

JavaScript

Table des matières

- Introduction
 - Le JavaScript, c'est quoi ?
 - Premiers pas en JavaScript
- Les variables
 - Déclaration de variable
 - Le typage
 - Les valeurs prédéfinies
- Interaction avec le DOM - Les Bases
 - Introduction au DOM
 - Interaction basique avec une page
- Les types de données
 - Number
 - BigInt
 - String
 - Boolean
 - Object
- Les Dates
 - L'objet de type « Date »
 - Méthodes d'un objet « Date »
- Les opérateurs
 - Les opérateurs d'égalité
 - Les opérateurs de comparaison
 - Les opérateurs logique
- Traitement conditionnelle
 - La structure « if...else »
 - La structure « switch »
 - L'opérateur « ternaire »
 - Les opérateurs logiques
 - L'opérateur « nullish coalescing »

Table des matières

- Interaction avec le DOM - Modifier la page
 - Créer un nouvel élément
 - Les méthodes d'interaction
- Traitement itératif
 - La structure « while »
 - La structure « do...while »
 - La structure « for »
 - Instructions « continue » et « break »
- Les collections indexées
 - Déclaration et utilisation
 - Les méthodes des collections
- Traitement itératif - Les collections
 - Les structures itératives
 - La méthode « Array.foreach »
 - La structure « for...of »
 - La structure « for...in »
- Les fonctions
 - Déclaration et utilisation
 - Les paramètres
 - Les fonctions fléchées
 - Les « Callback »
- Les timers
 - Timeout
 - Interval
- Interaction avec le DOM - Les événements
 - Abonnement aux events
 - Variable de l'événement
 - Types d'events des éléments du DOM
 - Types d'events de « Window »
- Les exceptions
 - L'instruction « try ... catch ... finally »
 - Déclencher une exception

Table des matières

- Les promesses
 - Déclaration et utilisation
 - Les méthodes des promesses
 - Les fonctions asynchrones
- Interaction avec le DOM - Les formulaires
 - Objectif des formulaires
 - Manipulation les formulaires
 - L'API de validation des contraintes
- Le destructuring
 - Objectif du destructuring
 - Décomposition de collection
 - Décomposition d'objet
- Annexes
 - Aperçu des méthodes sur les types
 - Les collections à clefs : Map et Set
 - Évaluation en court-circuit
- Références

Introduction

Le JavaScript, c'est quoi ?

Introduction

Le langage JavaScript

Créé en 1995 par Brendan Eich et intégré au navigateur Netscape, celui-ci est utilisé pour permettre de réaliser la validation des formulaires par le navigateur.

Ensuite, il a rapidement évolué pour devenir un langage puissant et polyvalent.

Quelques particularités du langage :

- Le code du JavaScript est interprété ou compilé à la volée.
- Il dispose d'un typage faible et dynamique.
- Il utilise le concept d'objet à prototype.

Utilisation du langage JavaScript

Bien que principalement connu en tant que langage de script Web, le langage Javascript n'est plus limité à une utilisation au sein d'un navigateur Web.

Aujourd'hui, le JavaScript permet de réaliser, par exemple :

- Site web dynamique (formulaire interactif, animation 3D, jeux vidéo ...)
- Application client Web, à l'aide des framework tels que NextJS, Angular,...
- Application server, grâce à des plateformes telles que NodeJS, Deno, Bun...
- Application Mobile, en utilisant les framework React Native, Ionic,...

Le standard ECMAScript

En 1997, le JavaScript est standardisé par la norme ECMAScript (ECMA-262).

Cette norme permet de définir :

- La syntaxe du langage.
- Les types de variables.
- Les objets et fonctions natives.
- Le mécanisme de traitement des erreurs.
- Etc...

Depuis 2015, le rythme de publication de la norme ECMAScript est annuel.

Site Web utile

- La documentation du JavaScript - MDN Web Docs
<https://developer.mozilla.org/fr/docs/Web/JavaScript>
- Spécification du langage ECMAScript
<https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
- Contribuer à la norme ECMAScript
<https://tc39.es/>
- Vérifier la compatibilité des navigateurs
<https://caniuse.com/?search=ECMAScript>

Développer en JavaScript

Dans le cadre du cours, nous allons faire du JavaScript destiné à une page Web.
Il nous faudra donc les deux outils suivants : Un éditeur de texte et un navigateur.

Il existe plusieurs éditeurs de texte adaptés au JavaScript :

- Visual Studio Code
- WebStorm ([gratuit pour une utilisation non commerciale](#))
- Vim
- Sublime Text
- ...

Premiers pas en JavaScript

Introduction

Emplacement du code

Pour intégrer du code JavaScript dans une page web, il y a plusieurs possibilités :

- Directement dans la page HTML

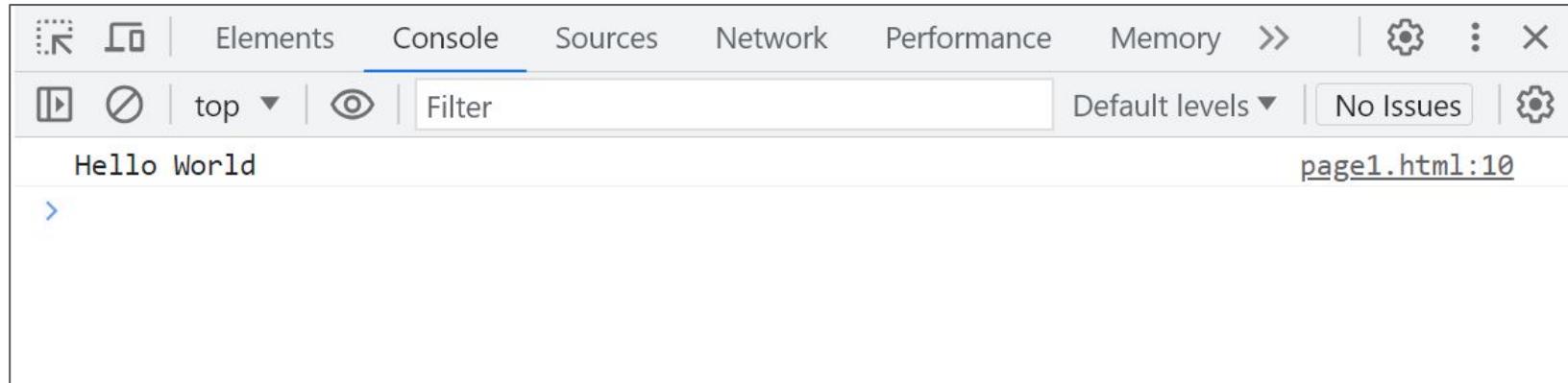
```
<script>
    const message = "Hello World";
    console.log(message);
</script>
```

- Depuis un fichier « js »

```
<script src="js/script.js" defer></script>
```

Déboguer son code

Les navigateurs web permettent d'avoir accès à un outil de développement.



La console permet de voir les messages envoyés par le code ou d'en exécuter.

Les raccourcis habituels pour l'ouvrir sont « F12 » ou « Ctrl Maj i ».

Utilisation de la console

Afficher un message dans la console :

```
console.log('Message log');
console.debug('Message de debug');
console.warn('Message d\'avertissement');
console.error('Message d\'erreur');
```

Les différentes méthodes permettent d'afficher un message dans la console avec un niveau de criticalité.

Résultat dans la console :

Message	Fichier	Ligne
Message log	script.js	1
Message de debug	script.js	2
Message d'avertissement	script.js	3
Message d'erreur	script.js	4

La console permet d'ajouter un filtre sur le niveau de criticalité des messages.

Interaction avec le navigateur

Les navigateurs Web permettent d'afficher des popups depuis le JavaScript.

Cela permet de :

- Afficher un message
- Réaliser une saisie
- Obtenir une confirmation

Bien que ces méthodes d'interaction soient simple à utiliser, il ne faut pas en abuser. Il sera toujours préférable d'interagir à l'aide de la page Web (via l'API du DOM), pour offrir une meilleure expérience à l'utilisateur.

Interaction avec le navigateur

Afficher un message via le navigateur :

```
alert('Hello World ❤');
```

Résultat dans le navigateur :

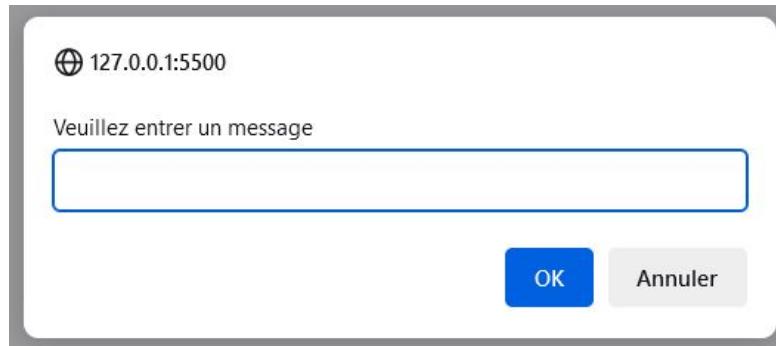


Interaction avec le navigateur

Permettre la saisie de l'utilisateur via le navigateur :

```
prompt('Veuillez entrer un message');
```

Résultat dans le navigateur :

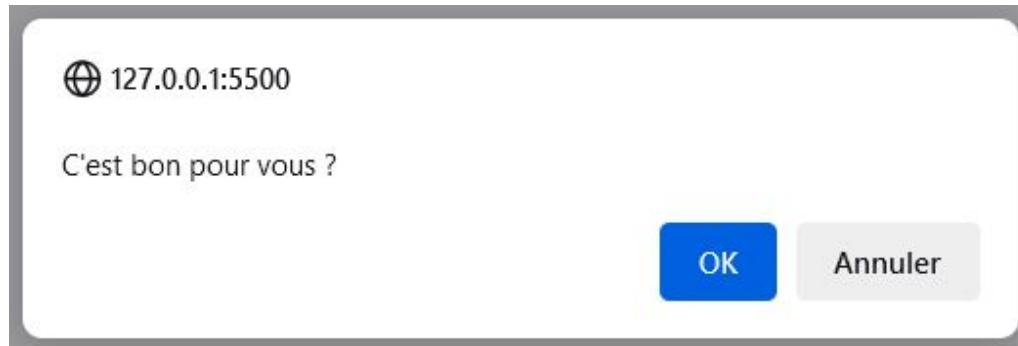


Interaction avec le navigateur

Obtenir une confirmation de l'utilisateur via le navigateur :

```
confirm('C\'est bon pour vous ?');
```

Résultat dans le navigateur :



Le mode strict

Depuis l'ECMAScript 5, il est possible d'utiliser une variante restrictive du JavaScript.

Cette variante est appelée le « Strict Mode », elle apporte les changements suivants :

- Élimination d'erreurs silencieuses, en les transformant en erreurs explicites.
- Interdiction de mots-clefs susceptibles d'être utilisés dans les futures versions.
- Correction des erreurs pour permettre une meilleure optimisation par les moteurs JavaScript (*Un même code sera donc exécuté plus rapidement en mode strict*).

Pour l'activer sur un fichier, ajouter « **'use strict';** » en première instruction.

Les commentaires

Ajouter des commentaires dans votre code JavaScript permettra de documenter votre code pour que celui-ci reste compréhensible.

- Commentaire sur une ligne

```
// Un commentaire
```

- Commentaire sur plusieurs lignes

```
/* Commentaire  
sur plusieurs  
lignes */
```

Bien évidemment, les commentaires ne sont pas interprétés par le moteur JavaScript.

Les variables

Déclaration de variable

Les variables

Déclaration de variable

Il existe trois manières de déclarer des variables en JavaScript :

- Variable globale modifiable « **var** »
- Variable locale modifiable « **let** »
- Variable locale immuable « **const** »

	Global	Portée limitée à la fonction	Portée limitée au bloc	Ré-assignable	Re-déclarable	Hoisting
var	✓	✓	✗	✓	✓	✓
let	✗	✓	✓	✓	✗	✗
const	✗	✓	✓	✗	✗	✗

Nommer une variable

Le nom d'une variable peut contenir les caractères suivants :

- des caractères alphanumériques
- des underscores
- des dollars

Le nom de la variable doit commencer par un dollar, un underscore ou une lettre.

Bonne pratique :

La convention de nommage pour les variables est le « camelCase ».

Le typage

Les variables

Le typage

Le JavaScript est un langage à typage dynamique.

Le type d'une variable sera défini dynamiquement lors de l'exécution en fonction du type du contenu qu'elle stockera.

De plus, il est possible que le type d'une variable change durant l'exécution

Bonne pratique :

Ne pas utiliser une même variable pour stocker plusieurs types de données !

Les types de valeurs

En JavaScript, ce sont des « pseudo-objets » qui permettent de définir les différents types de donnée avec leurs propriétés et leurs méthodes.

- Number : Un nombre (entier ou réel)
- BigInt : Un grand nombre entier
- String : Une chaîne de caractères
- Boolean : Type dont les valeurs possibles sont « true » ou « false »
- Undefined : Variable non initialisée
- Null : Variable initialisée sans aucune valeur
- Object : Élément complexe possédant plusieurs valeurs

Aperçu de quelques types de données

- Un nombre entier

```
const numberInteger = 42;
```

- Un nombre réel

```
const numberReal = 3.14;
```

- Une chaîne de caractères

```
const firstname = 'Della';
```

- Un booléen

```
const isValid = true;
```

- Un tableau de nombre

```
const tabNumber = [13, 42, 99, 42];
```

- Un tableau de chaîne de caractères

```
const tabName = ['Riri', 'Fifi', 'Zaza'];
```

- Une valeur « null »

```
const example = null;
```

- Un objet

```
const person = {  
    firstname : 'Balthazar',  
    lastname : 'Picsou'  
};
```

Obtenir le type d'une variable

L'opérateur « `typeof` » permet d'obtenir le type d'une variable.

Le type est renvoyé sous la forme d'une chaîne de caractères.

```
const response = 42;  
const responseType = typeof response;  
  
console.log(responseType); // → 'number'
```

Remarque : L'utilisation de l'opérateur « `typeof` » sur une variable de type « `Null` » renvoie la valeur « `'object'` » pour des raisons historiques.

Obtenir le type d'une variable - Les objets

L'opérateur « `typeof` » ne peut pas indiquer le prototype d'un objet (tableau, date, ...). Celui-ci renverra toujours le résultat « 'object' ».

