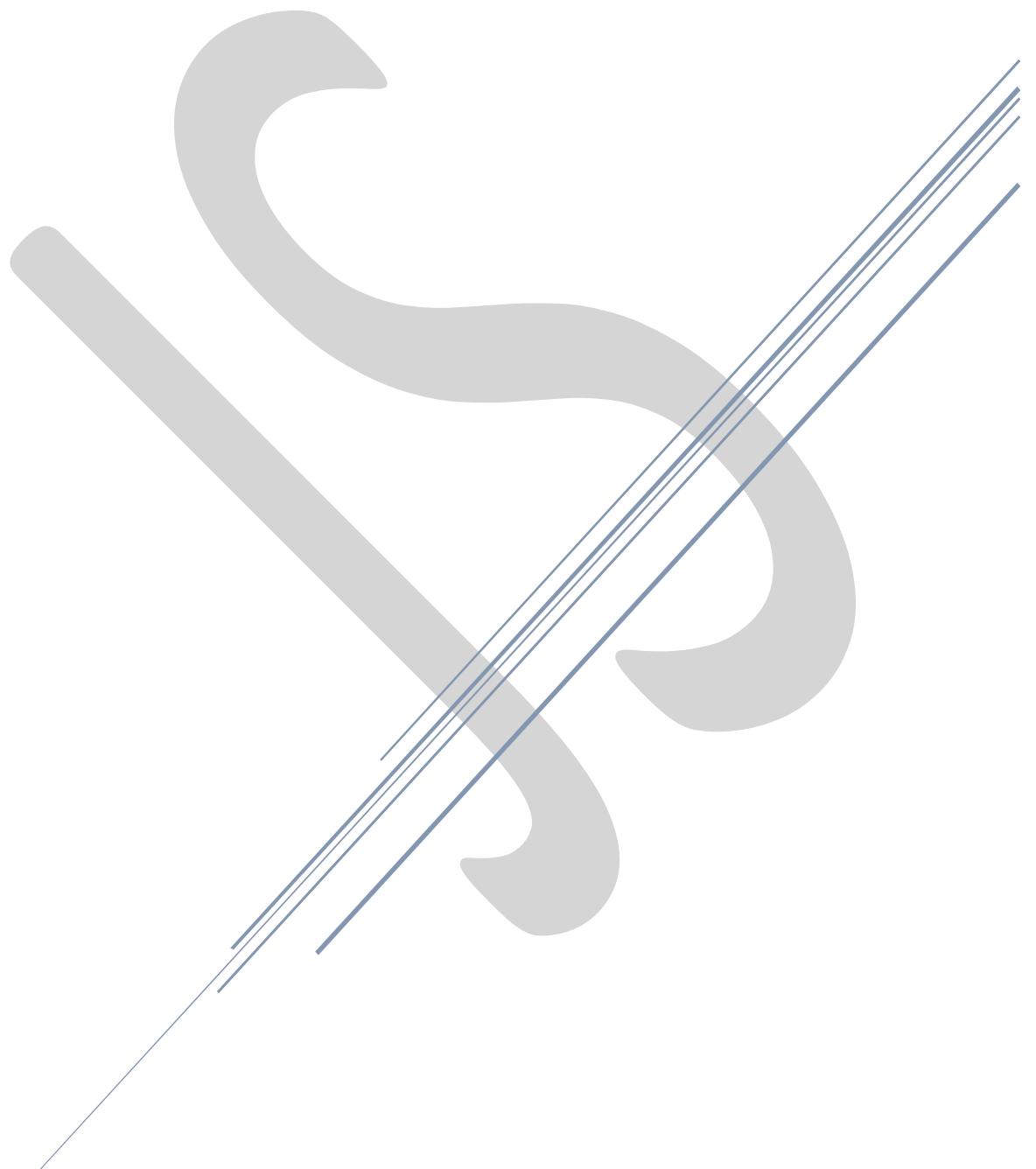


FROM BEGINNER TO PROFESSIONNAL IN JS

By Lévi Christ GOTÉNI



JAVA SCRIPT

SOMMAIRE

PARTIE 1 : INTRODUCTION À JAVASCRIPT

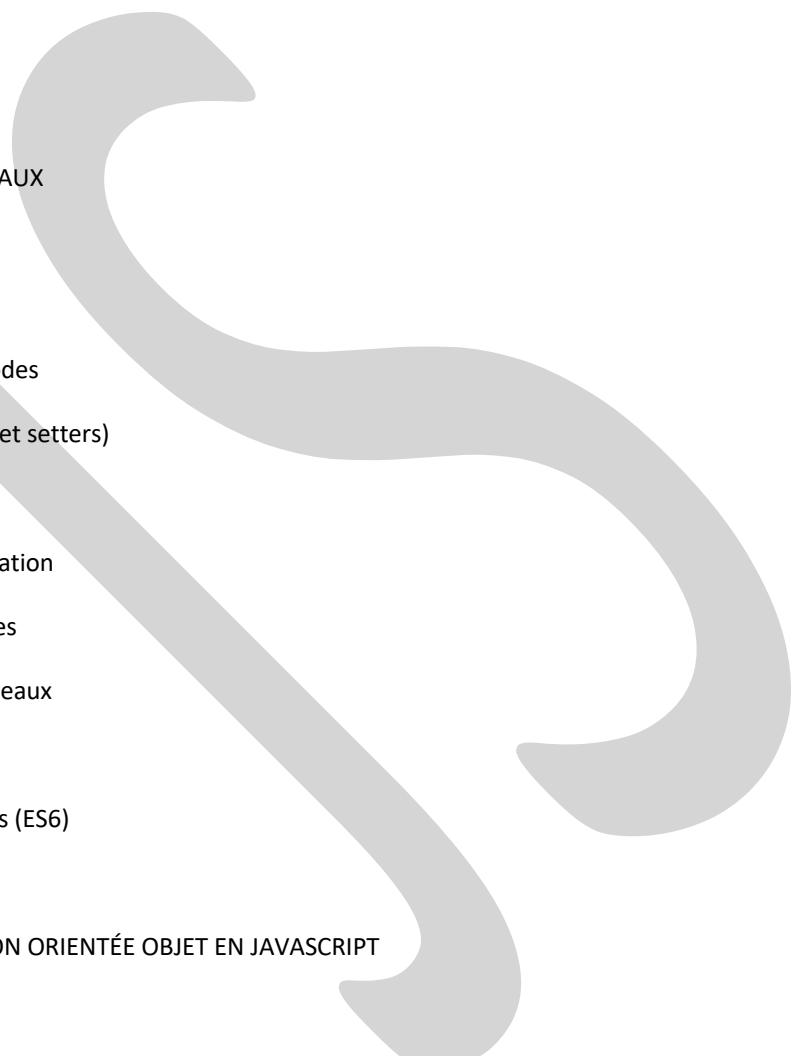
- 1.1. Histoire et évolution de JavaScript
- 1.2. Place de JavaScript dans le développement web moderne
- 1.3. Environnement de développement
 - 1.3.1. Éditeurs de code et IDE
 - 1.3.2. Les outils de débogage
 - 1.3.3. Configuration d'un environnement de travail

PARTIE 2 : LES FONDAMENTAUX

- 2.1. Syntaxe de base
 - 2.1.1. Structure d'un script JavaScript
 - 2.1.2. Commentaires
 - 2.1.3. Sensibilité à la casse
- 2.2. Variables et types de données
 - 2.2.1. Déclaration (var, let, const)
 - 2.2.2. Types primitifs (Number, String, Boolean, etc.)
 - 2.2.3. Conversion de types
- 2.3. Opérateurs
 - 2.3.1. Opérateurs arithmétiques
 - 2.3.2. Opérateurs de comparaison
 - 2.3.3. Opérateurs logiques
 - 2.3.4. Opérateurs d'affectation
- 2.4. Structures de contrôle
 - 2.4.1. Conditions (if, else, switch)
 - 2.4.2. Boucles (for, while, do-while)
 - 2.4.3. Instructions break et continue

PARTIE 3 : FONCTIONS ET PORTÉE

- 3.1. Définition et appel de fonctions
- 3.2. Paramètres et arguments
- 3.3. Retour de valeurs
- 3.4. Fonctions anonymes et expressions de fonction
- 3.5. Fonctions fléchées (ES6)
- 3.6. Portée des variables (scope)
- 3.7. Clôtures (closures)



PARTIE 4 : OBJETS ET TABLEAUX

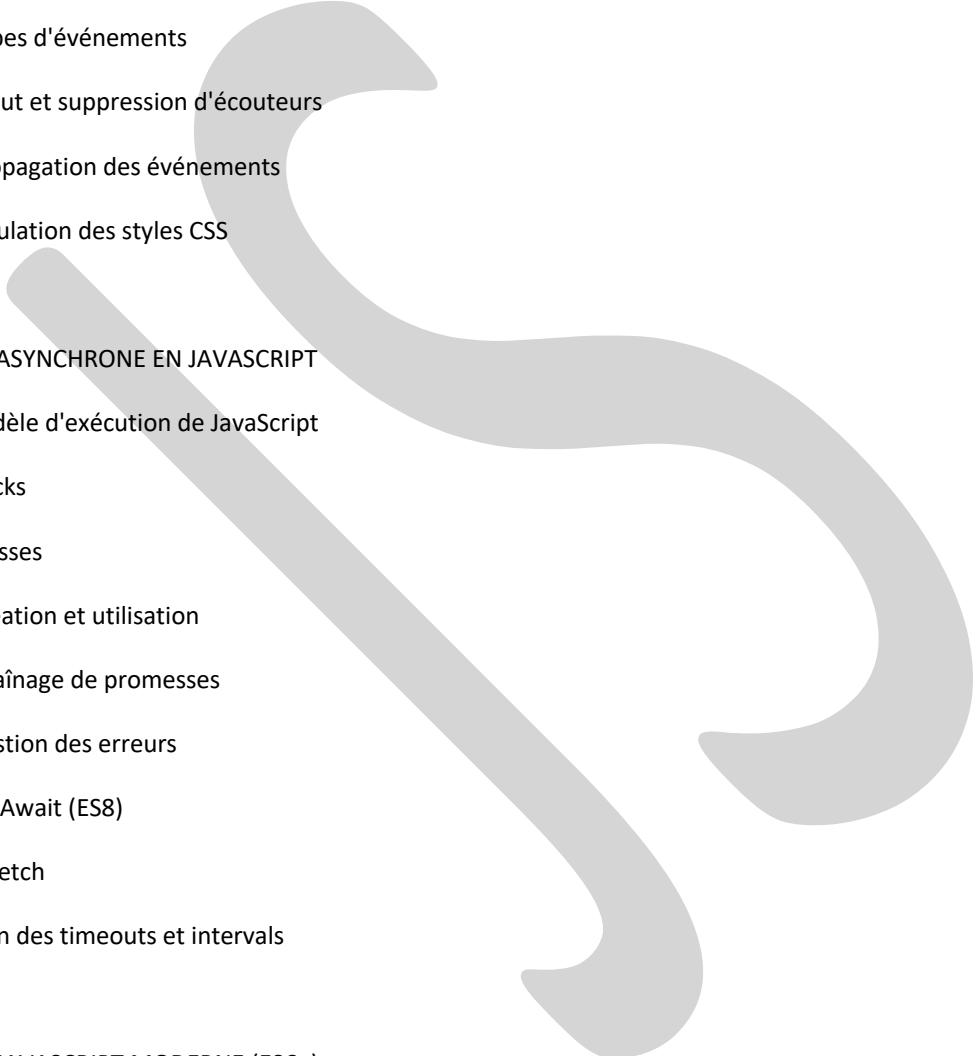
- 4.1. Objets littéraux
 - 4.1.1. Création d'objets
 - 4.1.2. Propriétés et méthodes
 - 4.1.3. Accesseurs (getters et setters)
- 4.2. Tableaux
 - 4.2.1. Création et manipulation
 - 4.2.2. Méthodes principales
 - 4.2.3. Itération sur les tableaux
- 4.3. Destructuration (ES6)
- 4.4. Spread et rest operators (ES6)

PARTIE 5 : PROGRAMMATION ORIENTÉE OBJET EN JAVASCRIPT

- 5.1. Prototypes et héritage
- 5.2. Constructeurs et l'opérateur new
- 5.3. Classes (ES6)
 - 5.3.1. Définition de classes
 - 5.3.2. Héritage avec extends
 - 5.3.3. Méthodes statiques
- 5.4. Encapsulation et polymorphisme

PARTIE 6 : MANIPULATION DU DOM

- 6.1. Le modèle objet du document
- 6.2. Sélection d'éléments
- 6.3. Modification du contenu et des attributs
- 6.4. Création et suppression d'éléments
- 6.5. Gestion des événements
 - 6.5.1. Types d'événements
 - 6.5.2. Ajout et suppression d'écouteurs
 - 6.5.3. Propagation des événements
- 6.6. Manipulation des styles CSS



PARTIE 7 : ASYNCHRONE EN JAVASCRIPT

- 7.1. Le modèle d'exécution de JavaScript
- 7.2. Callbacks
- 7.3. Promesses
 - 7.3.1. Crédit et utilisation
 - 7.3.2. Chaînage de promesses
 - 7.3.3. Gestion des erreurs
- 7.4. Async/Await (ES8)
- 7.5. L'API Fetch
- 7.6. Gestion des timeouts et intervals

PARTIE 8 : JAVASCRIPT MODERNE (ES6+)

- 8.1. Modules
 - 8.1.1. Export et import
 - 8.1.2. Modules par défaut et nommés
- 8.2. Template literals
- 8.3. Valeurs par défaut pour les paramètres

8.4. Maps et Sets

8.5. Symboles

8.6. Itérateurs et générateurs

8.7. Proxies et Reflect

PARTIE 9 : APIS WEB ET TECHNOLOGIES AVANCÉES

9.1. LocalStorage et SessionStorage

9.2. API de géolocalisation

9.3. API Canvas

9.4. Web Workers

9.5. Service Workers et PWA

9.6. WebSockets

PARTIE 10 : FRAMEWORKS ET BIBLIOTHÈQUES

10.1. Introduction aux frameworks JavaScript

10.2. React.js (aperçu)

10.3. Vue.js (aperçu)

10.4. Angular (aperçu)

10.5. Bibliothèques utiles

10.5.1. jQuery

10.5.2. Lodash/Underscore

10.5.3. Axios

PARTIE 11 : OUTILS ET ENVIRONNEMENT DE DÉVELOPPEMENT

11.1. Gestionnaires de paquets (npm, yarn)

11.2. Bundlers (Webpack, Parcel)

11.3. Transpilation avec Babel

11.4. Tests en JavaScript

11.4.1. Tests unitaires

11.4.2. Tests fonctionnels

11.5. Linting et formatage (ESLint, Prettier)

PARTIE 12 : BONNES PRATIQUES ET PATTERNS

12.1. Conventions de codage

12.2. Design patterns en JavaScript

12.3. Optimisation des performances

12.4. Gestion des erreurs

12.5. Débogage avancé

12.6. Sécurité en JavaScript

PARTIE 13 : PROJETS PRATIQUES

13.1. Application ToDo

13.2. Quiz interactif

13.3. Application météo

13.4. Jeu simple en JavaScript

13.5. Système de gestion de contenu

1.1. HISTOIRE ET ÉVOLUTION DE JAVASCRIPT

JavaScript, souvent abrégé en JS, est devenu l'un des langages de programmation les plus populaires et essentiels du développement web moderne. Son évolution depuis sa création jusqu'à aujourd'hui est une histoire fascinante de réussite, d'innovation et d'adaptation aux besoins changeants du web.

LES ORIGINES : NAISSANCE CHEZ NETSCAPE

JavaScript a été créé en mai 1995 par Brendan Eich alors qu'il travaillait chez Netscape Communications. À l'origine, le langage a été développé en seulement 10 jours sous le nom de "Mocha", puis renommé "LiveScript" avant de devenir "JavaScript".

Quelques faits importants sur la naissance de JavaScript :

- Il a été conçu comme un langage de script léger pour le navigateur Netscape Navigator
- Le nom "JavaScript" a été choisi pour des raisons marketing, afin de capitaliser sur la popularité de Java à l'époque
- Contrairement à ce que son nom suggère, JavaScript n'est pas lié techniquement à Java
- La première version de JavaScript était relativement limitée, permettant principalement la validation de formulaires et quelques interactions basiques

STANDARDISATION : ECMASCIPT

En 1996, Netscape a soumis JavaScript à l'ECMA International (European Computer Manufacturers Association) pour standardisation, donnant naissance à "ECMAScript". Cette standardisation a été cruciale pour l'adoption généralisée du langage.

Version ECMAScript	Année	Caractéristiques principales
ES1	1997	Première standardisation officielle
ES2	1998	Alignement avec la norme ISO/CEI 16262
ES3	1999	Ajout d'expressions régulières, try/catch, etc.
ES4	Abandonné	Jamais officiellement publié

ES5	2009	Strict mode, JSON support, méthodes Array
ES6 (ES2015)	2015	Classes, modules, arrow functions, promesses, etc.
ES2016	2016	Opérateur d'exponentiation, Array.prototype.includes
ES2017	2017	Async/await, Object.values/entries
ES2018	2018	Rest/spread pour objets, promise.finally()
ES2019	2019	Array.flat(), Object.fromEntries()
ES2020	2020	Nullish coalescing, Optional chaining
ES2021	2021	String.replaceAll(), Promise.any()
ES2022	2022	Top-level await, .at() méthode pour indexation
ES2023	2023	Array.findLast(), nouvelles méthodes pour objets

JAVASCRIPT ET LES GUERRES DES NAVIGATEURS

Dans les années 1990 et au début des années 2000, JavaScript a été au cœur de ce qu'on appelle "les guerres des navigateurs" :

- Microsoft a introduit JScript, sa propre implémentation
- Les différentes implémentations entre navigateurs ont créé des inconsistances et des défis pour les développeurs
- La célèbre phrase "Écrit une fois, teste partout" est devenue une réalité frustrante
- Des bibliothèques comme jQuery sont apparues pour résoudre ces problèmes de compatibilité

L'ÈRE MODERNE DE JAVASCRIPT (2009-AUJOURD'HUI)

Plusieurs facteurs ont contribué à la renaissance de JavaScript :

- a) La montée d'AJAX (2005-2006) : Permettant des mises à jour asynchrones des pages web
- b) L'arrivée de moteurs JS performants :
 - a. V8 de Google (2008) - utilisé dans Chrome et plus tard Node.js
 - b. SpiderMonkey de Mozilla
 - c. JavaScriptCore d'Apple
- c) Node.js (2009) : A permis d'utiliser JavaScript côté serveur
- d) ES6/ES2015 : Une mise à jour majeure qui a modernisé le langage
- e) Frameworks et bibliothèques modernes :
 - a. jQuery (2006)
 - b. AngularJS (2010)
 - c. React (2013)
 - d. Vue.js (2014)
- f) TypeScript (2012) : Sur-ensemble typé de JavaScript développé par Microsoft

L'ÉCOSYSTÈME JAVASCRIPT AUJOURD'HUI

Aujourd'hui, JavaScript fait partie d'un écosystème riche :

- ✉ Full-stack JavaScript : Du front-end au back-end avec Node.js
- ✉ Applications mobiles : React Native, Ionic, NativeScript
- ✉ Applications de bureau : Electron
- ✉ Internet des objets (IoT) : Johnny-Five, Node-RED
- ✉ Intelligence artificielle : TensorFlow.js

L'évolution de JavaScript reflète celle du web lui-même. D'un simple langage de script pour la validation de formulaires, il est devenu un langage polyvalent utilisé dans presque tous les domaines du développement logiciel. Sa capacité à s'adapter et à évoluer a assuré sa pertinence continue, et aujourd'hui, connaître JavaScript est une compétence essentielle pour tout développeur web.

