérateur pour soustraire num2 à num1 et stocker le résultat dans num1.
const result4 =
```

Opérateur de coalescence des nuls '??'

L'opérateur de coalescence des `nuls` retourne l'opérande de gauche si celle-ci ne vaut pas `null/undefined`, sinon il renvoie l'opérande de droite. Il s'écrit avec deux points d'interrogation `??`. Le résultat de `x ?? y` est:

- `x` si `x` est défini
- `y` si `x` n'est pas défini.

L'opérateur de coalescence des nuls est une implémentation récente dans le langage, il doit être substitué par un polyfill pour assurer sa compatibilité avec les anciens navigateurs.

Chapitre 4

Chaînes de caractères

Les chaînes de caractères en JavaScript partagent beaucoup de similitudes avec les implémentations d'autres langages de haut niveau. Celles-ci représentent des données basées sur le texte et des données textuelles. Dans ce chapitre, nous découvrirons les bases des chaînes de caractères. Par exemple, nous verrons comment créer de nouvelles chaînes et réaliser des opérations sur celles-ci.

Voici un exemple de chaîne de caractères:

```
"Hello World"
```

Les chaînes de caractères sont indexées sur une base 0. Cela signifie que la position du premier caractère est située à l'index `0` et les autres suivent dans un ordre incrémental. Différentes méthodes sont disponibles pour les chaînes de caractères et retournent une nouvelle valeur. Ces méthodes sont décrites ci-dessous.

Nom	Description
`charAt()`	Renvoie un caractère à l'index spécifié en paramètre
`charCodeAt()`	Renvoie le caractère Unicode à l'index spécifié en paramètre
`concat()`	Renvoie deux chaînes ou plus passées en paramètres combinées
`constructor`	Renvoie le constructeur d'une chaîne de caractères
`endsWith()`	Vérifie si une chaîne de caractères se termine par une valeur spécifiée en paramètre
`fromCharCode()`	Renvoie une chaîne de caractères créée à partir de valeurs Unicode passées en paramètres
`includes()`	Vérifie si une chaîne de caractères contient une valeur spécifiée en paramètre
`indexOf()`	Renvoie l'index de la première occurrence de la valeur en paramètre
`lastIndexOf()`	Renvoie l'index de la dernière occurrence de la valeur en paramètre
`length`	Renvoie la longueur d'une chaîne de caractères
`localeCompare()`	Compare deux chaînes de caractères en incorporant les règles lexicographiques de la locale
`match()`	Renvoie le tableau de correspondance entre une chaîne de caractères et une expression rationnelle passée en paramètres
`prototype`	Utilisé pour ajouter des propriétés et des méthodes à un objet
`repeat()`	Renvoie une nouvelle chaîne de caractères d'un nombre de copies spécifié en paramètre
`replace()`	Renvoie une nouvelle chaîne dans laquelle tout ou partie des valeurs sont remplacées par un modèle défini
`search()`	Exécute une recherche dans une chaîne de caractères grâce à une expression rationnelle et renvoie le premier index correspondant
`slice()`	Renvoie une chaîne de caractères contenant une partie de la chaîne initiale
`split()`	Divise une chaîne de caractères en un tableau de sous-chaînes
`startsWith()`	Vérifie si une chaîne de caractères commence par le caractère spécifié
`substr()`	Extrait une partie de chaîne, depuis un index de départ
`substring()`	Extrait une partie de chaîne, entre deux index spécifiés en paramètres
`toLocalLowerCase()`	Renvoie une chaîne avec des caractères en minuscule en utilisant la local lexicographique de l'utilisateur
`toLocalUpperCase()`	Renvoie une chaîne avec les caractères en majuscule en utilisant la local lexicographique de l'utilisateur
`toLowerCase()`	Renvoie une chaîne avec des caractères en minuscule
`toString()`	Renvoie une chaîne ou un objet chaîne de caractère sous forme de chaîne
`toUpperCase()`	Renvoie une chaîne avec des caractères en minuscule
`trim()`	Renvoie une chaîne de caractères dépourvue des espaces
`trimEnd()`	Renvoie une chaîne de caractères dépourvue des espaces à la fin
`trimStart()`	Renvoie une chaîne de caractères dépourvue des espaces au début
`valueOf()`	Renvoie la valeur primitive d'une chaîne ou d'un objet chaîne de caractères

Création de chaînes

Les chaînes de caractères peuvent être définies par du texte entouré par une paire d'apostrophes simples ou doubles:

```
// On peut tout aussi bien utiliser les apostrophes simples...
let str = "Our lovely string";

// ...Que les apostrophes doubles
let otherStr = "Another nice string";
```

En JavaScript, les chaînes de caractères peuvent contenir des caractères codés en UTF-8

```
"中文 español English हिन्दी العربية português বাংলা русский 日本語 ਪੰਜਾਬੀ 한국어";
```

Vous pouvez également utiliser le constructeur `String` pour créer une instance de l'objet chaîne de caractère:

```
const stringObject = new String("Je suis une chaîne");
```

Cependant, il n'est généralament pas recommandé d'utiliser le constructeur `String` pour créer des chaînes de caractères. En effet, cela peut être source de confusion entre le type primitif string et les objets chaînes de caractères. Il est davantage conseillé d'utiliser les It is usually better to use chaînes littérales pour créer des chaînes de caractères.

Vous avez également la possibilité d'utiliser d You can also use littéraux de gabarits pour créer des chaînes. Les littéraux de gabarits sont simplement des chaînes qui sont entourées par des backticks `(``)` et qui contiennent des espaces réservés pour recevoir les valeurs. Les espaces réservés sont déclarés avec la syntaxe suivante `` `${}` ``. Entre les crochets, on indiquera le nom de la variable à substituer. On appelle ce mécanisme l'interpolation de chaîne.

```
const name = "John";
const message = `Hello, ${name}!`;
```

Les littéraux de gabarit peuvent également contenir plusieurs lignes et inclure des expressions au sein des espaces réservés.

> Les chaînes de caractères ne peuvent être soustraites, multipliées ou divisées.

Exercise

Utilisez les littéraux de gabarit pour créer une chaîne qui inclus les valeurs de `name` et `age`. La chaîne finale doit ressembler à cela: "Mon nom est John et j'ai 25 ans."

```
let name = "John";
let age = 25;

// Mon nom est John et j'ai 25 ans.
let result =
```

Remplacement

La méthode `replace` nous permet de remplacer un caractère, un mot, ou une phrase entière dans une chaîne. Par exemple:

```
let str = "Hello World!";
let new_str = str.replace("Hello", "Hi");

console.log(new_str);

// Resultat: Hi World!
```

> Pour remplacer une valeur sur l'ensemble des occurrences d'une expression régulière on devra indiquer un drapeau `g` .

La méthode `replace` recherche une valeur ou la présence d'une expression rationnelle dans une chaîne et retourne une nouvelle chaîne avec les valeurs remplacées. Cette opération ne modifie pas la chaîne originelle. Voyons le cas d'un remplacement global non sensible à la casse. Ici on recherche les occurrences de l'expression rationnelle `/blue/` en lui précisant les drapeaux `g` (global) et `i` (non sensible à la casse). On remplace les occurrences trouvées par la chaîne en second paramètre (ici "red").

```
let text = "Mr Blue has a blue house and a blue car";
let result = text.replace(/blue/gi, "red");

console.log(result);
//Result: Mr red has a red house and a red car
```

Longueur des chaînes

En JavaScript, c'est un jeu d'enfant de savoir combien de caractères composent une chaîne. Pour ce faire, on utilise la propriété `.length` sur la chaîne en question. La propriété `length` renvoie le nombre de caractères contenus dans la chaîne, en incluant les espaces et les caractères spéciaux non alphanumériques.

```
let size = "Notre chaîne merveilleuse".length;
console.log(size);
// size: 25

let emptyStringSize = "".length;
console.log(emptyStringSize);
// emptyStringSize: 0
```

La propriété `length` d'une chaîne vide est `0`.

> La propriété `length` est en lecture seule, cela signifie qu'il est impossible de lui assigner une nouvelle valeur.

Concaténation

La concaténation signifie l'addition de deux ou plusieurs chaînes l'une à l'autre. Le résultat de la concaténation crée une chaîne plus grande contenant les données combinées des deux chaînes originales. La concaténation d'une chaîne ajoute celle-ci à une ou plusieurs autres. Voici comment on réalise cette opération en JavaScript.

- en utilisant l'opérateur `+`
- en utilisant la méthode `concat()`
- en utilisant la méthode de tableau `join()`
- en utilisant les littéraux de gabarits (depuis ES6)

La méthode de chaîne `concat()` accepte une liste de chaînes en paramètres et retourne une nouvelle chaîne, résultat de la concaténation. En substance, le résultat correspond à la combinaison de toutes les chaînes. Alors que la The string `concat()` method accepts the list of strings as parameters and returns a new string after concatenation i.e. combination of all the strings. La méthode de tableau `join()` est quant à elle utilisée pour concaténer tous les éléments présents dans un tableau en les convertissant en une seule et unique chaîne.

Les littéraux de gabarits utilisent les backticks `(``)` et procurent un moyen simple de créer des chaînes sur plusieurs lignes. De la même manière, c'est grâce à eux que l'on peut réaliser l'interpolation de chaîne. En effet, on peut utiliser une expression à l'intérieur de la syntaxe `${}` pour la remplacer par la valeur de l'expression `${expression}` .

```
const icon = "👋";
// en utilisant les littéraux de gabarits
`hi ${icon}`;

// en utilisant la méthode join()
["hi", icon].join(" ");

// en utilisant la méthode concat()
"".concat("hi ", icon);

//  en utilisant l'opérateur +
"hi " + icon;
// hi 👋
```

Chapitre 5

Logique conditionnelle

Une condition consiste en un test. Les conditions sont un point clé de tous les langages de programmation, de plusieurs manières:

Avant tout, les conditions peuvent être utilisées pour s'assurer de la fonctionnalité d'un programme, indépendamment du type de données qui sont lui sont passée. Si vous faites confiance aveuglément aux données et que vous ne testez pas ces derniers, vos programmes échoueront à coup sûr. À l'inverse, si vous mettez en place des conditions qui vous permettent de tester la donnée, votre programme gagnera en prédictivité et en stabilité. Prendre de telles dispositions à l'égard de sa conception s'appelle programmer en défense.

L'autre aspect incontournable des conditions est qu'elles nous permettent de mettre en place un code "en arbre". Vous avez certainement rencontré ce concept sans le savoir, par exemple en remplissant un formulaire. Basiquement, cela renvoie au fait que le code va s'exécuter selon différentes "branches" (ou parties) qui dépendent de la validité ou non d'une condition.

Dans ce chapitre, nous apprendrons les bases de la logique conditionnelle en JavaScript.

If

La condition la plus basique est la déclaration if et sa syntaxe est la suivante `if(condition){ do this … }`. La condition doit être évaluée à true pour que le code contenu entre crochets soit exécuté. Par exemple, vous pourriez vouloir tester la valeur d'une chaîne.

```
let country = "France";
let weather;
let food;
let currency;

if (country === "Angleterre") {
  weather = "horrible";
  food = "passable";
  currency = "livre sterling";
}

if (country === "France") {
  weather = "agréable";
  food = "formidable, mais peu adaptée aux végétariens";
  currency = "marrante, petite et colorée";
}

if (country === "Allemagne") {
  weather = "correct";
  food = "vraiment pas bon";
  currency = "marrante, petite et colorée";
}

let message =
  "Il s'agit de la " +
  country +
  ", le temps est " +
  weather +
  ", la nourriture est " +
  food +
  " et la " +
  "monnaie est " +
  currency;

console.log(message);
// 'Il s'agit de la France, le temps est agréable, la nourriture est formidable, mais peu adaptée aux végétarie
```

Les conditions peuvent également être imbriquées les unes dans les autres.

Else

De la même manière qu'il existe un `if`, il existe une condition appelée sinon `else`. Celle-ci s'applique quand la condition dans le if est évaluée à false. Cette combinaison si/sinon donne tout son sens aux conditions et est très puissante. En effet, elle permet de définir la manière de traiter n'importe quelle valeur, dans un cas ou dans l'autre.

```
let umbrellaMandatory;

if (country === "Angleterre") {
  umbrellaMandatory = true;
} else {
  umbrellaMandatory = false;
}
```

L'instruction `else` peut également être assortie d'un autre `if`, on parle dans ce cas d'un sinon si. Chaque condition va être testée à la suite l'une de l'autre. Dès lors que l'une d'entre elles sera évaluée à true, l'exécution du code se pousuivra dans le bloc de code correspondant et il en sera fini pour ce bloc. Dans ce cas, les conditions placées en dessous ne seront pas évaluées. Réécrivons l'exemple précédent en suivant cette logique du si/sinon si/sinon.

```
if (country === "Angleterre") {
  ...
} else if (country === "France") {
  ...
} else if (country === "Allemagne") {
  ...
}
```

Exercise

Avec les valeurs suivantes, écrivez une structure conditionnnelle qui vérifie si `num1` est plus grand que `num2`. Si c'est le cas, affectez "num1 est plus grand que num2" dans la variable `result`. Dans le cas contraire, la variable `result` devra être égale à "num1 est plus petit ou égal à num2"

```
let num1 = 10;
let num2 = 5;
let result;

// vérifie si num1 est plus grand que num2
if( condition ) {

}else {

}
```

Switch

Un bloc de condition de type `switch` permet de réaliser des opérations basées sur différentes conditions. Il utilise la comparaison stricte (`===`) pour déterminer si une condition est remplie ou non. Comme dans un bloc if/else if/else, le code placé après la condition évaluée à true est exécutée. La syntaxe du `switch` est écrite ci-dessous. Notez qu'à la différence du if/else, le code qui suit la condition n'est pas entouré par une paire de crochets.

```
switch (expression) {
  case x:
    // bloc d'instructions
    break;
  case y:
    // bloc d'instructions
    break;
  default:
  // bloc d'instructions
}
```

L'expression est évaluée une première fois avant d'être comparée à chaque case (cas). Si une condition est true, le code associé est exécuté. Dans le cas contraire, si chaque case renvoie false, c'est le bloc `default` qui est exécuté. Par ailleurs, le mot clé `break` permet d'indiquer à l'interpréteur que le bloc de code à exécuter est terminé et qu'il faut sortir du `switch` . Si le mot clé `break` est absent, l'exécution du `switch` se poursuivra à la fin du bloc d'instructions. Ce comportement est inutile est préjudiciable en termes de mémoire.

Voici l'exemple d'un code qui reçoit un jour de la semaine et qui utilise un switch pour renvoyer son nom.

```
switch (new Date().getDay()) {
  case 0:
    day = "Dimanche";
    break;
  case 1:
    day = "Lundi";
    break;
  case 2:
    day = "Mardi";
    break;
  case 3:
    day = "Mercredi";
    break;
  case 4:
    day = "Jeudi";
    break;
  case 5:
    day = "Vendredi";
    break;
  case 6:
    day = "Dimanche";
}
```

Dans le cas où plusieurs cas peuvent remplir la conditions, seul le **premier** cas true rencontré est exécuté. Dans le cas où aucun cas n'est évalué à true alors le bloc default est exécuté. En l'absence de bloc default et si aucun cas n'est évalué à true, le programme sort du switch sans aucun effet et poursuit son exécution.

Exercise

Avec les valeurs suivantes, écrivez un bloc `switch` qui vérifie la valeur d'un jour de la semaine stockée dans une variable dayOfWeek. Si dayOfWeek est égal à "Lundi", "Mardi", "Mercredi", "Jeudi", ou "Vendredi",

assignez "C'est un jour de la semaine" à la variable result. Si `dayOfWeek` est égal à "Samedi" ou "Dimanche", affectez "C'est un jour de week-end" à la variable résultat.