Pour connaître le prototype d'une variable de type « `Object` », il est possible :

- D'obtenir le nom du prototype via la propriété « `constructor` »
- De tester l'appartenance à un prototype via l'opérateur « `instanceof` »

```
const now = new Date();

console.log(typeof now);           // → 'object'
console.log(now.constructor.name); // → 'Date'
console.log(now instanceof Date);  // → true
```

Les valeurs prédéfinies

Les variables

Undefined

La valeur « undefined » est une propriété globale qui représente la primitive undefined.

Cette valeur primitive est affectée automatiquement :

- aux variables qui viennent d'être déclarées sans avoir été initialisée.
- aux paramètres de fonction qui ne possèdent pas de valeur par défaut.

Null

La valeur « null » est un littéral qui représente l'absence intentionnelle de valeur pour une variable objet. C'est une valeur primitive du langage JavaScript.

Contrairement à la valeur « undefined », ce n'est pas une propriété globale.

NaN

La valeur « NaN » (Not a Number) est une propriété globale qui permet de représenter une valeur numérique invalide.

Particularité de cette propriété « NaN » :

- Elle est de type « 'number' »

```
console.log(typeof NaN); // → 'number'
```

- Les opérateurs d'égalité ne peuvent pas être utilisés sur celle-ci.

Pour tester cette valeur, il est possible d'utiliser la méthode globale « isNaN(...) »

```
console.log(NaN === NaN); // → false  
console.log(isNaN(NaN)); // → true
```

Infinity

La valeur « Infinity » est une propriété globale qui permet de représenter l'infini.

Comportement de la valeur « Infinity » :

- Tout nombre multiplié par Infinity sera égal à Infinity
- Zéro multiplié par Infinity sera égal à NaN
- NaN multiplié par Infinity sera égal à NaN
- Tout nombre divisé par Infinity sera égal à zéro
- Infinity divisé par Infinity sera égal à NaN
- Infinity divisé par un nombre sera égal à Infinity

Interagir avec le DOM

- Les Bases -

Introduction au DOM

Interagir avec le DOM - Les Bases

Apprentissage progressif du DOM

Bien que le DOM ne soit pas une matière qui fait partie du langage JavaScript, interagir avec celui-ci à travers l'utilisation de l'API de DOM est essentiel.

Dans le cadre du cours, nous aborderons progressivement l'utilisation de l'api du DOM.

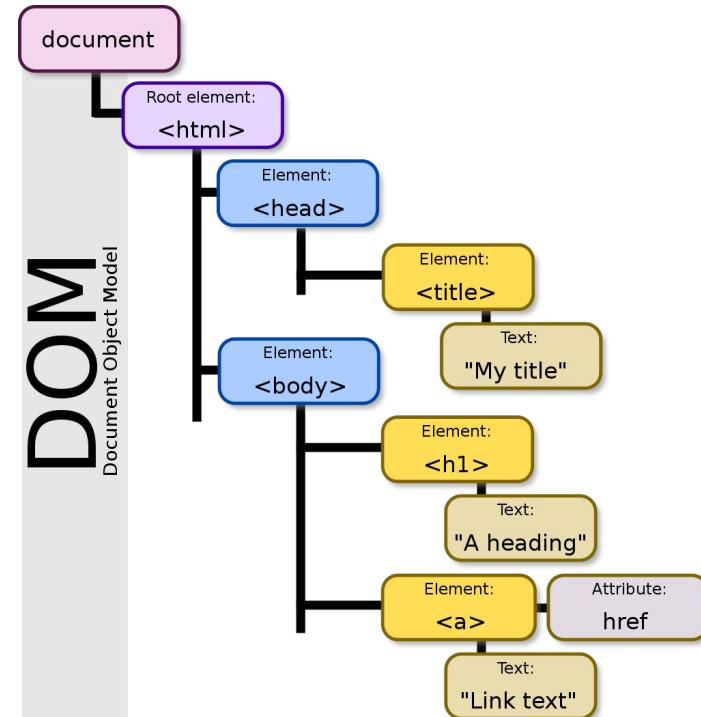
Les différents points de matière vu durant ce cours seront :

- Manipulation basique avec une page
- Modifier les éléments d'une page
- Interagir avec des événements
- Gestion de formulaire en JavaScript

C'est quoi le DOM ?

Le DOM (Document object Model) est la représentation sous la forme d'objet des données qui composent la structure et le contenu d'une page Web et de XML.

Le DOM n'est pas un langage de programmation, ce n'est qu'une API qui met à disposition des méthodes pour permettre de réaliser des interactions avec des pages Web et du XML.



Brève description des types fondamentaux du DOM

- Document

L'interface « Document » représente la page web chargée dans le navigateur. C'est le point d'entrée pour accéder au contenu de la page via l'arbre de DOM.

- Node

La classe abstraite « Node » est implémenté par chaque objet de « document ». Un « Node » peut être un nœud d'élément, un nœud texte ou un nœud d'attribut.

Brève description des types fondamentaux du DOM

- Element

La classe « Element » est la classe de base dont héritent tous les objets qui représentent des éléments d'un Document.

Elle contient les méthodes et les propriétés communes à l'ensemble des éléments.

- Attribute

Les valeurs de type « Attr » représentent un attribut d'un élément DOM.

C'est une référence à un objet qui expose une interface limitée aux attributs.

Brève description des types fondamentaux du DOM

- NodeList

Une « NodeList » est une collection de nœuds qui peut être renvoyée par des méthodes du DOM ou par la propriété « childNodes » d'un nœud.

Il est possible de récupérer un élément d'une « NodeList » par l'index :

- `list[index]`
- `list.item(index)`

Manipulation basique avec une page

Interagir avec le DOM - Les Bases

Récupérer un élément à l'aide de son Id

La méthode « getElementById(...) » retourne l'élément ayant l'attribut id spécifié

- HTML

```
<p id="display-message">Une balise d'un paragraphe</p>
```

- JavaScript

```
const displayMessage = document.getElementById('display-message');  
console.log(displayMessage);
```

Récupérer un élément à l'aide d'un sélecteur CSS

La méthode « `querySelector(...)` » retourne le premier élément qui correspond au sélecteur CSS

- HTML

```
<div class="demo">
  <p>Element 1</p>
  <p class="target">Element 2</p>
  <p>Element 3</p>
</div>
```

- JavaScript

```
const targetElement = document.querySelector('.demo > .target');
console.log(targetElement);
```

Récupérer des éléments à l'aide d'un sélecteur CSS

La méthode « querySelectorAll(...) » retourne une « NodeList » avec tous les éléments qui correspondent au sélecteur CSS.

- HTML

```
<ul class="people">
  <li>Della Duck</li>
  <li>Zaza Vanderquack</li>
  <li>Balthazar Picsou</li>
</ul>
```

- JavaScript

```
const people = document.querySelectorAll('ul.people > li:nth-child(odd)');
console.log(people);
```

Quelques propriétés des éléments du DOM

- .id

Permet d'obtenir ou modifier la valeur de l'attribut « id » de la balise.

- .innerHTML

Permet d'obtenir ou modifier le contenu html de la balise.

- .textContent

Permet d'obtenir ou modifier le contenu textuel de la balise.

Le code HTML est automatiquement échappé.

Quelques propriétés des éléments du DOM

- .className

Permet d'obtenir ou modifier la valeur de l'attribut « class » de la balise.

- .classList

Permet d'interagir avec la valeur de l'attribut « class » de la balise via les méthodes :

- ➔ add(...) : Ajoute les classes spécifiées
- ➔ remove(...) : Supprime les classes spécifiées
- ➔ contains(...) : Vérifie la présence de la classe spécifiée
- ➔ replace(..., ...) : Replace une classe par la nouvelle classe
- ➔ toggle(...) : Ajoute ou supprime la classe spécifiée

Quelques propriétés des éléments du DOM

- `.firstElementChild / .lastElementChild`

Permet d'obtenir respectivement le premier et le dernier élément de la balise.

- `.children`

Permet d'obtenir une collection avec tous les éléments enfants de la balise.

Remarque : Les propriétés ci-dessus permettent uniquement d'obtenir les nœuds de type « Element » de la balise ciblé. Les nœuds de type « Text » et « Commentaire » ne seront pas renvoyés.

Propriétés des balises de saisie de l'utilisateur

- .value

Permet d'obtenir ou modifier la valeur d'une balise « input / textarea / select ».

- .valueAsNumber

Permet d'obtenir ou modifier la valeur d'une balise « input » en type « number ».

Cette propriété n'est disponible que sur des balises de type « number » et « date ».

- .valueAsDate

Permet d'obtenir ou modifier la valeur d'une balise « input » en type « Date ».

Cette propriété n'est disponible que sur des balises de type « date ».

Propriétés des balises de saisie de l'utilisateur

- .checked

Permet d'obtenir ou modifier l'état une balise « input » de type « checkbox ».

- .disabled

Permet d'obtenir l'état de désactivation de la balise, et de modifier sa valeur.
Lorsqu'une balise est désactivé, elle n'est plus cliquable ou focusable.

- .readOnly

Permet de savoir si la balise est en lecture seul, et de modifier sa valeur.
Cette propriété fonctionne uniquement sur les balises de saisie de contenant du texte.

Interagir à l'événement « click » d'un élément

La méthode « `.addEventListener('click', ...)` » permet d'ajouter un code "en sommeil" sur l'événement clique de l'élément ciblé.

- HTML

```
<input id="input-name" type="text">  
<button id="btn-demo">Click here</button>
```

- JavaScript

```
const inputName = document.getElementById('input-name');  
const btnDemo = document.getElementById('btn-demo');  
  
btnDemo.addEventListener('click', function() {  
  const name = inputName.value;  
  console.log('Bonjour ' + name);  
});
```

Remarque : Les autres événements seront abordés ultérieurement dans le support.

Les types de données

Number

Les types de données

Déclaration d'une variable de type « Number »

Le type « Number » permet de stocker un nombre réel.

Exemple de déclaration de variable

```
const val1 = 42;  
const val2 = 4_357;  
const val3 = 9.99;  
const val4 = 2e3; // → 2000
```

Pour la lisibilité, le symbole « underscore » peut être utilisé comme séparateur.

Il est possible d'écrire un nombre sous la forme d'exposant décimal.

Exemple : 5e2 → 5×10^2

Convertir en nombre

Le JavaScript possède deux méthodes pour convertir une variable de type « String » :

- `parseInt(..., ...)`
- `parseFloat(...)`

En cas d'échec de la conversion, ces méthodes renvoient le résultat « `Nan` ».

Rappel :

Pour tester que la valeur est « `Nan` », il faut utiliser la méthode « `isNaN(...)` »

Convertir en nombre - Entier

La méthode « `parseInt` » permet d'obtenir la valeur entière d'une chaîne de caractères.

Celle-ci prend en paramètres :

- La chaîne de caractères à convertir
- La base (par défaut : 10)

```
const textNb = '42';
const nb1 = parseInt(textNb);      // → 42
const nb2 = parseInt(textNb, 10);  // → 42
const nb3 = parseInt(textNb, 16);  // → 66
const nb4 = parseInt('9.999');    // → 9

const textError = 'Dix';
const nb5 = parseInt(textError);   // → NaN
```

Convertir en nombre - Réel

La méthode « parseFloat » permet d'obtenir la valeur d'un réel.

Celle-ci ne prend qu'un seul paramètre :

- La chaîne de caractères à convertir.

```
const textReal = '3.14159';
const nb6 = parseFloat(textReal);      // → 3.14159

const textBad = '1,2345';
const nb7 = parseFloat(textBad);      // → 1

const textError2 = 'Demo';
const nb8 = parseFloat(textError2);   // → NaN
```

Les opérateurs arithmétiques

Le langage JavaScript supporte les opérateurs arithmétiques suivants :

- Addition
- Soustraction
- Multiplication
- Division
- Modulo
- Puissance

```
const nb1 = 14;
const nb2 = 3;

// Addition
const r1 = nb1 + nb2;    // → 17

// Soustraction
const r2 = nb1 - nb2;    // → 11

// Multiplication
const r3 = nb1 * nb2;    // → 42

// Division
const r4 = nb1 / nb2;    // → 4,66666...

// Modulo (Reste d'une division entière)
const r5 = nb1 % nb2;    // → 2

// Puissance
const r6 = nb1 ** nb2;  // → 2744
```

Les opérateurs arithmétiques

Le langage JavaScript permet de combiner un opérateur d'affectation avec un opérateur arithmétique.

```
let result = 14;
const nb = 3;

// Addition
result += nb;

// Soustraction
result -= nb;

// Multiplication
result *= nb;

// Division
result /= nb;

// Modulo
result %= nb;

// Puissance
result **= nb;
```

Les opérateurs d'incrément et de décrément

Le langage JavaScript permet de réaliser des actions d'incrémantation et de décrémantation à l'aide d'un raccourci d'écriture.

Deux possibilités :

- Post-Action :
Après l'utilisation de la variable.
- Pré-Action :
Avant l'utilisation de la variable.

```
// Post-Incrementation & Post-Decrementation
let nbA = 5, nbB = 3;
const r1 = nbA++;           // → r1: 5 / nbA: 6
const r2 = nbB--;           // → r2: 3 / nbB: 2

// Pré-Incrementation & Pré-Decrementation
let nbC = 5, nbD = 3;
const r3 = ++nbC;           // → r3: 6 / nbC: 6
const r4 = --nbD;           // → r4: 2 / nbD: 2

// Exemple
let nbE = 5, nbF = 3;
const r5 = (nbE-- * ++nbF) + nbE; // → r5 ?
```

Les constantes de « Number »

Quelques constantes utiles du pseudo-objet « Number » :

```
// La plus petite et la grande valeur numérique positive  
Number.MIN_VALUE;  
Number.MAX_VALUE;  
  
// Les valeurs entieres minimum et maximum représentable en JS  
Number.MIN_SAFE_INTEGER; // -(253 - 1)  
Number.MAX_SAFE_INTEGER; // (253 - 1)  
  
// Le plus petit intervalle possible entre deux valeurs numériques  
Number.EPSILON;
```

Les méthodes de « Number »

Quelques méthodes utiles du pseudo-objet « Number » :

```
// Détermine si la valeur est un entier
const r1 = Number.isInteger(42_000_000_000_123);      // → true

// Déterminer si la valeur peut être correctement représentée comme un entier
const r2 = Number.isSafeInteger(42_000_000_000_123); // → false

const v1 = 42.13;

// Détermine si la valeur est un nombre fini
const r3 = Number.isFinite(v1);      // → true

// Détermine si la valeur vaut NaN (Not a Number)
const r4 = Number.isNaN(v1);        // → false
```

Remarque : Contrairement aux méthodes globales, les méthodes “isFinite” et “isNaN” du pseudo-objet « Number » ne réalise pas de conversion avant de tester la valeur.