La section suivante explorera la place que JavaScript occupe dans l'écosystème du développement web moderne, et comment il s'intègre avec d'autres technologies pour créer des expériences web riches et interactives.

1.2. PLACE DE JAVASCRIPT DANS LE DÉVELOPPEMENT WEB MODERNE

Après avoir exploré l'histoire et l'évolution de JavaScript, nous allons maintenant examiner sa place centrale dans l'écosystème du développement web moderne. JavaScript n'est plus simplement un "langage de script pour le navigateur" mais est devenu l'épine dorsale de nombreuses applications et technologies web contemporaines.

LA TRIADE DU DÉVELOPPEMENT WEB

Le développement web moderne repose sur trois technologies fondamentales, souvent appelées "la triade du web" :

Technologie	Rôle
HTML	Structure et contenu des pages web
CSS	Présentation et design des pages web
JavaScript	Comportement et interactivité des pages web

Si HTML et CSS sont essentiels pour créer des pages web statiques, c'est JavaScript qui transforme ces pages en applications interactives et dynamiques.

LES DIFFÉRENTS RÔLES DE JAVASCRIPT

DÉVELOPPEMENT FRONT-END

Dans le navigateur, JavaScript est utilisé pour :

- Manipuler le DOM (Document Object Model) : modifier dynamiquement le contenu et la structure de la page
- Gérer les événements : répondre aux actions de l'utilisateur (clics, saisies, etc.)
- Effectuer des requêtes asynchrones : communiquer avec les serveurs sans recharger la page (AJAX, Fetch API)
- Créer des animations et effets visuels : rendre l'interface utilisateur plus attrayante et intuitive
- Gérer le stockage côté client : utiliser localStorage, sessionStorage et IndexedDB

- ☞ Valider les données des formulaires : vérifier les entrées utilisateur avant leur envoi

DÉVELOPPEMENT BACK-END

Avec l'avènement de Node.js en 2009, JavaScript a rompu les barrières traditionnelles :

- ☞ Serveurs web : création de serveurs HTTP avec Express.js
- ☞ API RESTful : développement d'interfaces de programmation pour les applications
- ☞ Applications en temps réel : création d'applications basées sur les WebSockets
- ☞ Microservices : développement d'architectures distribuées
- ☞ Accès aux bases de données : connexion à MongoDB, MySQL, PostgreSQL, etc.

DÉVELOPPEMENT FULL-STACK

L'utilisation de JavaScript à la fois pour le front-end et le back-end a donné naissance au concept de "développeur full-stack JavaScript" :

- ☞ Utilisation d'un seul langage pour l'ensemble de la pile technique
- ☞ Partage de code entre client et serveur (isomorphisme)
- ☞ Meilleure cohérence dans le développement

L'ÉCOSYSTÈME DES FRAMEWORKS ET BIBLIOTHÈQUES

JavaScript a engendré un vaste écosystème de frameworks et bibliothèques :

FRAMEWORKS FRONT-END

Framework	Année	Caractéristiques clés
React	2013	Composants réutilisables, DOM virtuel, one-way data flow
Angular	2016	Framework complet, two-way data binding, injection de dépendances

Vue.js	2014	Progressif, accessible, performances élevées
Svelte	2016	Compilation au moment de la construction, moins de code

FRAMEWORKS BACK-END

Framework	Caractéristiques
Express.js	Minimaliste, rapide, flexible
Nest.js	Inspiré d'Angular, architecture modulaire
Koa.js	Léger, utilise les générateurs et async/await
Hapi.js	Robuste, axé sur la configuration

FRAMEWORKS FULL-STACK

- ☞ Next.js : Framework React avec rendu côté serveur
- ☞ Nuxt.js : Version Vue.js de Next.js
- ☞ Meteor : Plateforme full-stack pour applications en temps réel
- ☞ Remix : Framework moderne avec accent sur les conventions web

JAVASCRIPT ET LES APPLICATIONS MODERNES

APPLICATIONS WEB PROGRESSIVES (PWA)

JavaScript est au cœur des PWA qui offrent :

- ☞ Expérience similaire à celle des applications natives
- ☞ Fonctionnement hors ligne grâce aux Service Workers
- ☞ Installation sur l'écran d'accueil
- ☞ Notifications push

APPLICATIONS À PAGE UNIQUE (SPA)

- ☞ Navigation fluide sans rechargement de page
- ☞ Expérience utilisateur améliorée
- ☞ Utilisation intensive de JavaScript pour gérer les états et le routage

APPLICATIONS JAMSTACK

- ☞ JavaScript, APIs et Markup (d'où JAM)
- ☞ Pré-rendu et découplage du back-end
- ☞ Sites statiques avec fonctionnalités dynamiques via JavaScript

TENDANCES ÉMERGENTES OÙ JAVASCRIPT EST CENTRAL

A- SERVERLESS

JavaScript est l'un des langages les plus populaires pour le développement de fonctions serverless (FaaS) :

- ☞ AWS Lambda
- ☞ Google Cloud Functions
- ☞ Azure Functions

B- WEB COMPONENTS

Standard W3C permettant de créer des composants web réutilisables :

- ☞ Custom Elements
- ☞ Shadow DOM
- ☞ HTML Templates

C- . WEBASSEMBLY (WASM)

Complément à JavaScript permettant d'exécuter du code compilé à des vitesses quasi natives :

- ☞ Utilisation de langages comme C, C++ et Rust dans le navigateur
- ☞ JavaScript reste le langage de "colle" qui interagit avec WASM

D- . DEVOPS ET AUTOMATISATION

- ☞ Utilisation de JavaScript pour les scripts de déploiement
- ☞ Configuration des pipelines CI/CD avec des outils comme GitHub Actions

DÉFIS ET CONTROVERSES

Malgré sa domination, JavaScript n'est pas sans controverses :

- ☞ Fatigue JavaScript : prolifération excessive de frameworks et d'outils
- ☞ Questions de performance : défis liés à l'optimisation
- ☞ Sécurité : vulnérabilités potentielles (XSS, injection, etc.)
- ☞ Accessibilité : sites trop dépendants de JavaScript peuvent poser des problèmes d'accessibilité

JavaScript occupe aujourd'hui une place incontournable dans le développement web moderne. De simple langage de script, il s'est transformé en une technologie polyvalente présente à tous les niveaux de la pile de développement. Sa capacité à évoluer et à s'adapter aux besoins changeants du web en fait un outil essentiel pour tout développeur moderne.

Dans la prochaine section, nous explorerons l'environnement de développement JavaScript, les outils et les configurations qui vous aideront à devenir un développeur JavaScript efficace.

1.3.1. ÉDITEURS DE CODE ET IDE

Un bon environnement de développement est essentiel pour programmer efficacement en JavaScript. Le choix d'un éditeur de code ou d'un environnement de développement intégré (IDE) adapté peut considérablement améliorer votre productivité, la qualité de votre code et votre expérience globale de développement.

DIFFÉRENCE ENTRE ÉDITEURS DE CODE ET IDE

Avant d'explorer les options disponibles, clarifions la différence entre un éditeur de code et un IDE :

Caractéristique	Éditeur de code	IDE (Environnement de Développement Intégré)
Objectif principal	Édition de fichiers texte avec fonctionnalités pour le code	Solution complète intégrant plusieurs outils
Performances	Généralement plus léger et rapide	Plus lourd mais avec davantage de fonctionnalités
Extensibilité	Extensible via plugins/extensions	Souvent hautement extensible également
Fonctionnalités intégrées	Limitées aux fonctions d'édition essentielles	Inclut débogueur, gestion de version, tests, etc.
Courbe d'apprentissage	Généralement plus facile à prendre en main	Peut nécessiter plus de temps pour maîtriser

Éditeurs de code populaires pour JavaScript

VISUAL STUDIO CODE (VS CODE)

VS Code est devenu l'éditeur de code le plus populaire pour le développement JavaScript :

- Avantages :
 - Gratuit et open-source (développé par Microsoft)
 - Très léger mais puissant
 - Écosystème riche d'extensions

- Excellente intégration avec Git
 - Support intégré pour JavaScript et TypeScript
 - Terminal intégré
 - IntelliSense pour l'auto-complétion
-
- Extensions essentielles pour JavaScript :
 - ESLint - Linting de code JavaScript
 - Prettier - Formatage de code
 - JavaScript (ES6) code snippets
 - Live Server - Serveur local avec rechargement en direct
 - Debugger for Chrome/Firefox/Edge

SUBLIME TEXT

Sublime Text est un éditeur léger et rapide, apprécié pour sa performance :

- Avantages :
 - Extrêmement rapide, même avec de grands fichiers
 - Interface minimaliste mais puissante
 - Système de packages extensible
 - Recherche et remplacement avancés

- Inconvénients :
 - Logiciel payant (bien qu'utilisable indéfiniment en version d'évaluation)
 - Moins de fonctionnalités intégrées que VS Code ou les IDE

IDE COMPLETS POUR JAVASCRIPT

WEBSTORM

WebStorm de JetBrains est un IDE spécialement conçu pour JavaScript :

- Avantages :
 - Support complet pour JavaScript, TypeScript, HTML, CSS
 - Refactoring intelligent

- Débogage avancé intégré
- Intégration avec les frameworks populaires (React, Angular, Vue.js)
- Navigation de code puissante
- Intégration des tests

- Inconvénients :
 - Payant (abonnement annuel, avec version gratuite pour étudiants)
 - Consommation de ressources plus élevée

VISUAL STUDIO

Visual Studio est l'IDE complet de Microsoft :

- Avantages :
 - Extrêmement puissant et complet
 - Excellent pour les grands projets
 - Support avancé de débogage
 - Parfait pour le développement .NET avec JavaScript

- Inconvénients:
 - Plus lourd que VS Code
 - Courbe d'apprentissage plus prononcée
 - La version complète est payante (Community Edition gratuite disponible)

Personnalisation de votre environnement

Quel que soit votre choix, personnalisez votre environnement pour maximiser votre productivité :

- ☞ Thèmes : Choisissez un thème qui réduit la fatigue oculaire.
- ☞ Extensions : Ajoutez seulement celles qui améliorent réellement votre flux de travail.
- ☞ Snippets de code : Créez des morceaux de code réutilisables pour les tâches répétitives.
- ☞ Raccourcis clavier : Apprenez et personnalisez les raccourcis pour accélérer votre travail.
- ☞ Synchronisation des paramètres : Utilisez des outils comme Settings Sync pour VS Code afin de conserver vos configurations entre différentes machines.

Le choix d'un éditeur ou d'un IDE est une décision personnelle qui dépend de vos préférences, de votre flux de travail et du type de projets sur lesquels vous travaillez. Pour les débutants en JavaScript, Visual Studio Code offre probablement le meilleur équilibre entre facilité d'utilisation, puissance et écosystème. Cependant, n'hésitez pas à explorer d'autres options pour trouver celle qui correspond le mieux à vos besoins.

1.3.2. LES OUTILS DE DÉBOGAGE

Le débogage est une étape cruciale du développement logiciel. En JavaScript, comme dans tout langage de programmation, les erreurs (ou "bugs") sont inévitables. Savoir utiliser efficacement les outils de débogage vous permettra d'identifier rapidement les problèmes, de comprendre leur origine et de les résoudre efficacement.

Dans cette section, nous explorerons les différents outils et techniques disponibles pour déboguer du code JavaScript, que ce soit dans un navigateur ou dans un environnement Node.js.

LES TYPES D'ERREURS EN JAVASCRIPT

Avant d'explorer les outils, familiarisons-nous avec les principaux types d'erreurs que vous pourriez rencontrer :

Type d'erreur	Description	Exemple
Erreurs de syntaxe	Problèmes dans la structure du code	Oublier un point-virgule, une parenthèse
Erreurs de référence	Utilisation de variables ou fonctions non définies	Faute de frappe dans un nom de variable
Erreurs de type	Opérations sur des types de données incompatibles	Tenter d'appeler comme une fonction quelque chose qui n'en est pas une
Erreurs logiques	Le code s'exécute mais ne produit pas le résultat attendu	Condition incorrecte dans une boucle
Erreurs asynchrones	Problèmes dans le code asynchrone	Promesse non gérée, callback mal utilisé

TECHNIQUES DE DÉBOGAGE DE BASE

1. UTILISATION DE CONSOLE.LOG()

La méthode la plus simple, mais toujours efficace :

```
console.log("Valeur de x:", x);
console.table(arrayOfObjects); // Affiche un tableau formaté
console.dir(object, {depth: null}); // Affiche une représentation interactive
console.time("Opération") et console.timeEnd("Opération"); // Mesure du temps
console.trace(); // Affiche la pile d'appels
// By Lévi GOTÉNI
```

Bonnes pratiques :

- ☞ Utilisez des libellés descriptifs
- ☞ N'oubliez pas de supprimer ou commenter les logs inutiles
- ☞ Pour les objets complexes, utilisez JSON.stringify(obj, null, 2)

2. ASSERTIONS

```
console.assert(condition, "Message d'erreur si la condition est fausse");
```

3. DÉBOGAGE VIA LES COMMENTAIRES

Commentez temporairement des portions de code pour isoler la source du problème.

OUTILS DE DÉBOGAGE DES NAVIGATEURS

CHROME DEVTOOLS

Les outils de développement de Chrome (similaires dans Edge, Brave, Opera) offrent :

Le panneau Console

- ☞ Exécution de code JavaScript en direct
- ☞ Affichage des erreurs avec lien vers le code source
- ☞ Filtrage des messages par niveau (erreurs, avertissements, etc.)
- ☞ Préservation des logs entre les rechargements avec "Preserve log"

LE PANNEAU SOURCES

- ☞ Affichage du code source
- ☞ Points d'arrêt (breakpoints)
- ☞ Exécution pas à pas du code (step over, step into, step out)
- ☞ Points d'arrêt conditionnels
- ☞ Modification du code à la volée
- ☞ Visualisation de la pile d'appels (call stack)

Type de point d'arrêt	Utilisation
Ligne de code	Arrêt sur une ligne spécifique
Conditionnel	Arrêt si une condition est vraie
XHR/Fetch	Arrêt sur des requêtes réseau
Événement	Arrêt sur des événements DOM
Exception	Arrêt lorsqu'une exception est levée

LE PANNEAU NETWORK

- ☞ Analyse des requêtes réseau
- ☞ Temps de chargement
- ☞ En-têtes et contenus des requêtes/réponses
- ☞ Simulation de conditions réseau (throttling)

FIREFOX DEVELOPER TOOLS

Similaires à Chrome DevTools avec quelques fonctionnalités uniques :

- ☞ Inspecteur de stockage pour cookies, localStorage, etc.
- ☞ Débogueur de Network Monitor plus détaillé

- ☞ Meilleur support pour CSS Grid et Flexbox

SAFARI WEB INSPECTOR

Pour les développeurs sur macOS et les tests sur Safari :

- ☞ Interface plus légère mais fonctionnalités similaires
- ☞ Parfait pour tester sur iOS via la connexion d'un appareil

DÉBOGAGE DANS LES ENVIRONNEMENTS NODE.JS

1. DÉBOGAGE EN LIGNE DE COMMANDE

« bash »

`node inspect monfichier.js`

Commandes importantes :

- ☞ cont ou c : continuer l'exécution
- ☞ next ou n : passer à la ligne suivante
- ☞ step ou s : entrer dans une fonction
- ☞ out ou o : sortir de la fonction courante
- ☞ watch('expression') : surveiller une expression

2. DÉBOGAGE AVEC L'INSPECTEUR CHROME

« bash »

`node --inspect monfichier.js`

ou pour arrêter l'exécution au début:

`node --inspect-brk monfichier.js`

Puis ouvrez `chrome://inspect` dans Chrome.