```
let dayOfWeek = "Lundi";
let result;
// Vérifie s'il s'agit d'un jour de la semaine or de week-end
switch(expression) {
case x:
// bloc d'instructions
break;
case y:
// bloc d'instructions
break;
default:
// bloc d'instructions
}
```

Comparaison

Concentrons nous sur la partie conditionnelle:

```
if (country === "France") {
    ...
}
```

La partie conditionnelle (ou la condition) est représentée par la variable `country` suivie par les trois signes égal (`===`). Cet opérateur d'égalité stricte teste si la variable `country` possède à la fois la valeur attendue (`France`) et le type requis (`String`). Bien sûr, il est possible de réaliser la condition suivante avec l'opérateur d'égalité à deux signes égal (`==`). Dans ce cas, on occultera le test du type de la variable. Ainsi, la condition `if (x == 5)` retournera true aussi bien pour `var x = 5;` que pour `var x = "5";` . Dans le premier cas x est du type `Number` et dans le second du type `String` . Vous en convenez, cela change pas mal de choses et cela peut créer des comportements indésirables. Pour prévenir ce genre de problème, il est recommandé de toujours tester l'égalité grâce à la comparaison stricte. Ainsi, on utilisera en priorité (`===` and `!==`) au lieu de (`==` and `!=`).

Autre test conditionnels fréquents:

- `x > a` : x est strictement supérieur à a?
- `x < a` : x est strictement inférieur à a?
- `x <= a` : x est inférieur ou égal à a?
- `x >=a` : x est supérieur ou égal à a?
- `x != a` : x est différent de a?
- `x` : est-ce que x existe?

Comparaison logique - ternaire

Dans le but de s'éviter l'écriture trop rébarbative des blocs if/else, la syntaxe des ternaires peut-être mise en oeuvre. En substance, il s'agit d'écrire un if/else simple de manière rapide.

```
let topper = marks > 85 ? "YES" : "NO";
```

Dans l'exemple ci-dessus, `?` est un opérateur logique. La condition à tester est l'opérande gauche (`marks > 85`). Si cette condition est remplie, la variable topper contiendra la chaîne "YES" et dans le cas contraire, elle se verra assigner la valeur "NO". En opérande droite, on fait apparaître les instructions à appliquer en fonction de l'évaluation de la condition. Les deux cas sont séparés par les `:` . Si la condition est évaluée à true, on accède à la partie gauche, à droite sinon.

Concaténation de conditions

De même, vous avez la possibilité d'associer les conditions entre elles grâce aux instructions " `or` " ou “ `and` ”. Cela permet de tester si toutes les conditions sont remplies, seulement l'une ou aucune d'entre elles.

En JavaScript "ou" s'écrit grâce au double pipe `||` quand "et" s'écrit avec le double esperluète `&&` .

Imaginons que vous vouliez tester si une valeur x est comprise entre 10 et 20. Voilà comment vous pourriez écrire cette condition globale en combinant deux sous-conditions:

```
if (x > 10 && x < 20) {
    ...
}
```

Si maintenant vous voulez vous assurer que le pays contenu dans la variable country est soit égal à "Angleterre" soit "Allemagne", vous utiliseriez: If you want to make sure that country is either “England” or “Germany” you use:

```
if (country === "Angleterre" || country === "Allemagne") {
    ...
}
```

> **Note**: À l'image des opérations sur les nombres, les conditions peuvent être priorisées en utilisant les parenthèses. Par exemple: `if ( (name === "John" || name === "Jennifer") && country === "France")` . Dans ce cas, on testera d'abord la condition relative au nom `name` avant d'évaluer la condition mettant en oeuvre le "et" logique.

Chapitre 6

Tableaux

Les tableaux sont une partie essentielle de la programmation. Un tableau n'est ni plus ni moins qu'une liste de données. Grâce à eux, on peut stocker de nombreuses données à l'intérieure d'une seule variable, ce qui rend notre code plus facile à lire et à comprendre. Les tableaux permettent aussi de réaliser de nombreuses actions sur les données qu'ils stockent.

Les données au sein d'un tableau s'appellent les **éléments**.

Voici un exemple simple de tableau:

```
// 1, 1, 2, 3, 5, et 8 sont les éléments de ce tableau
let numbers = [1, 1, 2, 3, 5, 8];
```

Les tableaux peuvent être aisément créés en utilisant la syntaxe littérale ou grâce au mot clé `new`.

```
const cars = ["Saab", "Volvo", "BMW"]; // en utilisant la syntaxe littérale entre crochets
const cars = new Array("Saab", "Volvo", "BMW"); // en utilisant le mot clé new
```

Un numéro d'index est utilisé pour accéder aux valeurs au sein d'un tableau. L'index du premier élément d'un tableau est toujours `0` comme le tableau commence toujours son indexation à `0`. Le numéro d'index peut aussi être utilisé pour mettre à jour les éléments au sein du tableau.

```
const cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]);
// Résultat: Saab

cars[0] = "Opel"; // changeons le premier élément du tableau
console.log(cars);
// Résultat: ['Opel', 'Volvo', 'BMW']
```

> Les tableaux sont un type spécial d'objets. On peut même stocker des objects dans un tableau.

La propriété `length` d'un tableau retourne le nombre d'éléments qui le constitue. Voici les méthodes supportées par les tableaux:

Nom	Description
`concat()`	Retourne deux ou plus tableaux concaténés
`join()`	Rassemble tous les éléments du tableau sous la forme d'une chaîne de caractères
`push()`	Ajoute un ou plusieurs éléments à la fin du tableau et renvoie la nouvelle longueur de celui-ci
`pop()`	Supprime le dernier élément d'un tableau et retourne l'élément supprimé
`shift()`	Supprime le premier élément d'un tableau et retourne l'élément supprimé
`unshift()`	Ajoute un ou plusieurs éléments au début d'un tableau et retourne la nouvelle longueur de celui-ci
`slice()`	Extrait la partie d'un tableau et retourne la partie extraite sous la forme d'un nouveau tableau
`at()`	Renvoie l'élément à l'index spécifié ou `undefined`
`splice()`	Supprime l'élément du tableau et (en option) remplace celui-ci avant de retourner le nouveau tableau
`reverse()`	Transpose les éléments d'un tableau en le retournant et renvoie le tableau modifié
`flat()`	Renvoie un nouveau tableau avec tous les éléments du sous-tableau concaténés de manière récursive jusqu'à la profondeur spécifiée
`sort()`	Trie les éléments d'un tableau et renvoie le tableau modifié
`indexOf()`	Renvoie l'index du premier match de l'élément recherché
`lastIndexOf()`	Renvoie l'index du dernier match de l'élément recherché
`forEach()`	Exécute une fonction de rappel sur chaque élément du tableau et renvoie undefined
`map()`	Returns a new array with a return value from executing `callback` on every array item.
`flatMap()`	Exécute un `map()` suivi d'un `flat()` de profondeur 1
`filter()`	Retourne un nouveau tableau contenant tous les éléments qui renvoient true à la fonction de `callback` passée
`find()`	Renvoie le premier élément qui renvoie true pour la `callback` passée
`findLast()`	Renvoie le dernier élément qui renvoie true pour la `callback` passée
`findIndex()`	Renvoie l'index du premier élément qui renvoie true pour la `callback` passée
`findLastIndex()`	Renvoie l'index du dernier élément qui renvoie true pour la `callback` passée
`every()`	Renvoie `true` si la `callback` renvoie `true` sur chaque élément du tableau
`some()`	Renvoie `true` si la `callback` renvoie `true` sur au moins 1 élément du tableau
`reduce()`	Utilise la `callback(accumulator, currentValue, currentIndex, array)` pour réduire le tableau et renvoie la valeur finale de la fonction `callback`
`reduceRight()`	Fonctionne de la même manière qu'un `reduce()` mais commence à partir du dernier élément

Unshift

La méthode `unshift` ajoute de nouveaux éléments séquentiellement au début du tableau. Il modifie le tableau d'origine et renvoie la nouvelle longueur du tableau. Par exemple.

```
let array = [0, 5, 10];
array.unshift(-5); // 4

// RESULT: array = [-5 , 0, 5, 10];
```

> La méthode `unshift()` écrase le tableau d'origine.

La méthode `unshift` prend un ou plusieurs arguments qui représentent les éléments à ajouter au début du tableau. Il ajoute les éléments dans l'ordre dans lequel ils sont fournis, le premier élément sera donc le premier élément du tableau.

Voici un exemple d'utilisation de `unshift` pour ajouter plusieurs éléments à un tableau :

```
const numbers = [1, 2, 3];
const newLength = numbers.unshift(-1, 0);
console.log(numbers); // [-1, 0, 1, 2, 3]
console.log(newLength); // 5
```

Map

La méthode `Array.prototype.map()` itère chaque élément d'un tableau et modifie l'élément courant en utilisant une fonction de rappel. Cette callback est appliquée sur chaque élément composant le tableau.

Voici la syntaxe pour utiliser `map` .

```
let newArray = oldArray.map(function (element, index, array) {
  // element: l'élément en cours de traitement dans le tableau
  // index: l'index de l'élément en cours de traitement dans le tableau
  // array: le tableau sur lequel la méthode est appelée
  // Retourne l'élément à ajouter à newArray
});
```

Par exemple, disons que vous disposez d'un tableau de nombres et que vous souhaitez créer un nouveau tableau qui double les valeurs des nombres du tableau d'origine. Vous pouvez le faire en utilisant `map` comme ceci.

```
const numbers = [2, 4, 6, 8];

const doubledNumbers = numbers.map((number) => number * 2);

console.log(doubledNumbers);

// Résultat: [4, 8, 12, 16]
```

Vous pouvez aussi utiliser une fonction fléchée pour définir la callback passée à `map` .

```
let doubledNumbers = numbers.map((number) => {
  return number * 2;
});
```

ou

```
let doubledNumbers = numbers.map((number) => number * 2);
```

> La méthode `map()` n'exécute pas la callback pour les éléments vides et ne modifie pas le tableau d'origine.

Opérateur de décomposition (spread)

Un tableau ou un objet peut être rapidement copié dans un autre tableau ou objet en utilisant l'opérateur de décomposition (spread) `(...)`. Il permet à un itérable tel qu'un tableau d'être développé aux endroits où zéro ou plusieurs arguments (pour les appels de fonction) ou éléments (pour les littéraux de tableau) sont attendus, ou à une expression d'objet d'être développée aux endroits où zéro ou plusieurs paires clé-valeur (pour les littéraux d'objet) sont attendus.

Voici certains exemples d'utilisation:

```
let arr = [1, 2, 3, 4, 5];

console.log(...arr);
// Résultat: 1 2 3 4 5

let arr1;
arr1 = [...arr]; //copie arr dans arr1

console.log(arr1); //Résultat: [1, 2, 3, 4, 5]

arr1 = [6, 7];
arr = [...arr, ...arr1];

console.log(arr); //Résultat: [1, 2, 3, 4, 5, 6, 7]
```

> L'opérateur de décomposition spread fonctionne uniquement avec les navigateurs modernes qui supportent la fonctionnalités. Si vous souhaitez l'utiliser avec d'anciens navigateurs, vous devrez utiliser un transpiler comme Babel pour convertir la syntaxe du spread dans son équivalent ES5.

Shift

La méthode `shift` supprime le premier élément d'index du tableau et déplace tous les index vers la gauche. Elle modifie le tableau d'origine. Voici la syntaxe de cette méthode

```
array.shift();
```

Par exemple:

```
let array = [1, 2, 3];
array.shift();

// Résultat: array = [2,3]
```

Vous pouvez également utiliser la méthode `shift` en combinaison avec une boucle pour supprimer tous les éléments d'un tableau. Voici un exemple de la façon dont vous pourriez procéder :

```
while (array.length > 0) {
  array.shift();
}

console.log(array); // Résultat: []
```

La méthode `shift` ne fonctionne que sur les tableaux, et pas sur d'autres objets similaires aux tableaux tels que les objets arguments ou les objets NodeList. Si vous devez déplacer des éléments de l'un de ces types d'objets, vous devrez d'abord le convertir en tableau à l'aide de la méthode `Array.prototype.slice()` .

Pop

La méthode `pop` supprime le dernier élément d'un tableau et renvoie l'élément supprimé. Cette méthode change la longueur du tableau.

Voici la syntaxe utilisée pour la méthode `pop` :

```
array.pop();
```

Par exemple:

```
let arr = ["one", "two", "three", "four", "five"];
arr.pop();

console.log(arr);

// Résultat: ['one', 'two', 'three', 'four']
```

Vous pouvez aussi utiliser la méthode `pop` en conjonction avec une boucle pour supprimer tous les éléments d'un tableau. Voici un exemple de comment vous pourriez faire cela:

```
while (array.length > 0) {
  array.pop();
}

console.log(array); // Résultat: []
```

> La méthode `pop` ne fonctionne que sur les tableaux, et non sur d'autres objets similaires aux tableaux tels que les objets arguments ou les objets NodeList. Si vous devez extraire des éléments de l'un de ces types d'objets, vous devrez d'abord le convertir en tableau à l'aide de la méthode `Array.prototype.slice()` .

Join

La méthode `join`, transforme un tableau en chaîne de caractère et le rassemble. La méthode ne modifie pas le tableau original. Voici la syntaxe de la méthode `join` :

```
array.join([separator]);
```

L'argument optionnel `separator` spécifie le caractère utilisé pour séparer les éléments dans la chaîne générée. S'il n'est pas précisé, les éléments seront séparés par une virgule (`,`).

Par exemple:

```
let array = ["one", "two", "three", "four"];

console.log(array.join(" "));

// Résultat: one two three four
```

> N'importe quel séparateur peut-être spécifié, mais la virgule est le séparateur par défaut `(,)` .

Dans l'exemple ci-dessus, un espace est utilisé comme séparateur. Vous pouvez aussi utiliser `join` pour convertir un objet de type tableau (tel qu'un objet d'arguments ou un objet NodeList) en chaîne en le convertissant d'abord en tableau à l'aide de la méthode `Array.prototype.slice()` :

```
function printArguments() {
  console.log(Array.prototype.slice.call(arguments).join(", "));
}

printArguments("a", "b", "c"); // Result: "a, b, c"
```

Length

Les tableaux possèdent une propriété `length` qui, comme son nom l'indique, permet de connaître la longueur d'un tableau.

```
let array = [1, 2, 3];

let l = array.length;

// Résultat: l = 3
```

La propriété `length` permet également de déterminer le nombre d'éléments d'un tableau. Par exemple.