L'objet « Math »

L'objet « Math » est un élément natif du Javascript qui permet d'utiliser :

- Des constantes
- Des fonctions mathématiques

Quelques exemples :

```
// La valeur de PI (π)
const v1 = Math.PI;           // → 3.141592653589793

// Arrondir un nombre
const v2 = 4.2;
const a1 = Math.round(v1);   // → 4 (l'arrondi mathématique)
const a2 = Math.floor(v1);   // → 4 (l'entier inférieur ou égal)
const a3 = Math.ceil(v1);    // → 5 (l'entier supérieur ou égal)

// Générer un nombre pseudo-aléatoire
const v3 = Math.random();
```

BigInt

Les types de données

Déclaration d'une variable de type « BigInt »

Le type « BigInt » permet de stocker des nombres entiers trop grands qui ne peuvent pas être représentés correctement par le type « Number ».

Exemple de déclaration de variable

```
const bi1 = 42n;  
const bi2 = 13_000_000_000_000_123n;  
  
const bi3 = BigInt('9007199254740991');  
const bi4 = BigInt('0x1FA5B');
```

Les opérateurs

Les « BigInt » supportent les opérateurs arithmétiques, d'incrément et de décrément.

```
const nb1 = 42_000_000_000_013n;  
const nb2 = 2n;  
  
const r1 = nb1 * nb2;    // → 8400000000000026n
```

Remarque : Il n'est pas autorisé d'utiliser les opérateurs arithmétiques avec un autre type de donnée. Les deux valeurs doivent être de type « BigInt ».

String

Les types de données

Les chaînes des caractères

Les chaînes des caractères sont stockées dans des variables de type « String ».

Le contenu d'une chaîne de caractères doit être déclaré entre des guillemets.

C'est un type de donnée immuable, c'est-à-dire qu'il n'est pas possible de modifier son contenu une fois que celle-ci a été créée.

Exemple de déclaration de variable

```
const name1 = 'Della';
const name2 = "Balthazar";
```

Il est possible d'utiliser les simples ou doubles quotes

La propriété « length » permet d'obtenir la longueur de la chaîne de caractères

```
console.log(name1.length);    // → 5
```

Concaténation

Pour concaténer des chaînes de caractères, il est possible d'utiliser trois méthodes :

```
const firstname = 'Zaza';
const lastname = 'Vanderquack';

// Opérateur « + »
const text1 = 'Bienvenue ' + firstname + ' ' + lastname + ' !';

// Méthode « .concat »
const text2 = 'Bienvenue '.concat(firstname).concat(' ').concat(lastname).concat(' !');
const text3 = ''.concat(firstname, ' ', lastname, ' !');

// Template String
const text4 = `Bienvenue ${firstname} ${lastname} !`;
```

Le « Template String » nécessite d'utiliser les « backtick » et non des guillemets !

Boolean

Les types de données

Déclaration d'une variable de type « Boolean »

Le type « Boolean » permet de stocker une variable booléenne.

Exemple de déclaration de variable

```
const boolVrai = true;  
const boolFaux = false;
```

Conversion implicite en booléen

Le Javascript réalise une conversion implicite lorsqu'une valeur est évaluée dans un contexte booléen (par exemple : dans la condition d'une structure conditionnelle).

```
if(person) {  
    // ...  
}
```

Il existe donc deux types de valeur possible :

- Truthy : Les valeurs considérées comme vraies.
- Falsy : Les valeurs considérées comme fausses.

Les valeurs « Truthy »

Voici un tableau d'exemple de valeur « Truthy » de JavaScript

Valeur	Type	Description
true	Boolean	Le mot clef « true »
42 / -3.14 / 13n	Number / BigInt	Des valeurs numériques différentes de 0
Infinity / -Infinity	Number	Une valeur représentant l'infinie
{ }	Object	Un objet
[]	Array	Un tableau
<code>new Date()</code>	Object	Une instance du prototype « Date »
"Della" / "false"	String	Une chaîne de caractères

Les valeurs « Falsy »

Voici un tableau avec la liste complète des valeurs « Falsy » de JavaScript

Valeur	Type	Description
null	Null	L'absence de valeur
undefined	Undefined	La valeur primitive « undefined »
false	Boolean	Le mot clef « false »
NaN	Number	Une valeur numérique invalide
0 / -0 / 0n	Number / BigInt	La valeur numérique zéro (positive ou négative)
"" / " " / ``	String	Une chaîne de caractères vide
document.all	Object	Le seul objet « Falsy » en JS → Pour des raisons de compatibilité

Object

Les types de données

Création d'un objet en JavaScript

Le type « Object » permet de stocker un ensemble de valeur sous forme de propriété.

Exemple de déclaration d'objet

```
// La méthode « create »  
const o1 = Object.create();  
  
// Le constructeur  
const o2 = new Object();  
  
// L'initialisateur d'objet  
const o3 = {};  
  
const person = {  
    firstname: 'Della',  
    lastname: 'Duck'  
};
```

Interagir avec les propriétés d'un objet

Les « Property accessors » (Accesseurs de propriétés) permettent d'accéder aux différentes propriétés d'un objet à l'aide d'une notation avec un point ou des crochets.

```
const person = {  
    firstname: 'Test',  
    lastname: null  
};  
  
// Ecriture  
person['firstname'] = 'Zaza';  
person.lastname = 'Vanderquack';  
  
// Lecture  
const fullname = person.firstname + ' ' + person['lastname'];
```

L'opérateur « Optional chaining »

Lorsqu'on lit la valeur d'une propriété d'un objet « null » ou « undefined », une erreur est levée par le JavaScript.

```
> let person = null;  
    person.firstname;  
  
✖ ▶ Uncaught TypeError: Cannot read properties of null (reading 'firstname')
```

Pour éviter cette erreur, il est possible d'utiliser l'opérateur « Optional chaining ».

L'opérateur fonctionne comme la notation « . », sauf qu'il ne déclenche pas d'erreur sur les objets « null » ou « undefined », à la place, la valeur « undefined » est renvoyé.

```
> let person = null;  
    person?.firstname;  
  
◀ undefined
```

Les Dates

L'objet de type « Date »

Les Dates

Les objets de type « Date »

Le type « Date » permet de représenter une date/heure en JavaScript.

Une date est stockée sous la forme d'un entier qui correspond au nombre de millisecondes écoulées depuis le premier janvier 1970 (UTC).

Pour initialiser une variable date, il est possible d'utiliser le constructeur :

- Sans paramètre (date et heure courante).
- Avec des valeurs individuelles pour la date et l'heure.
- Avec un nombre qui représente une valeur temporelle (en millisecondes).
- Avec une chaîne de caractère au format ISO 8601 : YYYY-MM-DDTHH:mm:ss.sssZ

Les objets de type « Date »

Exemple de déclaration de variable de type « Date »

```
// Constructeur sans paramètre
const now = new Date();

// Valeurs individuelles pour la date et l'heure
// ! La valeur des mois commence à partir de 0
const d1 = new Date(2022, 5);                      // → 01/06/2022 00:00:00
const d2 = new Date(2014, 8, 9);                     // → 09/09/2014 00:00:00
const d3 = new Date(1995, 11, 4, 9, 42, 0);        // → 04/12/1995 09:42:00

// Nombre qui représente une valeur temporelle
const d4 = new Date(366485442000);                  // → 12/08/1981 19:30:42

// Chaîne de caractère au format ISO 8601
const d5 = new Date("1995-12-17T03:24:00");       // → 17/12/1995 03:24:00
```

Méthodes d'un objet « Date »

Les Dates

Getter et setter d'une date

- `.getDate()` / `. setDate(nbJour)`

Permet d'obtenir ou de modifier la valeur du jour du mois d'une date.

- `.getMonth()` / `.setMonth(indexMois)`

Permet d'obtenir ou de modifier l'index (0-11) représentant le mois d'une date.

- `.getFullYear()` / `.setFullYear(nbAnnée)`

Permet d'obtenir ou de modifier la valeur de l'année d'une date.

Getter et setter d'une date

- `.getHours() / .setHours(nbHeure)`

Permet d'obtenir ou de modifier la valeur de l'heure d'une date.

- `.getMinutes() / .setMinutes(nbMinute)`

Permet d'obtenir ou de modifier la valeur des minutes d'une date.

- `.getSeconds() / .setSeconds(nbSeconde)`

Permet d'obtenir ou de modifier la valeur des secondes d'une date.

Getter et setter d'une date

- `.getMilliseconds() / .setMilliseconds(nbMilliseconde)`

Permet d'obtenir ou de modifier la valeur des millisecondes d'une date.

- `.getDay()`

Permet d'obtenir la valeur du jour de la semaine (0-6) d'une date. (0 = Dimanche)

- `.getTime()`

Permet d'obtenir en millisecondes le temps écoulées depuis le premier janvier 1970.

Convertir une date en chaîne de caractères

- `.toString()`

Renvoie la date formatée en chaîne de caractère.

Exemple : « Tue Aug 19 1975 23:15:30 GMT+0000 (Coordinated Universal Time) »

- `.toISOString()`

Renvoie la date formatée avec la norme ISO 8601 : YYYY-MM-DDTHH:mm:ss.sssZ

- `. toJSON()`

Renvoie la date formatée en JSON (en utilisant la méthode « `toISOString()` »).

Convertir une date en chaîne de caractères

- `.toLocaleString(locale, options)`

Renvoie la date formatée en chaîne de caractère en tenant compte de la locale.
Les « options » permet de préciser le contexte de formatage.

- `.toLocaleDateString(locale, options)`

Même comportement que « `toLocaleString` » limité à la date.

- `.toLocaleTimeString(locale, options)`

Même comportement que « `toLocaleString` » limité à l'heure.

Les opérateurs

Les opérateurs d'égalité

Les opérateurs

Les différents opérateurs d'égalité et d'inégalité

Le langage Javascript possède deux types opérateurs : faible et strict.

- Égalité (==) / Inégalité (!=)

L'opérateur tente de convertir les valeurs si celle-ci sont de type différents.
Ensuite, il compare les valeurs et renvoie le résultat booléen.

- Égalité stricte (===) / Inégalité stricte (!==)

L'opérateur strict considère comme fausse une égalité entre deux valeurs de type différents. Si le type est identique, il compare les valeurs et retourne le résultat.

Danger de l'utilisation des opérateurs « faibles »

L'utilisation des opérateurs d'égalités faibles est déconseillée car il est plus compliqué de prévoir son résultat dû au mécanisme de conversion réalisé.

Il est donc conseillé de privilégier l'utilisation des opérateurs stricts.



Opérateurs de comparaison

Les opérateurs

Les différents opérateurs de comparaison

Le langage Javascript possède quatre opérateurs de comparaison :

- < : Plus petit
- <= : Plus petit et égal
- > : Plus grand
- >= : Plus grand et égal

Les types comparés par l'opérateur

Les opérateurs de comparaison permettent de tester :

- Deux valeurs de type numérique (Number ou BigInt)
- Deux chaînes de caractères (String)

Si les valeurs comparées sont d'un autre type, le moteur Javascript tentera de convertir les valeurs non-numériques et ensuite de réaliser une comparaison numérique.

NB : Si une des valeurs vaut « NaN », la comparaison sera toujours fausse.

Les opérateurs logiques

Les opérateurs

Les opérateurs logiques

Le langage Javascript possède 3 opérateurs logiques :

- `&&` : Le ET logique

Renvoie « true » si les deux opérandes sont « true ».

- `||` : Le OU logique

Renvoie « true » si au moins une des deux opérandes est « true ».

- `!` : Le NON logique

Renvoie la valeur booléenne inversée de son opérande.

Traitement conditionnel

La structure « if...else »

Traitement conditionnel

Le « if...else »

La structure « if...else » permet d'exécuter du code en fonction de condition(s).

Elle peut être constituée de 3 blocs de code :

- if : Condition initiale à tester [Obligatoire]
- else if : Condition à tester si les précédentes sont fausses
- else : Si aucune condition n'a été validée

Syntaxe :

<pre>if (condition) { instruction1 }</pre>	<pre>if (condition) { instruction1 } else { instruction2 }</pre>
--	--

Exemple de structure « if...else »

```
btnDemo.addEventListener('click', function() {  
  
    const nb = document.getElementById('input-nb').valueAsNumber;  
    const result = document.getElementById('result-msg');  
  
    if(isNaN(nb)) {  
        result.innerText = 'La valeur est invalide !';  
    }  
    else if (nb === 42){  
        result.innerText = 'Oh ! C\'est la réponse ultime !';  
    }  
    else {  
        result.innerText = 'Merci pour votre participation...';  
    }  
});
```

La structure « switch »

Traitement conditionnel

Le « switch »

La structure « switch » permet d'exécuter du code en fonction d'une égalité stricte

Syntaxe :

```
switch (expression) {  
    case valeurA:  
        instructions1;  
        break;  
    case valeurB:  
    case valeurC:  
        instruction2;  
        break;  
    default:  
        instructions_def;  
        break;  
}
```

Exemple de structure « switch »

```
switch (expr) {  
    case 'Pommes':  
        console.log('Pommes : 0.32 € le kilo.');//  
        break;  
    case 'Cerises':  
        console.log('Cerises : 3.00 € le kilo.');//  
        break;  
    case 'Mangues':  
    case 'Papayes':  
        console.log('Mangues et papayes : 2.79 € le kilo.');//  
        break;  
    default:  
        console.log('Désolé, nous n\'avons plus de ' + expr + '.');//  
}
```

L'opérateur « ternaire »

Traitement conditionnel

L'opérateur ternaire

Cet opérateur permet d'affecter une variable sur base d'une condition.

Il est souvent utilisé comme raccourci pour la déclaration d'un « if...else ».

Sa syntaxe est composée de 3 opérantes :

- Une condition suivie d'un point d'interrogation (?)
- L'expression à réaliser si la condition est « Truthy », suivie de deux-point (:)
- Finalement, l'expression à réaliser si la condition est « Falsy »

```
const result = condition ? exprIfTrue : exprIfFalse;
```

Exemple d'utilisation de l'opérateur ternaire

```
// Ternaire simple
const r1 = isValid ? 'Validé !' : 'En attente de validation...';

// Ternaire sur un objet nullable
const r2 = person ? person.firstname : 'Anonyme';

// Ternaire enchaînée
const r3 = (choice === 1) ? 'Pommes'
  : (choice === 2) ? 'Cerises'
  : (choice === 3) ? 'Mangues'
  : 'Papayes';
```

L'opérateur « nullish coalescing »

Traitement conditionnel

L'opérateur « nullish coalescing »

Cet opérateur permet de réaliser une condition avec la valeur « null » ou « undefined ».