3. DÉBOGAGE INTÉGRÉ DANS LES IDE

VS Code

- ☞ Configuration via launch.json
- ☞ Points d'arrêt directement dans l'éditeur
- ☞ Variables, pile d'appels et expressions de surveillance
- ☞ Console de débogage intégrée

WebStorm et autres IDE JetBrains

- ☞ Débogage intégré sans configuration
- ☞ Prise en charge avancée des applications Node.js et front-end

OUTILS SPÉCIALISÉS DE DÉBOGAGE

1. MONITEURS D'ERREURS EN PRODUCTION

- ☞ Sentry : Capture des erreurs en production avec contexte
- ☞ LogRocket : Enregistre les sessions utilisateurs pour reproduire les problèmes
- ☞ Rollbar : Suivi d'erreurs avec alertes et rapports

2. OUTILS D'ANALYSE STATIQUE

- ☞ ESLint : Déetecte les problèmes potentiels avant l'exécution
- ☞ TypeScript : Ajoute une vérification de type statique
- ☞ SonarQube : Analyse complète de la qualité du code

3. TESTS AUTOMATISÉS COMME OUTILS DE DÉBOGAGE

Les tests peuvent servir d'outils de débogage:

- ☞ Tests unitaires pour isoler les problèmes
- ☞ Tests d'intégration pour détecter les problèmes d'interaction
- ☞ Tests de régression pour identifier les régressions

TECHNIQUES AVANCÉES DE DÉBOGAGE

1. DÉBOGAGE PAR BISECTION

- ☞ Utilisez git bisect pour localiser le commit qui a introduit un bug
- ☞ Particulièrement utile pour les régressions dans les grands projets

2. DÉBOGAGE ASYNCHRONE

- ☞ Points d'arrêt sur les promesses dans DevTools
- ☞ Utilisation de async/await pour rendre le code asynchrone plus lisible
- ☞ Extension "Async call stack" dans Chrome DevTools

3. DÉBOGAGE DES PERFORMANCES

- ☞ Panneau Performance dans DevTools
- ☞ Profilers mémoire pour détecter les fuites
- ☞ Outils de couverture de code pour identifier le code inutilisé

Problème de performance	Outil
CPU élevé	Profileur JavaScript
Mémoire excessive	Memory Heap Snapshot
Chargement lent	Network + Performance
Animation saccadée	Performance + Rendering

MÉTHODOLOGIE EFFICACE DE DÉBOGAGE

- E- Reproduire le problème : Créez un cas de test fiable qui reproduit l'erreur
- F- Isoler le problème : Réduisez le code au minimum nécessaire pour reproduire l'erreur
- G- Former des hypothèses : Proposez des explications possibles
- H- Tester vos hypothèses : Utilisez les outils de débogage pour vérifier
- I- Résoudre et vérifier : Corrigez le problème et assurez-vous qu'il reste résolu

BONNES PRATIQUES POUR FACILITER LE DÉBOGAGE

1. Écrire du code défensif :

```
function calcul(x, y) {
    // Validation des entrées
    if (typeof x !== 'number' || typeof y !== 'number') {
        throw new TypeError('Les paramètres doivent être des nombres');
    }
    return x * y;
}
```

2. Gérer les erreurs de manière appropriée :

```
try {
    // Code susceptible de générer une erreur
} catch (error) {
    console.error("Type d'erreur:", error.name);
    console.error("Message:", error.message);
    console.error("Stack trace:", error.stack);
}
```

3. Créer des messages d'erreur descriptifs :

```
throw new Error('Impossible de traiter la transaction: identifiant utilisateur manquant');
```

4. Utiliser le mode strict :

```
'use strict';  
  
// Le reste du code...
```

5. Structurer le code en modules testables

Maîtriser les outils de débogage est essentiel pour tout développeur JavaScript. Bien que déboguer puisse parfois être frustrant, les outils modernes rendent ce processus beaucoup plus efficace qu'auparavant.

En combinant une bonne compréhension de ces outils avec une méthodologie systématique de débogage, vous serez capable de résoudre rapidement les problèmes les plus complexes dans votre code.

Dans la prochaine section, nous verrons comment configurer un environnement de développement JavaScript complet, en intégrant tous ces outils dans un flux de travail cohérent et productif.

1.3.3. CONFIGURATION D'UN ENVIRONNEMENT DE TRAVAIL

Une configuration efficace de votre environnement de développement JavaScript est essentielle pour maximiser votre productivité et la qualité de votre code. Dans cette section, nous allons explorer les différentes étapes pour mettre en place un environnement de travail moderne et performant, adapté aux meilleures pratiques actuelles.

INSTALLATION DES OUTILS FONDAMENTAUX

1. NODE.JS ET NPM

Node.js est l'environnement d'exécution JavaScript côté serveur, mais il est également devenu l'outil central pour le développement front-end moderne.

Installation :

- Windows/macOS : Téléchargez et installez depuis nodejs.org
- Linux : Via le gestionnaire de paquets ou [nvm](https://github.com/nvm-sh/nvm) (recommandé)

Vérification de l'installation :

«bash»

node -v

npm -v

Utilisation de NVM (Node Version Manager) :

NVM permet de gérer plusieurs versions de Node.js simultanément :

« bash »

Installation sur Linux/macOS

```
curl -o https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash
```

Installation de Node.js via NVM

```
nvm install 18.16.0 # Version LTS (exemple)
```

```
nvm use 18.16.0
```

2. GESTIONNAIRES DE PAQUETS

NPM (NODE PACKAGE MANAGER)

Installé automatiquement avec Node.js :

```
« bash »  
#Initialisation d'un nouveau projet  
npm init  
#Installation d'un package  
npm install lodash  
# Installation globale  
npm install -g serve  
# Installation comme dépendance de développement  
npm install --save-dev jest
```

Yarn

ALTERNATIVE POPULAIRE À NPM :

```
“bash”  
# Installation  
npm install -g yarn  
# Initialisation d'un projet  
yarn init  
# Installation d'un package  
yarn add lodash
```

```
# Installation globale
```

```
yarn global add serve
```

Installation comme dépendance de développement

```
yarn add --dev jest
```

PNPM

Gestionnaire de paquets plus efficace en termes d'espace disque :

“bash”

Installation

```
npm install -g pnpm
```

Utilisation similaire à npm

```
pnpm install lodash
```

CONFIGURATION D'UN PROJET JAVASCRIPT MODERNE

1. STRUCTURE DE RÉPERTOIRES RECOMMANDÉE

Pour un projet front-end typique :

```
mon-projet/
```

```
    |--- node_modules/      # Dépendances (généré automatiquement)
```

```
    |--- public/            # Fichiers statiques
```

```
        |   |--- index.html
```

```
        |   |--- favicon.ico
```

```
        |   \--- assets/
```

```

└── src/          # Code source
  ├── components/  # Composants UI
  ├── services/    # Services/API
  ├── utils/       # Fonctions utilitaires
  ├── styles/      # CSS/SCSS
  ├── app.js       # Point d'entrée
  └── index.js

  ├── tests/        # Tests
  └── .gitignore    # Fichiers à ignorer par Git

  ├── package.json  # Configuration npm
  └── README.md     # Documentation

  └── [fichiers de configuration]

```



Pour un projet Node.js :

```

mon-api/
  ├── node_modules/
  └── src/
    ├── controllers/  # Contrôleurs
    ├── models/       # Modèles de données
    ├── routes/        # Définition des routes
    ├── middleware/   # Middleware
    ├── utils/         # Fonctions utilitaires
    ├── config/        # Configuration
    └── index.js       # Point d'entrée

    ├── tests/        # Tests
    └── .env.example  # Exemple de variables d'environnement

```



```

├-- package.json
└-- [fichiers de configuration]

```

2. FICHIERS DE CONFIGURATION ESSENTIELS

Fichier	Description
package.json	Configuration du projet, dépendances, scripts
.gitignore	Fichiers à ignorer par Git
.env	Variables d'environnement (ne pas versionner)
.eslintrc.js	Configuration ESLint
.prettierrc	Configuration Prettier
tsconfig.json	Configuration TypeScript
babel.config.js	Configuration Babel
webpack.config.js	Configuration Webpack
jest.config.js	Configuration Jest

3. CONFIGURATION DES OUTILS DE QUALITÉ DE CODE

ESLINT

```

« bash »

# Installation
npm install --save-dev eslint

# Initialisation avec assistant
npx eslint --init

```

Exemple de configuration .eslintrc.js :

```
« javascript »  
  
module.exports = {  
  
  env: {  
  
    browser: true,  
  
    es2021: true,  
  
    node: true,  
  
  },  
  
  extends: ['eslint:recommended'],  
  
  parserOptions: {  
  
    ecmaVersion: 'latest',  
  
    sourceType: 'module',  
  
  },  
  
  rules: {  
  
    'no-unused-vars': 'warn',  
  
    'no-console': process.env.NODE_ENV === 'production' ? 'warn' : 'off',  
  
  },  
  
};
```

PRETTIER

```
“bash”  
  
# Installation  
  
npm install --save-dev prettier  
  
  
# Intégration avec ESLint  
  
npm install --save-dev eslint-config-prettier eslint-plugin-prettier
```

Exemple de configuration .prettierrc :

```
"json"
{
  "singleQuote": true,
  "semi": true,
  "tabWidth": 2,
  "printWidth": 80,
  "trailingComma": "es5"
}
```

HUSKY (HOOKS GIT)

```
"bash"
# Installation
npm install --save-dev husky lint-staged
```

CONFIGURATION DANS PACKAGE.JSON

Exemple de configuration dans package.json :

```
"json"
{
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "files": [
      "src/**/*.{js,ts,jsx,tsx}",
      "tests/**/*.{js,ts,jsx,tsx}"
    ],
    "exec": "pnpm run lint"
  }
}
```

```

  "*.{js,jsx}": ["eslint --fix", "prettier --write"]

}

}

```

CONFIGURATION SPÉCIFIQUE PAR TYPE DE PROJET

1. Projet vanilla JavaScript

```

« bash »

# Initialisation

mkdir mon-projet && cd mon-projet

npm init -y

npm install --save-dev eslint prettier webpack webpack-cli babel-loader @babel/core @babel/preset-env

```

Exemple de webpack.config.js basique :

“javascript”

```

const path = require('path');

module.exports = {

  entry: './src/index.js',

  output: {

    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  module: {

    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {

```

```
    loader: 'babel-loader',  
  },  
},  
],  
},  
mode: 'development',  
};
```

2. PROJET REACT

“bash”

```
# Avec Create React App  
npx create-react-app mon-app-react  
  
# Ou avec Vite (plus rapide)  
npm create vite@latest mon-app-react --template react
```

3. PROJET VUE.JS

« bash »

```
# Avec Vue CLI  
npm install -g @vue/cli  
vue create mon-app-vue  
  
# Ou avec Vite  
npm create vite@latest mon-app-vue --template vue
```

4. PROJET NODE.JS

```
« bash »

# Initialisation

mkdir mon-api && cd mon-api

npm init -y

npm install express mongoose dotenv

npm install --save-dev nodemon eslint jest
```

Exemple de script dans package.json :

```
"json"
{
  "scripts": {
    "start": "node src/index.js",
    "dev": "nodemon src/index.js",
    "test": "jest"
  }
}
```

Optimisation de l'environnement de développement

1. SCRIPTS NPM UTILES

Dans votre package.json :

```
"json"
{
  "scripts": {
```

```
"start": "node src/index.js",
"dev": "nodemon src/index.js",
"build": "webpack --mode production",
"lint": "eslint src/**/*.js",
"lint:fix": "eslint src/**/*.js --fix",
"format": "prettier --write \"src/**/*.{js,jsx,json,css}\"",
"test": "jest",
"test:watch": "jest --watch",
"prepare": "husky install"
}
}
```

2. CONFIGURATION DE VS CODE

Créez un fichier `vscode/settings.json` dans votre projet :

```
"json"
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "eslint.validate": ["javascript", "javascriptreact"],
  "javascript.updateImportsOnFileMove.enabled": "always"
}
```

ENVIRONNEMENT DOCKER POUR JAVASCRIPT

Docker permet d'isoler votre environnement de développement pour une meilleure cohérence entre les machines.

1. FICHIER DOCKERFILE BASIQUE POUR NODE.JS

“dockerfile”

FROM node:18-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY ..

EXPOSE 3000

CMD ["npm", "start"]

2. CONFIGURATION DOCKER COMPOSE

Exemple de `docker-compose.yml` pour une application web avec base de données :

« yaml »

version: '3'

services:

app:

```
build: .  
  
ports:  
  - "3000:3000"  
  
volumes:  
  - ./app  
  - /app/node_modules  
  
depends_on:  
  - db  
  
environment:  
  - NODE_ENV=development  
  - DATABASE_URL=mongodb://db:27017/myapp
```

```
db:  
  image: mongo:latest  
  
  ports:  
    - "27017:27017"  
  
  volumes:  
    - mongodb_data:/data/db
```

```
volumes:  
  mongodb_data:
```

GESTION DES SECRETS ET VARIABLES D'ENVIRONNEMENT

1. CONFIGURATION AVEC DOTENV

« bash »

```
npm install dotenv
```

Exemple de fichier .env :

```
PORT=3000
```

```
DATABASE_URL=mongodb://localhost:27017/myapp
```

```
API_KEY=your_secret_api_key
```

```
NODE_ENV=development
```

Utilisation dans votre code :

« javascript »

```
require('dotenv').config();
```

```
const port = process.env.PORT || 3000;
```

2. Bonnes pratiques

- ☞ Ne jamais versionner les fichiers `.env` contenant des secrets (ajoutez-les à `.gitignore`)
- ☞ Fournir un fichier ``.env.example`` avec la structure mais sans valeurs sensibles
- ☞ Utiliser des solutions comme Vault ou les secrets de GitHub/GitLab pour les environnements CI/CD

Configuration avancée de l'écosystème JavaScript

1. TypeScript

« bash »

```
# Installation
```

```
npm install --save-dev typescript @types/node
```

```
# Initialisation
```

```
npx tsc --init
```

Exemple de configuration tsconfig.json :

```
"json"
{
  "compilerOptions": {
    "target": "es2018",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "./dist"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.{test}.ts"]
}
```

2. Babel pour la compatibilité navigateur

« bash »

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env core-js
```

Exemple de configuration babel.config.js :

```
"javascript"
module.exports = {
  presets: [
```

```
[  
  '@babel/preset-env',  
  {  
    targets: '> 0.25%, not dead',  
    useBuiltIns: 'usage',  
    corejs: 3,  
  },  
,  
,  
];
```

3. Configuration de tests avec Jest

“bash »
npm install --save-dev jest @types/jest babel-jest

Exemple de configuration `jest.config.js` :

« javascript »
module.exports = {
 testEnvironment: 'node',
 roots: ['<rootDir>/src'],
 testMatch: ['**/_tests_/**/*.{js,ts}', '**/?(*.)+(spec|test).{js,ts}'],
 coverageDirectory: 'coverage',
};

La configuration d'un environnement de développement JavaScript efficace nécessite plusieurs étapes, mais cet investissement initial vous fera gagner beaucoup de temps par la suite. Un environnement bien configuré favorise :

- Une meilleure qualité de code
- Une détection précoce des erreurs
- Une expérience de développement plus agréable
- Une collaboration plus fluide au sein des équipes

2.1.1. STRUCTURE D'UN SCRIPT JAVASCRIPT

Avant de plonger dans les concepts avancés de JavaScript, il est essentiel de comprendre comment structurer correctement un script JavaScript. Une bonne structure facilite la maintenance, la lisibilité et l'efficacité de votre code. Dans cette section, nous explorerons les bases de la structure d'un script JavaScript.

INTÉGRATION DE JAVASCRIPT DANS UNE PAGE WEB

Il existe plusieurs façons d'intégrer du code JavaScript dans une page HTML :

1. Script intégré directement dans HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Ma page avec JavaScript</title>
    <script>
        // Code JavaScript directement dans le HTML
        function saluer() {
            alert('Bonjour !');
        }
    </script>
</head>
<body>
    <button onclick="saluer()">Cliquez-moi</button>
</body>
</html>
//by @lgoteni242
```

2. Script externe (recommandé)

```
<!DOCTYPE html>
<html>
<head>
    <title>Ma page avec JavaScript</title>
    <!-- Chargement d'un fichier JavaScript externe -->
    <script src="monscript.js"></script>
</head>
<body>
    <button id="monBouton">Cliquez-moi</button>
</body>
</html>
```

Dans le fichier `monscript.js` :

```
// Le code est dans un fichier séparé
document.addEventListener('DOMContentLoaded', function() {
    document.getElementById('monBouton').addEventListener('click', function() {
        alert('Bonjour !');
    });
});
```

POSITION DES SCRIPTS DANS LE DOCUMENT HTML

La position de la balise `<script>` dans votre document HTML est importante car elle affecte le chargement et l'exécution de votre page :

EN-TÊTE (HEAD)

```
<!DOCTYPE html>

<html>
<head>
    <script src="monscript.js"></script>
    <!-- Le script est chargé avant le rendu de la page -->
</head>
<body>
```

```
<!-- Contenu de la page -->
</body>
</html>
```

Avantages :

- Regroupement des ressources
- Peut initialiser des variables globales avant le chargement du contenu

Inconvénients :

- Bloque le rendu de la page pendant le chargement (si « async » ou « defer » ne sont pas utilisés)
- Peut causer des erreurs si le script tente d'accéder à des éléments DOM non encore chargés

FIN DE BODY (RECOMMANDÉ SANS ATTRIBUTS SPÉCIAUX)

```
<!DOCTYPE html>
<html>
<head>
    <!-- En-tête de page -->
</head>
<body>
    <!-- Contenu de la page -->
    <!-- Scripts chargés après le contenu -->
    <script src="monscript.js"></script>
</body>
</html>
```

Avantages :

- Le DOM est entièrement chargé avant l'exécution du script
- Améliore le temps de chargement perçu par l'utilisateur

UTILISATION DES ATTRIBUTS ASYNC ET DEFER

```
<head>
  <script src="analytique.js" async></script>
  <script src="fonctionnalites.js" defer></script>
</head>
```

Attribut	Fonctionnement
Async	Charge le script de manière asynchrone pendant l'analyse du HTML, puis exécute le script dès qu'il est téléchargé (même si le HTML n'est pas encore complètement analysé)
defer	Charge le script pendant l'analyse du HTML, mais attend que le HTML soit complètement analysé avant d'exécuter le script

STRUCTURE D'UN FICHIER JAVASCRIPT

Un fichier JavaScript bien structuré suit généralement une organisation logique :

1. IMPORTS (DANS LES MODULES ES6)

```
import { fonction1 } from './module1.js';
```

2. DÉCLARATION DES CONSTANTES

```
const API_URL = 'https://api.exemple.com';
const MAX_ITEMS = 100;
```

3. DÉCLARATION DES VARIABLES GLOBALES (À MINIMISER)

```
let utilisateurCourant = null;  
let estConnecte = false;
```

4. DÉCLARATION DES FONCTIONS

```
function initialiser() {  
    // Code d'initialisation  
}  
  
function traiterDonnees(donnees) {  
    // Traitement des données  
}
```

5. CLASSES

```
class Utilisateur {  
  
    constructor(nom) {  
        this.nom = nom;  
    }  
  
    saluer() {  
        return `Bonjour, ${this.nom} !`;  
    }  
}
```

6. LOGIQUE PRINCIPALE / POINT D'ENTRÉE

```
function main() {
    initialiser();
    // Suite du code principal
}
```

7. EXÉCUTION DU POINT D'ENTRÉE (SELON LE CONTEXTE)

```
if (typeof window !== 'undefined') {
    // Code exécuté dans le navigateur
    window.addEventListener('DOMContentLoaded', main);
} else {
    // Code exécuté dans Node.js
    main();
}
```

8. EXPORTS (DANS LES MODULES ES6)

```
export { Utilisateur, traiterDonnees };
```

ORGANISATION DU CODE EN MODULES

JavaScript moderne encourage l'organisation du code en modules pour améliorer la maintenabilité et la réutilisabilité :

Modules ES6 (ECMAScript 2015+)