```
let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.length = 2;

console.log(fruits);
// Résultat: ['Banana', 'Orange']
```

Vous pouvez aussi utiliser la propriété `length` pour trouver le dernier élément d'un tableau en l'utilisant comme un index. Par exemple:

```
console.log(fruits[fruits.length - 1]); // Résultat: Orange
```

You can also use the `length` property to add elements to the end of an array. For example:

```
fruits[fruits.length] = "Pineapple";
console.log(fruits); // Result: ['Banana', 'Orange', 'Pineapple']
```

> La propriété `length` est automatiquement mise à jour lorsque des éléments sont ajoutés ou supprimés au sein du tableau.

Il convient également de noter que la propriété `length` n'est pas une méthode, vous n'avez donc pas besoin d'utiliser des parenthèses pour y accéder. Il s'agit simplement d'une propriété de l'objet tableau à laquelle vous pouvez accéder comme n'importe quelle autre propriété d'objet.

Push

On peut ajouter certains éléments dans un tableau, faisant du dernier index l'élément nouvellement ajouté. La méthode `psuh` modifie la longueur d'un tableau et renvoie une nouvelle longueur.

Voici la syntaxe pour utiliser `push` :

```
array.push(element1[, ...[, elementN]]);
```

Les arguments `element1, ..., elementN` représentent les éléments à ajouter à la fin d'un tableau.

Par exemple:

```
let array = [1, 2, 3];
array.push(4);

console.log(array);

// Résultat: array = [1, 2, 3, 4]
```

Vous pouvez également utiliser `push` pour ajouter des éléments à la fin d'un objet de type tableau (comme un objet arguments ou un objet NodeList) en le convertissant d'abord en tableau à l'aide de la méthode `Array.prototype.slice()` :

```
function printArguments() {
  let args = Array.prototype.slice.call(arguments);
  args.push("d", "e", "f");
  console.log(args);
}

printArguments("a", "b", "c"); // Résultat: ["a", "b", "c", "d", "e", "f"]
```

> **Note**: La méthode `push` modifie le tableau d'origine. Cela ne crée pas de nouveau tableau.

For Each

La méthode `forEach` (pour chaque) exécute une fonction sur chaque élément d'un tableau. Voici la syntaxe de la méthode `forEach` :

```
array.forEach(function (element, index, array) {
  // element: l'élément courant
  // index: l'index de l'élément courant
  // array: le tableau sur lequel la méthode forEach est appelée
});
```

Par exemple, imaginons que vous avez un tableau de nombre et que vous voulez afficher le double de chacun d'entre eux dans la console. Vous pouvez faire ceci en utilisant `forEach` :

```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(function (number) {
  console.log(number * 2);
});
```

Vous pouvez tout aussi bien utiliser une fonction fléchée pour définir la callback passée à `forEach` :

```
numbers.forEach((number) => {
  console.log(number * 2);
});
```

or

```
numbers.forEach((number) => console.log(number * 2));
```

La méthode `forEach` ne modifie pas le tableau original. Elle itère simplement les éléments les uns après les autres et exécute le colback sur chacun d'entre eux.

> La méthode `forEach()` n'est pas exécutée dans le cas d'une instruction vide.

Sort

La méthode `sort` trie les éléments d'un tableau dans un ordre spécifique (croissant ou décroissant). Par défaut, la méthode `sort` trie les éléments sous forme de chaînes et les organise par ordre croissant en fonction de leurs valeurs d'unité de code UTF-16.

Voici la syntaxe pour utiliser la méthode `sort` :

```
array.sort([compareFunction]);
```

L'argument `compareFunction` est facultatif et spécifie une fonction qui définit l'ordre de tri. S'il est absent, les éléments sont triés par ordre croissant en fonction de leur représentation sous forme de chaîne.

Par exemple:

```
let city = ["California", "Barcelona", "Paris", "Kathmandu"];
let sortedCity = city.sort();

console.log(sortedCity);

// Résultat: ['Barcelona', 'California', 'Kathmandu', 'Paris']
```

> Les nombres peuvent être mal triés numériquement lors de `sort`. Par exemple, "35" est plus grand que "100", car "3" est plus grand que "1".

Pour résoudre le problème de tri en nombres, des fonctions de comparaison sont utilisées. Les fonctions de comparaison définissent des ordres de tri et renvoient une valeur **négative**, **zéro** ou **positive** en fonction des arguments, comme ceci :

- Une valeur négative si `a` doit être trié avant `b`
- Une valeur positive si `a` doit être trié après `b`
- `0` si `a` et `b` sont égaux et leur ordre n'a pas d'importance

```
const points = [40, 100, 1, 5, 25, 10];
points.sort((a, b) => {
  return a - b;
});

// Résultat: [1, 5, 10, 25, 40, 100]
```

> La méthode `sort()` écrase le tableau original.

Indices

Désormais vous possédez un tableau stockant des données, mais comment faire pour accéder à un élément spécifique ? C'est ici que les indices entrent en jeu. Un **index** r So you have your array of data elements, but what if you want to access a specific element? That is where indices come in. Un **index** fait référence à un endroit du tableau. Les indices progressent logiquement un par un, mais il convient de noter que le premier index d'un tableau est 0, comme c'est le cas dans la plupart des langages. Les crochets `[]` sont utilisés pour signifier que vous faites référence à l'index d'un tableau.

```
// Voici un tableau stockant des chaînes de caractères
let fruits = ["apple", "banana", "pineapple", "strawberry"];

// On fixe la variable banana à la valeur du deuxième élément du
// tableau de fruits. Rappelez-vous que les indices commencent à 0, de ce fait 1
// pointe sur le deuxième élément. Résultat: banana = "banana"
let banana = fruits[1];
```

Vous pouvez aussi utiliser les index pour modifier la valeur d'un élément au sein du tableau:

```
let array = ["a", "b", "c", "d", "e"];
//  indices:  0    1    2    3    4
array[4] = "f";
console.log(array); // Résultat: ['a', 'b', 'c', 'd', 'f']
```

Notons que si vous essayez d'accéder à un élément en utilisant un index situé hors des limites du tableau (par exemple, un index plus petit que 0 ou plus grand ou égal à la longueur du tableau), vous recevrez une valeur de type `undefined` .

```
console.log(array[5]); // Sortie: undefined
array[5] = "g";
console.log(array); // Résultat: ['a', 'b', 'c', 'd', 'f', undefined, 'g']
```

Chapitre 7

Les boucles

Les boucles sont des instructions répétitives où une variable de la boucle change. Les boucles sont utiles: si vous voulez exécuter le même code encore et encore, à chaque fois avec une valeur différente.

Au lieu d'écrire:

```
faireQqch(voiture[0]);
faireQqch(voiture[1]);
faireQqch(voiture[2]);
faireQqch(voiture[3]);
faireQqch(voiture[4]);
```

Vous pouvez écrire:

```
for (var i = 0; i < voiture.length; i++) {
  faireQqch(voiture[i]);
}
```

For

La form la plus simple et intuitive des boucles est la boucle *for*. Sa syntaxe est presque similaire à celle de l'instruction `if()` mais avec plus d'options:

```
for (initialisation; condition de sortie; changement) {
    // do it, do it now
}
```

Explication:

- Dans la partie `initialisation`, exécutée avant la première itération, initialisez votre variable de boucle.
- Dans la partie `condition de sortie`, mettez une condition qui sera vérifiée avant chaque itération. La boucle s'arrête le moment où cette condition s'avère fausse.
- Dans la partie `changement`, dites au programme comment mettre à jour la variable de boucle.

Voyons comment exécuter un bloc d'instruction 10 fois avec une boucle for:

```
for (let i = 0; i < 10; i = i + 1) {
  // do this code ten-times
}
```

> ***Note***: `i = i + 1` peut être écrit: `i++`.

Les boucles `for in` peuvent aussi être utilisées pour parcourir un objet ou un tableau.

```
for (cle in object) {
  // code block to be executed
}
```

Exemple de boucles `for in` pour un objet:

```
const personne = {prenom:"John", nom:"Doe", age:25};
let info = "";
for (let x in personne) {
  info += personne[x];
}

// Resultat: info = "JohnDoe25"
```

Exemple de boucles `for in` pour un tableau:

```
const nombres = [45, 4, 9, 16, 25];
let txt = "";
for (let x in nombres) {
  txt += nombres[x];
}

// Resultat: txt = '45491625'
```

La valeur des objets itérables comme `Arrays`, `Strings`, `Maps`, `NodeLists` peuvent être parcourues en utilisant l'instruction `for of`.

```
let langage = "JavaScript";
let txt = "";
for (let x of langage) {
  txt += x;
}

// Resultat: langage = 'JavaScript'
```

While

La boucle while exécute de façon répétée un bloc de code tant qu'une codition spécifiée est vraie. Grâce à cette dernière, nous pouvons automatiser des tâches répétées ou effectuer des itérations en fonction de l'évalutation d'une condition.

```
while (condition) {
  // instruction à exécuter tant que la condition est vraie
}
```

Par exemple, le code de la boucle dans cet exemple sera exécuté tant que la valeur de la variable `i` est inférieure à 5.

```
var i = 0,
x = "";
while (i < 5) {
  x = x + "Le nombre est " + i + '\n';
  i++;
}
console.log(x);
```

> Faites attention d'éviter les boucles infinies qui arrivent lorsque la condition est indéfiniment évaluée à vraie.

Do...While

L'instruction do...while permet de créer une boucle qui exécute un bloc d'instructions spécifiques jusqu'à ce que la condition de test soit fausse. La condition est évalue après avoir exécuté le bloc. La syntaxe de la boucle est la suivante:

```
do {
  // instructions
} while (expression);
```

Affichons par exemple les nombres plus petits que 10 en utilisant la boucle `do...while` :

```
var i = 0;
do {
  document.write(i + " ");
  i++; // incrémentation de i by 1
} while (i < 10);
```

> ***Note***: `i = i + 1` peut être écrit: `i++` .

Chapitre 8

Les fonctions

Les fonctions sont l'une des notions les plus puissantes et les plus essentielles de la programmation. Les fonctions, comme les fonctions mathématiques, effectuent des transformations, prennent des valeurs d'entrée appelées **arguments** et **renvoient** une valeur de sortie

Les fonctions peuvent être créées de deux manières : en utilisant la "déclaration de fonction" ou l'"expression de fonction". Le *nom de la fonction* peut être omis dans l'expression de la fonction, ce qui en fait une fonction anonyme. Les fonctions, comme les variables, doivent être déclarées. Déclarons une fonction `double` qui accepte un *argument* appelé `x` et **retourne** le double de x :

```
// un exemple de déclaration de fonction
function double(x) {
  return 2 * x;
}
```

> La fonction ci-dessus **peut** être référencée avant d'avoir été définie.

Les fonctions sont également des valeurs en JavaScript; elles peuvent être stockées dans des variables (tout comme les nombres, les chaînes de caractères, etc ...) et données à d'autres fonctions en tant qu'arguments :

```
// un exemple d'expression de fonction
let double = function (x) {
  return 2 * x;
};
```

> La fonction ci-dessus **ne peut pas** être référencée avant d'être définie, comme n'importe quelle autre variable.

Une callback est une fonction passée en argument à une autre fonction.

Une fonction fléchée est une alternative compacte aux fonctions traditionnelles qui présente quelques différences sémantiques avec certaines limitations. Ces fonctions n'ont pas leurs propres liens avec `this`, `arguments` et `super`, et ne peuvent pas être utilisées comme constructeurs. Un exemple de fonction flèche.

```
const double = (x) => 2 * x;
```

Le mot-clé `this` dans la fonction fléchée représente l'objet qui a défini la fonction arrow.

Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur sont des fonctions qui manipulent d'autres fonctions. Par exemple, une fonction peut prendre d'autres fonctions comme arguments et/ou produire une fonction comme valeur de retour. Ces techniques fonctionnelles fantaisistes sont des constructions puissantes disponibles en JavaScript et dans d'autres langages de haut niveau comme python, lisp, etc.

Nous allons maintenant créer deux fonctions simples, `add_2` et `double`, et une fonction d'ordre supérieur appelée `map`. `map` acceptera deux arguments, `func` et `list` (sa déclaration commencera donc par `map(func,list)`), et retournera un tableau. `func` (le premier argument) sera une fonction qui sera appliquée à chacun des éléments du tableau `list` (le second argument).

```
// Définir deux fonctions simples
let add_2 = function (x) {
  return x + 2;
};
let double = function (x) {
  return 2 * x;
};

// map est une fonction qui accepte 2 arguments :
// func la fonction à appeler
// list un tableau de valeurs sur lequel appeler func
let map = function (func, list) {
  let output = []; // output est un tableau vide
  for (idx in list) {
    output.push(func(list[idx]));
  }
  return output;
};

// Nous utilisons map pour appliquer une fonction à une liste entière
// d'entrées pour les "mapper" à une liste de sorties correspondantes
map(add_2, [5, 6, 7]); // => [7, 8, 9]
map(double, [5, 6, 7]); // => [10, 12, 14]
```

Les fonctions de l'exemple ci-dessus sont simples. Cependant, lorsqu'elles sont transmises en tant qu'arguments à d'autres fonctions, elles peuvent être composées de manière imprévue pour construire des fonctions plus complexes.

Par exemple, si nous remarquons que nous utilisons les invocations `map(add_2, ...)` et `map(double, ...)` très souvent dans notre code, nous pourrions décider de créer deux fonctions spéciales de traitement de tableau qui ont l'opération désirée intégrée en elles. En utilisant la composition de fonctions, nous pourrions faire cela comme suit :

```
process_add_2 = function (list) {
  return map(add_2, list);
};
process_double = function (list) {
  return map(double, list);
};
process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Maintenant, créons une fonction appelée `buildProcessor` qui prend une fonction `func` en entrée et renvoie un `func`-processor, c'est-à-dire une fonction qui applique `func` à chaque entrée de la liste.

```
// une fonction qui génère un processeur de liste qui exécute
let buildProcessor = function (func) {
  let process_func = function (list) {
    return map(func, list);
  };
  return process_func;
};
// l'appel à buildProcessor renvoie une fonction qui est appelée avec une entrée de liste

// en utilisant buildProcessor, nous pourrions générer les processeurs add_2 et double list comme suit :
process_add_2 = buildProcessor(add_2);
process_double = buildProcessor(double);

process_add_2([5, 6, 7]); // => [7, 8, 9]
process_double([5, 6, 7]); // => [10, 12, 14]
```

Prenons un autre exemple. Nous allons créer une fonction appelée `buildMultiplier` qui prend un nombre `x` en entrée et renvoie une fonction qui multiplie son argument par `x` :

```
let buildMultiplier = function (x) {
  return function (y) {
    return x * y;
  };
};

let double = buildMultiplier(2);
let triple = buildMultiplier(3);

double(3); // => 6
triple(3); // => 9
```

Chapitre 9

Les objets

En JavaScript, les objets sont **mutables** car on change directement les valeurs pointées par l'objet de référence, alors que quand on change les valeurs des types primitifs, nous modifions en réalité leur référence qui pointe vers la nouvelle valeur (ce qui signifie que les types primitifs sont **non mutables**). Les types primitifs en JavaScript sont: `true` , `false` , `numbers` , `strings` , `null` et `undefined` . Toute autre valeur est de type `object` . Les objets contiennent des paires de : `nomPropriete` : `valeurPropriete` . Il existe trois façons de créer un objet en JavaScript.

1. De façon Litérale

```
let object = {};
// Oui, seulement des paires d'accolades.
```

> ***Note:*** Ceci est la méthode **recommandée**.

2. De façon orientée objet

```
let object = new Object();
```

> ***Note:*** A peu près la même syntaxe qu'en Java.

3. Et à travers la méthode: `Object.create()`

```
let object = Object.create(proto[, propertiesObject]);
```

> ***Note:*** Cette syntaxe pemet de créer un nouveau objet avec les propriétés du prototype spécifié.