Sa syntaxe se compose de 2 expressions séparées par deux points d'interrogation (??)

```
const result = student.yearResult ?? 5;
```

Il peut par exemple être utilisé pour :

- Affecter une valeur par défaut, si une variable ciblée n'a pas de valeur.
- Créer un court-circuit dans l'évaluation de l'instruction.

Remarque : contrairement à l'opérateur logique OU, celui-ci ne base pas son résultat sur les valeurs « falsy ». Il ne prend en compte que « null » ou « undefined » !

Interaction avec le DOM

- Modifier la page -

Créer un nouvel élément

Interaction avec le DOM - Modifier la page

Créer un élément HTML

La méthode « createElement » de « document » permet de générer un nouvel élément. Elle prend en paramètre le type de balise à créer.

```
const della = document.createElement('p');

della.textContent = 'Della Duck';
della.id = 'della-42'
della.classList.add('person');

console.log(della);
// <p id="della-42" class="person">Della Duck</p>
```

Remarque : L'élément sera créé en mémoire, mais il ne sera pas ajouté à la page Web.

Les méthodes d'interaction

Interaction avec le DOM - Modifier la page

Les méthodes de « Node » et « Element »

Les éléments HTML héritent des classes « Node » et « Element », celles-ci possèdent plusieurs méthodes qui permettent d'interagir avec la page Web.



Ces méthodes permettent de :

- Ajouter un élément enfant.
- Insérer un élément adjacent.
- Supprimer un élément.

Remarque : Un élément déjà présent dans la page sera déplacé, lorsqu'il est re-ajouté.

Ajouter un élément enfant [Node]

- `parentElement.appendChild(element)`

Permet d'ajouter un élément en tant que dernier enfant de la balise parent.

```
const parentElement = document.getElementById('people');
parentElement.appendChild(della);
```

- `parentElement.insertBefore(element, referenceElement)`

Permet d'ajouter un élément dans la balise parent avant un élément de référence.

```
const parentElement = document.getElementById('people');
parentElement.insertBefore(donald, della);
```

Ajouter un élément enfant [Element]

- `parentElement.append(element1, ..., elementN)`

Permet d'ajouter des éléments en tant que derniers enfants de la balise parent.

```
const parentElement = document.getElementById('people');
parentElement.append(balthazar, zaza);
parentElement.append(archibald);
```

- `parentElement.prepend(element1, ..., elementN)`

Permet d'ajouter des éléments en tant que premiers enfants de la balise parent.

```
const parentElement = document.getElementById('people');
parentElement.prepend(flagada, daisy);
parentElement.prepend(missTick);
```

Insérer un élément adjacent [Node]

- `referenceElement.insertAdjacentElement(position, element)`

Permet d'insérer un élément à une position par rapport à l'élément de référence.

La position où celui-ci est ajouté dépend de la valeur du paramètre :

- `beforebegin` : Avant l'élément de référence.
- `afterbegin` : En premier enfant à l'intérieur de l'élément de référence.
- `beforeend` : En dernier enfant à l'intérieur de l'élément de référence.
- `afterend` : Après l'élément de référence.

```
const parentElement = document.getElementById('people');
daisy.insertAdjacentElement('afterend', gontran);
```

Insérer un élément adjacent [Element]

- `referenceElement.before(element1, ..., elementN)`

Permet d'insérer des éléments avant l'élément référence.

```
balthazar.before(riri, fifi);
```

- `referenceElement.after(element1, ..., elementN)`

Permet d'insérer des éléments après l'élément référence.

```
balthazar.after(loulou, violette);
```

Supprimer un élément [Node / Element]

- `parentElement.removeChild(targetElement)`

Permet de supprimer l'élément ciblé de la balise parent.

```
const parentElement = document.getElementById('people');
parentElement.removeChild(archibald);
```

- `targetElement.remove()`

Permet de supprimer l'élément ciblé.

```
missTick.remove();
```

Traitement itératif

La structure « while »

Traitement itératif

Le « while »

La structure « while » permet de répéter l'exécution d'un bloc de code.

Fonctionnement de la boucle :

- Test de la condition.
- Si la condition est « vraie », exécution du code
- Retour à la condition

Syntaxe :

```
while (condition) {  
    instruction  
}
```

Exemple de structure « while »

```
btnDemo.addEventListener('click', function() {  
    const nb = document.getElementById('input-nb').valueAsNumber;  
    let count = 0;  
  
    while (count < nb) {  
        const elem = document.createElement('p');  
        elem.textContent = count;  
        zone.append(elem);  
  
        count++;  
    }  
});
```

La structure « do...while »

Traitement itératif

Le « do...while »

La structure « do...while » permet d'exécuter un bloc de code et de le répéter.

Fonctionnement de la boucle :

- Exécution du code
- Test de la condition.
- Si la condition est « vraie », retour à l'exécution du code

Syntaxe :

```
do {  
    instruction  
} while (condition);
```

Exemple de structure « do...while »

```
btnDemo.addEventListener('click', function() {  
    const nb = document.getElementById('input-nb').valueAsNumber;  
    let count = 0;  
  
    do {  
        const elem = document.createElement('p');  
        elem.textContent = count;  
        zone.append(elem);  
  
        count++;  
    } while (count < nb);  
});
```

La structure « for »

Traitement itératif

Le « for »

La structure « for » est une boucle qui possède un bloc de code d'initialisation et final.

Fonctionnement de la boucle :

- Exécution du code d'initialisation
- Test de la condition.
- Si la condition est « vraie », exécution du code de la boucle et du code final.
- Retour à la condition

Syntaxe :

```
for (initialisation; condition; final) {  
    instruction  
}
```

Exemple de structure « for »

```
btnDemo.addEventListener('click', function() {  
    const nb = document.getElementById('input-nb').valueAsNumber;  
    for (let count = 0; count < nb; count++) {  
        const elem = document.createElement('p');  
        elem.textContent = count;  
        zone.append(elem);  
    }  
});
```

Instructions « continue » et « break »

Traitement itératif

Les instructions d'une structure itérative

Les mots clefs « continue » et « break » sont deux instructions pouvant être utilisées dans une structure itérative.

- continue

Permet d'arrêter le traitement actuel et de passer à l'itération suivante de la boucle.

- break

Permet de terminer la boucle et de passer au code suivant.

Les collections indexées

Déclaration et utilisation

Les collections indexées

C'est quoi une collection indexée ?

Une collection indexée est une structure de données qui permet de stocker un ensemble de valeurs dans une variable et de les rendre accessibles à l'aide d'un index.

Valeur :	5	9	1	7	2	6	9	3
Index :	0	1	2	3	4	5	6	7
Longueur de la collection : 8								

La valeur d'index commence à 0 et celui-ci est une valeur entière positive.

Les collections indexées en JavaScript

En JavaScript, les collections indexées sont implémentées par l'objet global « `Array` ».

Particularités d'une collection de type « `Array` » :

- On peut accéder à leur contenu à l'aide de l'opérateur d'accès `[]`.
- Le nombre d'éléments est dynamique.
- Les valeurs stockées peuvent être de type différents (non conseillé).
- Les structures itératives permettent de parcourir leur contenu.
- Elles possèdent des méthodes.

Les objets de type « Array »

Exemple de déclaration de variable de type « Array »

```
// Déclaration à l'aide de la notation littérale :  
const emptyTab1 = [];  
const drake = ['Ludwig', 'Klara', 'Corvus', 'Anya'];  
  
// Déclaration à l'aide du constructeur :  
// - Sans paramètre  
const emptyTab2 = new Array();  
  
// - Un seul paramètre de type number (de 0 à 2^32-1)  
//   Crée un tableau avec X éléments vides  
const zone = new Array(5);  
  
// - Plusieurs paramètres  
const vals = new Array(5, 3, 1);  
const rapetou = new Array('La Science', 'La Gonflette', 'Burger');
```

Les objets de type « Array »

L'opérateur d'accès [] permet d'obtenir ou de modifier la valeur d'un élément du tableau à l'aide de son index.

```
const people = ['Della', 'Balthazar', 'Zaza', 'Gontran', 'Daisy'];

// Obtenir la valeur de l'élément à l'index 1
console.log(people[1]);    // → Balthazar

// Modifier l'élément à l'index 3
people[3] = 'Donald';

// Ajouter un nouvel élément (index cible > index max)
people[6] = 'Riri'
```

Si on affecte une valeur à un index supérieur à l'index maximum, l'élément est ajouté.

Les objets de type « Array »

Les méthodes « console.log(...) » et « console.table(...) » permettent de visualiser facilement le contenu du tableau dans le terminal

```
console.log(people);
```

```
▶ (7) ['Della', 'Balthazar', 'Zaza', 'Donald', 'Daisy', empty, 'Riri']
```

```
console.table(people)
```

(index)	Value
0	'Della'
1	'Balthazar'
2	'Zaza'
3	'Donald'
4	'Daisy'
6	'Riri'

```
▶ Array(7)
```

Les objets de type « Array »

La propriété « length » permet d'obtenir la longueur du tableau.

```
// Trois éléments
const tab1 = ['Riri', 'Fifi', 'Loulou'];
console.log(tab1.length);    // → 3

// Cinq éléments vides
const tab2 = new Array(5);
console.log(tab2.length);    // → 5

// Trois éléments et six éléments vides
const tab3 = ['Archiblad', 'Gabby'];
tab3[8] = 'Miss Tick';
console.log(tab3.length);    // → 9
```

Remarque : La propriété prend en compte les éléments vides du tableau.

Les méthodes des collections

Les collections indexées

Ajouter des éléments

- `.push(element1, ..., elementN)`

Permet d'ajouter un ou plusieurs éléments à la fin de la collection.

```
const personnages = ['Balthazar', 'Zaza'];
personnages.push('Donald', 'Daisy');      // → ['Balthazar', 'Zaza', 'Donald', 'Daisy']
personnages.push('Della');                 // → ['Balthazar', 'Zaza', 'Donald', 'Daisy', 'Della']
```

- `.unshift(element1, ..., elementN)`

Permet d'ajouter un ou plusieurs éléments au début de la collection

```
const personnages = ['Balthazar', 'Zaza'];
personnages.unshift('Donald', 'Daisy'); // → ['Donald', 'Daisy', 'Balthazar', 'Zaza']
personnages.unshift('Della');          // → ['Della', 'Donald', 'Daisy', 'Balthazar', 'Zaza']
```

Supprimer un élément

- `.pop()`

Permet d'obtenir et de supprimer le dernier élément de la collection

```
const personnages = ['Balthazar', 'Zaza', 'Della', 'Miss Tick'];
const target = personnages.pop();
console.log(personnages);           // → ['Balthazar', 'Zaza', 'Della']
console.log(target);                // → 'Miss Tick'
```

- `.shift()`

Permet d'obtenir et de supprimer le premier élément de la collection

```
const personnages = ['Balthazar', 'Zaza', 'Della', 'Miss Tick'];
const target = personnages.shift();
console.log(personnages);           // → ['Zaza', 'Della', 'Miss Tick']
console.log(target);                // → 'Balthazar'
```

Ajouter et supprimer des éléments

- `.splice(indexDebut, deleteCount [, element1, ..., elementN])`

Permet de supprimer de 0 à plusieurs éléments à partir de l'index reçu et ensuite d'ajouter de 0 à plusieurs éléments sur la collection.

```
const personnages = ['Balthazar', 'Miss Tick', 'Archibald', 'Della'];
const removed = personnages.splice(1, 2, 'Riri', 'Fifi', 'Loulou');
console.log(personnages);           // → ['Balthazar', 'Riri', 'Fifi', 'Loulou', 'Della']
console.log(removed);              // → ['Miss Tick', 'Archibald']
```

- `.toSpliced(indexDebut, deleteCount, element1, ..., elementN)`

Traitement identique à la méthode « `.splice(...)` » sauf que celle-ci crée une copie de la collection qui est modifiée et renvoyée.

Convertir en chaîne de caractères

- `.join([séparateur])`

Renvoie une chaîne de caractères en concaténant tous les éléments d'un tableau séparés par le séparateur (par défaut : « , »)

```
const personnages = ['Balthazar', 'Zaza', 'Donald', 'Daisy', 'Della'];
const text1 = personnages.join();
console.log(text1);           // → 'Balthazar,Zaza,Donald,Daisy,Della'
const text2 = personnages.join(' - ');
console.log(text2);           // → 'Balthazar - Zaza - Riri - Fifi - Loulou - Della'
```

- `.toString()`

Renvoie une chaîne de caractères représentant le tableau spécifié et ses éléments. Cette méthode génère le même résultat que la méthode « `.join()` ».

Trier les éléments

- `.sort([methodeComparaison])`

Permet de trier les éléments d'un tableau sur base d'une méthode qui doit retourner un nombre positif, négatif ou zéro.

```
const vals = [42, 13, -1, 4];
vals.sort(function(a, b) {
  return a - b;
});
console.log(vals);
// → [ -1, 4, 13, 42 ]
```

```
const names = ['Balthazar', 'Léon', 'Édith', 'Lena', 'Zaza', 'Lily'];
names.sort(function(a, b) {
  return a.localeCompare(b);
});
console.log(names);
// → [ 'Balthazar', 'Édith', 'Lena', 'Léon', 'Lily', 'Zaza' ]
```

- `.toSorted([methodeComparaison])`

Traitement identique à la méthode « `.sort(...)` » sauf que celle-ci crée une copie de la collection qui est modifiée et renvoyée.

Inverser l'ordre des éléments

- `.reverse()`

Permet d'inverser l'ordre des éléments d'un tableau sur base d'une fonction qui doit retourner un nombre positif, négatif ou zéro.

```
const names = ['Balthazar', 'Léon', 'Édith', 'Lena', 'Zaza', 'Lily'];
names.reverse();
console.log(names);
// → ['Lily', 'Zaza', 'Lena', 'Édith', 'Léon', 'Balthazar']
```

- `.toReversed()`

Traitement identique à la méthode « `.reverse(...)` » sauf que celle-ci crée une copie de la collection qui est modifiée et renvoyée.