MATH.JS - MODULE POUR LES FONCTIONS MATHÉMATIQUES

```
export function additionner(a, b) {  
    return a + b;  
}
```

```
export function multiplier(a, b) {  
    return a * b;  
}
```

```
export const PI = 3.14159;
```

UTILISATION DANS UN AUTRE FICHIER

```
import { additionner, multiplier, PI } from './math.js';  
  
import * as Math from './math.js';  
  
console.log(additionner(2, 3)); // 5  
console.log(Math.multiplier(2, 3)); // 6
```

Pour utiliser les modules ES6 dans le navigateur :

“html”

`<script type="module" src="app.js"></script>`

MODÈLE COMMONJS (NODE.JS)

MATH.JS - STYLE NODE.JS

```
function additionner(a, b) {
    return a + b;
}
```

```
function multiplier(a, b) {
    return a * b;
}
```

```
const PI = 3.14159;
```

```
module.exports = {
    additionner,
    multiplier,
    PI
};
```

UTILISATION DANS UN AUTRE FICHIER NODE.JS

```
const math = require('./math.js');
console.log(math.additionner(2, 3)); // 5
```

PATTERNS D'ORGANISATION COURANTS

PATTERN MODULE (VIA IIFE)

Le pattern module utilise une fonction immédiatement invoquée (IIFE) pour créer un scope privé :

```
const MonModule = (function() {  
    // Variables privées  
    let compteur = 0;
```

FONCTIONS PRIVÉES

```
        function incrementerCompteur() {  
            compteur++;  
        }
```

INTERFACE PUBLIQUE

```
        return {  
            incrementer: function() {  
                incrementerCompteur();  
            },  
            getCompteur: function() {  
                return compteur;  
            },  
            reinitialiser: function() {  
                compteur = 0;  
            }  
        };  
    }();
```

UTILISATION

```
MonModule.incrementer();

console.log(MonModule.getCompteur()); // 1

// console.log(compteur); // Erreur: compteur n'est pas accessible
```

PATTERN RÉVÉLATEUR

Une variation du pattern module qui définit toutes les fonctions privées et publiques à l'intérieur du scope privé :

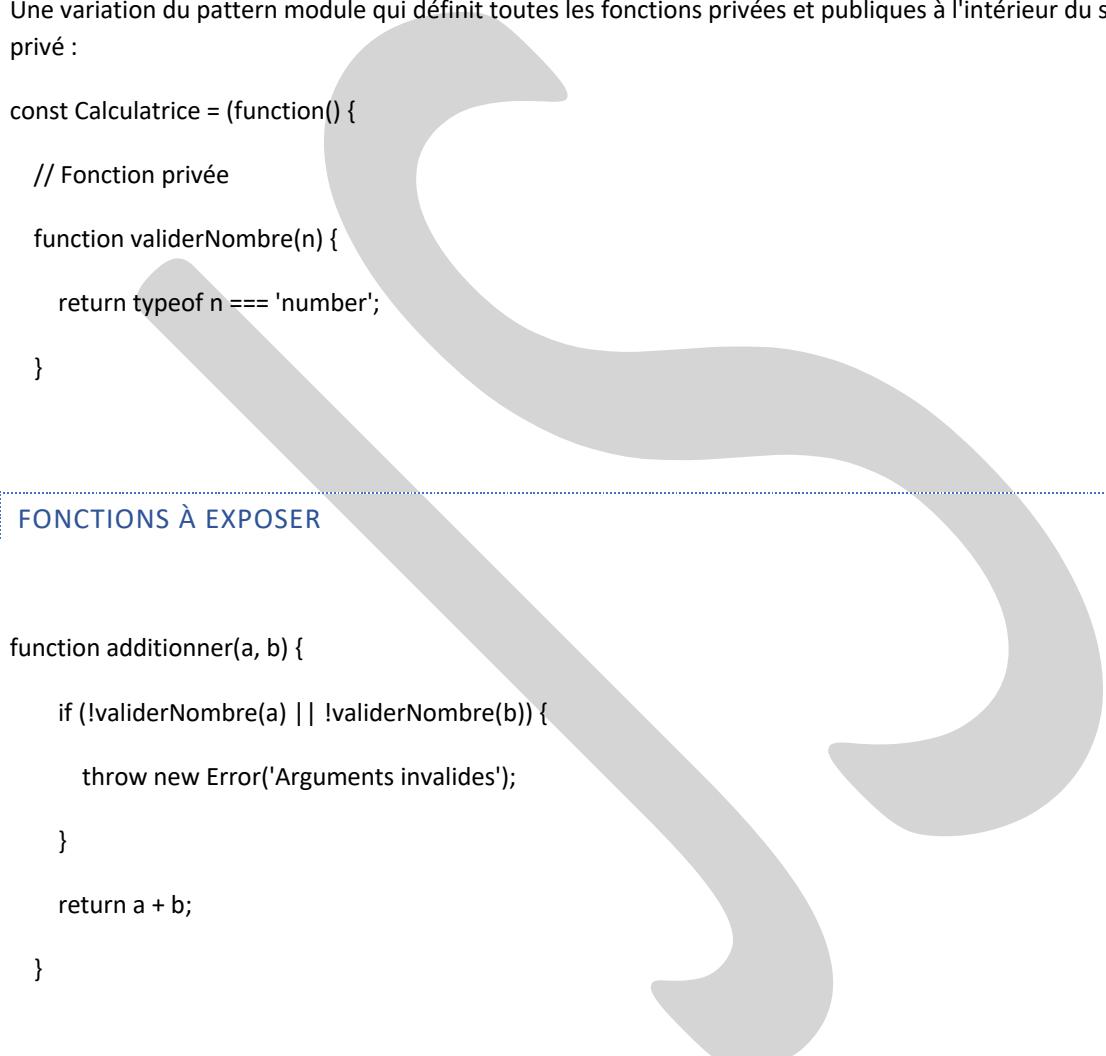
```
const Calculatrice = (function() {

    // Fonction privée

    function validerNombre(n) {
        return typeof n === 'number';
    }

    }
```

FONCTIONS À EXPOSER



```
function additionner(a, b) {

    if (!validerNombre(a) || !validerNombre(b)) {
        throw new Error('Arguments invalides');
    }

    return a + b;
}

function multiplier(a, b) {

    if (!validerNombre(a) || !validerNombre(b)) {
        throw new Error('Arguments invalides');
    }

    return a * b;
}
```

INTERFACE PUBLIQUE

```
return {  
    additionner: additionner,  
    multiplier: multiplier  
};  
}();
```

BONNES PRATIQUES DE STRUCTURATION

1. Séparation des préoccupations : Divisez votre code en fichiers ou modules logiques selon leur fonction.
2. Principes SOLID: Appliquez les principes de conception orientée objet.
3. Évitez les variables globales : Limitez l'utilisation de variables dans le scope global.
4. Organisation cohérente : Suivez une structure cohérente dans tous vos fichiers.
5. Documentation : Incluez des commentaires explicatifs pour les parties complexes.
6. Gestion des dépendances : Explicitez clairement les dépendances entre modules.

Une bonne structure de script JavaScript est fondamentale pour développer des applications maintenables et évolutives. En suivant des conventions établies et en organisant logiquement votre code, vous facilitez non seulement votre propre travail mais aussi celui des autres développeurs qui pourraient collaborer sur votre projet.

2.1.2. COMMENTAIRES

Les commentaires sont une partie essentielle de tout code JavaScript bien écrit. Ils permettent d'expliquer le fonctionnement de votre code, de documenter les décisions techniques et de faciliter la maintenance. Dans cette section, nous explorerons les différents types de commentaires en JavaScript et les bonnes pratiques pour les utiliser efficacement.

TYPES DE COMMENTAIRES EN JAVASCRIPT

JavaScript prend en charge deux types principaux de commentaires :

1. COMMENTAIRES SUR UNE LIGNE

Utilisez deux barres obliques (`//`) pour créer un commentaire sur une seule ligne :

```
// Ceci est un commentaire sur une ligne
```

```
let compteur = 0; // Les commentaires peuvent aussi être placés après le code
```

2. COMMENTAIRES SUR PLUSIEURS LIGNES

Utilisez /* pour commencer un commentaire multi-lignes et */ pour le terminer :

```
/*
 * Ceci est un commentaire
 * sur plusieurs lignes.
 * Il peut s'étendre sur autant de lignes que nécessaire.
*/
function calculerSomme(a, b) {
    return a + b;
}
```

```
}
```

ANNOTATIONS DE DOCUMENTATION (JSDOC)

JSDoc est un système de documentation pour JavaScript qui utilise des commentaires spéciaux pour générer de la documentation API. Ces commentaires commencent par `/**` et se terminent par `*/` :

```
/**
 * Calcule la somme de deux nombres.
 *
 * @param {number} a - Premier nombre à additionner
 * @param {number} b - Second nombre à additionner
 * @returns {number} La somme des deux nombres
 * @throws {TypeError} Si les paramètres ne sont pas des nombres
 * @example
 * // Retourne 8
 * calcularSomme(3, 5);
 */
function calcularSomme(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new TypeError('Les paramètres doivent être des nombres');
  }
  return a + b;
}
```

TAGS JSDOC COURANTS

Tag	Description	Exemple
@param	Décrit un paramètre de fonction	@param {string} nom - Nom de l'utilisateur
@returns	Décrit la valeur de retour	@returns {boolean} True si réussi, false sinon
@throws	Documente les exceptions possibles	@throws {Error} Si la connexion échoue
@example	Fournit un exemple d'utilisation	@example getValue(5)
@deprecated	Indique que la fonctionnalité est obsolète	@deprecated Utilisez newMethod() à la place
@see	Référence à une documentation connexe	@see OtherClass pour plus de détails
@todo	Indique les tâches à accomplir	@todo Implémenter la gestion des erreurs
@type	Spécifie le type d'une variable	@type {string[]}
@typedef	Définit un type personnalisé	@typedef {Object} UserOptions
@callback	Documente un type de fonction de rappel	@callback RequestCallback

UTILISATION DE COMMENTAIRES POUR LE DÉBOGAGE

Les commentaires peuvent être utilisés temporairement pour désactiver des portions de code lors du débogage :

```
function testFunction() {
    // Version active
    doSomething();

    // Version désactivée pour déboguer
    // doSomethingElse();

    /* Bloc de code commenté pour tests
    */
}
```

```
if (condition) {  
    complexOperation();  
}  
*/  
}
```

Commentaires spéciaux pour les outils de développement

Certains commentaires ont une signification particulière pour les outils de développement :

DIRECTIVES DE SOURCE MAP

```
« javascript »  
//# sourceMappingURL=app.min.js.map
```

DIRECTIVES POUR ESLINT

```
« javascript »  
/* eslint-disable no-console */  
console.log('Débogage');  
/* eslint-enable no-console */  
  
// Désactiver ESLint pour une ligne spécifique  
console.log('Important'); // eslint-disable-line no-console
```

Directives pour TypeScript

« typescript »

```
// @ts-ignore - Ignore l'erreur TypeScript sur la ligne suivante
element.nonExistentMethod();
```

```
// @ts-expect-error - Indique qu'une erreur est attendue
invalidOperation();
```

BONNES PRATIQUES POUR LES COMMENTAIRES

1. Commenter le "pourquoi", pas le "quoi"

« javascript »

```
// Mauvais : Indique ce que fait le code (évident)
// Incrémente i de 1
i++;
```

// Bon : Explique pourquoi cette action est nécessaire

```
// Passe à l'élément suivant de la collection
i++;
```

2. Maintenir les commentaires à jour

Les commentaires obsolètes sont pires que l'absence de commentaires car ils peuvent induire en erreur.
Assurez-vous de mettre à jour les commentaires lorsque vous modifiez le code.

3. Utiliser des commentaires en bloc pour la documentation API

« javascript »

```
/**
```

- * Cette fonction fait partie de l'API publique.
- * Elle est utilisée pour [...]
- */

4. Éviter les commentaires redondants

« javascript »

```
// Mauvais

// Cette fonction retourne la somme
function somme(a, b) {
    return a + b; // Retourne la somme
}

// Bon

// Calcule la somme avec arrondi au centième près
function somme(a, b) {
    return Math.round((a + b) * 100) / 100;
}
```

5. Commentaires TODO et FIXME

Les commentaires TODO et FIXME sont souvent utilisés pour marquer du code qui nécessite des améliorations futures :

« javascript »

```
// TODO: Implémenter la validation des entrées
function processInput(input) {
    // Code temporaire
}
```

```
// FIXME: Cette fonction échoue avec des entrées négatives

function calculateSquareRoot(num) {

    return Math.sqrt(num);

}
```

De nombreux IDE détectent automatiquement ces balises et les mettent en évidence.

6. Commentaires d'en-tête de fichier

Inclure un commentaire en début de fichier peut aider à comprendre son objectif et son contexte :

```
« javascript »

/**

 * @fileoverview Module de gestion des utilisateurs

 * @author Prénom Nom <email@example.com>

 * @version 1.2.0

 * @license MIT

 * @description

 * Ce module contient les fonctionnalités de gestion des utilisateurs,
 * notamment l'authentification, l'autorisation et les opérations CRUD.

 */

```

```

## 7. Commentaires pour code complexe

Lorsque vous écrivez du code complexe, ajoutez des commentaires pour expliquer le raisonnement :

```
« javascript »
```

```

function optimizerAlgorithme(data) {
 // Utilise l'algorithme de tri fusion pour maximiser les performances
 // avec de grands ensembles de données, complexité O(n log n)

 // Première étape : préparation des données
 // ...

 // Deuxième étape : application de l'heuristique de Knuth
 // Cette approche évite le problème de [...] qui surviendrait autrement
 // ...
}

}

```

## GÉNÉRATION DE DOCUMENTATION

Les commentaires JSDoc peuvent être utilisés avec des outils comme JSDoc, Docco, ou ESDoc pour générer une documentation HTML :

```

« bash »
Installation de JSDoc
npm install -g jsdoc

```

Génération de documentation

```

jsdoc chemin/vers/mon/fichier.js -d chemin/vers/documentation

```

Équilibre entre commentaires et code lisible

Un code bien écrit devrait être en grande partie auto-documenté. Visez un équilibre :

« javascript »

```
// Mauvais : code obscur avec commentaires excessifs

function c(a, b) { // Fonction pour calculer la valeur de l'hypothénuse

 var r = 0; // Initialise le résultat

 var sa = a * a; // Calcule le carré de a

 var sb = b * b; // Calcule le carré de b

 r = Math.sqrt(sa + sb); // Calcule la racine carrée de la somme

 return r; // Retourne le résultat

}
```

// Bon : code clair avec commentaires stratégiques

```
/**
 * Calcule la longueur de l'hypothénuse d'un triangle rectangle.
 */

function calculerHypotenuse(coteA, coteB) {

 // Utilisation du théorème de Pythagore

 return Math.sqrt(coteA * coteA + coteB * coteB);

}
```

Les commentaires sont un outil puissant pour améliorer la qualité et la maintenabilité de votre code JavaScript. En suivant les bonnes pratiques présentées dans cette section, vous pourrez créer une documentation claire et utile qui facilitera la compréhension de votre code, tant pour vous-même que pour les autres développeurs travaillant sur votre projet.

N'oubliez pas qu'un bon code est auto-documenté grâce à des noms de variables et de fonctions significatifs, mais que des commentaires bien placés peuvent fournir un contexte supplémentaire essentiel pour comprendre les décisions et les intentions derrière le code.

Dans la prochaine section, nous aborderons la sensibilité à la casse en JavaScript, une caractéristique fondamentale du langage à comprendre pour éviter des erreurs courantes.

### 2.1.3. SENSIBILITÉ À LA CASSE

JavaScript est un langage sensible à la casse, ce qui signifie qu'il distingue les lettres majuscules des lettres minuscules. Cette caractéristique est fondamentale à comprendre pour éviter des erreurs courantes dans votre code. Dans cette section, nous explorerons en détail ce concept et son impact sur divers aspects du langage.

#### PRINCIPES DE BASE

En JavaScript, les identifiants suivants sont considérés comme différents :

```
let nombre = 10;
let Nombre = 20;
let NOMBRE = 30;

console.log(nombre); // 10
console.log(Nombre); // 20
console.log(NOMBRE); // 30
```

Cette sensibilité à la casse s'applique à tous les identifiants JavaScript :

- Noms de variables
- Noms de fonctions
- Noms de classes
- Mots-clés et instructions réservés
- Noms de propriétés d'objets (avec quelques nuances)

#### ERREURS COURANTES LIÉES À LA CASSE

##### 1. Erreurs de référence

L'une des erreurs les plus fréquentes est de référencer une variable avec la mauvaise casse :

```
let utilisateur = { nom: 'Alice' };

// Erreur: Uncaught ReferenceError: Utilisateur is not defined
console.log(Utilisateur.nom);
```

## 2. Confusion dans les noms similaires

```
function calculerTotal() {
 // Code pour calculer le total
}

// Tentative d'appel avec la mauvaise casse
// Erreur: Uncaught TypeError: calculerTOTAL is not a function
calculerTOTAL();
```

## 3. Mots-clés réservés

Tous les mots-clés réservés JavaScript doivent être écrits exactement comme spécifié :

```
// Correct
if (condition) {
 // Code
}

// Incorrect - erreur de syntaxe
IF (condition) {
 // Code
```

```
}
```

## CAS PARTICULIERS

### 1. PROPRIÉTÉS D'OBJETS DANS LE CODE

La sensibilité à la casse s'applique également aux propriétés d'objets dans le code JavaScript :

```
const personne = {
 nom: 'Alice',
 age: 30,
 Adresse: 'Paris'
};

console.log(personne.nom); // "Alice"
console.log(personne.Nom); // undefined
console.log(personne.Adresse); // "Paris"
console.log(personne.adresse); // undefined
```

### 2. PROPRIÉTÉS DOM ET API WEB

Historiquement, certaines API du navigateur ne sont pas strictement sensibles à la casse :

```
// Ces deux lignes sont équivalentes dans certains navigateurs
document.getElementById('monElement');

document.getElementByID('monElement'); // Fonctionne dans certains navigateurs anciens
```

Cependant, les spécifications modernes et les bonnes pratiques imposent désormais le respect de la casse :