Les propriétés

On appelle propriété une paire `nom propriété : valeur propriété` où **nom propriété** est une chaîne de caractères. Si ce n'est pas le cas, il est transformé en chaîne. On précise la propriété lors de la création de l'objet ou plutard. On peut avoir plusieurs propriétés, séparées par des virgules ou aucune.

```
let language = {
  name: "JavaScript",
  isSupportedByBrowsers: true,
  createdIn: 1995,
  author: {
    firstName: "Brendan",
    lastName: "Eich",
  },

  // Oui ! On peut imbriquer les objets!
  getAuthorFullName: function () {
    return this.author.firstName + " " + this.author.lastName;
  },
  // Oui ! On peut mettre des fonctions comme valeurs
};
```

Le code suivant démontre comment on récupère la valeur d'une propriété.

```
let variable = language.name;
// variable contient maintenant "JavaScript"
variable = language["name"];
// Les lignes ci-dessus font la même chose. La seule différence est que la seconde syntaxe vous donne la possib
variable = language.newProperty;
// variable est maintenant undefined, parce que nous n'avons pas assigné cette propriété encore.
```

L'exemple suivant nous montre comment **ajouter** ou **modifier** une propriété qui existe.

```
language.newProperty = "new value";
// Désormais, l'objet a une propriété newProperty. Si la propriété existait, sa valeur aurait été remplacée.
language["newProperty"] = "changed value";
// Une fois de plus, vous pouvez accéder aux propriété des deux façons, mais la première (notation avec un poin
```

La mutabilité

La **mutabilité** signifie que l'objet ou le type de données peut être changé après sa création tandis que l'**immutabilité** signifie qu'on ne peut le changer. Les objets mutables autorisent la modification de leur état interne alors que les objets immutables renvoient une nouvelle instance avec les modifications effectuées, laissant l'état original inchangé.

La différence entre les objets et les valeurs primitives est que l'on peut **changer les objets**, alors que les valeurs primitives sont **immutables**.

Par exemple :

```
let myPrimitive = "première valeur";
myPrimitive = "autre valeur";
// myPrimitive pointe maintenant sur une autre chaîne.

let myObject = { key: "première valeur" };
myObject.key = "autre valeur";
// myObject réfère toujours le même objet !.
```

Vous pouvez ajouter, modifier ou supprimer les propriéts d'un objet en utilisant la notation pointée ou les crochets:

```
let object = {};
object.foo = 'bar'; // Ajouter la propriété 'foo'
object['baz'] = 'qux'; // Ajouter la propriété 'baz'
object.foo = 'quux'; // Changer la propriété 'foo'
delete object.baz; // Supprimer la propriété 'baz'
```

> Les valeurs primitives comme les nombres et les chaînes sont immutables tandis que les objets comme les tableaux sont mutables.

Les références

Les objets ne sont **jamais copiés**. On les manipule par leur référence. La référence d'un objet est une valeur qui référence l'objet. Lorsqu'on crée un objet en utilisant l'opérateur `new`, ou la syntaxe littérale des objets (`{}`), JavaScript crée un objet et lui assigne une référence.

Voici un exemple de création d'un objet avec avec les accolades:

```
var object = {
  foo: 'bar'
};
```

Voici un exemple de création d'un objet avec l'opérateur `new` :

```
var object = new Object();
object.foo = 'bar';
```

Quand ou lorsqu'on assigne la référence d'un objet à une variable, la variable contient simplement une référence de l'objet, mais pas l'objet lui-même. Cela signifie que lorsqu'on assigne la référence à une autre variable, les deux variables pointent vers le même objet.

Par exemple:

```
var object1 = {
  foo: 'bar'
};

var object2 = object1;

console.log(object1 === object2); // Output: true
```

Dans l'exemple ci-dessus, et `object1` , et `object2` sont des variables qui contiennent une référence vers le même objet. L'opérateur `===` est utilisé pour comparer les références, pas les objets eux-mêmes. Vu que les deux variables contiennent la même référence, voilà pourquoi le résultat sera `true` .

> Vous pouvez utiliser la méthode `Object.assign()` pour créer un nouveau objet qui représente une copie d'un objet existant.

Voici un exemple de création d'un objet par référence.

```
// Imagine que j'aie une pizza
let myPizza = { slices: 5 };
// Et que je la partage avec toi
let yourPizza = myPizza;
// Si je mange une part
myPizza.slices = myPizza.slices - 1;
// Il nous resterait 4 parts, car myPizza te yourPizza font référence au même objet.
let numberOfSlicesLeft = yourPizza.slices;
console.log(numberOfSlicesLeft); // 4
console.log(myPizza.slices === yourPizza.slices); // true

let a = {}, b = {}, c = {}; // a, b, et c réfèrent chacun à un objet différent vide.
a = b = c = {}; // a, b, et c réfèrent tous maintenant au même objet vide.
```

Prototype

Chaque objet est lié à une propriété `prototype` qui est un autre objet duquel il hérite des propriéts. Les objets créés directement avec les accolades (`{}`) sont automatiquement associés à `Object.prototype` , qui est un standard du JavaScript.

Lorsque l'interpréteur JavaScrit essaye de trouver une propriété à laquelle on essaie d'accéder comme à travers ce code:

```
let adult = { age: 26 },
  retrievedProperty = adult.age;
// The line above
```

L'interpréteur recherche d'abord à travers toutes les propriétés de l'objet lui-même. Par exemple l'objet `adult` n'a qu'une seule propriété — `age` . Mais à côté de celui-là, en réalité il en a d'autres qu'il a hérité de `Object.prototype.`

```
let stringRepresentation = adult.toString();
// the variable has value of '[object Object]'
```

La méthode `toString` est une propriété de l'objet `Object.prototype` dont `adult` a hérité. C'est une fonction qui renvoie la représentation textuelle de l'objet. Si vous voulez avoir une représentation plus personnalisée, alors vous pouvez surcharger la méthode (ou la fonction); c'est-à-dire, définir votre propre version de la méthode. Et pour ce faire, on procède comme suit:

```
adult.toString = function () {
  return "J'ai " + this.age + ' ans';
};
```

Si maintenant nous appelons la méthode `toString` , l'interpréteur va retrouver cette propriété dans l'objet `adult` lui-même.

Ainsi, l'interpréteur retourne la première valeur qu'il retrouve en partant de l'objet lui-même et descendant prototype par prototype.

Pour déclarer notre objet `adult` comme prototype au lieu de `Object.prototype` , nous pouvons invoquerla méthode `Object.create` comme suit:

```
let child = Object.create(adult);
/**
 * Cette façon de créer les objets nous laisse le champ libre de remplacer l'objet par défaut qui est Object.pr
 * Dans ce cas-ci, le prototype de child est l'objet adult.
*/
child.age = 8;
/**
 * Avant, child n'avait pas sa propriété age, donc pour la trouver, l'interpréteur devait aller chercher sur so
 * Maintenant, quand nous affectons un age à child, l'interpréteur ne va pas trop loin pour la trouver.
 * Note: La propriété age de adult démeure 26.
*/
let stringRepresentation = child.toString();
// Maintenant, le résultat de cet appel est évidemment : "J'ai 8 ans".
/**
 * Note: Nous n'avons pas ici surchargé la propriété toString, donc c'est l'implémentation qu'on a mise dans ad
 * Si adult n'avait pas de méthode toString, alors il s'agit de celle de Object.prototype qui serait invoquée e
*/
```

Le prototype de `child` est `adult`, dont le prototype est lui même `Object.prototype`. Cette séquence de prototype est appelée **chaîne de prototypes**.

Le mot clé delete

L'opérateur `delete` peut être utilisé pour **supprimer une propriété**. Lorsqu'une propriété est supprimée, elle est enlevée de l'objet et ne peut ni être accédée ni énumérée (i.e., n'apparaît pas dans une boucle for-in).

Voici la syntaxe d'utilisation de l'opérateur `delete` :

```
delete object.property;
```

Par exemple:

```
let adult = { age: 26 }, child = Object.create(adult);

child.age = 8;

delete child.age;

/* Supprime la propriété age de child, permet d'accéder à l'age du protype car il n'est plus masqué */

let prototypeAge = child.age;
// 26, car child n'a pas de propriété age.
```

> L'opérateur `delete` marche uniquement sur les propriétés de l'objet (définis sur l'objet) mais pas ceux qui sont hérités. Il ne marche pas non plus sur les propriété qui ont l'attribut `configurable` mis à `false` .

L'opérateur `delete` ne modifie pas la chaîne de prototypage. Elle supprime purement et simplement la propriété spécifiée de l'objet et n'agit pas sur l'objet lui-même. Cela permet de rendre la propriété inaccessible comme si elle n'avait pas été attribuée. Si vous désirez supprimer un objet et libérer la mémoire occupée, vous pouvez tout simplement affecter `null` à la variable.

L'énumération

On appelle *Enumération*, le fait d'itérer sur les propriétés d'un objet et d'exécuter un bloc d'instructions pour chaque propriété de l'objet. Il y a diverses façons d'énumérer les propriétés d'un objet en JavaScript:

Une façon d'énumérer les propriétés d'un objet est d'utiliser une boucle `for in`. La boucle `for in` permet d'itérer de façon stochastique (arbitraire) sur les propriétés d'un objet qui est `énumérable`, et pour chaque propriété on exécute un certain bloc d'instructions.

> **Note :** La boucle `for in` peut itérer sur tous les nom de propriétés d'un objet. L'énumération inclut autant les fonctions que les propriétés issus du prototype.

```
let fruit = {
    apple: 2,
    orange: 5,
    pear: 1,
  },
  sentence = "I have ",
  quantity;

for (kind in fruit) {
  quantity = fruit[kind];
  sentence += quantity + " " + kind + (quantity === 1 ? "" : "s") + ", ";
}
// La ligne suivante supprime la virgule finale..
sentence = sentence.substr(0, sentence.length - 2) + ".";
// I have 2 apples, 5 oranges, 1 pear.
```

Un autre moyen d'énumérer les propriétés d'un objet est d'utiliser la méthode: `Object.keys()` qui retourne un tableau contenant les noms des propriétés appartenant à l'objet lui-même.

Par exemple:

```
let object = {
  foo: 'bar',
  baz: 'qux'
};

let properties = Object.keys(object);
properties.forEach(function(property) {
  console.log(property + ': ' + object[property]);
});

// foo: bar
// baz: qux
```

Chapitre 10

Date et temps

L'objet `date` stocke des dates et le temps et fournit les méthodes pour les manipuler. Les objets date sont statiques et utilisent le fuseau horaire du navigateur pour afficher la date sous forme de texte.

Pour créer un objet `date` on utilise le constructeur avec le mot clé `new`. En plus de la syntaxe de base `new Date()`, on peut passer des paramètres pour affiner le retour.

```
new Date()
new Date(date string)
new Date(year,month)
new Date(year,month,day)
new Date(year,month,day,hours)
new Date(year,month,day,hours,minutes)
new Date(year,month,day,hours,minutes,seconds)
new Date(year,month,day,hours,minutes,seconds,ms)
new Date(milliseconds)
```

> Les mois sont spécifiés de `0` à `11`, si on excède ces index, on débordera sur l'année suivante.

Les méthodes et les propriétés supportées par `date` sont décrites ci-dessous:

Name	Description
`constructor`	Renvoie la fonction qui a créé le prototype de l'objet Date
`getDate()`	Renvoie le jour (1-31) du mois en cours
`getDay()`	Renvoie le jour (0-6) de la semaine en cours
`getFullYear()`	Renvoie l'année (sur 4 chiffres)
`getHours()`	Renvoie l'heure (0-23)
`getMilliseconds()`	Renvoie les millisecondes (0-999)
`getMinutes()`	Renvoie les minutes (0-59)
`getMonth()`	Renvoie le mois (0-11)
`getSeconds()`	Renvoie les secondes (0-59)
`getTime()`	Renvoie la valeur de la date en millisecondes écoulées depuis le 1er janvier 1970
`getTimezoneOffset()`	Renvoie l'écart en minutes entre la date et le fuseau horaire UTC
`getUTCDate()`	Renvoie le jour (1-31) du mois de la date d'après UTC
`getUTCDay()`	Renvoie le jour (0-6) de la semaine d'après UTC
`getUTCFullYear()`	Renvoie l'année (sur 4 chiffres) d'après UTC
`getUTCHours()`	Renvoie les heures (0-23) d'après UTC
`getUTCMilliseconds()`	Renvoie les millisecondes (0-999) d'après UTC
`getUTCMinutes()`	Renvoie les minutes (0-59) d'après UTC
`getUTCMonth()`	Renvoie les mois (0-11) d'après UTC
`getUTCSeconds()`	Renvoie les secondes (0-59) d'après UTC
`now()`	Renvoie le nombre de millisecondes écoulées depuis le 1er janvier 1970
`parse()`	Formate une date passée en paramètre et renvoie sa valeur en millisecondes écoulées depuis le 1er janvier 1970
`prototype`	Permet d'ajouter des propriétés
`setDate()`	Définit le jour du mois (relatif au début du mois courant) pour une date passée en paramètre
`setFullYear()`	Définit l'année
`setHours()`	Définit l'heure
`setMilliseconds()`	Définit les millisecondes
`setMinutes()`	Définit les minutes
`setMonth()`	Définit le mois
`setSeconds()`	Définit les secondes
`setTime()`	Définit le temps
`setUTCDate()`	Définit le jour du mois d'après UTC
`setUTCFullYear()`	Définit l'année d'après UTC
`setUTCHours()`	Définit l'heure d'après UTC

Name	Description
`setUTCMilliseconds()`	Définit les millisecondes d'après UTC
`setUTCMinutes()`	Définit les minutes d'après UTC
`setUTCMonth()`	Définit le mois d'après UTC
`setUTCSeconds()`	Définit les secondes d'après UTC
`toDateString()`	Renvoie la date dans un format lisible par un humain
`toISOString()`	Renvoie la date au format ISO
`toJSON()`	Renvoie la date en tant que chaîne, au format JSON
`toLocaleDateString()`	Renvoie la date en tant que chaîne en utilisant les conventions de la locale
`toLocaleTimeString()`	Renvoie le temps en tant que chaîne en utilisant les conventions de la locale
`toLocaleString()`	Renvoie la date en utilisant les conventions de la locale
`toString()`	Renvoie une chaîne de caractères représentant l'objet
`toTimeString()`	Renvoie la partie *temps* de la date dans un format lisible par un humain
`toUTCString()`	Convertit la date en chaîne selon le format universel
`toUTC()`	Renvoie les millisecondes écoulées depuis le 1er janvier 1970 au format UTC
`valueOf()`	Renvoie la valeur primitive de l'objet `Date`

Chapitre 11

JSON

Java**S**cript **O**bject **N**otation (JSON) est un format textuele dévolu à stocker et transporter des données. Les objets Javascript peuvent être facilement convertis en JSON et vice versa. Par exemple:

```
//  un objet JavaScript
let myObj = { name: "Ryan", age: 30, city: "Austin" };

// convertit en JSON:
let myJSON = JSON.stringify(myObj);
console.log(myJSON);
// Résultat: '{"name":"Ryan","age":30,"city":"Austin"}'

//convertit à nouveau en objet JavaScript
let originalJSON = JSON.parse(myJSON);
console.log(originalJSON);

// Résultat: {name: 'Ryan', age: 30, city: 'Austin'}
```

`stringify` et `parse` sont deux méthodes disponibles pour JSON.

Method	Description
`parse()`	Renvoie un objet JavaScript depuis une chaîne JSON analysée
`stringify()`	Renvoie une chaîne au format JSON depuis un objet JavaScript

Les types de données suivants sont supportés en JSON.

- chaîne de caractères
- nombre
- tableau
- booléen
- un objet contenant des valeurs JSON valides
- `null`

Par contre, JSON ne supporte pas les `function`, les `date` ou les `undefined`.