Fusionner des tableaux

- `.concat(element1, ..., elementN)`

Permet de générer un tableau en fusionnant deux ou plusieurs tableaux.

```
const picsou = ['Balthazar', 'Hortense'];
const duck = ['Della', 'Donald'];
const guest = ['Zaza', 'Mamie Baba'];
const family1 = picsou.concat(duck).concat(guest);
const family2 = [].concat(picsou, duck, guest);
// → ['Balthazar', 'Hortense', 'Della', 'Donald', 'Zaza', 'Mamie Baba']
```

Extraire des portions de tableaux

- `.slice([start, [end]])`

Permet de générer un tableau qui contient la copie d'une portion du tableau cible.
La portion sélectionnée dans le tableau est définie par des index.

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];
// Index de début (inclusif) et de fin (exclusif)
const grp1 = personnages.slice(1, 3);
console.log(grp1); // → ['Balthazar', 'Zaza']
// Index de début uniquement
const grp2 = personnages.slice(3);
console.log(grp2); // → ['Della', 'Riri', 'Fifi', 'Loulou']
// Copie complète du tableau (pas d'index)
const grp3 = personnages.slice();
console.log(grp3); // → ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou']
```

Rechercher un élément

- `.includes(elementRecherché [, indiceDebut]);`

Permet de déterminer si le tableau contient une valeur recherchée.

```
const vals = [3, -5, 22, 39, -3, 22, 42, 14, 0];
const test1 = vals.includes(42);           // → true
const test2 = vals.includes(-5, 3);        // → false
const test3 = vals.includes(14, -3);       // → true
```

- `.some(methodeRecherche)`

Permet de déterminer si le tableau contient une valeur sur base d'une méthode de test.

```
const vals = [3, -5, 22, 39, -3, 22, 42, 14, 0];
const test4 = vals.some(function(nb) {
  return nb % 13 === 0;                  // → true
});
```

Obtenir un élément

- `.find(methodeRecherche) / .findLast(methodeRecherche)`

Permet d'obtenir la première ou dernière valeur correspondant à une méthode de test.

```
const personages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];
// Le premier personnage avec un prénom de moins de 5 lettres
const r1 = personages.find(function(prenom) {
  return prenom.length < 5
});
console.log(r1);                                // → Zaza
// Le dernier personnage avec un prénom de moins de 5 lettres
const r2 = personages.findLast(function(prenom) {
  return prenom.length < 5;
});
console.log(r2);                                // → Fifi
// Le premier personnage avec les lettres 'za' dans son prénom
const r3 = personages.find(function(prenom) {
  return prenom.includes('za');
});
// console.log(r3);                                // → Balthazar
```

Appliquer un filtre à la collection

- `.filter(methodeFiltre)`

Permet d'obtenir la copie filtrée d'un tableau sur base d'une fonction.

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];
// Obtenir tous les prénoms avec la lettre 'a'
const r1 = personnages.filter(function(prenom) {
  return prenom.includes('a');
})
console.log(r1);          // → ['Donald', 'Balthazar', 'Zaza', 'Della']

const vals = [3, -5, 22, 39, -3, 22, 42, 14, 0];
// Obtenir tous les nombres pairs
const r2 = vals.filter(function(nb) {
  return nb % 2 === 0;
});
console.log(r2);          // → [22, 22, 42, 14, 0]
```

Transformer la collection

- `.map(methodeTransformation)`

Permet d'obtenir la copie modifiée d'un tableau sur base d'une fonction.

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della'];
// Obtenir tous les prénoms en majuscule
const r1 = personnages.map(function(prenom) {
  return prenom.toLocaleUpperCase('fr-be');
})
console.log(r1);          // → ['DONALD', 'BALTAZAR', 'ZAZA', 'DELLA']

const vals = [3, -5, 22, 39, -3, 22, 42, 14, 0];
// Obtenir le triple de tous les nombres
const r2 = vals.map(function(nb) {
  return nb * 3;
});
console.log(r2);          // → [9, -15, 66, 117, -9, 66, 126, 42, 0]
```

Traitement itératif

- Les collections -

Les structures itératives

Traitement itératif - Les collections

Parcourir une collection avec une boucle

Pour parcourir les différents éléments d'une collection indexée, il est possible d'utiliser les structures itératives précédemment abordées :

- « while »
- « do...while »
- « for »

Pour cela, il sera souvent nécessaire de créer une variable qui nous permettra de définir la condition de fin de boucle, pour éviter de "sortir" de la collection.

Exemple avec la structure « while »

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];

// Variable avec la valeur du dernier index de la collection
let index = personnages.length - 1;

// Traitement itératif
while(index >= 0) {

    const perso = personnages[index];
    console.log(`> ${perso}`);

    index--;
}
```

Exemple avec la structure « for »

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];

// Traitement itératif
for(let i = 0; i < personnages.length; i++) {
    const perso = personnages[index];
    console.log(`> ${perso}`);
}
```

Les autres boucles pour les collections

Bien que ces structures itératives soient suffisantes, le langage JavaScript propose d'autres solutions pour manipuler les collections.

Celui-ci permet également d'utiliser d'autres boucles adaptées aux collections :

- La méthode « `forEach(...)` » du prototype `Array`
- La structure « `for .. of` »
- La structure « `for .. in` »

La méthode « Array.foreach »

Traitement itératif - Les collections

Le « forEach »

La méthode « forEach » du prototype « Array » permet d'exécuter un bloc de code pour chaque élément d'une collection à l'aide d'une méthode reçue en paramètre.

La méthode peut posséder jusqu'à 3 paramètres :

- Premier : La valeur de l'élément en cours de traitement.
- Deuxième (Optionnel) : L'indice de l'élément en cours de traitement.
- Troisième (Optionnel) : La collection utilisée par la méthode « forEach ».

Syntaxe :

```
collection.forEach(function (value, index, array) {  
    instruction  
});
```

Exemple de la méthode « forEach »

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];

// Traitement itératif
personnages.forEach(function (perso) {
    console.log(`> ${perso}`);
});
```

Limitation de la méthode « forEach »

La méthode « forEach » parcourt la totalité des éléments de la collection.

Il n'est pas possible d'arrêter le traitement d'une boucle « forEach », à part en déclenchant une exception (code d'erreur).

Si vous avez besoin d'arrêter la boucle, utiliser plutôt :

- Une boucle classique (for, while, do .. while)
- Une boucle for .. of
- Une boucle for .. in
- Une méthode des collections (filter, find, findIndex, ...)

La structure « for...of »

Traitement itératif - Les collections

Le « for .. of »

La structure « for .. of » permet de parcourir des objets itérables (Array, Map, String, ...).

Cette boucle permet de parcourir tous les éléments d'un itérable.

La variable initialisée par la structure reçoit une copie de la valeur parcourue.

Syntaxe :

```
for (const value of element) {  
    instruction  
}
```

Exemple de structure « for .. of »

```
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];

// Traitement itératif
for (const perso of personnages) {
    console.log(`> ${perso}`);
}
```

La structure « for...in »

Traitement itératif - Les collections

Le « for .. in »

La structure « for .. in » permet de parcourir les propriétés énumérables d'un objet.

Bien que cette boucle permet de parcourir les collections, il n'est pas recommandé de l'utiliser car :

- Elle ne permet pas de garantir l'ordre de parcours de la collection.
- Elle manipule les index (int) sous la forme de chaîne de caractères (string)

Syntaxe :

```
for (const key in element) {  
    instruction  
}
```

Exemple de structure « for .. in »

```
// - Exemple avec une collection
const personnages = ['Donald', 'Balthazar', 'Zaza', 'Della', 'Riri', 'Fifi', 'Loulou'];

// Traitement itératif
for (const key in personnages) {
    const perso = personnages[key];
    console.log(`> ${perso}`);
}

// - Exemple avec un objet
const students = { 'zava42': 'Zaza Vanderquack', 'ridu19': 'Riri Duck', 'cama01': 'Castor Major' };

// Traitement itératif
for (const key in students) {
    const student = students[key];
    console.log(`${key} -> ${student}`);
}
```

Les fonctions

Déclaration et utilisation

Les fonctions

Déclaration d'une fonction

Les fonctions permettent de stocker un bloc de code et de l'exécuter.

Les fonctions ont comme objectif :

- D'éviter de répéter du code.
- Diviser le code en un ensemble de fonction simple.

La fonction renvoie la valeur renournée par l'instruction « return », ou « undefined » si la fin du code a été atteint.

```
// Déclaration d'une fonction d'addition
function addTwoNumber(nb1, nb2) {
    // Traitement
    const result = nb1 + nb2;
    // Valeur renournée par la fonction
    return result;
}

// Déclaration d'une fonction de log
function logger(message) {
    // Traitement
    const now = new Date().toISOString();
    console.log(`[${now}] ${message}`);
}

// Utilisation des fonctions
const val = addTwoNumber(13, 29);
logger(`Le résultat vaut ${val}`);
```

Déclaration d'une fonction

Ils est également possible de déclarer une fonction en l'affectant à une variable. Cela permet, par exemple, de garantir son unicité en l'affectant à une constante.

```
// Déclaration d'une fonction dans une variable
const numberIsEven = function (nb) {
    return nb % 2 === 0;
}

// Utilisation de la fonction
const test1 = numberIsEven(42);
console.log(`Est ce que 42 est pair : ${test1 ? 'oui' : 'non'}');
```

Remarque : contrairement à la syntaxe classique, une fonction déclaré à l'aide d'une variable ne peut pas être utilisé avant sa déclaration.

Les paramètres

Les fonctions

Paramètres d'une fonction

Lors de la déclaration d'une fonction, il est possible de définir ses paramètres.

Les paramètres peuvent être :

- Une valeur primitive (number, ...)
- Un objet
- Une collection
- Une fonction

```
function getCartPrice(cart, tva) {  
    let total = 0;  
    for (const product in cart) {  
        total += (product.price * product.quantity);  
    }  
  
    const tax = total * (tva / 100);  
    return total + tax;  
}  
  
const cart = [  
    { name: 'Pomme', quantity: 10, price: 0.32 },  
    { name: 'Cerise', quantity: 5, price: 3.00 },  
    { name: 'Mangue', quantity: 5, price: 2.79 },  
];  
  
const price = getCartPrice(cart, 6);
```

Paramètres avec une valeur par défaut

Il est possible de définir une valeur par défaut aux paramètres d'une fonction.

Pour cela, il faut définir la valeur par défaut lors de la déclaration du paramètre à l'aide de l'opérateur égal.

Cette valeur sera utilisée si :

- Aucune valeur n'a été définie
- La valeur « undefined » a été passée

```
function sayHello(firstname = 'John', lastname = 'Smith') {  
    console.log(`Bonjour ${firstname} ${lastname} !`);  
}  
  
sayHello('Della');  
// → Bonjour Della Smith !  
  
sayHello('Zaza', 'Vanderquack');  
// → Bonjour Zaza Vanderquack !  
  
sayHello(undefined, 'Snow');  
// → Bonjour John Snow !  
  
sayHello();  
// → Bonjour John Smith !
```

Paramètre avec l'opérateur « rest »

L'opérateur « rest » permet de définir un paramètre qui peut obtenir un nombre indéfini d'arguments.

Les valeurs reçues seront stockées sous la forme d'une collection (Array).

La syntaxe de l'opérateur est « ... »

```
function combine(separator = ' > ', ...names) {  
  if(names.length === 0) {  
    return 'No data !';  
  }  
  
  let result = names[0];  
  
  for (let i = 1; i < names.length; i++) {  
    result += separator + names[i];  
  }  
  
  return result;  
}  
  
const r1 = combine(', ', 'Della', 'Riri', 'Zaza');  
// → "Della, Riri, Zaza"  
  
const r2 = combine(undefined, 'Balthazar', 'Donald');  
// → "Balthazar > Donald"
```

Les fonctions fléchées

Les fonctions

Déclaration d'une fonction fléchée

L'écriture fléchée permet de déclarer une fonction à l'aide de la syntaxe :

« (...)=>{ ... } ».

Note : lorsqu'il n'y a qu'un seul paramètre, les parenthèses sont optionnelles.

Lors de la déclaration d'une fonction fléchée, il est possible ne pas définir de corps à celle-ci.

Dans ce cas, il ne faut pas mettre les accolades et le « return » est implicite.

```
// Exemple de fonction fléchée
const example01 = () => {
  return 42;
}

const example02 = (x) => {
  return x + 5;
}

const example03 = (x, y) => {
  return x * y;
}

// Exemple de fonction "lambda"
const example04 = () => 42;
const example05 = (x) => x + 5;
const example06 = (x, y) => x * y;
```

Les « Callback »

Les fonctions

Principe de « Callback »

Les callback (fonction de rappel en français) consistent à envoyer une fonction “interne” à une autre fonction en tant qu’argument.

Cette fonction “interne” sera ensuite invoquée par la fonction pour réaliser une action.

Exemple de fonction (qu’on a utilisé) qui attendent un callback en paramètre :

- elementDOM.addEventListener('click', callback)
- array.find(callback)
- array.filter(callback)

Exemple d'implémentation d'un « Callback »

```
// Déclaration d'une fonction avec un callback
function numbersValidator(array, conditionCallback) {
    const results = [];
    for(const elem of array) {
        const nb = parseFloat(elem);
        if(!isNaN(nb) && (!conditionCallback || conditionCallback(nb))) {
            results.push(nb);
        }
    }
    return results;
}

// Utilisation de la fonction avec un callback
const elements = [42, 13, 9, 6, 40, 22];
const evenNumbers = numbersValidator(elements, (nb) => nb % 2 === 0);
```

Les timers

Timeout

Les timers

Déclencher un timer « Timeout »

La méthode « setTimeout » permet l'appel d'une fonction après un délai.

```
const timeoutId = setTimeout(functionRef, delay, param1, param2, /* ..., */ paramN);
```

Cette méthode prend en arguments :

- functionRef (**obligatoire**)
La fonction à exécuter.
- delay
Le délai à appliquer.
- param1, ..., paramN
Les arguments à envoyer à la fonction exécutée.

Cette méthode retourne un entier positif qui représente le « Timeout Id »

Arrêter un timer « Timeout »

La méthode « clearTimeout » permet d'annuler un timer généré par « setTimeout ».

```
clearTimeout(timeoutId);
```

Cette méthode prend en argument le « Timeout Id » du timer à annuler.

Remarque : Si le paramètre « Timeout Id » fourni n'identifie pas un timer précédemment créé, l'appel de cette méthode ne fait rien.