```
// La forme standard et recommandée
document.getElementById('monElement');
```

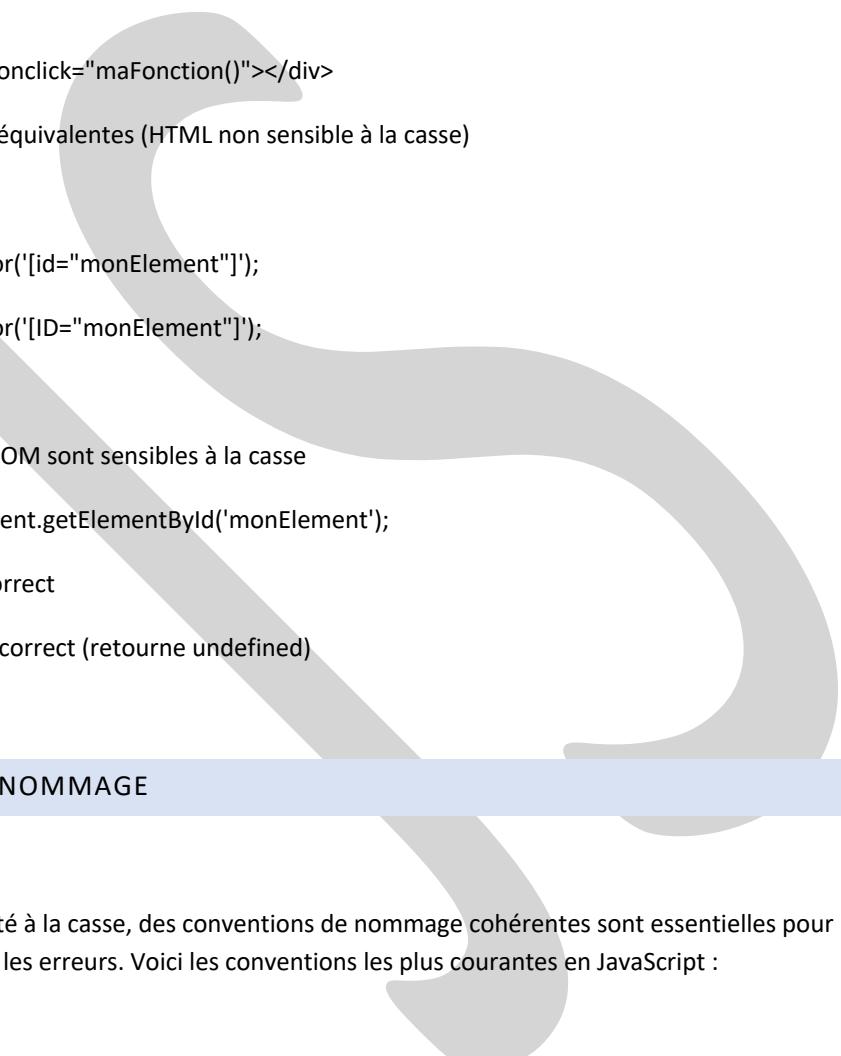
### 3. ATTRIBUTS HTML VS PROPRIÉTÉS DOM

---

Les attributs HTML ne sont pas sensibles à la casse, mais les propriétés DOM correspondantes le sont :

```
<div id="monElement" onclick="maFonction()"></div>
// Ces deux lignes sont équivalentes (HTML non sensible à la casse)
```

```
document.querySelector('[id="monElement"]);
document.querySelector('[ID="monElement"]');
```



```
// Mais les propriétés DOM sont sensibles à la casse
const element = document.getElementById('monElement');
element.onclick; // Correct
element.onClick; // Incorrect (retourne undefined)
```

## CONVENTIONS DE NOMMAGE

Étant donné la sensibilité à la casse, des conventions de nommage cohérentes sont essentielles pour maintenir un code lisible et éviter les erreurs. Voici les conventions les plus courantes en JavaScript :

### 1. CAMELCASE

---

Utilisé pour les noms de variables, fonctions et propriétés d'objets :

```
let nombreUtilisateurs = 42;
function calculerMoyenne(a, b) { /* ... */ }
const utilisateur = {
```

```
nomComplet: 'Marie Dupont',
ageUtilisateur: 28
};
```

## 2. PASCALCASE

---

Utilisé pour les classes et constructeurs :

```
class Utilisateur {
 constructor(nom) {
 this.nom = nom;
 }
}

const admin = new Utilisateur('Admin');
```

## 3. UPPER\_SNAKE\_CASE

---

Utilisé pour les constantes (vraiment constantes, pas simplement déclarées avec const) :

```
const PI = 3.14159;
const MAX_UTILISATEURS = 100;
const CODES_ERREUR = {
 NOT_FOUND: 404,
 SERVER_ERROR: 500
};
```

## 4. PRÉFIXES SPÉCIAUX

---

Certaines conventions utilisent des préfixes pour indiquer la portée ou le type :

```
// Variables privées (convention, pas une protection réelle)
```

```
let _compteurInterne = 0;
```

// Éléments DOM

```
const $bouton = document.getElementById('monBouton');
```

// Classes abstraites (convention TypeScript)

```
class AbstractModel { /* ... */ }
```

## BONNES PRATIQUES

1. Soyez cohérent : Adhérez à une convention de nommage dans tout votre code.

2. Utilisez des noms significatifs : Évitez les noms qui ne diffèrent que par la casse.

// À éviter

```
let user = { /* ... */ };
```

```
let User = { /* ... */ };
```

3. Respectez les conventions établies : Suivez les conventions de l'écosystème JavaScript.

4. Faites attention aux API externes : Vérifiez la documentation pour la casse exacte des API que vous utilisez.

5. Utilisez des outils de linting : Des outils comme ESLint peuvent vous aider à maintenir des conventions cohérentes.

## PIÈGES COURANTS

## PROPRIÉTÉS D'OBJETS DYNAMIQUES

Lorsque vous accédez aux propriétés d'un objet de manière dynamique, veillez à respecter la casse :

```
const utilisateur = {
 nom: 'Alice',
 age: 30
};
```

// Accès dynamique à une propriété

```
const proprietee = 'nom';
console.log(utilisateur[proprietee]); // "Alice"
```

// Attention à la casse

```
const proprieteeIncorrecte = 'Nom';
console.log(utilisateur[proprieteeIncorrecte]); // undefined
```

## IMPORTS ET EXPORTS DE MODULES

La sensibilité à la casse est également importante lors de l'import/export de modules :

```
// fichier user.js
export const User = { /* ... */};
```

// Dans un autre fichier

```
import { User } from './user.js'; // Correct
import { user } from './user.js'; // Erreur: user n'est pas exporté
```

## APIS DU NAVIGATEUR DANS DIFFÉRENTS ENVIRONNEMENTS

Certaines APIs peuvent avoir différentes casses selon l'environnement :

```
// Navigateur moderne
```

```
navigator.serviceWorker;
```

```
// WebView Android ancien
```

```
navigator.ServiceWorker;
```

### CAS D'INSENSIBILITÉ À LA CASSE

Il existe quelques cas où JavaScript ne fait pas la distinction entre majuscules et minuscules :

#### 1. NOMS D'ÉTIQUETTES (LABELS)

Les noms d'étiquettes pour les instructions `break` et `continue` sont insensibles à la casse dans certains navigateurs (mais pas selon la spécification) :

```
bucleExterieure: for (let i = 0; i < 3; i++) {
 for (let j = 0; j < 3; j++) {
 if (i * j >= 4) {
 break bucleExterieure; // Conforme à la spécification
 // break BUCLEEXTERIEURE; // Peut fonctionner dans certains navigateurs mais non standard
 }
 }
}
```

## 2. HTML ET CSS

---

Bien que cela ne concerne pas directement JavaScript, il est utile de noter que HTML et CSS sont insensibles à la casse :

```
<DIV id="monElement" CLASS="important">Contenu</DIV>
```

```
// Fonctionne même si la casse dans le HTML est différente
```

```
document.getElementById('monElement');
```

```
document.querySelector('.important');
```

### DÉBOGAGE DES PROBLÈMES LIÉS À LA CASSE

Si vous rencontrez des erreurs liées à la casse, voici quelques conseils :

1. Vérifiez les messages d'erreur : Les erreurs comme « ReferenceError: X is not defined » indiquent souvent un problème de casse.

2. Utilisez l'autocomplétion : Les IDE modernes suggèrent les noms avec la casse correcte.

3. Inspectez les objets : Utilisez « console.dir(objet) » ou « Object.keys(objet) » pour voir les propriétés disponibles avec leur casse exacte.

4. Activez les linters : ESLint peut signaler les identifiants non déclarés ou mal orthographiés.

### OUTILS ET CONFIGURATION

#### ESLINT

Configurez ESLint pour détecter les problèmes de casse :

```
{
 "rules": {
 "camelcase": ["error", { "properties": "never" }],
 "new-cap": ["error", { "newIsCap": true }]
 }
}
```

## ÉDITEURS DE CODE

Activez les fonctionnalités d'auto-complétion et de vérification dans votre éditeur :

- ☞ VS Code : Utilisez l'extension "JavaScript Language Features"
- ☞ WebStorm : Activez les inspections JavaScript
- ☞ Sublime Text : Installez des packages comme "SublimeLinter" et "SublimeLinter-eslint"

La sensibilité à la casse en JavaScript est un aspect fondamental du langage qui peut être source d'erreurs subtiles mais frustrantes. En comprenant bien ce concept et en suivant des conventions cohérentes, vous pouvez éviter ces problèmes et écrire du code plus robuste et maintenable.

En résumé, gardez toujours à l'esprit que variable, Variable et VARIABLE sont trois identifiants distincts en JavaScript, et prêtez une attention particulière à la casse lorsque vous travaillez avec des APIs externes ou des frameworks.

Dans la prochaine section, nous explorerons les variables et les types de données en JavaScript, en construisant sur les bases que nous avons établies jusqu'à présent.

### 2.2.1. DÉCLARATION DE VARIABLES (VAR, LET, CONST)

Les variables sont des conteneurs nommés qui permettent de stocker des données dans un programme JavaScript. La façon dont vous déclarez vos variables a un impact significatif sur leur comportement en termes de portée, réaffectation et initialisation. JavaScript propose trois mots-clés pour déclarer des variables : var, let et const, chacun avec ses propres caractéristiques et cas d'utilisation.

### Vue d'ensemble des méthodes de déclaration

Caractéristique	var	let	const
<b>Introduction</b>	ES1 (1997)	ES6 (2015)	ES6 (2015)
<b>Portée</b>	Fonction	Bloc	Bloc
<b>Hissage (Hoisting)</b>	Oui, avec initialisation « undefined »	Oui, sans initialisation (TDZ)	Oui, sans initialisation (TDZ)
<b>Redéclaration</b>	Autorisée	Non autorisée dans le même bloc	Non autorisée dans le même bloc
<b>Réassignation</b>	Autorisée	Autorisée	Non autorisée
<b>Initialisation requise</b>	Non	Non	Oui
<b>Usage moderne</b>	Déconseillé	Pour les valeurs qui changent	Pour les valeurs constantes

## LA DÉCLARATION AVEC « VAR »

### SYNTAXE ET UTILISATIONS BASIQUES

```
var age = 25; // Déclaration avec initialisation
var nom = "Alice"; // Variable contenant une chaîne de caractères
var estActif; // Déclaration sans initialisation (valeur undefined)

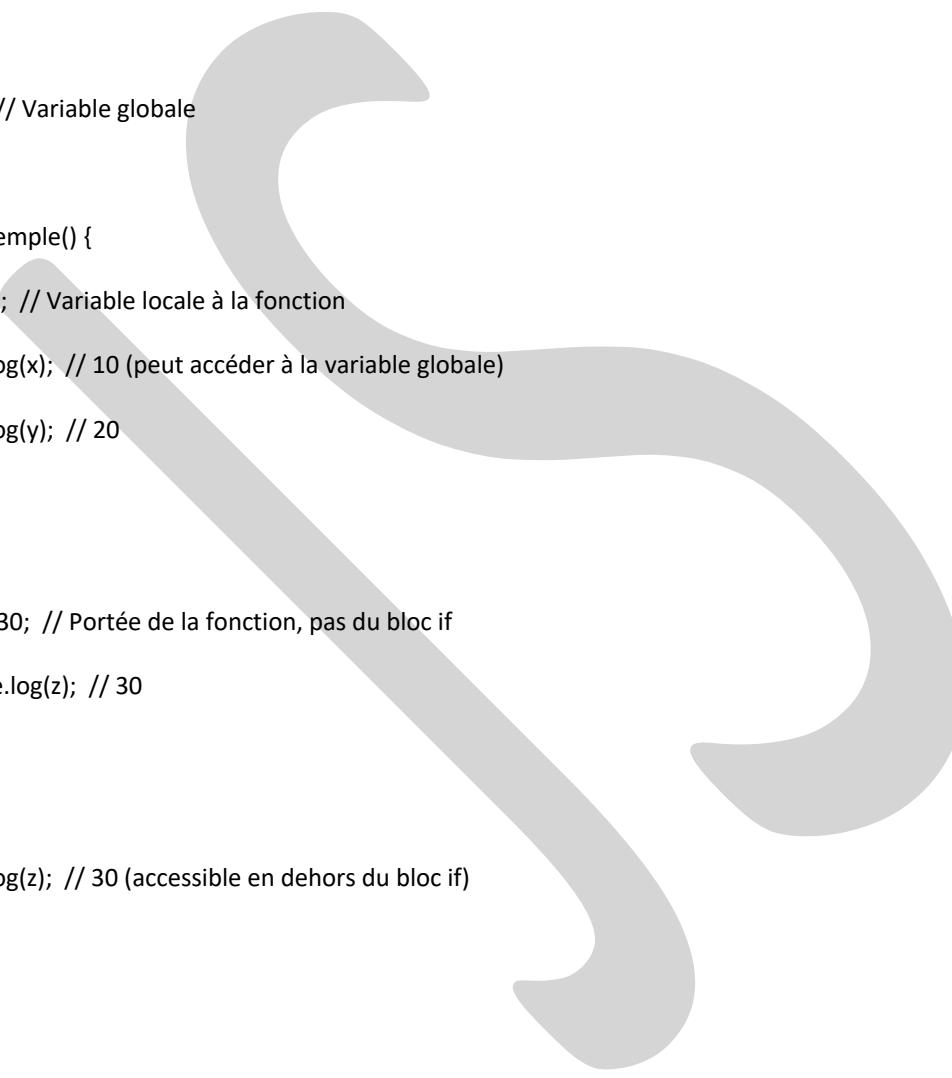
// Redéclaration autorisée
var age = 26; // Pas d'erreur, la variable est mise à jour
```

```
// Réassignton autorisée
age = 27; // Valeur modifiée
```

## PORTEE DE « VAR »

---

Les variables déclarées avec « var » ont une portée de fonction, ou globale si elles sont déclarées en dehors d'une fonction :



```
var x = 10; // Variable globale

function exemple() {
 var y = 20; // Variable locale à la fonction
 console.log(x); // 10 (peut accéder à la variable globale)
 console.log(y); // 20

 if (true) {
 var z = 30; // Portée de la fonction, pas du bloc if
 console.log(z); // 30
 }
}

console.log(z); // 30 (accessible en dehors du bloc if)

}
exemple();
console.log(x); // 10
console.log(y); // ReferenceError: y is not defined
```

## HISSEAGE (HOISTING) AVEC VAR

---

Les déclarations var sont "hissées" au sommet de leur contexte d'exécution (fonction ou global), avec une initialisation à « undefined » :

```
console.log(a); // undefined (pas d'erreur)
```

```
var a = 5;
```

```
console.log(a); // 5
```

// Équivalent à :

```
var a; // Hissée au sommet
```

```
console.log(a); // undefined
```

```
a = 5;
```

```
console.log(a); // 5
```

## PROBLÈMES LIÉS À VAR

---

### 1. FUITE DE PORTÉE DANS LES BLOCS :

---

```
for (var i = 0; i < 3; i++) {
 // i est accessible dans toute la fonction
}

console.log(i); // 3 (i est accessible en dehors de la boucle)
```

### 2. REDÉCLARATIONS ACCIDENTELLES :

---

```
var utilisateur = "Alice";
// Plus tard dans le code, par inadvertance

var utilisateur = "Bob"; // Écrase la valeur précédente sans erreur
```

### 3. PROBLÈMES AVEC LES CLOSURES DANS LES BOUCLES :

---

```
function createFunctions() {
 var functions = [];

 for (var i = 0; i < 3; i++) {
 functions.push(function() {
 console.log(i);
 });
 }

 return functions;
}

var fs = createFunctions();
fs[0](); // 3 (pas 0 comme on pourrait s'attendre)
fs[1](); // 3
fs[2](); // 3
```

#### LA DÉCLARATION AVEC LET

#### SYNTAXE ET UTILISATIONS BASIQUES

---

```
let age = 25; // Déclaration avec initialisation
let nom = "Alice"; // Variable contenant une chaîne
let estActif; // Déclaration sans initialisation (valeur undefined)

// Redéclaration interdite dans le même bloc
```

```
// let age = 26; // SyntaxError

// Réassiguation autorisée
age = 27; // Valeur modifiée
```

## PORTEE DE BLOC AVEC LET

Les variables déclarées avec let ont une portée de bloc (entre accolades {}), ce qui les rend plus prévisibles et sécurisées :

```
let x = 10; // Variable globale ou au niveau du module
```

```
function exemple() {
 let y = 20; // Variable locale à la fonction
 console.log(x); // 10
 console.log(y); // 20
```

```
 if (true) {
 let z = 30; // Variable locale au bloc if
 console.log(z); // 30
 }
```

```
// console.log(z); // ReferenceError: z is not defined
}
```

```
exemple();
console.log(x); // 10
// console.log(y); // ReferenceError: y is not defined
```

## ZONE MORTE TEMPORELLE (TEMPORAL DEAD ZONE)

---

Bien que les déclarations let soient également hissées, elles ne sont pas initialisées, créant une "zone morte temporelle" :

```
// console.log(a); // ReferenceError: Cannot access 'a' before initialization
let a = 5;
console.log(a); // 5
```

## AVANTAGES DE LET PAR RAPPORT À VAR

---

### 1. PORTÉE DE BLOC :

```
for (let i = 0; i < 3; i++) {
 // i est limité à ce bloc
}
// console.log(i); // ReferenceError: i is not defined
```

### 2. MEILLEURE GESTION DES ERREURS :

```
let utilisateur = "Alice";
// let utilisateur = "Bob"; // SyntaxError: Identifier 'utilisateur' has already been declared
```

### 3. CLOSURES CORRECTES DANS LES BOUCLES :

```
function createFunctions() {
 let functions = [];
 for (let i = 0; i < 3; i++) {
```

```

functions.push(function() {
 console.log(i);
});

}

return functions;
}

```

```

const fs = createFunctions();
fs[0](); // 0 (comme attendu)
fs[1](); // 1
fs[2](); // 2

```

## LA DÉCLARATION AVEC « CONST »

---

```

const PI = 3.14159; // Constante avec valeur primitive
const JOURS = ["Lun", "Mar"]; // Constante avec valeur de référence
// const ERREUR; // SyntaxError: Missing initializer in const declaration

```

## IMMUTABILITÉ DE L'ASSIGNATION

---

Les variables déclarées avec const ne peuvent pas être réassignées, mais les propriétés des objets peuvent être modifiées :

```

const PI = 3.14159;
// PI = 3.14; // TypeError: Assignment to constant variable

```

```
// Mais pour les objets et tableaux, le contenu peut être modifié
const utilisateur = { nom: "Alice" };

utilisateur.nom = "Bob"; // Autorisé

// utilisateur = { nom: "Charlie" }; // TypeError: Assignment to constant variable
```

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

nombres.push(4); // Autorisé

// nombres = [5, 6, 7]; // TypeError: Assignment to constant variable
```

## PORTEE DE BLOC

---

Comme let, les variables const ont une portée de bloc :

```
if (true) {
 const CONSTANTE_LOCALE = "Valeur locale";
 console.log(CONSTANTE_LOCALE); // "Valeur locale"
}
// console.log(CONSTANTE_LOCALE); // ReferenceError: CONSTANTE_LOCALE is not defined
```

## ZONE MORTE TEMPORELLE

---

Les constantes sont également soumises à la zone morte temporelle :

```
// console.log(MAX_USERS); // ReferenceError: Cannot access 'MAX_USERS' before initialization
const MAX_USERS = 100;
console.log(MAX_USERS); // 100
```

## QUAND UTILISER CONST

---

## 1. POUR LES VALEURS QUI NE DEVRAIENT JAMAIS CHANGER:

---

```
const PI = 3.14159;
const URL_API = "https://api.exemple.com";
const COULEURS = {
 PRIMAIRE: "#3498db",
 SECONDAIRE: "#2ecc71"
};
```

## 2. POUR LES RÉFÉRENCES D'OBJETS ET TABLEAUX STABLES :

---

```
const configuration = {
 theme: "sombre",
 notifications: true
};

// Les propriétés peuvent être modifiées
configuration.theme = "clair";
```

## 3. Pour les importations de modules :

```
const express = require('express');
import { Component } from 'react';
```

## CHOISIR ENTRE VAR, LET ET CONST

---

1. Privilégiez `const` par défaut pour toutes les variables dont la valeur ne change pas après initialisation.

2. Utilisez `let` lorsque vous avez besoin de réaffecter la variable (compteurs, accumulateurs, etc.).

3. Évitez `var` dans le code moderne, sauf pour la compatibilité avec d'anciens navigateurs sans transpilation.