Chapitre 12

Gestion des erreurs

En programmation, les erreurs surviennent pour diverses raisons, certaines sont dues à des erreurs de code, d'autres à des entrées erronées, et d'autres encore sont imprévisibles. Lorsqu'une erreur se produit, le code s'arrête et génère un message d'erreur généralement affiché dans la console.

try... catch

Au lieu d'arrêter l'exécution du code, nous pouvons utiliser la construction `try...catch` qui permet d'attraper les erreurs sans interrompre le script. La construction `try...catch` a deux blocs principaux; `try` et ensuite `catch`.

```
try {
  // code...
} catch (err) {
  // gestion des erreurs
}
```

Dans un premier temps, le code du bloc `try` est exécuté. Si aucune erreur n'est rencontrée, il saute le bloc `catch`. Si une erreur survient, l'exécution du bloc `try` est arrêtée, déplaçant la séquence de contrôle vers le bloc `catch`. La cause de l'erreur est capturée dans la variable `err`.

```
try {
  // code...
  alert("Welcome to Learn JavaScript");
  asdk; // erreur la variable asdk n'est pas définie
} catch (err) {
  console.log("Une erreur s'est produite");
}
```

`try...catch` fonctionne pour les erreurs d'exécution, ce qui signifie que le code doit être exécutable et synchrone.

Pour lancer une erreur personnalisée, une instruction `throw` peut être utilisée. L'objet error, qui est généré par les erreurs, a deux propriétés principales.

name : nom de l'erreur **message** : détails sur l'erreur

Si nous n'avons pas besoin d'un message d'erreur, la capture peut être omise.

try...catch...finally

Nous pouvons ajouter une construction supplémentaire à `try...catch` appelée `finally`, ce code s'exécute dans tous les cas, c'est-à-dire après `try` lorsqu'il n'y a pas d'erreur et après un `catch` en cas d'erreur. La syntaxe pour `try ...catch...finally` est la suivante.

```
try {
  // essaie d'exécuter le code
} catch (err) {
  // gère les erreurs
} finally {
  // s'exécute quoi qu'il arrive
}
```

Exécution du code de l'exemple du monde réel.

```
try {
  alert("try");
} catch (err) {
  alert("catch");
} finally {
  alert("finally");
}
```

Dans l'exemple ci-dessus, le bloc `try` est exécuté en premier, puis il est suivi par `finally` puisqu'il n'y a pas d'erreur.

Exercise

Ecrivez une fonction `divideNumbers()` qui prend deux arguments "numerator" et "denominator" et renvoie le résultat de la division du numerator par le denominator en utilisant les paramètres suivants.

```
function divideNumbers(numerator, denominator) {
try {
// instruction try pour diviser le numérateur par le dénominateur.
} catch (error) {
// impression du message d'erreur
} finally {
// afficher que l'exécution est terminée
}
// return result
}
let answer = divideNumbers(10, 2);
```

Chapitre 13

Modules

Dans le monde réel, un programme grandit organiquement au gré de ses besoins en fonctionnalité. Avec une base de code qui s'élargit, la structuration et le besoin en maintenance de celui-ci réclame un travail supplémentaire. En dépit que cela puisse sembler fastidieux, il est tentant de le négliger et de permettre au code de s'enchevêtrer sans fin. En réalité, cela va rapidement accroître la complexité de l'application, nous obligeant à construire une compréhension globale du système dans laquelle on peinera à comprendre chaque élément de manière isolée.

Les *modules* sont une solution à ce problème. Un `module` spécifie de quels morceaux de code il dépend, ainsi que les fonctionnalités qu'il permet aux autres modules d'utiliser. Les modules qui dépendent d'autres modules s'appellent des *dépendances*. De nombreuses bibliothèques de modules existent pour organiser le code en modules et les charger à la demande.

- AMD - est l'un des plus anciens systèmes de modules, utilisé à l'origine par require.js.
- CommonJS - est un système de modules créés pour les serveurs Node.js.
- UMD - est un système de modules compatible avec AMD et CommonJS.

Les modules peuvent se charger l'un l'autre, avec les directives spéciales `import` et `export` pour partager leurs fonctionnalités et appeler mutuellement leurs propres fonctions.

- `export` - identifie les fonctions et les variables qui seront accessibles depuis l'extérieur du module courant
- `import` - importe les fonctionnalités depuis un module extérieur

Découvrons le fonctionnement de `import` et `export` au sein des modules. Nous avons ici la fonction `sayHi` qui est exportée depuis le fichier `sayHi.js`.

```
// 📁 sayHi.js
export const sayHi = (user) => {
  alert(`Bonjour, ${user}!`);
};
```

La fonction `sayHi` est utilisée dans le fichier `main.js` grâce à la directive `import`.

```
// 📁 main.js
import { sayHi } from "./sayHi.js";

alert(sayHi); // function...
sayHi("Kelvin"); // Bonjour, Kelvin!
```

Ici, la directive d'importation charge le module en important le chemin relatif et en assignant la variable `sayHi`.

Les modules peuvent être exportés de deux manières: **nommés** et **default (par défaut)**. De plus, les exports nommés peuvent être assignés en ligne ou individuellement.

```
// 📁 person.js

// exports nommés en ligne
export const name = "Kelvin";
export const age = 30;

// immédiatement
const name = "Kelvin";
const age = 30;
export { name, age };
```

On ne peut avoir qu'un seul `export` par défaut dans un fichier.

```
// 📁 message.js
const message = (name, age) => {
    return `${name} is ${age} years old.`;
};
export default message;
```

En se basant sur le type d'export, on peut importer de deux manières. Les exports nommés sont construits en utilisant les crochets alors que les exports par défaut non. En conséquence, on procédera aux imports selon la même logique.

```
import { name, age } from "./person.js"; // import d'export nommé
import message from "./message.js"; // import d'export par défaut
```

Lors de l'assignation des modules, on doit veiller à empêcher la *dépendance circulaire*. La dépendance circulaire est une situation où un module A dépend d'un module B, et le B dépend lui aussi de A directement ou indirectement.

Chapitre 14

Expression régulières

Une expression régulière est un objet qui peut être indifféremment construit grâce au constructeur `RegEx` ou écrit comme une valeur littérale délimitée par des slash avant `(/)`. Les syntaxes pour créer une expression rationnelle sont expliquées ci-dessous.

```
// en utilisant le constructeur de l'objet `RegExp`
new RegExp(pattern[, flags]);

// en utilisant les littéraux
/pattern/modifiers
```

Les drapeaux (ou modificateurs) sont optionnels lorsque l'on crée une regex en utilisant les littéraux. Un exemple de création d'expression régulière à l'aide de la méthode mentionnée ci-dessus est le suivant.

```
let re1 = new RegExp("xyz");
let re2 = /xyz/;
```

Ces deux manières créeront un objet regex et auront les mêmes méthodes et propriétés. Il existe des cas où nous pourrions avoir besoin de valeurs dynamiques pour créer une expression régulière, dans ce cas, les littéraux ne fonctionneront pas et les regex devront être créés via la méthode du constructeur.

> Dans les cas où nous voulons qu'un slash fasse partie d'une expression régulière, nous devons échapper le slash `(/)` par un antislash `(\)`.

Les différents modificateurs utilisés pour effectuer des recherches insensibles à la casse sont:

- `g` - recherche globale (trouve toutes les occurrences au lieu de s'arrêter à la première occurrence)
- `i` - recherche insensible à la casse
- `m` - correspondance multiligne

Les *crochets* sont utilisés à l'intérieur d'une expression régulière pour trouver une plage de caractères. Certaines d'entre elles sont mentionnées ci-dessous.

- `[abc]` - trouve n'importe quel caractère entre les crochets
- `[^abc]` - trouve n'importe quel caractère qui n'est pas entre les crochets
- `[0-9]` - trouve n'importe quel chiffre entre les crochets
- `[^0-9]` - trouve n'importe quel caractère qui n'est pas entre les crochets (non numérique)
- `(x|y)` - trouve l'une des alternatives séparées par |

Les *métacaractères* sont des caractères spéciaux qui possèdent une signification particulière au sein de l'expression. Ces caractères sont décrits en détail ci-après :

Métacaractère	Description
`.`	Matche tous les caractères sauf une nouvelle ligne ou un terminateur
`\w`	Matche un caractère alphanumérique `[a-zA-Z0-9_]`
`\W`	Matche tous les caractères non alphanumériques (identique à `[^a-zA-Z0-9_]`)
`\d`	Matche tous les caractères numériques (same as `[0-9]`)
`\D`	Matche tous les caractères non numériques
`\s`	Matche tous les caractères d'espacement (espaces, tabulations, etc)
`\S`	Matche tous les caractères qui ne sont pas des caractères d'espacement
`\b`	Matche le début / la fin d'un mot
`\B`	Matche tous les caractères hormis le début / la fin d'un mot
`\0`	Matche un caractère `NULL`
`\n`	Matche un caractère de nouvelle ligne
`\f`	Matche un caractère de saut de page
`\r`	Matche un caractère de type retour chariot
`\t`	Matche un caractère tabulaire
`\v`	Matche un caractère tabulaire vertical
`\xxx`	Matche un caractère spécifié par un nombre octal `xxx`
`\xdd`	Matche un caractère spécifié par un nombre hexadécimal `dd`
`\udddd`	Matche un caractère Unicode spécifié par un nombre hexadécimal `dddd`

Les propriétés et les méthodes supportées par les regex sont listées ci-dessous:

Nom	Description
`constructor`	Renvoie la fonction qui a créé le prototype de l'objet `RegExp`
`global`	Vérifie si le modificateur `g` est défini
`ignoreCase`	Vérifie si le modificateur `i` est défini
`lastIndex`	Spécifie l'index auquel commencer le prochain match
`multiline`	Vérifie si le modificateur `m` est défini
`source`	Retourne le texte d'une chaîne
`exec()`	Teste la correspondance et renvoie la première occurrence. S'il n'y en a pas, renvoie "null"
`test()`	Teste la correspondance et renvoie `true` ou `false`
`toString()`	Renvoie la valeur de chaîne de l'expression régulière

> La méthode `compile()` recompile une expression régulière et est dépréciée. Elle ne doit plus être utilisée.

Quelques exemples d'expressions régulières sont présentés ici.

```
let text = "Les meilleures choses dans la vie sont gratuites";
let result = /e/.exec(text); // cherche le matche d'un e dans la chaîne "text"
// résultat: e

let helloWorldText = "Hello world!";
// Recherchons "Hello"
let pattern1 = /Hello/g;
let result1 = pattern1.test(helloWorldText);
// Résultat: true

let pattern1String = pattern1.toString();
// pattern1String : '/Hello/g'
```

Statique

Le mot-clé `static` définit les méthodes ou propriétés statiques d'une classe. Ces méthodes et propriétés sont appelées dans la classe elle-même.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Bonjour " + x.name;
  }
}
let myCar = new Car("Toyota");

console.log(myCar.hello()); // Ceci va générer une erreur
console.log(Car.hello(myCar));
// Résultat : Bonjour Toyota
```

On peut accéder à la méthode statique ou à la propriété d'une autre méthode statique de la même classe en utilisant le mot-clé `this`.

L'héritage

L'héritage est utile pour la réutilisation du code car il étend les propriétés et les méthodes existantes d'une classe. Le mot-clé `extends` est utilisé pour créer un héritage de classe.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return "J'ai une " + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ", c'est un " + this.model;
  }
}

let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()); // J'ai une Camry, c'est une Toyota.
```

> Le prototype de la classe parente doit être un `Objet` ou un `null`.

La méthode `super` est utilisée à l'intérieur d'un constructeur et fait référence à la classe parente. Elle permet d'accéder aux propriétés et aux méthodes de la classe parente.

Modificateurs d'accès

`public` , `private` , et `protected` sont les trois modificateurs d'accès utilisés dans la classe pour contrôler son accès depuis l'extérieur. Par défaut, tous les membres (propriétés, champs, méthodes ou fonctions) sont accessibles publiquement depuis l'extérieur de la classe.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Bonjour " + x.name;
  }
}
let myCar = new Car("Toyota");
console.log(Car.hello(myCar)); // Hello Toyota
```

Les membres `private` ne peuvent être accédés qu'à l'intérieur de la classe et ne peuvent être accessibles de l'extérieur. Private doit commencer par `#` .

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Bonjour " + x.name;
  }
  #present(carname) {
    return "J'ai une " + this.carname;
  }
}
let myCar = new Car("Toyota");
console.log(myCar.#present("Camry")); // Erreur
console.log(Car.hello(myCar)); // Hello Toyota
```

Les champs `protected` ne sont accessibles qu'à l'intérieur de la classe et de celles qui l'étendent. Ils sont utiles pour l'interface interne, car la classe qui en hérite a également accès à la classe mère. Les champs protégés sont préfixés par `_` .

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  _present() {
    return "J'ai une " + this.carname;
  }
}

class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this._present() + ", c'est un " + this.model;
  }
}
let myCar = new Model("Toyota", "Camry");
console.log(myCar.show()); // J'ai une Toyota, c'est une Camry
```

Chapitre 16

Browser Object Model (BOM)

Le modèle objet du navigateur (Browser Object Model, BOM) nous permet d'interagir avec la fenêtre du navigateur. L'objet `window` représente la fenêtre du navigateur et est supporté par tous les navigateurs.

L'objet `window` est l'objet par défaut d'un navigateur, nous pouvons donc spécifier `window` ou appeler directement toutes les fonctions.

```
window.alert("Bienvenue dans Apprendre JavaScript");

alert("Bienvenue dans Apprendre JavaScript");
```

De la même manière, nous pouvons appeler d'autres propriétés sous l'objet fenêtre, telles que l'historique, l'écran, le navigateur, l'emplacement, etc.

Window (fenêtre)

L'objet `window` représente la fenêtre du navigateur et est supporté par les navigateurs. Les variables globales, les objets et les fonctions font également partie de l'objet window.

Les **variables** globales sont des **propriétés** et les **fonctions** sont des **méthodes** de l'objet window.

Prenons l'exemple des propriétés de l'écran. Elle est utilisée pour déterminer la taille de la fenêtre du navigateur et est mesurée en pixels.

- `window.innerHeight` - la hauteur intérieure de la fenêtre du navigateur
- `window.innerWidth` - la largeur intérieure de la fenêtre du navigateur

> ***Note*** : `window.document` est identique à `document.location` car le document object model(DOM) fait partie de l'objet window.

Quelques exemples de méthodes de fenêtre:

- `window.open()` - ouvre une nouvelle fenêtre
- `window.close()` - fermer la fenêtre courante
- `window.moveTo()` - déplace la fenêtre courante
- `window.resizeTo()` - redimensionne la fenêtre courante

Popup (modale)

Les fenêtres contextuelles (ou modales), aussi appelées "popup", sont un moyen supplémentaire d'afficher des informations, d'obtenir la confirmation de l'utilisateur ou de prendre en compte les données de l'utilisateur à partir de documents supplémentaires. Une fenêtre contextuelle peut naviguer vers un nouvel URL et envoyer des informations à la fenêtre qui s'ouvre. Les fonctions globales dans lesquelles nous pouvons afficher les informations de la fenêtre contextuelle sont les suivantes : **Alerte**, **Confirmation** et **Prompt (demande)**.

1. **alert()** : Elle affiche des informations à l'utilisateur et dispose d'un bouton "**OK**" pour interragir.

   ```
   alert("Exemple de message d'alerte");
   ```

2. **confirm()** : S'utilise comme une boîte de dialogue pour confirmer ou accepter quelque chose. Elle dispose de "**Ok**" et "**Cancel**" pour interragir. Si l'utilisateur clique sur "**Ok**", elle renvoie `true`, si l'utilisateur clique sur "**Cancel**", elle renvoie `false`.

   ```
   let txt;
   if (confirm("Cliquez sur un bouton !")) {
     txt = "Vous avez appuyé sur OK !";
   } else {
     txt = "Vous avez appuyé sur Cancel !";
   }
   ```

3. **prompt()** : Prend la valeur d'entrée de l'utilisateur avec les boutons "**Ok**" et "**Cancel**". Il renvoie `null` si l'utilisateur ne fournit aucune valeur d'entrée.

   ```
   //syntaxe
   //window.prompt("sometext", "defaultText");

   let person = prompt("Entrez votre nom", "Harry Potter");

   if (person == null || person == "") {
     txt = "L'utilisateur a annulé l'invitation";
   } else {
     txt = "Bonjour " + person + " ! Comment allez-vous aujourd'hui ?";
   }
   ```

Screen (écran)

L'objet `screen` contient des informations sur l'écran sur lequel la fenêtre courante est affichée. Pour accéder à l'objet `screen`, nous pouvons utiliser la propriété `screen` de l'objet `window`.