Interval

Les timers

Déclencher un timer « Interval »

La méthode « setInterval » permet l'appel répété d'une fonction espacé d'un délai fixe.

```
const intervalId = setInterval(functionRef, delay, param1, param2, /* ..., */ paramN);
```

Cette méthode prend en arguments :

- functionRef (**obligatoire**)
La fonction à exécuter.
- delay
Le délai à appliquer.
- param1, ..., paramN
Les arguments à envoyer à la fonction exécutée.

Cette méthode retourne un entier positif qui représente le « Interval Id »

Arrêter un timer « Interval »

La méthode « clearInterval » permet d'annuler un timer généré par « setInterval ».

```
clearInterval(intervalId);
```

Cette méthode prend en argument le « Interval Id » du timer à annuler.

Remarque : Si le paramètre « Interval Id » fourni n'identifie pas un timer précédemment créé, l'appel de cette méthode ne fait rien.

Les exceptions

L'instruction « try ... catch ... finally »

Les exceptions

L'instruction « try ... catch ... finally »

Le « try » permet d'exécuter un bloc de code dans lequel une erreur pourrait survenir.

Cette instruction ne peut pas être utilisée seule. Celle-ci doit être suivie de minimum une des clauses suivantes :

- La clause « catch » : Code permettant de réagir à l'exception.
- La clause « finally » : Code qui sera toujours exécuté.

Il peut donc y avoir trois formes suivantes pour cette instruction :

- try ... catch
- try ... finally
- try ... catch ... finally

La clause « catch »

La clause « catch » permet de définir un bloc de code qui s'exécutera si une exception se déclenche dans le « try ».

```
try {  
    myProcess(); // Fonction pouvant générer une exception  
}  
catch (err) {  
    console.error('Une erreur a été provoquée par « myProcess »');  
    console.error(err);  
}
```

La clause « catch »

S'il est nécessaire de réaliser un traitement différent en fonction du type d'erreur, il faut utiliser une structure conditionnelle avec l'opérateur « instanceof ».

```
try {
    myProcess(); // Fonction pouvant générer plusieurs types d'exceptions
}
catch (e) {
    if (e instanceof TypeError) {
        // Gestion de l'exception TypeError
    }
    else if (e instanceof RangeError) {
        // Gestion de l'exception RangeError
    }
    else {
        // Gestion des exceptions non spécifiées
    }
}
```

La clause « finally »

La clause « finally » permet de définir du code qui s'exécutera après l'exécution du code du « try » et de la clause « catch », si celle-ci est présente.

```
try {
    // Accède à une ressource
    const fileStream = readFile('...');
    // Utilisation de la ressource
    fileStream.write(data);
}
finally {
    // Toujours libérer la ressource
    fileStream?.close();
}
```

Déclencher une exception

Les exceptions

Déclencher une exception

L'instruction « throw » permet de déclencher une exception.

Celle-ci entraîne l'arrêt de l'exécution de la fonction actuelle et le premier bloc « catch » rencontré reprend la suite de l'exécution du script.

Si aucun « catch » n'a été trouvé, l'exécution du script se termine.

Lorsqu'on souhaite déclencher une exception, il est nécessaire de définir une expression avec la valeur à renvoyer

```
throw new Error("Boum 💣");
throw "Unexpected error";
throw 42;
```

Déclencher une exception - Best Practice

Bien qu'il n'y ai pas de limitation de type pour l'expression envoyée par le « throw », la valeur envoyée par l'exception devrait toujours être d'un type « Error ». La raison étant que le code qui traite l'exception pourrait s'attendre à utiliser certaines propriétés, telles que le message.

Les types d'exceptions pour du JS client :

- Les ECMAScript exceptions.
- La « DOMException »

Liste des exceptions de ECMAScript :

- Error
- EvalError
- RangeError
- ReferenceError
- SyntaxError
- TypeError
- URIError
- AggregateError

Interaction avec le DOM

- Les événements -

Abonnement aux events

Interaction avec le DOM - Les événements

Interagir aux événements d'une page Web

Les éléments d'une page Web peuvent provoquer des événements.

L'API de DOM permet au JavaScript de réagir à ceux-ci à l'aide de fonction.

Pour s'abonner à un event, il est possible d'utiliser deux techniques :

- La propriété de gestion d'event (*Spécification du DOM 0 - Ancienne manière*)
Permet de remplacer l'écouteur d'événement existant de l'élément.
- Utilisation des Event Listeners (*Spécification du DOM 2*)
Permet d'ajouter ou retirer l'enregistrement d'un nouvel écouteur d'événements.

La propriété de gestion d'événement - Via le code HTML

Pour lier une fonction à un événement depuis le code HTML, il faut :

- Créer une fonction dans le code JS
- Définir un attribut commençant par « on... » suivi du type d'événement dans la balise.
- Affecter à cet attribut, le nom de la fonction sans oublier les parenthèses.

- HTML

```
<button onclick="handleDemoEvent()">Click Here !</button>
```

- JavaScript

```
function handleDemoEvent() {  
    // Traitement ...  
}
```

La propriété de gestion d'évent - Via le code JS

Pour lier une fonction à un event à l'aide de la propriété de gestion d'event, il faut :

- Récupérer la balise HTML dans le code JS.
- Utiliser une propriété d'event commençant par « on... » suivi du type d'event.
- Affecter à cette propriété, le callback à exécuter.

```
const btnDemo = document.getElementById('btn-demo');

btnDemo.onclick = handleDemoEvent;
// ou
btnDemo.onclick = function () {
    // Traitement ...
}
```

Les Event Listeners

S'abonner à des events à l'aide de « Event Listener » est la manière d'enregistrer un écouteur d'événements telle que spécifiée dans le DOM du W3C.

Les Event Listeners ont comme avantages :

- de pouvoir ajouter plusieurs « listener » sur un même événement,
- de supprimer les instances dupliquées si des « listener » identiques sont ajoutés plusieurs fois avec les mêmes paramètres,
- de fonctionner avec tout élément DOM (pas uniquement les éléments HTML).
- de pouvoir retirer un ou plusieurs « listener » d'un élément.

Les Event Listeners - S'abonner à un événement

La méthode « addEventListener » permet d'ajouter un « Listener » à un élément.

Cette méthode prend en arguments :

- Le type d'événement
- Une fonction de callback
- Un objet « options » ou un booléen « useCapture » (Optionnel)

```
const btnDemo = document.getElementById('btn-demo');

btnDemo.addEventListener('click', handleDemoEvent);
// ou
btnDemo.addEventListener('click', function () {
    // Traitement ...
});
```

Les Event Listeners - S'abonner à un événement

L'objet « options » permet de spécifier les caractéristiques du « listener ».

Les options disponibles sont :

- **once** (Booléen)
Permet d'indiquer que le listener doit être supprimé après son premier appel
- **passive** (Booléen)
Permet de spécifier que le listener ne doit pas appeler la méthode « preventDefault »
- **capture** (Booléen)
Permet de définir que l'événement sera distribué au « Listener » avant d'être distribué aux « EventTarget » situés en dessous dans l'arborescence DOM.

Les Event Listeners - Se désabonner à un événement

La méthode « `removeEventListener` » permet de retirer un « Listener » à un élément.

Cette méthode prend en arguments :

- Le type d'événement
- Une fonction de callback
- Un objet « options » ou un booléen « `useCapture` » (**Optionnel**)

```
const btnEvent = document.getElementById('btn-event');
btnEvent.removeEventListener('click', handleDemoEvent);
```

Remarque : Si l'abonnement du listener a été configuré avec l'option de « capture » (*via l'objet `options` ou `useCapture`*), il sera nécessaire de le définir lors du désabonnement.

Variable de l'événement

Interaction avec le DOM - Les événements

La variable de l'événement

Une fonction abonnée à un événement reçoit en paramètre la variable d'évènement.

Cette variable permet :

- D'accéder à des information relative à l'évènement
- Déclencher des comportement sur l'évènement.

```
const btnEvent = document.getElementById('btn-event');

btnEvent.addEventListener('click', (event) => {
    console.log('Variable d\'event', event);
});
```

Quelques propriétés de la variable de l'événement

- .type

Permet d'obtenir le type d'événement déclenché.

- .target

Identifie l'élément à partir duquel l'événement s'est déclenché.

- .currentTarget

Identifié la cible pour l'événement lorsque celui-ci traverse le DOM (via la propagation). Celle-ci fait toujours référence à l'élément sur lequel l'abonnement a été réalisé.

Quelques méthodes de la variable de l'événement

- `.preventDefault()`

Annule l'action par défaut de l'événement.

- `.stopPropagation()`

Arrête la propagation de l'événement aux éléments parents.

- `.stopImmediatePropagation()`

Arrête la propagation de l'événement à élément en cours et aux éléments parents.

Types d'events des éléments du DOM

Interaction avec le DOM - Les événements

Liste non exhaustive des events

- click

Clique sur un élément.

- keydown

Une touche est pressée.

- dblclick

Double clique sur un élément.

- keyup

Une touche est relâchée.

- contextmenu

Ouverture du menu via le clique droit.

- ~~keypress~~(deprecated)

Une touche de caractère est pressée.

Liste non exhaustive des events

- focus / focusin

L'élément reçoit le focus.

La différence entre les événements de focus est que « focus / blur » ne se propage pas, à l'inverse de « focusin / focusout ».

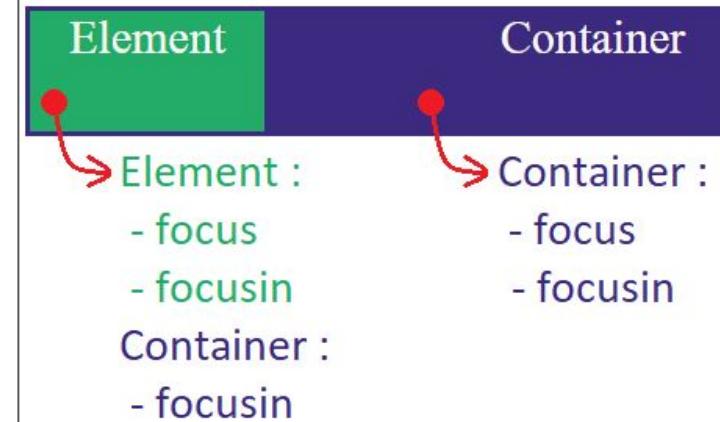
- blur / focusout

L'élément perd le focus.

- mousemove

Le curseur est déplacé.

Donner le focus à un élément :



Liste non exhaustive des events

- change

La valeur d'une balise a été modifiée.

- input

La valeur d'une balise est modifiée,
cet event se déclenche en temps réel.

- invalid

Le contenu d'un balise input est invalide.

- submit

Le formulaire a été validé.

- reset

Le formulaire a été réinitialisé.

- formdata

Génération d'un objet « FormData »,
celui-ci contient les données du form.

Types d'events de « Window »

Interaction avec le DOM - Les événements

Liste non exhaustive des events de Window

- load

La page et toutes ses dépendances sont complètement chargées.

- online

Le navigateur a obtenu l'accès au réseau et « `Navigator.onLine` » passe à true.

- beforeunload

La page va être déchargée. Permet d'interrompre la fermeture de celle-ci.

- offline

Le navigateur perd l'accès au réseau et « `Navigator.onLine` » bascule à false.

- languagechange

La langue préférée (via le navigateur) de l'utilisateur a été changée.

Les promesses

Déclaration et utilisation

Les promesses

C'est quoi une promesse ?

Le mécanisme des promesses permet à une méthode de renvoyer un objet de type « Promise », alors que celle-ci n'a pas encore terminé sa tâche.

Cet objet représente littéralement la promesse faite par cette méthode de transmettre un résultat une fois sa tâche accomplie.

Les promesses sont généralement utilisées pour réaliser des actions asynchrones en JavaScript et cela permet d'éviter d'avoir recours au mécanisme des callbacks.

L'objet de type « Promise »

Un objet « promise » représente donc une valeur qui peut être disponible maintenant, dans le futur, voire jamais.

Cet objet peut être dans l'un de ces trois états :

- pending (en attente) : la promesse est en cours de traitement.
- fulfilled (tenue) : la promesse est résolue et a réussi.
- rejected (rompue) : la promesse est résolue et a échoué.

Création d'une promesse

Pour créer un objet du type « Promise », il est nécessaire de l'instancier à l'aide de son constructeur en lui fournissant en paramètre une fonction de callback.

Cette fonction contiendra le code à exécuter et possède jusqu'à deux paramètres :

- **resolve :** Promesse « fulfilled »
*Fonction pour terminer une promesse avec succès.
Celle-ci prend un paramètre pour renvoyer une valeur.*
- **reject :** Promesse « rejected »
*Fonction pour terminer une promesse avec une erreur.
Celle-ci prend un paramètre pour renvoyer la raison de l'échec de la promesse.*

Exemple de création d'une fonction avec une promesse

```
function examplePromise(dividend, divisor) {  
    // La fonction renvoie un objet du type « Promise »  
    return new Promise((resolve, reject) => {  
        // Utilisation d'un timer de 500 ms pour "simuler" du code asynchrone  
        setTimeout(() => {  
            // Promesse rompue si le diviseur vaut zéro  
            if(divisor === 0) {  
                reject(new Error('Division by zero !'))  
            }  
            // Traitement  
            const result = dividend / divisor;  
            // Promesse tenue  
            resolve(result);  
        }, 500);  
    });  
}
```

Utilisation d'une promesse

Pour utiliser une promesse, il est possible d'utiliser trois méthodes sur cet objet :

- `.then(resolveCallback [, rejectCallback])`

Permet de traiter le succès et l'erreur (si le callback est défini) d'une promesse.

- `.catch(rejectCallback)`

Permet de traiter l'erreur d'une promesse.

- `.finally(finallyCallback)`

Permet de traiter la complétion d'une promesse (en succès ou en erreur).

Ces méthodes renvoient également une promesse, ce qui permet de les enchaîner.