## DESTRUCTURATION ET DÉCLARATION

Les déclarations `let` et `const` fonctionnent bien avec la destructuration :

```
// Destructuration avec const
```

```
const { nom, age } = personne;
```

```
// Destructuration avec let
```

```
let { x, y } = position;
```

```
// Plus tard, réassiguation possible
```

```
({ x, y } = nouvellePosition);
```

```
// Destructuration dans les boucles
```

```
for (const [cle, valeur] of Object.entries(objet)) {
```

```
 console.log(` ${cle}: ${valeur}`);
```

```
}
```

## BONNES PRATIQUES DE DÉCLARATION

1. Déclarez les variables au niveau de portée le plus restreint possible

2. Initialisez les variables lors de leur déclaration quand c'est possible

3. Utilisez des noms de variables significatifs

4. Groupez les déclarations similaires

```
// Bon

const NOM = "Alice";
const AGE = 30;
const ADMIN = true;

let compteur = 0;
let resultat = null;
```

5. Utilisez des conventions de nommage cohérentes

```
// Pour les constantes avec valeurs vraiment fixes

const MAX_USERS = 100;

// Pour les "constantes" qui sont des objets

const config = {
 apiUrl: 'https://example.com'
};
```

Le choix entre var, let et const a un impact significatif sur la clarté, la maintenabilité et la robustesse de votre code JavaScript. Les déclarations modernes (let et const) offrent une gestion de portée plus prévisible et contribuent à prévenir de nombreuses erreurs communes.

En adoptant la règle simple "utiliser const par défaut, let au besoin, et éviter var", vous pouvez écrire un code JavaScript plus sûr et plus lisible. Comprendre les nuances de chaque type de déclaration vous permettra de faire des choix éclairés en fonction des besoins spécifiques de votre code.

Dans la prochaine section, nous explorerons les différents types de données primitifs en JavaScript, comme les nombres, les chaînes de caractères et les booléens, et comment ils sont manipulés par le langage.

### 2.2.2. TYPES PRIMITIFS (NUMBER, STRING, BOOLEAN, ETC.)

JavaScript est un langage à typage dynamique, ce qui signifie que les variables ne sont pas associées à un type spécifique et peuvent contenir différentes valeurs de différents types au cours du temps. Malgré cette flexibilité, comprendre les types de données est essentiel pour écrire du code JavaScript efficace et éviter les erreurs courantes.

Dans cette section, nous explorerons les types de données primitifs en JavaScript, leurs caractéristiques, et comment les manipuler correctement.

#### VUE D'ENSEMBLE DES TYPES EN JAVASCRIPT

JavaScript possède huit types de données fondamentaux, dont sept sont considérés comme primitifs :

Type	Description	Exemple
<b>Number</b>	Nombres entiers et à virgule flottante	42, 3.14
<b>String</b>	Séquences de caractères	"Hello", 'JavaScript'
<b>Boolean</b>	Valeurs logiques	true, false
<b>Undefined</b>	Variable déclarée mais sans valeur assignée	undefined
<b>Null</b>	Valeur intentionnellement vide	null
<b>Symbol (ES6)</b>	Identifiant unique et immuable	Symbol('description')
<b>BigInt (ES2020)</b>	Nombres entiers de précision arbitraire	9007199254740991n
<b>Object</b>	Collection de propriétés (non primitif)	{}, [], function(){}{}

#### LE TYPE NUMBER

En JavaScript, tous les nombres sont représentés sous forme de valeurs en virgule flottante double précision (format IEEE 754), ce qui signifie qu'il n'y a pas de distinction entre les entiers et les nombres décimaux.

## REPRÉSENTATION DES NOMBRES

```
// Entiers

const entier = 42;

const negatif = -17;

// Décimaux

const decimal = 3.14159;

const negatifDecimal = -0.5;

// Notation scientifique

const grand = 1e6; // 1 million (1 * 10^6)

const petit = 1e-6; // 0.000001 (1 * 10^-6)

// Notation hexadécimale

const hex = 0xFF; // 255

const octal = 0o77; // 63

const binaire = 0b1010; // 10
```

## LIMITES ET VALEURS SPÉCIALES

```
// Valeurs spéciales

const infini = Infinity;

const negatifInfini = -Infinity;

const pasUnNombre = NaN; // Not a Number
```

```
// Limites

const nombreMax = Number.MAX_VALUE; // Le plus grand nombre représentable

const nombreMin = Number.MIN_VALUE; // Le plus petit nombre positif représentable

const entierSur = Number.MAX_SAFE_INTEGER; // 9007199254740991 (2^53 - 1)

const entierSousMin = Number.MIN_SAFE_INTEGER; // -9007199254740991 (-2^53 + 1)
```

## OPÉRATIONS NUMÉRIQUES

// Opérations arithmétiques de base

```
const somme = 5 + 3; // 8
const difference = 5 - 3; // 2
const produit = 5 * 3; // 15
const quotient = 5 / 3; // 1.6666666666666667
const reste = 5 % 3; // 2 (modulo)
const puissance = 5 ** 3; // 125 (exponentiation)
```

// Incrémentation et décrémentation

```
let compteur = 1;
compteur++; // 2
compteur--; // 1
```

// +=, -=, \*=, /=, %=, \*\*=

```
let x = 5;
x += 3; // équivalent à x = x + 3
```

## PRÉCISION ET PROBLÈMES DE VIRGULE FLOTTANTE

En raison de la représentation binaire des nombres en virgule flottante, JavaScript peut produire des résultats surprenants :

```
console.log(0.1 + 0.2); // 0.30000000000000004, pas exactement 0.3
console.log(0.1 + 0.2 === 0.3); // false
```

## L'OBJET MATH

JavaScript fournit l'objet Math pour des opérations mathématiques avancées :

```
// Constantes
console.log(Math.PI); // 3.141592653589793
console.log(Math.E); // 2.718281828459045
```

```
// Arrondi
console.log(Math.round(3.7)); // 4
console.log(Math.floor(3.7)); // 3
console.log(Math.ceil(3.2)); // 4
console.log(Math.trunc(3.7)); // 3 (partie entière)
```

```
// Min, Max, Absolu
console.log(Math.min(5, 2, 8)); // 2
console.log(Math.max(5, 2, 8)); // 8
console.log(Math.abs(-5)); // 5
```

```
// Racines et puissances
console.log(Math.sqrt(16)); // 4
console.log(Math.pow(2, 3)); // 8 (équivalent à 2 ** 3)
```

```
// Trigonométrie

console.log(Math.sin(Math.PI / 2)); // 1

console.log(Math.cos(0)); // 1

console.log(Math.tan(Math.PI / 4)); // 0.9999999999999999 (environ 1)
```

// Nombres aléatoires

```
console.log(Math.random()); // nombre entre 0 (inclus) et 1 (exclu)
```

// Nombre aléatoire entre min et max

```
function getRandomInt(min, max) {

 min = Math.ceil(min);

 max = Math.floor(max);

 return Math.floor(Math.random() * (max - min + 1)) + min;

}
```

### BIGINT (ES2020)

Introduit pour manipuler des entiers de taille arbitraire, au-delà des limites de `Number.MAX\_SAFE\_INTEGER` :

// Création de BigInt

```
const grandEntier = 9007199254740991n; // Notation littérale avec 'n'

const autreFacon = BigInt(9007199254740991);
```

// Opérations

```
console.log(grandEntier + 1n); // 9007199254740992n

console.log(grandEntier * 2n); // 18014398509481982n
```

// Limitations

```
// console.log(grandEntier + 1); // TypeError: ne peut pas mélanger BigInt et autres types

// console.log(Math.sqrt(grandEntier)); // TypeError: Math ne fonctionne pas avec BigInt
```

## LE TYPE STRING

Les chaînes de caractères (strings) représentent du texte. En JavaScript, elles sont immuables, ce qui signifie qu'une fois créées, leur contenu ne peut pas être modifié.

### Création de chaînes

// Guillemets simples

```
const simple = 'JavaScript';
```

// Guillemets doubles

```
const double = "est un langage";
```

// Backticks (littéraux de gabarits, ES6)

```
const template = `dynamique`;
```

// Chaînes multi-lignes

```
const multiLigne = `Ceci est
```

```
sur plusieurs
```

```
lignes`;
```

// Caractères d'échappement

```
const echappement = 'L\'apostrophe est échappée';
```

```
const tabulation = 'Un\tdeux\ttrois';
```

```
const nouvelle_ligne = 'Ligne 1\nLigne 2';
```

## LITTÉRAUX DE GABARITS (TEMPLATE LITERALS)

Introduits en ES6, ils offrent une syntaxe plus flexible :

```
```javascript
const nom = 'JavaScript';

const annee = 1995;

// Interpolation
const phrase = `${nom} a été créé en ${annee}.`;

// Expressions
const prix = 19.99;
const taxe = 0.2;
const total = `Total: ${prix * (1 + taxe)}.toFixed(2) €`;

// Fonctions de balises (tagged templates)
function surligner(strings, ...valeurs) {
    return strings.reduce((resultat, str, i) => {
        return `${resultat}${str}${valeurs[i] ? '<strong>' + valeurs[i] + '</strong>' : ''}`;
    }, "");
}

const message = surligner`Le langage ${nom} est très populaire.`;
// "Le langage <strong>JavaScript</strong> est très populaire."
```

PROPRIÉTÉS ET MÉTHODES DE CHAÎNE

```
const texte = "JavaScript est puissant";
```

```
// Propriété length
```

```

console.log(texte.length); // 23

// Accès aux caractères

console.log(texte[0]); // "J"

console.log(texte.charAt(0)); // "J"

// Recherche

console.log(texte.indexOf('Script')); // 4

console.log(texte.includes('puissant')); // true

console.log(texte.startsWith('Java')); // true

console.log(texte.endsWith('sant')); // true

// Extraction

console.log(texte.slice(4, 10)); // "Script"

console.log(texte.substring(4, 10)); // "Script"

console.log(texte.substr(4, 6)); // "Script" (déprécié)

// Modification (crée une nouvelle chaîne)

console.log(texte.replace('puissant', 'génial')); // "JavaScript est génial"

console.log(texte.toUpperCase()); // "JAVASCRIPT EST PUISSANT"

console.log(texte.toLowerCase()); // "javascript est puissant"

console.log(texte.trim()); // Supprime les espaces au début et à la fin

// Division

console.log(texte.split(' ')); // ["JavaScript", "est", "puissant"]

// Répétition (ES6)

console.log('*'.repeat(5)); // "*****"

```

```
// Remplissage (ES2017)

console.log('5'.padStart(3, '0')); // "005"
console.log('5'.padEnd(3, '0')); // "500"

// Extraction de caractères Unicode (ES6)

console.log('😊'.codePointAt(0)); // 128512
console.log(String.fromCodePoint(128512)); // "😊"
```

CHAÎNES DE CARACTÈRES ET UNICODE

JavaScript utilise UTF-16 pour représenter les chaînes, ce qui peut causer des problèmes avec les caractères en dehors du plan multilingue de base :

```
// Les emojis et certains caractères occupent 2 positions

const emoji = "😊";
console.log(emoji.length); // 2
console.log(emoji.codePointAt(0)); // 128512

// Solution moderne pour itérer sur les points de code Unicode

for (const char of "😊🌍🌈") {
    console.log(char); // Affiche chaque caractère correctement
}
```

LE TYPE BOOLEAN

Les booléens représentent des valeurs logiques : true (vrai) ou false (faux).

OPÉRATIONS BOOLÉENNES

```
// Opérateurs logiques
```

```
const et = true && false; // false
const ou = true || false; // true
const non = !true; // false
```

```
// Opérateurs de comparaison
```

```
const egal = 5 == 5; // true
const strictEgal = 5 === '5'; // false
const different = 5 != '5'; // false
const strictDifferent = 5 !== '5'; // true
const plus_grand = 5 > 3; // true
const plus_petit = 5 < 3; // false
const plus_grand_egal = 5 >= 5; // true
const plus_petit_egal = 5 <= 5; // true
```

CONVERSION IMPLICITE EN BOOLÉEN

Les valeurs sont automatiquement converties en booléens dans les contextes conditionnels :

```
// Valeurs falsy (évaluées à false)
console.log(Boolean(false)); // false
console.log(Boolean(0)); // false
console.log(Boolean("")); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
console.log(Boolean(NaN)); // false
```

```
// Toutes les autres valeurs sont truthy (évaluées à true)

console.log(Boolean(true)); // true

console.log(Boolean(1)); // true

console.log(Boolean('hello')); // true

console.log(Boolean([])); // true

console.log(Boolean({})); // true

console.log(Boolean(function(){})); // true
```

UNDEFINED ET NULL

Ces deux types représentent l'absence de valeur, mais avec des nuances importantes.

UNDEFINED

« undefined » représente une valeur qui n'a pas été assignée :

```
// Variables déclarées sans valeur

let variable;

console.log(variable); // undefined
```

// Paramètres de fonction manquants

```
function exemple(param) {

    console.log(param);

}

exemple(); // undefined
```

// Propriétés d'objet inexistantes

```
const obj = {};
```

```
console.log(obj.propriete); // undefined
```

// Fonctions sans return explicite

```
function sanRetour() {}
```

```
console.log(sanRetour()); // undefined
```

NULL

« null » représente l'absence intentionnelle de valeur :

```
// Utilisation typique de null
```

```
let utilisateur = null; // utilisateur actuellement non défini, mais attendu plus tard
```

// Vérification

```
console.log(utilisateur === null); // true
```

DIFFÉRENCES ENTRE NULL ET UNDEFINED

```
console.log(typeof undefined); // "undefined"
```

```
console.log(typeof null); // "object" (bug historique en JavaScript)
```

```
console.log(null == undefined); // true (équivalence faible)
```

```
console.log(null === undefined); // false (équivalence stricte)
```

// Comportement dans les opérations

```
console.log(null + 1); // 1 (null se comporte comme 0)
```

```
console.log(undefined + 1); // NaN
```

LE TYPE SYMBOL (ES6)

Les symboles, introduits en ES6, sont des identifiants uniques et immuables, principalement utilisés comme clés de propriétés d'objets.

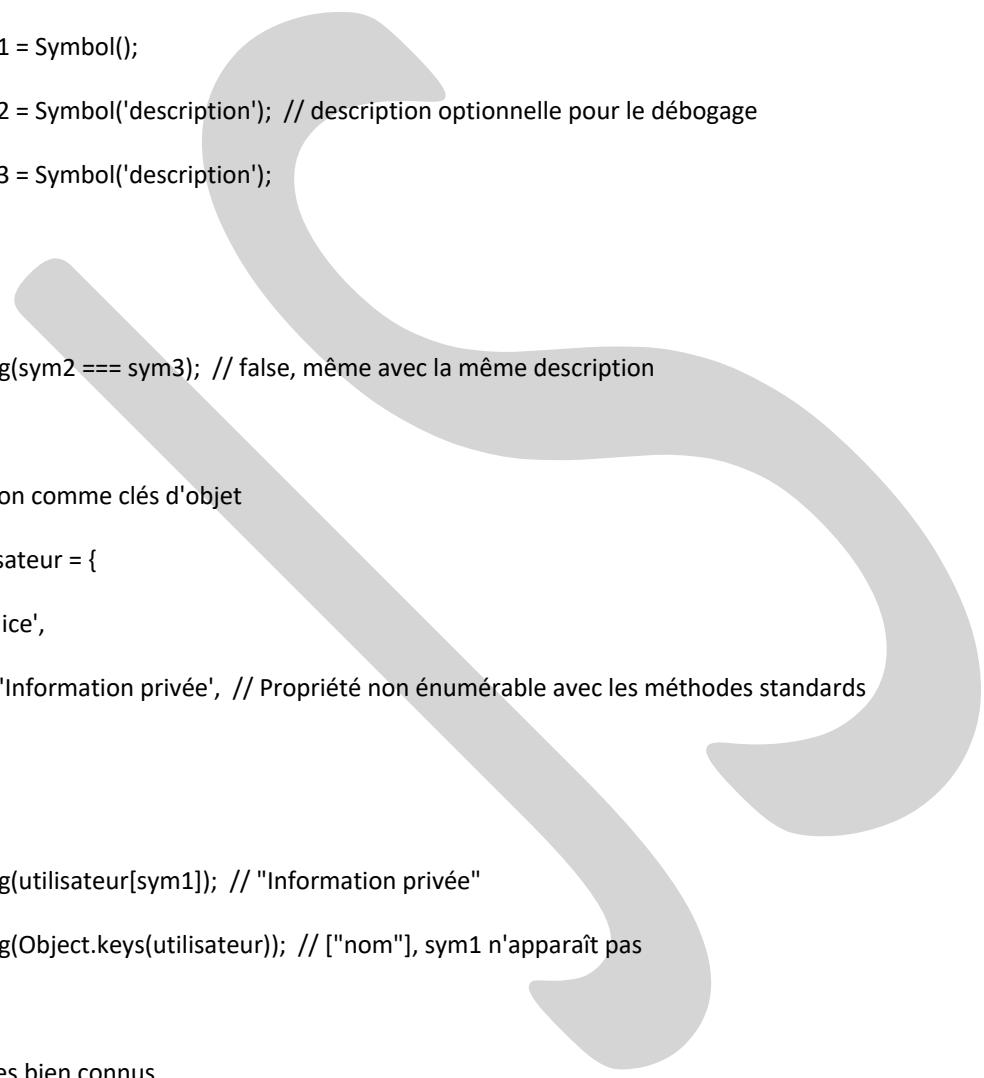
```
// Création de symboles
const sym1 = Symbol();
const sym2 = Symbol('description'); // description optionnelle pour le débogage
const sym3 = Symbol('description');