```
window.screen;
//ou
screen;
```

L'objet `window.screen` possède différentes propriétés, dont certaines sont listées ici :

Propriété	Description
`height`	Représente la hauteur en pixels de l'écran
`left`	Représente la distance en pixels du côté gauche de l'écran actuel
`pixelDepth`	Propriété en lecture seule qui renvoie la profondeur de bits de l'écran
`top`	Représente la distance en pixels du haut de l'écran actuel
`width`	Représente la largeur en pixels de l'écran
`orientation`	Retourne l'orientation de l'écran telle que spécifiée dans l'API Orientation de l'écran
`availTop`	Propriété en lecture seule qui renvoie le premier pixel du haut qui n'est pas occupé par des éléments du système
`availWidth`	Propriété en lecture seule qui renvoie la largeur en pixels de l'écran sans les éléments du système
`colorDepth`	Une propriété en lecture seule qui renvoie le nombre de bits utilisés pour représenter les couleurs

Navigateur

Le `window.navigator` ou `navigator` est une propriété **en lecture seule** et contient différentes méthodes et fonctions liées au navigateur.

Voyons quelques exemples de navigation.

1. **navigator.appName**: Fournit le nom du navigateur utilisé

```
navigator.appName;
// "Netscape"
```

> ***Note:*** "Netscape" est le nom de l'application pour IE11, Chrome, Firefox et Safari.

2. **navigator.cookieEnabled**: Renvoie un booléen basée sur la valeur du cookie dans le navigateur.

```
navigator.cookieEnabled;
//true
```

3. **navigator.platform**: Fournit des informations sur le système d'exploitation du navigateur.

```
navigator.platform;
("MacIntel");
```

Cookies 🍪

Les cookies sont des petits fichiers stockés sur un ordinateur et accessibles par le navigateur.

La communication entre un navigateur Web et le serveur est sans état, ce qui signifie qu'elle traite chaque requête indépendamment. Il existe des cas où nous devons stocker les informations de l'utilisateur et mettre ces informations à la disposition du navigateur. Avec les cookies, les informations peuvent être récupérées de l'ordinateur chaque fois que cela est nécessaire.

Les cookies sont sauvegardés sous la forme de paires clé-valeur

```
book = Learn Javascript
```

La propriété `document.cookie` permet de créer, lire et supprimer des cookies. Créer un cookie est assez simple, vous devez fournir son nom suivi de sa valeur.

```
document.cookie = "book=Learn Javacript";
```

Par défaut, un cookie est supprimé quand le navigateur est fermé. Pour le faire persister, nous devons spécifier une date d'expiration (en temps UTC).

```
document.cookie = "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC";
```

Il est possible d'ajouter un paramètre pour préciser à quel chemin appartient le cookie. Par défaut, celui-ci appartient à la page courante.

```
document.cookie =
  "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
```

Voici l'example simple d'un cookie.

```
let cookies = document.cookie;
// un moyen simple de récupérer tous les cookies.

document.cookie =
  "book=Learn Javacript; expires=Fri, 08 Jan 2022 12:00:00 UTC; path=/";
// enregistrer un cookie.
```

Historique

Lorsque nous ouvrons un navigateur web et que nous surfons sur une page web, une nouvelle entrée est créée dans la pile de l'historique. Au fur et à mesure que nous naviguons sur des pages différentes, de nouvelles entrées sont ajoutées à cette pile.

Pour accéder à l'objet historique, nous pouvons utiliser

```
window.history;
// ou
history;
```

Pour naviguer entre les différentes piles d'historique, nous pouvons utiliser les méthodes `go()` , `foward()` et `back()` de l'objet **history**.

1. **go()**: est utilisé pour naviguer dans l'URL spécifique de la pile d'historique.

   ```
   history.go(-1); // fait reculer d'une page
   history.go(0); // rafraîchit la page actuelle
   history.go(); // rafraîchit la page actuelle
   history.go(1); // déplace la page vers l'avant
   ```

   > ***Note:*** la position de la page actuelle dans l'historique est **0**.

2. **back()** : Pour naviguer vers l'arrière de la page, nous utilisons la méthode `back()` .

   ```
   history.back();
   ```

3. **forward()**: Il charge la liste suivante dans l'historique du navigateur. Ceci est similaire à cliquer sur le bouton "avancer" dans le navigateur.

   ```
   history.forward();
   ```

Location

L'objet `location` est utilisé pour récupérer l'emplacement actuel (URL) du document et fournit différentes méthodes pour manipuler l'emplacement du document. On peut accéder à l'emplacement actuel par:

```
window.location
//ou
document.location
//ou
location
```

> ***Note*** : `window.location` et `document.location` font référence au même objet `location`.

Prenons l'exemple de l'URL suivante et explorons les différentes propriétés de `location` :

`http://localhost:3000/js/index.html?type=listing&page=2#title`

```
location.href; //affiche l'URL du document actuel
location.protocol; //affiche le protocole comme http : ou https :
location.host; //affiche le nom de l'hôte avec le port comme localhost ou localhost:3000
location.hostname; //affiche le nom d'hôte comme localhost ou www.example.com
location.port; //affiche le numéro de port comme 3000
location.pathname; //affiche le nom du chemin d'accès comme /js/index.html
location.search; //affiche la chaîne de requête comme ?type=listing&page=2
location.hash; //affiche l'identifiant du fragment comme #title
```

Chapitre 17

Événements

En programmation, les *événements* sont les actions ou les occurrences qui se produisent dans un système et pour lesquels le système vous indique que vous pouvez y répondre. Par exemple, quand vous cliquez sur un bouton de réinitialisation, cela vide le champ lié.

Les interactions issues du clavier, comme le fait d'appuyer sur une touche ont besoin d'être lues constamment pour connaître l'état de la touche avant que celle-ci soit relâchée. Réaliser d'autres calculs qui sollicitent le facteur temps peuvent vous faire manquer l'appui sur une touche. Ceci était utilisé par le passé pour gérer les entrées sur certaines machines primitives. Une étape supplémentaire consiste à utiliser une file d'attente (on parle alors de queue). Il s'agit d'un programme qui vérifie périodiquement les nouveaux événements et permet d'y réagir. Cette approche est appelée le *polling*.

Le principal inconvénient de cette approche est qu'on doit surveiller la file d'attente en permanence ce qui peut provoquer des désagréments dès lors qu'un événement est déclenché. Le meilleur mécanisme à mettre en oeuvre consiste à notifier le code dès lors qu'un événement se produit. C'est ce que font les navigateurs modernes en nous autorisant l'usage de fonctions spécifiques appelées *handlers* (qu'on peut traduire par "gestionnaire") charger de se déclencher lors de la survenue d'un événement particulier.

```
<p>Cliquez moi pour activer le handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("cliqué");
  });
</script>
```

Ici, le `addEventListener` est appelé sur l'objet `window` (objet intégré fourni par le navigateur) pour enregistrer un handler sur l'ensemble de `window` . La méthode `addEventListener` fonctionne la manière suivante: lorsque l'événement passé en premier argument (ici `click`) est détecté, le second argument est déclenché.

> Les écouteurs d'événements sont appelés uniquement quand l'événement survient dans le contexte de l'objet sur lequel ils sont enregistrés.

Some of the common HTML events are mentioned here.

Événement	Description
`onchange`	Quand l'utilisateur change ou modifie la valeur dans une entrée de formulaire
`onclick`	Quand l'utilisateur clique sur un élément
`onmouseover`	Quand le curseur de la souris entre sur un élément
`onmouseout`	Quand le curseur de la souris quitte un élément
`onkeydown`	Quand l'utilisateur presse une touche et relâche une touche
`onload`	Quand le navigateur a terminé son chargement

Il est courant pour les handlers enregistrés sur des noeuds avec enfants de recevoir également recevoir les événements de leurs enfants. Par exemple, si un bouton à l'intérieur d'un paragraphe est cliqué, les handlers enregistré sur le paragraphe recevront également l'événement clic. Dans le cas ou un gestionnaire est présent sur les deux éléments, celui le plus bas dans la hiérarchie sera déclenché le premier. On dit que l'événement se *propage* à l'extérieur, du noeud initial vers son parent et à la racine du document.

Le gestionnaire d'événement peut appeler la méthode `stopPropagation` de l'objet `event` pour empêcher les gestionnaires supérieurs de recevoir l'événement. C'est utile dans un grand nombre de cas, par exemple quand vous avez un bouton à l'intérieur d'un élément cliquable et que vous ne souhaitez pas déclencher l'événement extérieur lors du clic sur le bouton interne.

```
<p>Un paragraphe avec un <button>bouton</button>.</p>
<script>
  let para = document.querySelector("p"),
      button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler du paragraphe.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler du bouton.");
    event.stopPropagation();
  });
</script>
```

Ici, les handlers `mousedown` sont enregistrés aussi bien par le paragraphe que le bouton. Lors d'un clic sur le bouton, le handler du bouton appelle la méthode `stopPropagation` ce qui va empêcher le handler du paragraphe de se lancer à son tour.

Les événements peuvent avoir un comportement par défaut. Par exemple, les liens cliquables permettent d'atteindre une cible, vous êtes dirigé vers le bas d'une page en cliquant sur une flèche vers le bas, et ainsi de suite. Ces comportements par défaut peuvent être évités en appelant la méthode `preventDefault` sur l'objet `event` .

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
  let link = document.querySelector("a");
  link.addEventListener("click", event => {
    console.log("Non.");
    event.preventDefault();
  });
</script>
```

Ici, le comportement par défaut du lien est annulé, dans ce cas, se rendre vers la cible du lien en question lors du clic. Seul le message de la console est déclenché.

Chapitre 18

Les Promesses, async/await

Alors, mettez-vous dans la peau d'un écrivain et vous êtes actuellement en train de planifier la sortie de votre prochaine oeuvre. Les lecteurs qui s'intéressent à ce livre l'ajoutent à leur liste de souhait et sont notifiés lorsque le livre paraît ou même quand la date de sortie est repoussée. Le jour de la sortie, tout le monde peut l'acheter et tout le monde est content. Ceci est une analogie avec le fonctionnement des promesses (autrement appelées **Promises**) en JavaScript.

1. Le "*producteur*" est un bloc d'instructions qui prend du temps à s'exécuter et accomplit quelque chose. Ici, c'est l'écrivain.
2. Le "*consommateur*" est quelque chose qui consomme ce que produit le producteur une fois que c'est prêt. Dans notre parabole, il s'agit du "lecteur".
3. Ce qui lie le "*producteur*" et le "*consommateur*" peut être appellé une *promesse* parce qu'elle permet de fournir le résultat du "*producteur*" au "*consommateur*".

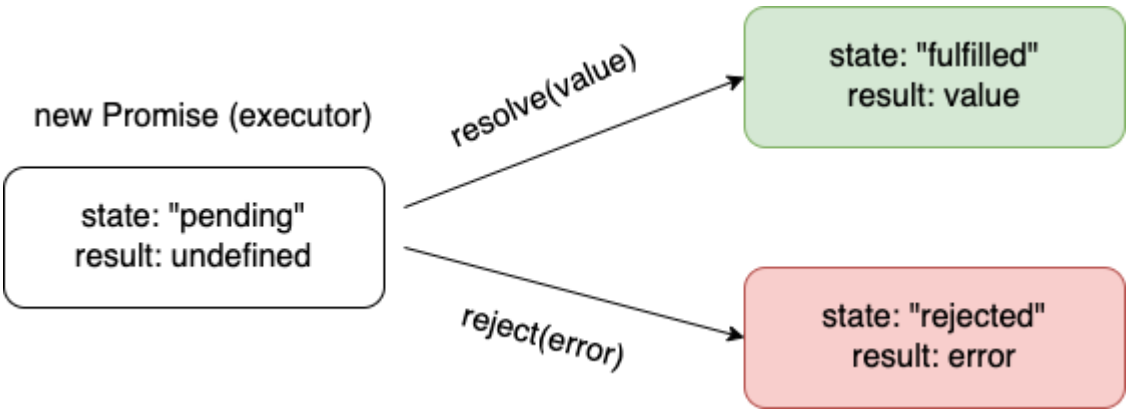
Promise

L'analogie qui nous avons faite est aussi vraie pour les objets de type `promise` en JavaScript. La syntaxe de création à partir du constructeur est comme suit:

```
let promise = new Promise(function(resolve, reject) {
  // Le producteur ici
});
```

Ici, une fonction est passée au constructeur `new Promise`, aussi connu sous le nom d'*exécutant* et s'exécute automatiquement dès l'appel du constructeur. Il contient le code de production qui retourne le résultat. `resolve` et `reject` sont les arguments fournis par JavaScript lui-même et sont appelés dans les conditions suivantes:

- `resolve(value):` Une fonction de callback qui renvoie `value` comme résultat.
- `reject(error)` : Une fonction de callback qui renvoie `error` en cas d'erreur.



Les propriétés internes de la promesse retournée par l'appel du constructeur `new Promise` sont :

- `state` - initialement `pending` (en cours), ensuite change à soit `fulfill` (accomplie) lorsque `resolve` est appelée, soit `rejected` (rejetée) lorsque c'est `reject` qui est appelée.

- `result` - initialement `undefined`, ensuite change à `value` en cas d'appel à `resolve` ou `error` lorsque c'est `reject` qui est appelée.

> Nous ne pouvons pas accéder aux propriétés : `state` et `result`. Les méthodes spécifiques au type Promise sont nécessaires pour gérer les promesses.

Exemple d'une promesse :

```
let promiseOne = new Promise(function(resolve, reject) {
  // La fonction est exécutée automatiquement après l'appel du constructeur

  // Là nous lui disons : attends une seconde et renvoie un signal "done" comme résultat
  setTimeout(() => resolve("done"), 1000);
})

let promiseTwo = new Promise(function(resolve, reject) {
  // La fonction est exécutée automatiquement lorsque la promesse est initiée (appel du constructeur)

  // Là nous lui disons: attends une seconde et renvoie une erreur: "Whoops!"
  setTimeout(() => reject(new Error("Whoops!")), 1000);
})
```

Ici, la promesse `promiseOne` est un exemple de promesse "*fulfilled*", (i.e *accomplie*) parce qu'elle est résolue (appel de *resolve()*) tandis que la promesse `promiseTwo` est une promesse "*rejected*" (i.e *rejetée*) parce que c'est la fonction *reject* qui fut appelée par l'exécuteur.