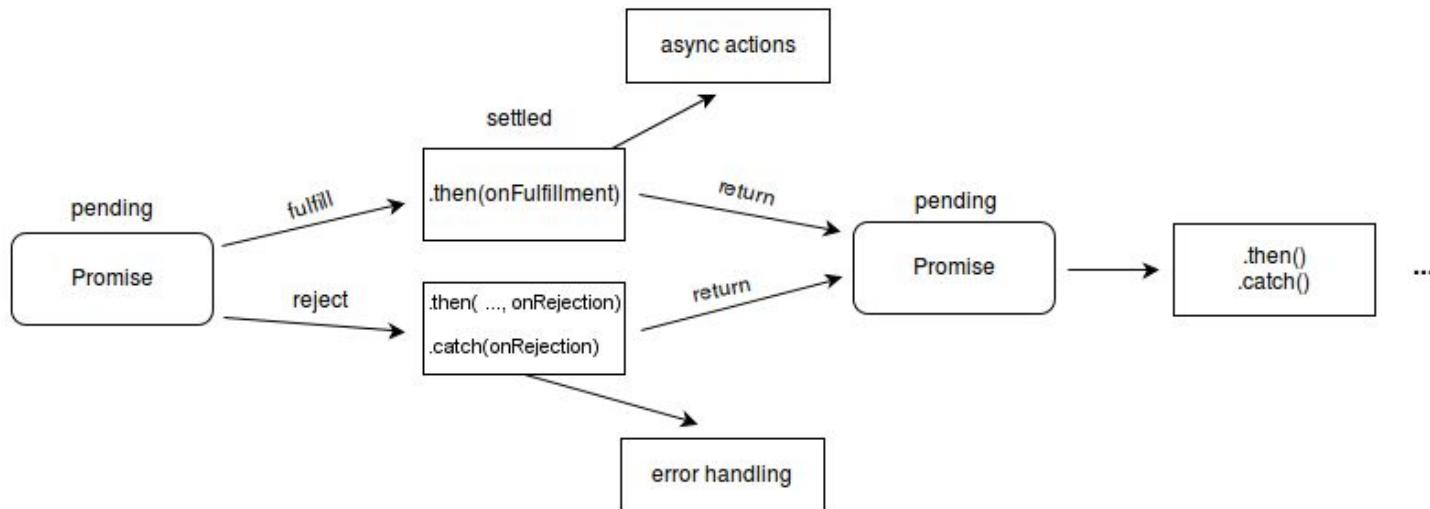
Exemple d'utilisation d'une promesse

```
const dividend = inputDividend.valueAsNumber;
const divisor = inputDivisor.valueAsNumber;

examplePromise(dividend, divisor)
  .then((data) => {
    result.value = `Le résultat de ${dividend} / ${divisor} est ${data}`;
  })
  .catch((reason) => {
    result.value = `Erreur : ${reason.message}`;
  })
  .finally(() => {
    console.log('Traitement de la promesse terminé !')
  });
}
```

Enchaîner des promesses

Les méthodes « `then` », « `catch` » et « `finally` » renvoient toutes des promesses.
Ce qui permet de pouvoir les chaînées.



Enchaîner des promesses

Exemple avec quatre promesses chaînés

```
function exampleChainPromise(nb) {
  return addition(nb, 3)
    .then(r1 => multi2(r1))
    .then(r2 => addition(r2, 42))
    .then(r3 => multi3(r3))
    .then(rf => {
      | return `Resultat : ${rf}`;
    })
    .catch(err => {
      | return 'Erreur...';
    });
}
```

```
function addition(nb1, nb2) {
  return new Promise((resolve) => {
    const result = Math.round(nb1 + nb2);
    setTimeout(resolve, 500, result);
  })
}

function multi2 (nb) {
  return new Promise((resolve) => {
    const result = nb * 3;
    setTimeout(resolve, 500, result);
  })
}

function multi3 (nb) {
  return new Promise((resolve) => {
    const result = nb * 3;
    setTimeout(resolve, 1_000, result);
  })
}
```

Les méthodes des promesses

Les promesses

Méthode pour créer des promesses terminées

- `Promise.resolve([data])`

Créer un objet du type « Promise » qui est résolu avec la valeur donnée.

- `Promise.reject([reason])`

Créer un objet du type « Promise » qui est rejeté avec la cause.

Méthode pour utiliser des promesses en parallèle

- Promise.all(promiseCollection)

Créer une promesse qui attend que toutes les promesses reçues soit résolue.

Elle renvoie le résultat de chaque promesse dans une collection.

Dès qu'une des promesse est rejeté, celle-ci passe en état « rejected ».

Si la collection reçu est vide, la promesse sera résolue avec une collection vide.

```
const p1 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 1'); }, 200); });
const p2 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 2'); }, 50); });
const p3 = new Promise((resolve, reject) => { setTimeout(() => { reject(new Error('Boum !')); }, 100); });

Promise.all([p1, p2]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → ['Promesse 1', 'Promesse 2']

Promise.all([p1, p3]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → {Error: 'Boum !'}
```

Méthode pour utiliser des promesses en parallèle

- Promise.allSettled(promiseCollection)

Créer une promesse qui attend que toutes les promesses reçues soit terminé.

Elle renvoie le résultat de chaque promesse (fulfilled ou rejected) dans une collection.

Si la collection reçu est vide, la promesse sera résolue avec une collection vide.

```
const p1 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 1'); }, 200); });
const p2 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 2'); }, 50); });
const p3 = new Promise((resolve, reject) => { setTimeout(() => { reject(new Error('Boum !'))}, 100); });

Promise.allSettled([p1, p2]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → [{status: 'fulfilled', value: 'Promesse 1'}, {status: 'fulfilled', value: 'Promesse 2'}]

Promise.allSettled([p1, p3]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → [{status: 'fulfilled', value: 'Promesse 1'}, {status: 'rejected', reason: {Error: 'Boum !'}}]
```

Méthode pour utiliser des promesses en parallèle

- `Promise.any(promisesCollection)`

Créer une promesse qui attend qu'une des promesses reçues soit résolue.

Elle renvoie le résultat de la première promesse résolue.

Si toutes les promesses sont rejetées, celle-ci passe en état « rejected ».

Si la collection reçue est vide, la promesse sera rejetée avec une "AggregateError".

```
const p1 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 1'); }, 200); });
const p2 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 2'); }, 50); });
const p3 = new Promise((resolve, reject) => { setTimeout(() => { reject(new Error('Boum !')); }, 100); });

Promise.any([p1, p2]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → 'Promesse 2'

Promise.any([p1, p3]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → 'Promesse 1'
```

Méthode pour utiliser des promesses en parallèle

- Promise.race(promiseCollection)

Créer une promesse qui attend qu'une des promesses reçues soit terminée.

Elle renvoie le résultat de la première promesse terminé (fulfilled ou rejected).

Si la collection reçue est vide, la promesse sera continuellement en attente.

```
const p1 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 1'); }, 200); });
const p2 = new Promise((resolve, reject) => { setTimeout(() => { resolve('Promesse 2'); }, 50); });
const p3 = new Promise((resolve, reject) => { setTimeout(() => { reject(new Error('Boum !')); }, 100); });

Promise.race([p1, p2]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → 'Promesse 2'

Promise.race([p1, p3]).then(data => console.log(data)).catch(error => console.log(error));
// Résultat → {Error: 'Boum !'}
```

Les fonctions asynchrones

Les promesses

Les fonctions asynchrones

Une fonction peut être déclaré en tant que fonction asynchrone, pour cela, il est nécessaire de la précédéer de « `async` ».

Celle-ci peut contenir des promesses précédées du mot clef « `await` ».

Ce qui interrompt son exécution et attend la résolution de la promesse.

La fonction renvoie une promesse :

- Résolue lorsque elle est terminé.
- Rompue en cas d'exception.

```
// Fonction qui renvoie une promesse
function doSomething() {
  return new Promise((resolve) => {
    setTimeout(resolve, 2_000, '42')
  });
}

// Fonction Asynchrone
async function exampleAsyncAwait() {
  console.log('Début du traitement...');

  // Traitement durant de 2s
  const result = await doSomething();

  console.log(`Résutlat : ${result}`);
}
```

Les fonctions asynchrones

Les fonctions asynchrones permettent de simplifier l'utilisation des promesses.
Elle évite de devoir configurer explicitement les chaînes de promesse.

Pour traiter les exceptions provoquées par des promesses en attente (via await) lorsque celles-ci sont rompu, il est nécessaire d'utiliser un bloc « try / catch ».

```
async function exampleErrorHandler() {
  let result;
  try {
    result = await doSomething();
  }
  catch(err) {
    result = 'Erreur lors du traitement';
  }
  return result;
}
```

Enchaîner des promesses avec async/await

Exemple avec quatre promesses chaînés

```
async function exampleSequentialPromise(nb) {
  try {
    const r1 = await addition(nb, 3);
    const r2 = await multi2(r1);
    const r3 = await addition(r2, 42);
    const rf = await multi2(r3);
    console.log(`Résultat : ${rf}`);
  }
  catch (err) {
    console.log('Erreur...');

  }
}
```

```
function addition(nb1, nb2) {
  return new Promise((resolve) => {
    const result = Math.round(nb1 + nb2);
    setTimeout(resolve, 500, result);
  })
}

function multi2 (nb) {
  return new Promise((resolve) => {
    const result = nb * 3;
    setTimeout(resolve, 500, result);
  })
}

function multi3 (nb) {
  return new Promise((resolve) => {
    const result = nb * 3;
    setTimeout(resolve, 1_000, result);
  })
}
```

Interaction avec le DOM

- Les formulaires -

Objectif des formulaires

Interaction avec le DOM - Les formulaires

Objectif des formulaires

Les formulaires sont utilisés pour qu'un utilisateur puisse interagir avec une page web.
Ils permettent d'envoyer des données au serveur.

- Par défaut, via le comportement de la balise « form » (en « GET » ou « POST »)
- En JS, via des requêtes Ajax (abordé ultérieurement)

Le langage JavaScript permet :

- D'accéder aux données du formulaire (en lecture et écriture)
- D'interrompre ou de déclencher les comportements « Submit » et « Reset »
- D'ajouter des règles de validation sur les différentes inputs

Structure d'un formulaire

Les formulaires sont généralement constitués :

- D'une balise « form »
- De balise de saisie pour l'utilisateur
 - input(text, number, radio, checkbox, ...)
 - textarea
 - select
- D'un bouton de validation et d'annulation

Structure du formulaire utilisé dans le support

```
<form id="example-form" name="registration" method="post">
  <div>
    <label for="field-email">Email</label>
    <input id="field-email" type="email" name="email" required>
  </div>
  <div>
    <label for="field-pseudo">Pseudo</label>
    <input id="field-pseudo" type="text" name="pseudo">
  </div>
  <div>
    <label for="field-guest">Nombre de personne</label>
    <input id="field-guest" type="number" name="nbGuest" min="1" required>
  </div>
  <div>
    <button type="submit">Valider l'inscription</button>
    <button type="reset">Réinitialiser</button>
  </div>
</form>
```

Manipulation les formulaires

Interaction avec le DOM - Les formulaires

Récupérer un formulaire

Pour récupérer une balise de formulaire, il existe de plusieurs possibilité :

- À l'aide des méthodes « getElementById » et « querySelector ».
- Via la propriété « forms » de l'objet « document » (via son nom ou son index).
- En utilisant le nom du formulaire en tant que propriété de « document ».

```
const form1 = document.getElementById('example-form');
const form2 = document.querySelector('#example-form');

const form3 = document.forms['registration'];
const form4 = document.forms[0];

const form5 = document.registration;
```

Récupérer les champs d'un formulaire

La propriété « elements » d'un objet de type « `HTMLFormElement` » permet d'obtenir une collection qui contient tous les champs (`input`, `textarea`, `button`) du formulaire.

- `Javascript`

```
for (const elem of form1.elements) {  
    console.log(elem);  
}
```

- `Résultat`

```
<input id="field-email" type="email" name="email" required>  
  
<input id="field-pseudo" type="text" name="pseudo">  
  
<input id="field-guest" type="number" name="nbGuest" min="1" required>  
  
<button type="submit">Valider l'inscription</button>  
  
<button type="reset">Réinitialiser</button>
```

Récupérer un champ d'un formulaire

Pour récupérer un des champs d'un formulaire, il est possible de l'obtenir :

- Depuis la propriété « element ».
- Directement depuis l'objet formulaire.

Ces deux méthodes fonctionnent avec le nom et l'index du champ ciblé.

```
const fieldPseudo1 = form1.elements['pseudo'];
const fieldPseudo2 = form1.elements[1];

const fieldPseudo3 = form1['pseudo'];
const fieldPseudo4 = form1[1];
```

Interagir avec les événements d'un formulaire

L'événement « submit » permet d'interagir sur la validation d'un formulaire avant que les données de celui-ci ne soient envoyées.

```
form1.addEventListener('submit', (event) => {  
    // Traitement ...  
});
```

L'événement « reset » permet d'interagir sur la remise à zéro d'un formulaire avant que les données ne soient effacées.

```
form1.addEventListener('reset', (event) => {  
    // Traitement ...  
});
```

Rappel : L'utilisation de « event.preventDefault() » annule leurs comportements.

Interagir avec les événements d'un formulaire

L'event « formdata » se déclenche quand l'objet avec les valeurs du formulaire est créé. Il est donc possible d'ajouter, de modifier et de supprimer des valeurs du formData.

```
form1.addEventListener('formdata', (event) => {  
    event.formData.append('message', 'The cake is a lie 🎂');  
});
```

Données reçues par le serveur

```
{  
    email: 'della.duck@demo.com',  
    pseudo: 'Della',  
    nbGuest: '4',  
    message: 'The cake is a lie 🎂'  
}
```

Attention, cette événement n'est pas annulable !

L'API de validation des contraintes

Interaction avec le DOM - Les formulaires

Objectif de l'API de validation des contraintes

Les données d'un formulaire peuvent être validé par règle définie dans la balise html telle que : required, minlength, maxlength, pattern, etc...

Si vous souhaitez ajouter des règles custom, il est nécessaire d'utiliser le JavaScript.

L'API de validation des contraintes permet d'ajouter facilement des règles de validation sans devoir gérer manuellement le comportement du navigateur.

Elle permet :

- D'obtenir l'état de validation des champs.
- De définir un champs en erreur.

Les propriétés de l'API de validation des contraintes

- .validity

Un objet avec l'états (booléen) des différentes contraintes à valider

- .validationMessage

Le message d'erreur lorsqu'une contrainte est invalide.

Sinon, la propriété contient une chaîne de caractère vide

- .willValidate

Valeur booléen qui indique si l'élément doit être validé.

Les méthodes de l'API de validation des contraintes

- `.checkValidity()`

La méthode renvoi « true » si l'élément n'a pas de problème de validation.
Sinon, elle renvoi « false » et déclenche également l'évent « invalid ».

- `.setCustomValidity(message)`

Permet de définir une erreur personnalisé sur un élément avec son message d'erreur.
Cet élément sera considéré comme invalide par le formulaire.