// Unicité
console.log(sym2 === sym3); // false, même avec la même description

// Utilisation comme clés d'objet
const utilisateur = {
  nom: 'Alice',
  [sym1]: 'Information privée', // Propriété non énumérable avec les méthodes standards
};

console.log(utilisateur[sym1]); // "Information privée"
console.log(Object.keys(utilisateur)); // ["nom"], sym1 n'apparaît pas

// Symboles bien connus
const iterableObj = {
  [Symbol.iterator]: function* () {
    yield 1;
    yield 2;
    yield 3;
  }
};
```



```

    }
};

for (const valeur of iterableObj) {
  console.log(valeur); // Affiche 1, 2, 3
}

```

```

// Registre global de symboles
const symGlobal = Symbol.for('identifiant');
const symGlobal2 = Symbol.for('identifiant');
console.log(symGlobal === symGlobal2); // true
console.log(Symbol.keyFor(symGlobal)); // "identifiant"

```

DÉTECTER ET CONVERTIR LES TYPES

L'OPÉRATEUR TYPEOF

```

console.log(typeof 42);      // "number"
console.log(typeof "texte"); // "string"
console.log(typeof true);   // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null);   // "object" (bug historique)
console.log(typeof {});     // "object"
console.log(typeof []);     // "object" (les tableaux sont des objets)

```

```
console.log(typeof function(){}); // "function"
console.log(typeof Symbol()); // "symbol"
console.log(typeof 42n); // "bigint"
```

CONVERSION EXPLICITE DE TYPES

// Vers String

```
console.log(String(42)); // "42"
console.log(String(true)); // "true"
console.log(String(null)); // "null"
console.log(String(undefined)); // "undefined"
console.log(String({})); // "[object Object]"
console.log(String([1, 2, 3])); // "1,2,3"
```

// Vers Number

```
console.log(Number("42")); // 42
console.log(Number("42.5")); // 42.5
console.log(Number ""); // 0
console.log(Number("texte")); // NaN
console.log(Number(true)); // 1
console.log(Number(false)); // 0
console.log(Number(null)); // 0
console.log(Number(undefined)); // NaN
```

// Vers Boolean

```
console.log(Boolean(42)); // true
console.log(Boolean(0)); // false
console.log(Boolean("texte")); // true
console.log(Boolean ""); // false
```

```
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
```

MÉTHODES DE CONVERSION ALTERNATIVES

// Conversion en nombre

```
console.log(parseInt("42px")); // 42 (s'arrête aux caractères non numériques)
console.log(parseFloat("42.5px")); // 42.5
console.log(+ "42.5"); // 42.5 (opérateur unaire +)
```

// Conversion en chaîne

```
console.log(42..toString()); // "42" (notez le double point)
console.log((42).toString()); // "42"
console.log(42 + ""); // "42" (concaténation implicite)
```

// Conversion en booléen

```
console.log(!!"texte"); // true (double négation)
```

VÉRIFICATION DE TYPE AVANCÉE

Pour une vérification plus précise, en particulier pour les objets :

// Vérifier si une valeur est un tableau

```
console.log(Array.isArray([])); // true
console.log(Array.isArray({})); // false
```

// Vérifier si une valeur est NaN

```
console.log(isNaN(NaN)); // true
```

```
console.log(isNaN("texte"));      // true (conversion implicite)
console.log(Number.isNaN("texte")); // false (pas de conversion implicite)
```

// Vérifier si une valeur est finie

```
console.log(isFinite(42));      // true
console.log(isFinite(Infinity)); // false
console.log(isFinite("42"));    // true (conversion implicite)
console.log(Number.isFinite("42")); // false (pas de conversion implicite)
```

// Vérifier si un nombre est un entier

```
console.log(Number.isInteger(42)); // true
console.log(Number.isInteger(42.5)); // false
```

// Vérifier si un nombre est dans les limites sûres

```
console.log(Number.isSafeInteger(9007199254740991)); // true
console.log(Number.isSafeInteger(9007199254740992)); // false
```

BONNES PRATIQUES POUR TRAVAILLER AVEC LES TYPES PRIMITIFS

1. Utilisez l'égalité stricte (`==` et `!=`) au lieu de l'égalité faible (`==` et `!=`) pour éviter les conversions implicites surprenantes.
2. Soyez explicite dans vos conversions de type plutôt que de compter sur la conversion implicite.
3. Utilisez des fonctions d'assistance pour les comparaisons de nombres à virgule flottante.
4. Évitez les expressions mixtes avec des types différents si possible.
5. Utilisez des outils comme TypeScript pour une vérification de type au moment de la compilation.

6. Utilisez `Object.is()` pour des comparaisons spéciales :

```
console.log(Object.is(0, -0));      // false
console.log(Object.is(NaN, NaN));   // true
console.log(Object.is(null, undefined)); // false
```

7. Faites attention aux opérations qui peuvent produire `Nan` et vérifiez les résultats quand nécessaire.

Comprendre les types primitifs en JavaScript est essentiel pour écrire du code fiable. Bien que JavaScript soit à typage dynamique, une bonne compréhension de la façon dont les valeurs sont traitées vous permettra d'éviter de nombreuses erreurs subtiles.

Les types primitifs forment la base sur laquelle sont construites des structures de données plus complexes comme les objets et les tableaux, que nous explorerons dans les prochaines sections. En maîtrisant ces types fondamentaux, vous serez mieux préparé pour manipuler efficacement les données dans vos applications JavaScript.

Dans la section suivante, nous verrons comment convertir entre ces différents types de données pour adapter les valeurs à vos besoins spécifiques

2.2.4. OPÉRATEURS

Les opérateurs sont des symboles qui effectuent des opérations sur des variables et des valeurs. JavaScript dispose d'un large éventail d'opérateurs qui permettent d'effectuer des calculs arithmétiques, des comparaisons, des opérations logiques, et bien plus encore.

Dans cette section, nous examinerons en détail les différents types d'opérateurs en JavaScript, leur fonctionnement, et les bonnes pratiques pour les utiliser efficacement.

LES OPÉRATEURS ARITHMÉTIQUES

Les opérateurs arithmétiques effectuent des opérations mathématiques sur des valeurs numériques.

Opérateur	Description	Exemple	Résultat
+	Addition	5 + 3	8
-	Soustraction	5-3	2
*	Multiplication	5*3	15
/	Division	15/3	5
%	Modulo (reste de division)	5 % 2	1
**	Exponentiation (ES2016)	5**2	25

Exemples d'utilisation

```
let a = 10;
```

```
let b = 3;
```

// Opérations de base

```
console.log(a + b); // 13
```

```
console.log(a - b); // 7
```

```
console.log(a * b); // 30
```

```
console.log(a / b); // 3.333333333333335
```

```
console.log(a % b); // 1 (reste de 10 divisé par 3)
```

```
console.log(a ** b); // 1000 (10 à la puissance 3)
```

// Cas particuliers

```
console.log(5 / 0); // Infinity
```

```
console.log(-5 / 0); // -Infinity
```

```
console.log(0 / 0); // NaN (Not a Number)
```

OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

Ces opérateurs augmentent ou diminuent une variable de 1.

Opérateur	Description	Exemple
++	Incrémantation	a++ ou ++a
--	Décrémentation	a--ou --a

Il existe deux formes pour chaque opérateur :

- Forme préfixe (++a) : incrémente puis retourne la valeur
- Forme postfixe (a++) : retourne la valeur puis incrémente

```
let x = 5;
let y;
// Préfixe
y = ++x; // x est incrémenté à 6, puis y prend la valeur 6
console.log(x, y); // 6, 6
// Postfixe
x = 5;
y = x++; // y prend la valeur 5, puis x est incrémenté à 6
console.log(x, y); // 6, 5
```

OPÉRATEURS D'AFFECTATION COMBINÉS

Ces opérateurs combinent une opération arithmétique avec une affectation.

Opérateur	Équivalent	Exemple
+=	$a = a + b$	a += b

<code>-=</code>	<code>a = a - b</code>	<code>a -= b</code>
<code>*=</code>	<code>a = a * b</code>	<code>a *= b</code>
<code>/=</code>	<code>a = a / b</code>	<code>a /= b</code>
<code>%=</code>	<code>a = a % b</code>	<code>a %= b</code>
<code>**</code>	<code>a = a ** b</code>	<code>a **= b</code>

```
let n = 10;
```

```
n += 5; // n = 15
n -= 3; // n = 12
n *= 2; // n = 24
n /= 6; // n = 4
n %= 3; // n = 1
n **= 3; // n = 1 (1 à la puissance 3)
```

LES OPÉRATEURS DE COMPARAISON

Les opérateurs de comparaison comparent deux valeurs et renvoient un booléen ('true' ou 'false').

Opérateur	Description	Exemple	Résultat
<code>==</code>	Égalité (avec coercition de type)	<code>5 == "5"</code>	true
<code>===</code>	Égalité stricte (sans coercition)	<code>5 === "5"</code>	false
<code>!=</code>	Inégalité (avec coercition)	<code>5 != "6"</code>	true
<code>!==</code>	Inégalité stricte (sans coercition)	<code>5 !== 5</code>	false
<code>></code>	Supérieur à	<code>5 > 3</code>	true
<code><</code>	Inférieur à	<code>5 < 3</code>	false

<code>>=</code>	Supérieur ou égal à	<code>5 >= 5</code>	true
<code><=</code>	Inférieur ou égal à	<code>5 <= 4</code>	false

L'ÉGALITÉ FAIBLE VS. L'ÉGALITÉ STRICTE

L'opérateur d'égalité faible (`==`) effectue une conversion de type avant la comparaison, tandis que l'égalité stricte (`===`) compare à la fois la valeur et le type.

```
// Égalité faible (==)

console.log(5 == 5);    // true

console.log(5 == "5");  // true (conversion de type)

console.log(0 == "");   // true (conversion de type)

console.log(0 == false); // true (conversion de type)

console.log(null == undefined); // true (cas spécial)
```

```
// Égalité stricte (===)

console.log(5 === 5);    // true

console.log(5 === "5");  // false (types différents)

console.log(0 === "");   // false (types différents)

console.log(0 === false); // false (types différents)

console.log(null === undefined); // false (types différents)
```

COMPARAISON DES OBJETS

Les comparaisons d'objets vérifient si les références pointent vers le même objet en mémoire, pas si leurs contenus sont identiques.

```
const obj1 = { a: 1 };

const obj2 = { a: 1 };
```

```

const obj3 = obj1;

console.log(obj1 == obj2); // false (références différentes)
console.log(obj1 === obj2); // false (références différentes)
console.log(obj1 == obj3); // true (même référence)
console.log(obj1 === obj3); // true (même référence)

...

```

LES OPÉRATEURS LOGIQUES

Les opérateurs logiques sont utilisés pour combiner ou inverser des valeurs booléennes.

Opérateur	Description	Exemple
&&	ET logique	a && b
 	OU logique	a b
!	NON logique	!a

ÉVALUATION COURT-CIRCUIT

Les opérateurs **&&** et **||** utilisent l'évaluation court-circuit : si le résultat peut être déterminé par la première opérande, la seconde n'est pas évaluée.

```

// ET logique (&&)

console.log(true && true); // true
console.log(true && false); // false
console.log(false && true); // false (court-circuit, la 2ème opérande n'est pas évaluée)
console.log(false && false); // false (court-circuit)

```

```
// OU logique (||)
```

```

console.log(true || true); // true (court-circuit)

console.log(true || false); // true (court-circuit)

console.log(false || true); // true

console.log(false || false); // false

// NON logique (!)

console.log(!true); // false

console.log(!false); // true

console.log (!!0); // false (double négation, convertit en booléen)

```

UTILISATION AVEC DES VALEURS NON BOOLÉENNES

Les opérateurs logiques fonctionnent avec n'importe quelle valeur, qui est d'abord convertie en booléen.

Ce qui est important de noter, c'est que :

- && retourne la première valeur falsy rencontrée, ou la dernière valeur si toutes sont truthy
- || retourne la première valeur truthy rencontrée, ou la dernière valeur si toutes sont falsy

// && retourne la première valeur falsy, ou la dernière si toutes sont truthy

```

console.log("hello" && 42); // 42

console.log(0 && "hello"); // 0 (court-circuit)

console.log(null && "anything"); // null (court-circuit)

```

// || retourne la première valeur truthy, ou la dernière si toutes sont falsy

```

console.log("hello" || 42); // "hello" (court-circuit)

console.log(0 || "hello"); // "hello"

console.log(null || undefined); // undefined (toutes falsy)

```

// Applications pratiques

```
const name = inputName || "Invité"; // Valeur par défaut
const canEdit = isAdmin && !isLocked; // Autorisation conditionnelle
```

NULLISH COALESCING OPERATOR (??) (ES2020)

L'opérateur de coalescence des nuls (??) est similaire à ||, mais il ne se déclenche que pour null ou undefined, pas pour d'autres valeurs falsy comme 0 ou "".

```
console.log(null ?? "défaut");    // "défaut"
console.log(undefined ?? "défaut"); // "défaut"
console.log(0 ?? "défaut");        // 0 (0 n'est pas null ou undefined)
console.log("") ?? "défaut");     // "" (chaîne vide n'est pas null ou undefined)

// Comparaison avec ||
console.log(0 || "défaut");       // "défaut" (0 est falsy)
console.log("") || "défaut");     // "défaut" (chaîne vide est falsy)
```

OPÉRATEUR CONDITIONNEL (TERNAIRE)

L'opérateur conditionnel (? :) est le seul opérateur JavaScript qui prend trois opérandes. Il agit comme un raccourci pour l'instruction `if-else`.

condition ? expression_si_vraie : expression_si_fausse

Exemples d'utilisation

```
// Syntaxe de base
const age = 20;
const statut = age >= 18 ? "adulte" : "mineur";
```

```

console.log(statut); // "adulte"

// Utilisation avec affectation
let message;
age < 16 ? message = "Trop jeune" : message = "Bienvenue";

// Chaînage de ternaires (à utiliser avec modération)
const categorie = age < 13 ? "enfant" : age < 18 ? "adolescent" : "adulte";

// Dans des expressions
const prix = isMember ? 20 : 50;
const total = `Total: ${prix + (isMember ? 0 : 10)} €`;

// Avec valeurs par défaut
const username = input ? input : "Anonyme";
// Équivalent à: const username = input || "Anonyme";

```

BONNES PRATIQUES

```

// Ne pas abuser du chaînage de ternaires - préférer if-else pour la lisibilité
// Difficile à lire:
const message = age < 13 ? "Enfant" : age < 18 ? "Ado" : age < 65 ? "Adulte" : "Senior";

// Plus lisible:
let message;
if (age < 13) {
    message = "Enfant";
} else if (age < 18) {
    message = "Ado";
}

```

```

} else if (age < 65) {

    message = "Adulte";

} else {

    message = "Senior";

}

// Utilisez des parenthèses pour clarifier l'intention lors de l'imbrication

const prix = isPremium ? (hasDiscount ? 40 : 50) : (hasDiscount ? 80 : 100);

```

OPÉRATEURS D'ENCHAÎNEMENT OPTIONNEL (OPTIONAL CHAINING) (ES2020)

L'opérateur d'enchaînement optionnel (?.) permet d'accéder en toute sécurité aux propriétés imbriquées d'un objet, sans risquer une erreur si une propriété intermédiaire est null ou undefined.

```

// Sans enchaînement optionnel (risque d'erreur)

function sansOperateur(user) {

    const admin = user.profile.admin; // Erreur si user ou profile est undefined

    return admin;

}

```

```

// Avec enchaînement optionnel (sécurisé)

function avecOperateur(user) {

    const admin = user?.profile?.admin; // undefined si user ou profile est undefined

    return admin;

}

```

```

// Utilisation avec fonctions et tableaux

const result = obj.method?(); // undefined si method n'existe pas

```

```
const value = arr?.[0]; // undefined si arr est null ou undefined
```

OPÉRATEUR DE DÉCOMPOSITION (SPREAD OPERATOR) (ES6)

L'opérateur de décomposition (...) permet de décomposer un itérable (comme un tableau ou une chaîne) en ses éléments individuels.