Une promesse qui n'est ni *rejected*, ni *resolved* est appelée *settled* en opposition à l'état *pending* initial. On peut consommer la promesse en utilisant les méthodes `.then` et `.catch`. Nous pouvons également ajouter `.finally` pour effectuer des actions après l'appel de l'une des méthodes précédentes.

```
let promiseOne = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve exécute la première fonction qu'on fourni à .then
promiseOne.then(
  result => alert(result), // affiche "done!" après 1 seconde
  error => alert(error) // Ne s'exécute pas
);

let promiseTwo = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject exécute la deuxième fonction qu'on fournit à .then
promiseTwo.then(
  result => console.log(result), // Ne s'exécute pas
  error => console.log(error) // Affiche l'erreur "Error: Whoops!" après une seconde
);

let promiseThree = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) est pareil que promise.then(null, f)
promiseThree.catch(alert); // montre l'erreur "Error: Whoops!" après une seconde
```

> Dans la méthode `Promise.then()`, toutes les deux fonctions de callback sont optionels.

Async/Await

Avec les promesses, nous pouvons utiliser le mot clé `async` pour déclarer les fonctions asynchrones qui vont renvoyer une promesse. Tandis que le mot clé `await` fait attendre JavaScript jusqu'à ce que la promesse s'achève et renvoie une réponse. Ces deux mots clés rendent l'écriture des promesses plus facile. Voici un exemple:

```
// Une fonction asynchrone f
async function f() {
  return 1;
}
// La résolution d'une promesse
f().then(alert); // 1
```

L'exemple ci-dessus peut être écrit comme suit :

```
function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

`async` fait en sorte que la fonction renvoie une promesse, cela permet d'encapsuler les valeurs qui ne sont pas des promesses. Avec `await`, nous pouvons indiquer à JavaScript d'attendre jusqu'à ce que la promesse soit résulue et renvoie sa valeur.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Welcome to Learn JavaScript!"), 1000)
  });

  let result = await promise; // Attend la résolution de la promesse.
  alert(result); // "Welcome to Learn JavaScript!"
}

f();
```

Le mot clé `await` net peut être utilisé que dans une fonction déclarée avec `async`.

Chapitre 19

Divers

Dans ce chapitre, nous parlerons davantage de sujets divers qu'on rencontre lorsqu'on commence à écrire du code. Les sujets sont les suivants:

- Les Templates Literals
- Le Hoisting
- Le Currying
- Les Polyfills et les Transpileurs
- Les Listes Chaînées
- L'Empreinte Globale
- Le débogage
- Développer et Déployer les applications JavaScript

Les Template Literals

Les templates literals sont des literales (expressions) délimitées par des backticks (accent grâve) `(``)` et sont utilisées lorsque l'on fait l'interpolation de varibles dans une chaîne de caractères.

```
let text = `Hello World!`;
// Les templates literals avec simple quote et double quote sur une même chaîne (sans échappement)
let text = `He's often called "Johnny"`;
// Les templates litterals en multiligne
let text =
`The quick
brown fox
jumps over
the lazy dog`;

// Les templates literals avec l'interpolation de variable
const firstName = "John";
const lastName = "Doe";

const welcomeText = `Welcome ${firstName}, ${lastName}!`;

// Les templates literals avec l'interpolation d'une expression
const price = 10;
const VAT = 0.25;

const total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
```

Le hoisting

Le Hosting est un comportement par défaut en JavaScript qui consiste à déplacer les déclaration en haut. En exécutant un code, le programme crée un contexte global d'exécution : création et exécution. Lors de la phase de création, JavaScript déplace les déclarations de fonctions et de variables en haut du fichier, ce principe est appelé hoisting.

> `Hoist` est un mot anglais qui signifie `éléver` ou `déplacer en haut` en français.

```
// Le hoisting
console.log(counter);
let counter = 1; // Déclenche une erreur ReferenceError: Cannot access 'counter' before initialization
```

L'erreur est déclenché car même si `counter` est présent en mémoire mais parce qu'il n'oa pas été initialisé. A cause du `hoisting`, la variable `counter` est élevée ici.

```
// function hoisting
const x = 20,
    y = 10;

let result = add(x,y); // ❌ Uncaught ReferenceError: add is not defined
console.log(result);

let add = (x, y) => x + y;
```

Ici, la fonction `add` function est déplacée en haut (`hoisting`) et initialisée avec `undefined` dans la mémoire lors de la phase création du contexte global d'exécution. C'est pourquoi, le déclenchement de l'erreur.

Le currying

Le `currying` est une technique avancée dans la programmation modulaire qui consiste à transformer une fonction avec des arguments multiples en une séquence de fonctions imbriquées. Cela transforme la fonction de `f(a, b, c)` en `f(a)(b)(c)` . Cela n'appelle pas la fonction, mais plutôt la transforme.

En d'autres termes, le `currying` convertit une fonction avec plusieurs paramètres en une chaîne de fonctions unaires (à un seul argument).

Pour avoir une meilleure compréhension du principe, créons une simple fonction `add` qui additionne trois arguments et retourne leur somme.

```
// Sans le currying
const add = (a, b, c)=>{
    return a+ b + c;
}
console.log(add(2, 3, 5))
```

Maintenant créons la version avec le `currying` de la fonction précédente qu'on va appeler `addCurrying` . Elle prend une seule entrée et renvoie une série de fonctions avec sa somme.

```
// Avec le currying
const addCurry = (a) => {
    return (b)=>{
        return (c)=>{
            return a+b+c
        }
    }
}
console.log(addCurry(2)(3)(5)) // 10
```

Nous constatons que les deux fonctions avec ou sans `currying` ont retourné le même résultat. Cette technique peut être bénéfique pous plusieurs raisons mentionnées ci-dessous:

- Cela permet d'éviter de passer la même variable encore et encore.
- Cela permet de diviser la fonctions en plusieurs morceaux avec une tâche spécifique, rendant la fonction moins sujette aux erreurs.
- Cela est utilisé dans la programmation fonctionnelle pour créer une fonction d'ordre supérieur. C'est-à-dire une fonction qui peut prendre comme paramètre d'autres fonctions et se charge de les exécuter. Ou encore une fonction qui retourne une fonction comme avec notre exemple ci-haut.

Les Polyfills et les Transpileurs

Le JavaScript évolue à chaque instant. Très souvent, les propositions de nouvelles fonctionnalités sont soumises, analysées et ajoutées à https://tc39.github.io/ecma262/ et ensuite incorporées dans la spécification. Il se peut qu'il y ait des différences sur la façon dont la fonctionnalité est implémentée dans les moteurs JavaScript en fonction du navigateur. Certains pourraient implémenter des version d'essai, tandis que d'autres attendent jusqu'à ce que la spécification entière sorte. Des soucis liés compatibilité avec les anciennes versions peuvent survenir lorsque des nouvelles fonctionnalités sont introduites.

Pour prendre en charge le code moderne dans les anciens navigateurs, nous utilisons deux outils : `transpileurs` et `polyfills` .

Les transpileurs

Un transpileur est un programme qui traduit un code *moderne* et le réécrit en utilisant les anciennes syntaxes pour faire en sorte que les les anciennes versions des moteurs d'exécution du JavaScript puisse le comprendre. Par exemple, l'opérateur de fusion nulle `??` a été introduit en 2020 et les navigateurs obsolètes ne pouvaient comprendre cette syntaxe.

Maintenant, c'est le job du transpileur de faire en sorte que l'opérateur de fusion nulle soit compréhensible par les navigateurs obsolètes.

```
// Avant l'exécution du transpileur
height = height ?? 200;

// Après la transpilation
height = (height !== undefined && height !== null) ? height: 200;
```

> Babel est l'un des tranpileurs les plus en avant. Dans le process de développement, nous pouvons utiliser les outils de compilation tels que webpack ou parcel pour transpiler le code.

Les polyfills

Il arrive qu'une nouvelle fonctionnalité ne soit pas disponnible dans les moteurs obsolètes. Dans ce cas, le code qui utilise la nouvelle fonctionnalité ne fonctionnera pas. Pour éviter cela, nous ajoutons les fonctionnalités manquantes (qu'on appelle `polyfills`). Par exemple, la méthode `filter()` qui a été introduite avec ES5 et qui n'est pas supportée par certains navigateurs. Cette méthode accepte une fonction et retourne un tableau qui contient uniquement les valeurs du tableau original pour lesquelles la fonction a retourné `true` .

```
const arr = [1, 2, 3, 4, 5, 6];
const filtered = arr.filter((e) => e % 2 === 0); // se débarasse des valeurs paires.
console.log(filtered);

// [2, 4, 6]
```

Un polyfill pour la méthode `filter` est:

```
Array.prototype.filter = function (callback) {
  const result = []; // le tableau à retourner
  for (let i = 0; i < this.length; i++) {
    // Appeller la fonction callback avec l'élement courant, l'indexe et le contexte.
    // Si le test est bon, ajouter l'élement dans le nouveau tableau.
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }
  return result; // Retourner le nouveau tableau (filtré)
}
```

Le site caniuse montre les versions mises à jour et les syntaxes supportées par différents moteurs JavaScript.

Listes Chaînées

Les Listes Chaînées représentent une des structures de données courantes qu'on retrouve dans tous les langages de programmation. Une Liste Chaînée est très simimlaire à un tableau normal en JavaScript. Il agit juste un petit peu différemment.

Ici, chaque élément dans la liste est un objet séparé contenant un lien ou pointeur vers l'élément suivant. Il n'y pas de méthode ou fonction natif en JavaScript propre aux Listes Chaînées, c'est donc à nous de l'implémenter. En voici un exemple d'implémentation.

```
["un", "deux", "trois", "quatre"];
```

Les types de Listes Chaînées

Il y a trois différents types de listes chaînées:

1. **Les Listes Simplement Chaînées :** Chaque noeud contient seulement un pointeur vers l'élément suivant (noeud).
2. **Les Listes Doublement Chaînées :** Il y a deux noeuds pour chaque noeud, un vers l'élément suivant et l'autre vers l'élement précedent.
3. **Les Listes Chaînées Circulaires :** Une Liste Chaînée Circulaire forme une boucle en ayant le dernier noeud qui pointe vers le premier ou tout autre élément derrière lui.

Ajouter à la fin

La méthode `add` est utilisée pour ajouter un noeud à la fin de la liste chaînée.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    append = (value) => {
        const newNode = new Node(value)
        let current = this.head
        if (!this.head) {
            this.head = newNode
            return
        }
        while (current.next) {
            current = current.next
        }
        current.next = newNode
    }
}
```

Retirer à la fin

La méthode `pop` est utilisée pour supprimer le dernier noeud de la liste chaînée.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    pop = () => {
        let current = this.head
        while (current.next.next) {
            current = current.next
        }
        current.next = current.next.next
    }
}
```

Ajouter au début

La méthode `prepend` est créée pour ajouter une valeur au début de la liste chaînée.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    prepend = (value) => {
        const newNode = new Node(value)
        if (!this.head) {
            this.head = newNode
        }
        else {
            newNode.next = this.head
            this.head = newNode
        }
    }
}
```

Retirer au début

La méthode `shift` est créée pour retirer le premier élément de la Liste Chaînée.

```
class Node {
    constructor(data) {
        this.data = data
        this.next = null
    }
}

class LinkedList {
    constructor(head) {
        this.head = head
    }
    shift = () => {
        this.head = this.head.next
    }
}
```

L'Empreinte Globale

Quant vous développeé un module, qui pourrait s'exécuter sur une page web, qui exécute également d'autres modules, alors vous devez faire attention au chevauchement du nom des variables.

Imaginez qu'on soit en train de développer un module `counter` :

```
let myCounter = {
  number: 0,
  plusPlus: function () {
    this.number = this.number + 1;
  },
  isGreaterThanTen: function () {
    return this.number > 10;
  },
};
```

> ***Note:*** Cette technique est utilisée la plupart du temps avec les closures afin de rendre l'état interne du module inaccessible depuis l'extérieur.

Le module maintenant consiste en une seule variable `myCounter` . Si un autre module sur la page web utilise des noms de variables tels que `number` ou `isGreaterThanTen` , chaque module sera protégé car il ne risque pas d'écraser les variables d'un autre.

Le débogage

Dans la programmation, des erreurs peuvent arriver lorsque vous écrivez du code. Ces erreurs peuvent être d'origine syntaxique ou logique. Le processus qui vous mène à retrouver ces erreurs peut être très pénible et prendre du temps; ceci est appelé le débogage.

Heureusement, la plupart des navigateurs modernes de nos jours ont des débogeurs internes. On peut les activer ou pas, forçant ainsi le signalement des erreurs. Il est aussi possible de mettre des points d'arrêts durant l'exécution du code pour stopper l'exécution et examiner les variables. Pour ce faire, il suffit d'ouvrir la fenêtre d'inspection et placer le mot clé `debugger` dans le code JavaScript. Au niveau de chaque breakpoint, le flux d'exécution s'interrompt permettant aux développeurs d'examiner les variables et continuer l'exécution du code.

Vous pouvez aussi utiliser la fonction `console.log()` pour laisser des messages sur la console (au niveau de la fenêtre de débogage).