Si l'argument est une chaîne de caractères vide, elle supprime l'erreur personnalisé.

Exemple d'erreur personnalisé sur un champ

```
form1['email'].addEventListener('input', (event) => {
  const field = event.currentTarget;
  const email = field.value.toLowerCase();
  const validityState = field.validity;

  if (validityState.typeMismatch) {
    field.setCustomValidity('Votre email est invalide !');
  }
  else if (email.includes('gripsou')) {
    field.setCustomValidity('Vous n\'êtes pas autorisé !');
  }
  else {
    field.setCustomValidity('');
  }
});
```

Le destructuring

Objectif du destructuring

Le destructuring

Objectif du destructuring

Le destructuring est une syntaxe qui permet de réaliser une affectation par décomposition. Celle-ci permet d'extraire des données depuis un objet ou un tableau. L'objectif du destructuring est donc de pouvoir affecter les valeurs d'une structure de données à des variables en une seule instruction.

```
// Exemple simple

// - Ensemble de données
const fruits = ['Pomme', 'Cerise', 'Mangues'];
const person = { firstname: 'Della', lastname: 'Duck'};

// - Utilisation du destructuring
const [f1, f2, f3] = fruits;
const { firstname, lastname } = person;
```

Décomposition de collection

Le destructuring

Décomposition de collection

- Déclaration de variable et initialisation

La valeur des variables est basée sur la position des éléments de la collection.

```
const people = ['Della', 'Balthazar', 'Zaza', 'Gontran'];

// Affection sans décomposition
const duck01 = people[0];
const duck02 = people[1];
const duck03 = people[2];

// Affection avec décomposition
const [duck01, duck02, duck03] = people;
```

Décomposition de collection

- Affectation sans déclaration

Il est possible d'utiliser le destructuring pour uniquement affecter des variables.

```
const numbers = [42, -5, 13];
let val1, val2, val3;
[val1, val2, val3] = numbers;
```

- Valeurs par défaut

On peut définir une valeur qui sera utilisée si la valeur extraite est « undefined ».

```
const numbers = [-9, 3.14];
const [val1, val2 = 10, val3 = 42] = numbers;
```

Décomposition de collection

- Affecter le reste d'une collection à une variable

Il est possible d'utiliser l'opérateur « rest » avec le destructuring.

```
const people = ['Donald', 'Daisy', 'Riri', 'Fifi', 'Loulou'];

// Affection sans décomposition
const duck01 = people[0];
const duck02 = people[1];
const neveux = people.slice(2);

// Affection avec décomposition
const [duck01, duck02, ...neveux] = people;
```

Décomposition de collection

- Ignorer des valeurs

Il est possible d'ignorer certaines des valeurs de la collection.

```
const numbers = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50];
const [val1, , val3, , , val6, val7] = numbers;
```

- Échange de variables

Le destructuring permet également d'inverser le contenu de variables.

```
let a = 42;
let b = 13;
[a, b] = [b, a];
```

Décomposition d'objet

Le destructuring

Décomposition d'objet

- Déclaration de variable et initialisation

La valeur des variables est basée sur le nom des clefs de l'objet ciblé.

```
const person = { firstname: 'Della', lastname: 'Duck', job: 'Pilote' };

// Affection sans décomposition
const firstname = person.firstname;
const lastname = person.lastname;
const job = person.job;

// Affection avec décomposition
const { firstname, lastname, job } = person;
```

Décomposition d'objet

- Affectation sans déclaration

Il est possible d'utiliser le destructuring pour uniquement affecter des variables.

```
const course = { title: 'JavaScript', acronym: 'JS', section: 'Dev' };
let title, acronym;

({ title, acronym } = course);
```

- Valeurs par défaut

On peut définir une valeur qui sera utilisée si la valeur de la clef est « undefined ».

```
const course = { title: 'Information', acronym: 'IC' };
const { title, section = 'N/A' } = course;
```

Décomposition d'objet

- Affecter le reste

Il est possible d'utiliser l'opérateur « rest » avec le destructuring.

```
const company = {
  name: 'Bstorm',
  lat: 50.66072,
  lon: 4.632701
};

// Affection sans décomposition
const name = company.name;
const coord = { lat: company.lat, lon: company.lon };

// Affection avec décomposition
const { name, ...coord } = company;
```

Décomposition d'objet

- Affectation avec un nom différent

Il est possible définir un nom de variable différent que celui de la clef ciblé.

```
const person = { firstname: 'Donald', lastname: 'Duck', job: 'Matelot' };
const { firstname: prenom, lastname: nom, job } = person;
```

Il est également possible de définir une valeur par défaut en plus du nom de variable.

```
const position1 = { x: 5, y: -9, z: 2 };
const position2 = { x: 3, y: 6 };

const { x: x1, y: y1, z: z1 = 0 } = position1;
const { x: x2, y: y2, z: z2 = 0 } = position2;
```

Décomposition d'objet

- Décomposition imbriquée

```
const user = {
  id: 42,
  pseudo: 'zavand',
  name: {
    firstname: 'Zaza',
    lastname: 'Vanderquack',
  },
  langagues: [
    { code: 'en', name: 'English', isFav: true },
    { code: 'fr', name: 'French' },
  ]
};

const { name : { firstname }, langagues: [{ name: lang1}, { name: lang2}]} = user;
```

Décomposition d'objet

- Décomposition d'un paramètre d'une fonction

Lorsqu'on créer une méthode, il est possible d'utiliser le destructuring pour définir le nom des paramètre de l'objet attendu et si nécessaire la valeur par défaut de ceux-ci.

```
function showBook({ title, author, category = 'Others' }) {  
    // ...  
}  
  
const book1 = {title: 'Tout JavaScript', author: 'Olivier Hondermarck', category: 'Dev'};  
showBook(book1);  
  
const book2 = { title: 'Le Petit Prince', author: 'Antoine de Saint-Exupéry' };  
showBook(book2);
```

Annexes

Aperçu de méthodes sur les types

Annexes

Méthodes des variables de type « number »

- `.toFixed(nbChiffre)`

Renvoie le nombre formaté en chaîne de caractère avec une notation à point-fixe.

- `.toLocaleString(locale, options)`

Renvoie le nombre formaté en chaîne de caractère en tenant compte de la locale.

Les « options » permettent de préciser le contexte de formatage (décimal, monnaie...).

```
const nb = 42;
const nbFormatted = nb.toLocaleString('fr-be', {
  style: 'unit',
  unit: 'celsius'
});
// → '42 °C'
```

Méthodes des variables de type « string »

- `.indexOf('valeur cherchée') / .lastIndexOf('valeur cherchée')`

Permet d'obtenir l'indice de la première ou dernière occurrence de la valeur cherchée dans la chaîne de caractères. Si la valeur cherchée n'est pas trouvée, elle renvoie -1.

- `.includes('chaîne recherchée')`

Permet de déterminer si la chaîne recherchée est contenue dans la variable.

- `.startsWith('chaîne recherchée') / .endsWith('chaîne recherchée')`

Permet de déterminer si la variable commence ou termine par la chaîne recherchée

Méthodes des variables de type « string »

- `.localeCompare('chaîne cible', locale)`

Renvoie un nombre qui permet de déterminer la position de la variable par rapport à la chaîne cible dans la locale choisie :

- Nombre positif : Elle se situe après
- Valeur « 0 » : Les deux chaînes de caractères sont identiques
- Nombre négatif : Elle se situe avant

- `.trim() / .trimStart() / .trimEnd()`

Renvoie une copie de la variable sans les caractères blancs (espace, tabulation, ...)

Méthodes des variables de type « string »

- .toUpperCase() / .toLowerCase()

Renvoie une copie de la variable en majuscule ou minuscule.

- .toLocaleUpperCase(locale) / .toLocaleLowerCase(locale)

Renvoie une copie de la variable en majuscule ou minuscule en respectant la locale.

- .padStart(longueur [, 'chaîne compl']) / .padEnd(longueur [, 'chaîne compl'])

Permet d'obtenir la chaîne complétée à la longueur demandée. Pour cela, la valeur du deuxième paramètre (par défaut : un espace) sera ajouté en début ou fin de la chaîne.

Méthodes des variables de type « string »

- `.replace('cible', 'remplacement')`

Renvoie une nouvelle chaîne dans laquelle la 1ère occurrence de la chaîne 'cible' est remplacée par la chaîne 'remplacement'.

- `.replaceAll('cible', 'remplacement')`

Renvoie une nouvelle chaîne dans laquelle toutes les occurrences de la chaîne 'cible' sont remplacées par la chaîne 'remplacement'.

Remarque : Il est possible de remplacer la chaîne cible par une regex.

Pour la méthode « replaceAll » la regex est utilisé doit être global « /regex/g ».

Méthodes des variables de type « string »

- `.substring(indiceA [, indiceB])`

Renvoie une chaîne de caractère commençant au plus petit indice et jusqu'au plus grand indice. Le deuxième indice est optionnel (par défaut : la longueur de la chaîne).

- `.slice(indiceDebut [, indiceFin])`

Renvoie une chaîne de caractère commençant à l'indice de début et jusqu'à l'indice de fin. Contrairement à la méthode `substring`, celle-ci peut utiliser des valeurs négatives.

- `.split('separateur')`

Permet d'obtenir un tableau avec le contenu de la chaîne divisé par le séparateur.

La collection à clefs « Map »

Annexes

Les objets de type « Map »

Un objet « Map » permet de stocker des éléments sous la forme de « Clef / Valeur ». Les clefs peuvent être de n'importe quel type de valeur (primitive ou objet).

Celui-ci est un itérable, il peut être parcouru à l'aide d'une boucle.
L'ordre d'insertion des éléments est mémorisé.

Aperçu d'un objet « Map » :

```
const dico = new Map();
dico.set('Pomme', 250);
dico.set('Poire', 300);
dico.set('Mangues', 70);
▶ Map(3) { 'Pomme' => 250, 'Poire' => 300, 'Mangues' => 70}
```

Méthodes sur un objet de type « Map »

- `.set(clef, valeur)`

Permet d'ajouter une combinaison clef / valeur dans l'objet « Map ».

Si la clef existe déjà, la valeur sera mise à jour avec celle reçue.

- `.get(clef)`

Renvoie la valeur liée à la clef. Si aucune valeur n'est trouvée, elle renvoie « undefined ».

- `.has(clef)`

Renvoie un booléen pour indiquer si la clef est présente dans l'objet « Map ».

Méthodes sur un objet de type « Map »

- .delete(clef)

Permet de supprimer un élément à l'aide de sa clef.

Cette méthode renvoie « true » si l'élément existe et qu'il a bien été supprimé.

- .clear()

Permet de supprimer tous les éléments.

- .forEach(méthode)

Permet d'effectuer un traitement itératif. (*Identique à la méthode forEach() des Array*)

Propriété sur un objet de type « Map »

La propriété « size » permet d'obtenir le nombre d'éléments de l'objet « Map ».

```
const dico = new Map();
const s1 = dico.size;    // → 0

dico.set('z1', 'Zaza');
dico.set('d1', 'Donald');
dico.set('r1', 'Riri');
const s2 = dico.size;    // → 3

dico.set('d1', 'Della');
dico.delete('r1');
const s3 = dico.size;    // → 2

dico.clear();
const s4 = dico.size;    // → 0
```

Exemple de l'utilisation d'un objet « Map »

```
// Déclaration
const students = new Map();

// Ajouter des éléments (clef : string / valeur : objet)
students.set('zava42', { firsname: 'Zaza', lastname: 'Vanderquack' });
students.set('ridu19', { firsname: 'Riri', lastname: 'Duck' });

// Obtenir le nombre d'élément
const nbElem = students.size; // → 2

// Obtenir des valeurs via leur clef
const v1 = students.get('zava42'); // → L'objet : {firsname: 'Zaza', lastname: 'Vanderquack'}
const v2 = students.get('cama01'); // → undefined

// Parcourir les éléments
for(const [key, value] of students) {
    console.log(` ${key} => ${value}`);
}
```

La collection à clefs « Set »

Annexes

Les objets de type « Set »

Un objet « Set » permet de stocker des éléments sous la forme de clef unique.
Les clefs peuvent être de n'importe quel type de valeur (primitive ou objet).

Celui-ci est un itérable, il peut être parcouru à l'aide d'une boucle.
L'ordre d'ajout des éléments est mémorisé.

Aperçu d'un objet « Set » :

```
const fruits = new Set();

fruits.add('Pomme');
fruits.add('Poire');
fruits.add('Mangues');

▶ Set(3) {'Pomme', 'Poire', 'Mangues'}
```

Méthodes sur un objet de type « Set »

- `.add(clef)`

Permet d'ajouter une clef dans l'objet « Set ».

Si la clef existe déjà, la méthode sera ignorée.

- `.has(clef)`

Renvoie un booléen pour indiquer si la clef est présente dans l'objet « Set ».

Méthodes sur un objet de type « Set »

- .delete(clef)

Permet de supprimer une clef.

Cette méthode renvoie « true » si l'élément existe et qu'il a bien été supprimé.

- .clear()

Permet de supprimer toutes les clefs.

- .forEach(méthode)

Permet d'effectuer un traitement itératif. (*Identique à la méthode forEach() des Array*)

Propriété sur un objet de type « Set »

La propriété « size » permet d'obtenir le nombre d'éléments de l'objet « Set ».

```
const ensemble = new Set();
const s1 = ensemble.size;    // → 0

ensemble.add('Zaza');
ensemble.add('Riri');
ensemble.add('Della');
const s2 = ensemble.size;    // → 3

ensemble.delete('Riri');
const s3 = ensemble.size;    // → 2

ensemble.clear();
const s4 = ensemble.size;    // → 0
```

Évaluation en court-circuit

Annexes

Évaluation en court-circuit

Le JavaScript permet d'utiliser les opérateurs logiques pour créer un « court-circuit » lors de l'évaluation d'une instruction.

Cela permet de rendre l'exécution d'instruction de code conditionnel.

- Opérateur logique ET : si l'expression est « Truthy », la suite sera interprétée
`callback && callback();`
- Opérateur logique OU : si l'expression est « Truthy », la suite ne sera pas interprétée

```
const message = box.lastMessage || 'Aucun message';
```

Références

Références

- MDN Web Docs

<https://developer.mozilla.org/fr/docs/Web/JavaScript>

- Ecma International's TC39

<https://tc39.es/>

<https://github.com/tc39>

Merci pour votre attention.