```
const a = 5, b = 6;
const c = a + b;
console.log(c);
// Result : c = 11;
```

Développer et déployer des applications JavaScript

Développer et déployer des applications JavaScript implique une série d'étapes qui vont de la mise en place de l'environnement de développement au déploiement de l'application sur un serveur web ou la plateforme d'hébergement. Sur les lignes qui suivent, vous trouverez un guide détaillé qui vous assitera dans ce processus.

Mise en place de l'environnement de développement

Avant de débuter le développement, il est essentiel pour le développeur de s'assurer que Node.js et npm (Gestionnaire de Package Node) sont installés sur son système. Ces outils vitaux peuvent être téléchargés depuis le site officiel de Node.js.

En plus de cela, le développeur doit choisir un *éditeur de texte* approprié ou un *IDE* (Environnement de Développement Intégré) pour la programmation JavaScript. Les choix les plus populaires sont:

- Visual Studio Code
- Sublime Text
- WebStorm

L'installation de Node.js et npm donnent accès aux outils essentiels ainsi qu'aux librairies requises pour la programmation JavaScript. La sélection minutieuse de l'éditeur de text ou environnement de développement augmente la productivité et la qualité de code écrit.

Choisir un Framework JavaScript ou une Bibliothèque

Le choix d'un Framework JavaScript ou Bibliothèque dépent des exigences du projet qu'on a sous la main. Les développeurs peuvent opter de travailler avec des framework déjà établis tels que:

- React
- Angular
- Vue.js

Ou alors, certains choisissent d'utiliser `Vanilla JavaScript` (c'est-à-dire du JS pur), en fonction de la complexité et la demande du projet. La sélection est fondamentallement guidée par le besoin de structuration et des composant fonctionnels de base qui peuvent accélérer le processus de développement et renforcer la maintenabilité.

Créer un projet

L'initialisation d'un projet est facilitée par l'utilisation du `package manager` tel que `npm` ou `yarn` pour déclarer un nouveau projet. Par exemple la commande `npm init` peut être employée pour mettre en place un nouveau projet Node.js. L'emploi d'un gestionnaire de packets lors de l'initialisation du projet assure la structuration du projet et permet une gestion intelligente des dépendances. Cette approche aide de façon significative à la maintenance du projet, son organisation et sa gestion.

Le développement d'application

Au cours du processus de codage d'une application JavaScript, le développeur est conseillé de sciemment organiser les modules et les composants de son projet. Cette pratique est cruciale pour faciliter la maintenance dans le future. le développement d'un code organisé et modulaire est essentiel pour assurer que l'application demeure maintenable très facilement facilite la localisation des bugs. En plus de cela, cette approche favorise la réutilisabilité du code et encourage la collaboration entre les développeurs qui travaillent sur le projet.

Le test d'application

Le développeur est encouragé à créer des tests unitaires et des tests d'intégration en utilisant les frameworks de test tels que:

- Jest
- Mocha
- Jasmine

Cette pratique a pour but de vérifier que l'application s'exécute en fonction des besoins spécifiés. La création des tests démeure une pratique proactive d'identification et de prévention de bugs potentiels, et par conséquent insuffle la confiance sur l'application.

Builder l'application

Afin d'optimiser le code JavaScript, CSS et les ressources (fichiers) pour la production, il est recommendé d'utiliser un outil de compilation tel que :

- Webpack
- Rollup
- Parcel

Ces outils compressent et optimissent le code et les ressources, impliquant un temps de chargement reduit et une performance accrue. Mais en plus de cela, ils contribue à l'organisation du code et la séparation des tâches au sein de l'application.

La configuration du déploiement

Le développeur doit également prendre des décision intelligentes en ce qui concerne le lieu de déploiement. Les options de déploiement sont entre autres les traditionnels hébergeurs web, les services cloud tels que : Amazon Web Services, Google Cloud ou bien des plateformes telles que: Netlify, GitHub Pages.

Générer un build (ou version) de production

Générer une version de production de l'application suit l'exécution (ou la compilation) du projet. Cela implique généralement la compression (minification) et l'optimisation des ressources, resultant à un usage reduit de la bande-passante et une amélioration de l'expérience utilisateur. En plus de ça, générer une version de production nous assure que l'application s'exécute bien dans l'environnement de production.

Déployer l'application

Le processus de déploiment nécessite d'accepter strictement les instructions fournies par la plateforme d'hébergement. Cela peut impliquer l'utilisation des protocoles FTP, SSH et bien d'autres outils. S'accomoder aux bonnes pratiques durant le déploiement est crucial pour s'assurer l'accès en temps réel à la plateforme. Le déploiement peut être effectué de plusieurs façons, y compris les téléchargements manuels ou les processus déploiement automatisés.

Configuration de domain et de DNS (si applicable)

Pour ceux qui utilisent des nom de domaines personnalisés, configurer les paramètre de DNS pour diriger le traffic chez l'hébergeeur ou le serveur est une étape obligatoire. Cette configuration permet aux utilisateurs d'accéder à l'application à travers un nom de domaine facile à mémoriser. Cela permet également d'améliorer l'image de marque et l'accessibilité.

L'intégration continu et le déploiement continu (CI/CD)

Le développeur peut opter d'établir une procédure de Continuous Integration et de Continuous Deployment (CI/CD) (i.e intégration continu / déploiement continu). Cela peut être fait grâce à des outils CI/CD tels que Jenkins, Travis CI, CircleCI, ou GitHub Actions. L'automatisation des tests et du déploiement en réponse aux changement infini du code minimise les erreurs humaines potentielles et assure que la modification du code est soumise à un test rigoureux avant d'atteindre l'état de production. Cette approche augmente la qualité du code et sa fiabilité.

Le monitoring et la maintenance

La vigilance post-déploiement est requise pour monitorer une application en proie aux erreurs, aux bugs, aux attaques et bien d'autres. La mise à jour régulière des dépendances est essentielle afin d'augmenter la sécurité et tirer partie des nouvelles fonctionnalités. Cette approche proactive garantit que l'application conserve sa fiabilité, sa sécurité et sa performance au cours du temps.

Mise à l'échelle (si nécessaire)

Dans les scénarios où l'application connaît une croissance et une augmentation du trafic et de la charge de travail, la mise à l'échelle de l'infrastructure peut devenir impérative. Les fournisseur de services basés sur le Cloud proposent des solutions conçues pour répondre à ces exigences d'évolutivité. Ces solutions permettent à l'application de gérer de manière transparente des charges accrues tout en préservant les performances et la disponibilité.

Sauvegarde et restauration (si nécessaire)

La mise en œuvre de stratégies de sauvegarde et de restauration en cas d'incident est indispensable pour sauvegarder les données de l'application en cas de perturbations imprévues. Ces stratégies permettent d'assurer la continuité des activités et d'atténuer le risque de perte de données en cas d'événements inattendus.

Chapitre 21

Exercises

Dans ce chapitre, nous allons faire quelques exercices pour tester notre connaissance en JavaScript. Les exercices que nous aurons à faire sont listés ci-dessous :

- La console
- La multiplication
- La saisie utilisateur
- Les constantes
- La concaténation
- Les fonctions
- Les instructions contiditionnelles
- Les objets
- Le problème de FizzBuzz
- Récupérer les Titres!

La console

En JavaScript, nous utilisons `console.log()` pour écrire un message (le contenu d'une variable, une chaîne de caractères, etc.) sur la `console`. Cette procédure est principalement utilisée pour des raisons de débogage, soit pour laisser l'historique du contenu d'une variable durant l'exécution de notre programme.

Example:

```
console.log("Bienvenue dans Apprendre JavaScript, La Version Pour Débutants");
let age = 30;
console.log(age);
```

📝 Tâches :

- [ ] Ecrire un programme pour afficher `Hello World` sur la `console`. N'hésitez pas à essayer d'afficher d'autres messages !
- [ ] Ecrivez un programme pour afficher le contenu de variables sur la `console`.
  1. Déclarez une variable `animal` et assignez-lui la valeur `"dragon"`.
  2. Affichez la variable `animal` sur la `console`.

💡 Indices :

- Visitez le chapitre variables pour en apprendre plus sur les variables.

La multiplication

En JavaScript, nous pouvons effectuer la multiplication de deux nombres en utilisant l'opérateur arithmétique `(*)` appelé aussi *astérisque*.

Exemple:

```
let resultat = 3 * 2;
```

Ici, nous avons stocké le produit `3 * 2` dans une variable `resultat`.

📝 Tache :

- [ ] Ecrire un programme qui permet de stocker le produit `23` fois `41` dans une variable et affiche le résultat.

💡 Indices :

- Visitez le chapitre the Opérateurs de base pour mieux comprendre les opérations mathématiques.

La saisie utilisateur

En JavaScript, nous pouvons récupérer les entrées utilisateur et les stocker dans une variable. C'est un usage assez pratique si nous voulons mettre en place un programme dynamique et nous n'avons pas nécessairement besoin de connaître une valeur avant l'exécution du programme en question.

📝 Tâches :

- [ ] Ecrire un programme qui récupère la valeur sasie par l'utilisateur et y ajoute `10` avant de reafficher le résultat trouvé.

💡 Indices :

- Le contenu initial de la variable est saisi par l'utilisateur. La fonction `prompt()` enregistre la valeur saisie comme un `string`.
- Vous devez vous assurer que la chaîne de caractères est convertie en nombre pour effectuer les calculs.
- Visitez le chapitre Opérateurs de base pour en apprendre plus sur la conversion de type entre `string` et `int`.

Les constantes

Les constantes ont été introduites avec la norme ES6(2015). Pour déclarer une constante on utilise le mot clé `const` . Les variables déclarées avec `const` ne peuvent ni être modifiées (affectation) ou redéclarées.

Exemple:

```
const VERSION = '1.2';
```

📝 Tâches :

- [ ] Exécutez le programme écrit ci-dessous et corrigez l'erreur que vous verrez sur la console. Faites en sorte qu'après la correction, le résultat d'exécution sur la console soit `0.9` .

```
const VERSION = '0.7';
VERSION = '0.9';
console.log(VERSION);
```

💡 Indices :

- Visitez le chapitre sur les Variables pour plus d'informations concernant `const` et également faites des recherches à propos de "*TypeError assignment to constant variable*" sur les moteurs de recherches pour éviter ce genre d'erreur.

La concaténation

Dans tous les langages de programmation, la `concaténation` signifie tout simplement `"coller bout-à-bout"` une ou plusieurs chaînes de caractères à une autre. Par exemple, quand les chaînes "*World*" et "*Good Afternoon*" sont concatenées avec la chaîne "*Hello*", elles forment "*Hello World, Good Afternoon*". Nous pouvons concatener des chaînes de plusieurs façons en JavaScript.

Example:

```
const icon = '👋';

// En utilisant les templates
`hi ${icon}`;

// En utilisant la méthode join()
['hi', icon].join(' ');

// En utilisant la méthode concat()
''.concat('hi ', icon);

//  En utilisant l'opérateur +
'hi ' + icon;

// RESULTAT
// hi 👋
```

📝 Tâches :

- [ ] Ecrire un programme qui initialise deux variables `str1` et `str2` pour que le code affiche '*Hello World*' sur la console.

💡 Indices :

- Visitez le chapitre sur la concaténation des chaînes de caractères pour plus d'informations.

Les fonctions

Une fonction est un bloc de code destiné à exécuter une tâche spécifique quand elle est exécutée. Vous pouvez apprendre plus sur les fonctions en lisant le chapitre qui y est consacré.

📝 Tâches :

- [ ] Ecrire un programme qui crée une fonction appelée `estImpair` qui reçoit nombre et détermine si ce dernier est impair ou pas. Le résultat doit être un booléen: `true | false`
- [ ] Testez la fonction en lui passant le nombre `45345` comme argument. Le résultat attendu est `true`. Afficher cette valeur sur la `console`.

💡 Indices :

- Consultez le chapitre sur les functions pour comprendre davantage sur les fonctions et comment les créer.

Les expressions conditionnelles

La logique conditionnelle est vitale voire necessaire en programmation parce que cela nous assure que le programme fonctionne peut importe les données qu'il reçoit; elle permet également les branchements. Cet exercice consiste à implémenter diverses logiques conditionnelles dans des problèmes de la vie réelle.

📝 Tâches :

- [ ] Ecrire un programme qui demande "*Combien de km reste-t-il à faire ?*" et en fonction de la saisie de l'utilisateur et des conditions suivantes, affiche le résultat au niveau de la `console`.
  - S'il reste plus de 100 km, afficher `"Il me reste de la route à faire encore."`.
  - S'il reste plus de 50 km, afficher `"Je serais là dans 5 minutes."`.
  - S'il reste au plus 50 km `"Je suis en train de garer. Tu peux sortir."`.

💡 Indices :

- Lisez le chapitre sur la logique conditionnelle pour savoir comment utiliser les instructions conditionnelles et la logique conditionnelle.

Les objets

Les objets sont des collections de paires `clé` , `valeur` et chaque paire de clé-valeur est appelée propriété de l'objet en question. Ici, la `clé` peut être `string` , alors que la `valeur` peut être une valeur de n'importe quel type (`string` , `number` , etc.).

📝 Tâches :

Nous vous présentons la famille Dupon avec deux membres. Et pour chaque membre, ses informations sont données sous la forme d'un objet.

```
let personne = {
    prenom: "Jean",                    //String
    nom: "Dupon",
    age: 35,                          //Number
    genre: "masculin",
    nombresFetiches : [ 7, 11, 13, 17], //Array
    autrePersonneImportante: personne2       //Object,
};

let personne2 = {
    prenom: "Jeanne",
    nom: "Dupon",
    age: 38,
    genre: "feminin",
    nombresFetiches : [ 2, 4, 6, 8],
    autrePersonneImportante: personne
};

let famille = {
    nom: "Dupon",
    membres : [personne, personne2]       //Array of objects
};
```

- [ ] Trouvez un moyen d'afficher sur la `console` le prénom du premier membre de la famille `Dupon` .
- [ ] Remplacer le quatrième `nombresFetiches` du deuxième membre de la famille `Dupon` par `33` .
- [ ] Ajouter un nouveau membre à la famille en créant une nouvelle personne : (`Jimmy` `Dupon` , `13` , `masculin` , `[1, 2, 3, 4]` , `null`) et mettez à jour la liste.
- [ ] Afficher la `SOMME` des nombres fétiches de la famille `Dupon` sur la `console` .

💡 Indices :

- Visitez le chapitre sur les objets pour en apprendre plus sur comment fonctionnent les objets.
- Vous pouvez récupérer la propriété `nombresFetiches` de chaque personne dans l'objet famille.
- Pour chauque personne, une fois que vous avez récupéré ses nombres fétiches, utilisez une boucle pour sommer ses valeurs et par après, additionnez les sommes partielles obtenues.

Le problème de FizzBuzz

Le problème de *FizzBuzz* une question courante d'entretien sur le codage. Ici, l'intéressé devra créer un programme qui affiche *Fizz* et *Buzz* selon certaines conditions.

📝 Tâches :

- [ ] Ecrire un programme qui affiche tous les nombre entre 1 et 100 de façon à ce que les condtions suivantes soient remplies.
  - Pour les multiples de 3, afficher `Fizz` au lieu du nombre.
  - Pour les multiples de 5, afficher `Buzz` au lieu du nombre.
  - Pour les multiples de 3 et 5, afficher `FizzBuzz` .

Exemple:

```
/
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
....
....
98
Fizz
Buzz
/
```

💡 Indices :

- Consultez le chapitre sur les boucles pour comprendre comment fonctionnent les boucles et leur cas d'utilisations.

Récupérer les Titres!

Le problème *Récupérer les Titres!* est un problème intéressant où nous avons à récupérer le titre depuis une liste de livres. C'est un bon exercice pour comprendre le fonctionnement des tableaux et des objets.

📝 Tâches :

Etant donné un tableau d'objet qui reppresente des livres avec leurs auteurs.

```
const livres = [
  {
    titre: "Eloquent JavaScript, Third Edition",
    auteur: "Marijn Haverbeke"
  },
  {
    titre: "Practical Modern JavaScript",
    auteur: "Nicolás Bevacqua"
  }
]
```

- [ ] Ecrire un programme pour créer une fonction `recupererLesTitres` qui prends comme paramètre un tableau de livres et retourne un tableau contenant uniquement les titres.
- [ ] Invoquer la fonction sur notre variable afin de récupérer les titre et afficher le résultat sur la `console`.

💡 Indices :

- Consulter le chapitre sur les tableaux et les objets pour comprendre comment les tableaux et les objets fonctionnent.

References

Ballard, P. (2018). JavaScript in 24 Hours, Sams Teach Yourself. Sams Publishing.

Crockford, D. (2008). JavaScript: The Good Parts. O'Reilly Media.

Duckett, J. (2011). HTML & CSS: Design and Build Websites. Wiley.

Duckett, J. (2014). JavaScript and JQuery: Interactive Front-End Web Development. Wiley.

Flanagan, D. (2011). JavaScript: The Definitive Guide. O'Reilly Media.

Freeman, E., & Robson, E. (2014). Head First JavaScript Programming: A Brain-Friendly Guide. O'Reilly Media.

Frisbie, M. (2019). Professional JavaScript for Web Developers. Wrox.

Haverbeke, M. (2019). Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press.

Herman, D. (2012). Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript. Addison-Wesley Professional.

McPeak, J., & Wilton, P. (2015). Beginning JavaScript. Wiley.

Morgan, N. (2014). JavaScript for Kids: A Playful Introduction to Programming. No Starch Press.

Murphy C, Clark R, Studholme O, et al. (2014). Beginning HTML5 and CSS3: The Web Evolved. Apress.

Nixon, R. (2021). Learning PHP, MySQL & JavaScript: With jQuery, CSS & HTML5. O'Reilly Media.

Powell, T., & Schneider, F. (2012). JavaScript: The Complete Reference. McGraw-Hill Education.

Resig, J. (2007). Pro JavaScript Techniques. Apress.

Resig, J., & Bibeault, B. (2016). Secrets of the JavaScript Ninja. Manning Publications.

Robbins, J. N. (2014). Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics. O'Reilly Media.