```
// Décomposition de tableaux
const array1 = [1, 2, 3];
const array2 = [...array1, 4, 5]; // [1, 2, 3, 4, 5]
```

```
// Copie de tableau
const copie = [...array1]; // Crée une copie superficielle
```

```
// Fusion de tableaux
const fusion = [...array1, ...array2];
```

```
// Avec des fonctions
function somme(a, b, c) {
    return a + b + c;
}
console.log(somme(...[1, 2, 3])); // 6
```

```
// Avec des objets (ES2018)
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }
const fusion_obj = { ...obj1, ...obj2 }; // Fusionne les objets
const override = { ...obj1, a: 42 }; // { a: 42, b: 2 }
```

OPÉRATEURS DE DÉSTRUCTURATION (ES6)

La déstructuration permet d'extraire des données de tableaux ou d'objets dans des variables distinctes.

DÉSTRUCTURATION DE TABLEAUX

// Syntaxe de base

```
const numbers = [1, 2, 3];
const [a, b, c] = numbers;
console.log(a, b, c); // 1 2 3
```

// Ignorer certains éléments

```
const [first, , third] = numbers;
console.log(first, third); // 1 3
```

// Avec valeur par défaut

```
const [x = 0, y = 0, z = 0, w = 0] = numbers;
console.log(x, y, z, w); // 1 2 3 0
```

// Rest pattern

```
const [head, ...tail] = numbers;
console.log(head, tail); // 1 [2, 3]
```

// Échange de variables

```
let p = 1, q = 2;
[p, q] = [q, p];
console.log(p, q); // 2 1
```

DÉSTRUCTURATION D'OBJETS

// Syntaxe de base

```
const user = { name: "Alice", age: 30, city: "Paris" };

const { name, age } = user;

console.log(name, age); // "Alice" 30
```

// Renommer des propriétés

```
const { name: userName, age: userAge } = user;

console.log(userName, userAge); // "Alice" 30
```

// Valeurs par défaut

```
const { name, country = "France" } = user;

console.log(name, country); // "Alice" "France"
```

// Combinaison renommage et valeur par défaut

```
const { city: residence = "Inconnu" } = user;

console.log(residence); // "Paris"
```

// Rest pattern

```
const { name, ...details } = user;

console.log(name, details); // "Alice" { age: 30, city: "Paris" }
```

// Déstructuration imbriquée

```
const metadata = {

  title: "Mon titre",

  info: {

    year: 2023,

    format: "digital"
  }
}
```

```

    }
};

const { title, info: { year, format } } = metadata;
console.log(title, year, format); // "Mon titre" 2023 "digital"

```

```

// Paramètres de fonction déstructurés

function printUserInfo({ name, age, country = "Inconnu" }) {
  console.log(` ${name}, ${age} ans, ${country}`);
}

printUserInfo(user); // "Alice, 30 ans, Inconnu"

```

Opérateurs de propriété (Property Operators)

L'OPÉRATEUR IN

L'opérateur « in » vérifie si une propriété spécifiée existe dans un objet ou dans sa chaîne de prototypes.

```

```javascript
const car = { make: "Toyota", model: "Corolla", year: 2020 };

console.log("make" in car); // true
console.log("price" in car); // false
console.log("toString" in car); // true (hérité de Object.prototype)

```

```

// Vérification de l'existence d'un index dans un tableau

const arr = ["a", "b", "c"];

console.log(0 in arr); // true (l'index 0 existe)
console.log(5 in arr); // false (l'index 5 n'existe pas)

```

## ACCÈS AUX PROPRIÉTÉS DYNAMIQUES

```
// Notation par points (statique)
```

```
console.log(car.make); // "Toyota"
```

```
// Notation par crochets (dynamique)
```

```
const prop = "make";
```

```
console.log(car[prop]); // "Toyota"
```

```
// Crédit dynamique de propriétés
```

```
const key = "color";
```

```
car[key] = "red";
```

```
console.log(car.color); // "red"
```

Opérateurs de type (Type Operators)

## L'OPÉRATEUR TYPEOF

L'opérateur typeof retourne une chaîne indiquant le type de l'opérande.

```
console.log(typeof 42); // "number"
```

```
console.log(typeof "hello"); // "string"
```

```
console.log(typeof true); // "boolean"
```

```
console.log(typeof undefined); // "undefined"
```

```
console.log(typeof null); // "object" (bug historique)
```

```
console.log(typeof {}); // "object"
```

```
console.log(typeof []); // "object"
```

```
console.log(typeof function(){}); // "function"
```

```
console.log(typeof Symbol()); // "symbol"
```

```
console.log(typeof 42n); // "bigint"
```

## L'OPÉRATEUR INSTANCEOF

L'opérateur instanceof vérifie si un objet est une instance d'une classe ou d'un constructeur spécifique.

```
const arr = [1, 2, 3];
const date = new Date();
const regex = /abc/;

console.log(arr instanceof Array); // true
console.log(arr instanceof Object); // true (héritage)
console.log(date instanceof Date); // true
console.log(date instanceof Object); // true (héritage)
console.log(regex instanceof RegExp); // true
console.log({} instanceof Object); // true

// Les primitives ne sont pas des instances
console.log("hello" instanceof String); // false (primitive, pas objet)
console.log(new String("hello") instanceof String); // true (objet String)
```

Les opérateurs en JavaScript sont des outils puissants qui permettent d'effectuer une grande variété d'opérations sur les données. Comprendre leur fonctionnement, leur précédence et leurs particularités est essentiel pour écrire du code JavaScript efficace et sans erreur.

En particulier, il est important de bien comprendre les comportements liés à la coercition de type, qui a été détaillée dans la section précédente, ainsi que les nouvelles possibilités offertes par les opérateurs modernes introduits dans les récentes versions de JavaScript (comme l'enchaînement optionnel, la coalescence des nuls et les opérateurs de décomposition).

Dans la section suivante, nous explorerons les structures de contrôle en JavaScript, qui vous permettront d'organiser la logique de votre code en utilisant ces opérateurs de manière efficace.

### 2.3.1. STRUCTURES DE CONTRÔLE CONDITIONNELLES

Les structures de contrôle conditionnelles permettent à votre programme de prendre des décisions en fonction de conditions spécifiques. Elles constituent un élément fondamental de la programmation, permettant d'exécuter différents blocs de code selon que certaines conditions sont remplies ou non.

Dans cette section, nous examinerons les différentes structures conditionnelles disponibles en JavaScript, leurs syntaxes, leurs cas d'utilisation, et les bonnes pratiques pour les utiliser efficacement.

#### L'INSTRUCTION IF

L'instruction « if » est la structure conditionnelle la plus fondamentale. Elle évalue une condition et exécute un bloc de code si cette condition est évaluée comme « true ».

#### SYNTAXE DE BASE

```
if (condition) {
 // Code exécuté si la condition est true
}
```

La condition est convertie en valeur booléenne si elle n'est pas déjà de ce type. Rappelez-vous que les valeurs suivantes sont évaluées comme `false` (falsy) :

- false
- 0
- "" (chaîne vide)
- null
- undefined
- NaN

Toutes les autres valeurs sont évaluées comme true (truthy).

Exemples d'utilisation

```
// Condition simple

if (age >= 18) {
 console.log("Vous êtes majeur");
}

// Condition avec opérateurs logiques

if (username && username.length > 3) {
 console.log("Nom d'utilisateur valide");
}

// Condition avec expression

if (totalPanier > 100) {
 appliquerRemise(15);
}
```

### L'INSTRUCTION IF...ELSE

La structure if...else permet d'exécuter un bloc de code alternatif lorsque la condition n'est pas remplie.

### SYNTAXE

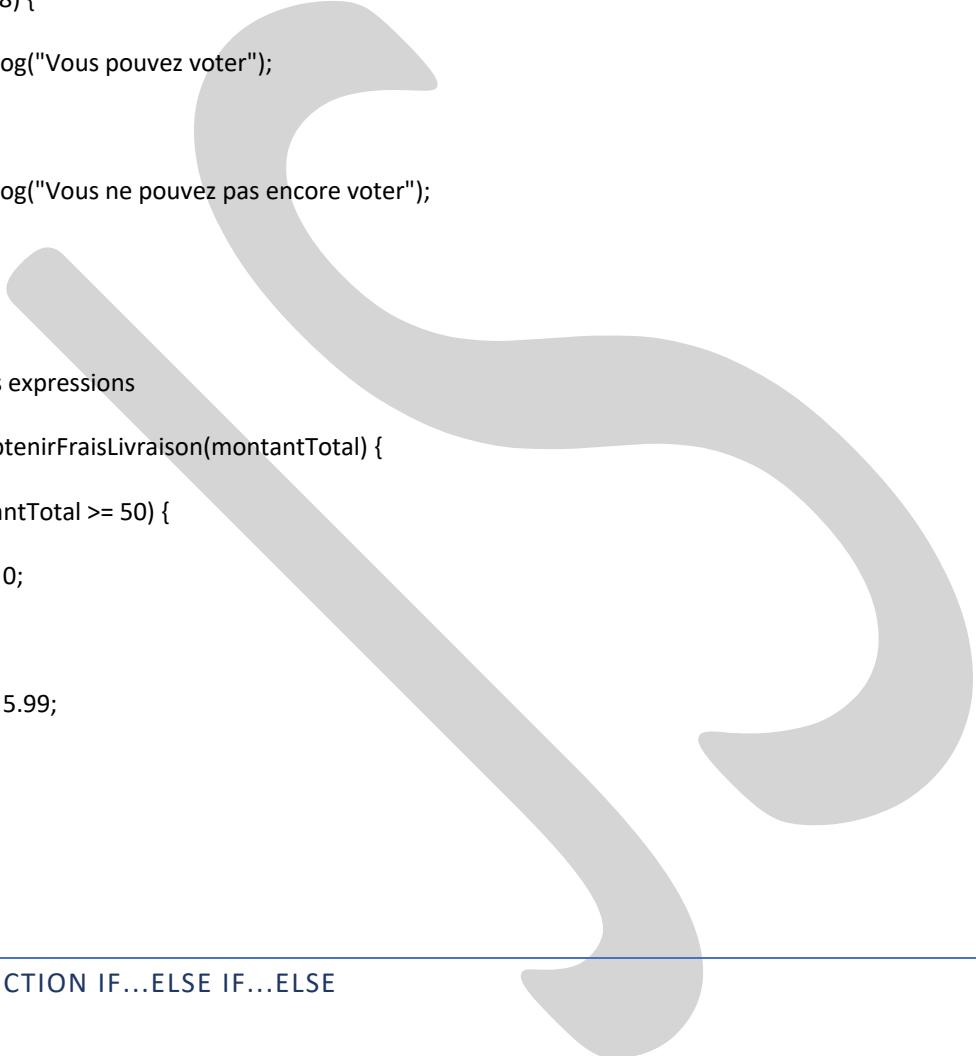
```
if (condition) {
 // Code exécuté si la condition est true
} else {
```

```
// Code exécuté si la condition est false
}
```

## EXEMPLES

```
const age = 15;

if (age >= 18) {
 console.log("Vous pouvez voter");
} else {
 console.log("Vous ne pouvez pas encore voter");
}
```



```
// Avec des expressions

function obtenirFraisLivraison(montantTotal) {
 if (montantTotal >= 50) {
 return 0;
 } else {
 return 5.99;
 }
}
```

## L'INSTRUCTION IF...ELSE IF...ELSE

Pour tester plusieurs conditions séquentiellement, on utilise la structure if...else if...else.

## SYNTAXE

```
if (condition1) {
```

```
// Code exécuté si condition1 est true
} else if (condition2) {

// Code exécuté si condition1 est false et condition2 est true

} else if (condition3) {

// Code exécuté si condition1 et condition2 sont false, et condition3 est true

} else {

// Code exécuté si toutes les conditions sont false
}
```

### EXEMPLES

```
const note = 85;

if (note >= 90) {
 console.log("Excellent (A)");
} else if (note >= 80) {
 console.log("Très bien (B)");
} else if (note >= 70) {
 console.log("Bien (C)");
} else if (note >= 60) {
 console.log("Passable (D)");
} else {
 console.log("Insuffisant (E)");
}
```

```
// Exemple avec des conditions plus complexes
```

```
const age = 25;
const estEtudiant = true;
```

```
if (age < 18) {
```

```
console.log("Tarif enfant");

} else if (age >= 65) {

 console.log("Tarif senior");

} else if (estEtudiant) {

 console.log("Tarif étudiant");

} else {

 console.log("Tarif normal");

}
```

### CONDITIONS IMBRIQUÉES

Il est possible d'imbriquer des instructions conditionnelles les unes dans les autres, mais cela peut rapidement rendre le code difficile à lire.

#### EXEMPLE

```
const age = 20;

const permis = true;

if (age >= 18) {

 if (permis) {

 console.log("Vous pouvez conduire");

 } else {

 console.log("Vous devez obtenir votre permis");

 }

} else {

 console.log("Vous êtes trop jeune pour conduire");

}
```

## MEILLEURES PRATIQUES

En général, il est préférable d'éviter les imbriques profondes en utilisant des conditions composées avec des opérateurs logiques :

```
// Version améliorée de l'exemple précédent

if (age >= 18 && permis) {

 console.log("Vous pouvez conduire");

} else if (age >= 18) {

 console.log("Vous devez obtenir votre permis");

} else {

 console.log("Vous êtes trop jeune pour conduire");

}
```

## L'INSTRUCTION SWITCH

L'instruction switch offre une alternative à multiples if...else if lorsqu'on veut comparer une valeur à plusieurs cas possibles.

### SYNTAXE

```
switch (expression) {

 case valeur1:

 // Code exécuté si expression === valeur1

 break;

 case valeur2:

 // Code exécuté si expression === valeur2

 break;

 // Autres cas...
```

default:

```
// Code exécuté si aucun cas ne correspond
}
```

Le mot-clé break est important : sans lui, l'exécution continue dans le cas suivant (comportement dit de "fall-through").

### EXEMPLES

```
const jour = new Date().getDay(); // 0 pour dimanche, 1 pour lundi, etc.
```

```
switch (jour) {
 case 0:
 console.log("Dimanche");
 break;

 case 1:
 console.log("Lundi");
 break;

 case 2:
 console.log("Mardi");
 break;

 case 3:
 console.log("Mercredi");
 break;

 case 4:
 console.log("Jeudi");
 break;

 case 5:
 console.log("Vendredi");
}
```

```
break;

case 6:
 console.log("Samedi");
 break;

default:
 console.log("Jour invalide");

}
```

### GROUPEMENT DE CAS

On peut regrouper plusieurs cas qui partagent le même traitement :

```
const jour = new Date().getDay();

switch (jour) {

 case 0:

 case 6:
 console.log("C'est le week-end!");
 break;

 case 1:

 case 2:

 case 3:

 case 4:

 case 5:
 console.log("C'est un jour de semaine");
 break;

 default:
 console.log("Jour invalide");

}
```

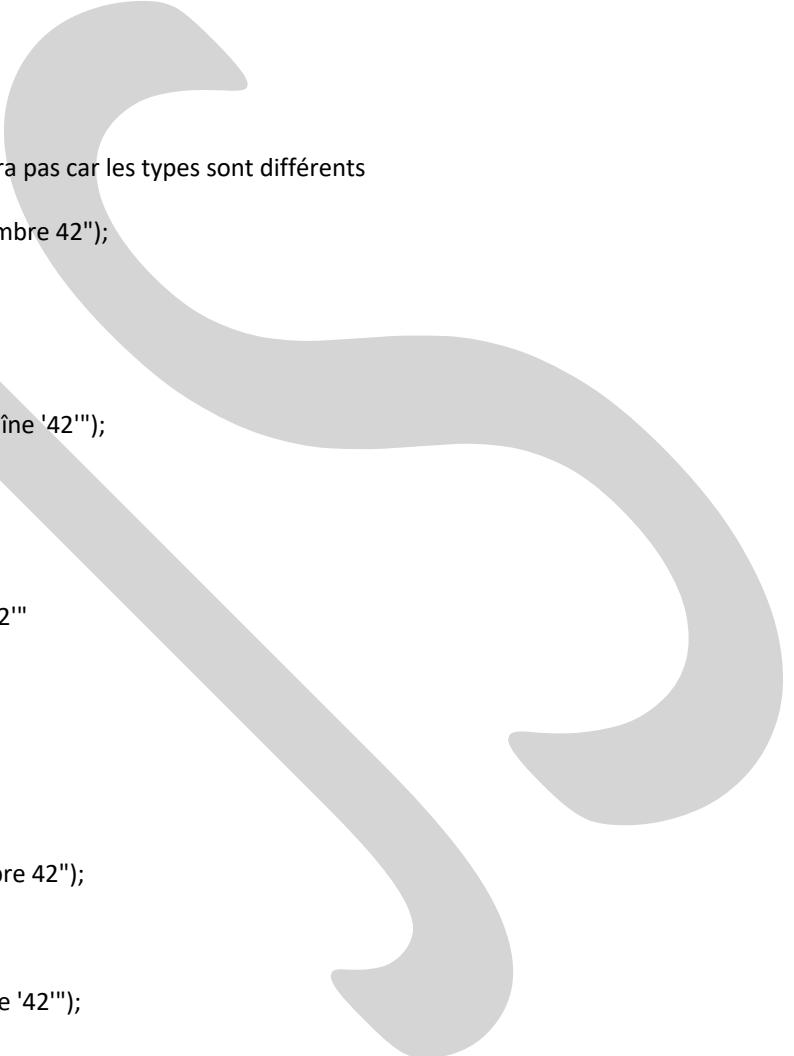
## COMPARAISON AVEC IF...ELSE

L'instruction switch utilise une comparaison d'égalité stricte (==). Cela signifie qu'elle ne fait pas de conversion de type, contrairement à l'égalité abstraite (==).

```
// Avec switch
const valeur = "42";
switch (valeur) {
 case 42: // Ne correspondra pas car les types sont différents
 console.log("C'est le nombre 42");
 break;
 case "42":
 console.log("C'est la chaîne '42'");
 break;
}
// Affiche: "C'est la chaîne '42'"
```

```
// Équivalent avec if...else
if (valeur === 42) {
 console.log("C'est le nombre 42");
} else if (valeur === "42") {
 console.log("C'est la chaîne '42'");
}
// Affiche également: "C'est la chaîne '42'"
```



## L'INSTRUCTION IF AVEC CONDITION PRÉALABLE

Une technique courante consiste à effectuer une vérification préalable au début d'une fonction pour retourner rapidement en cas de conditions non satisfaites. Cette approche est souvent appelée "early return" ou "guard clause".

### EXEMPLE

// Avec imbrication (moins lisible)

```
function processUser(user) {
 if (user) {
 if (user.isActive) {
 if (user.hasPermission) {
 // Traitement principal...
 return result;
 } else {
 return "Permission denied";
 }
 } else {
 return "User not active";
 }
 } else {
 return "No user provided";
 }
}
```

// Avec early returns (plus lisible)

```
function processUser(user) {
 if (!user) return "No user provided";
 if (!user.isActive) return "User not active";
 if (!user.hasPermission) return "Permission denied";
```

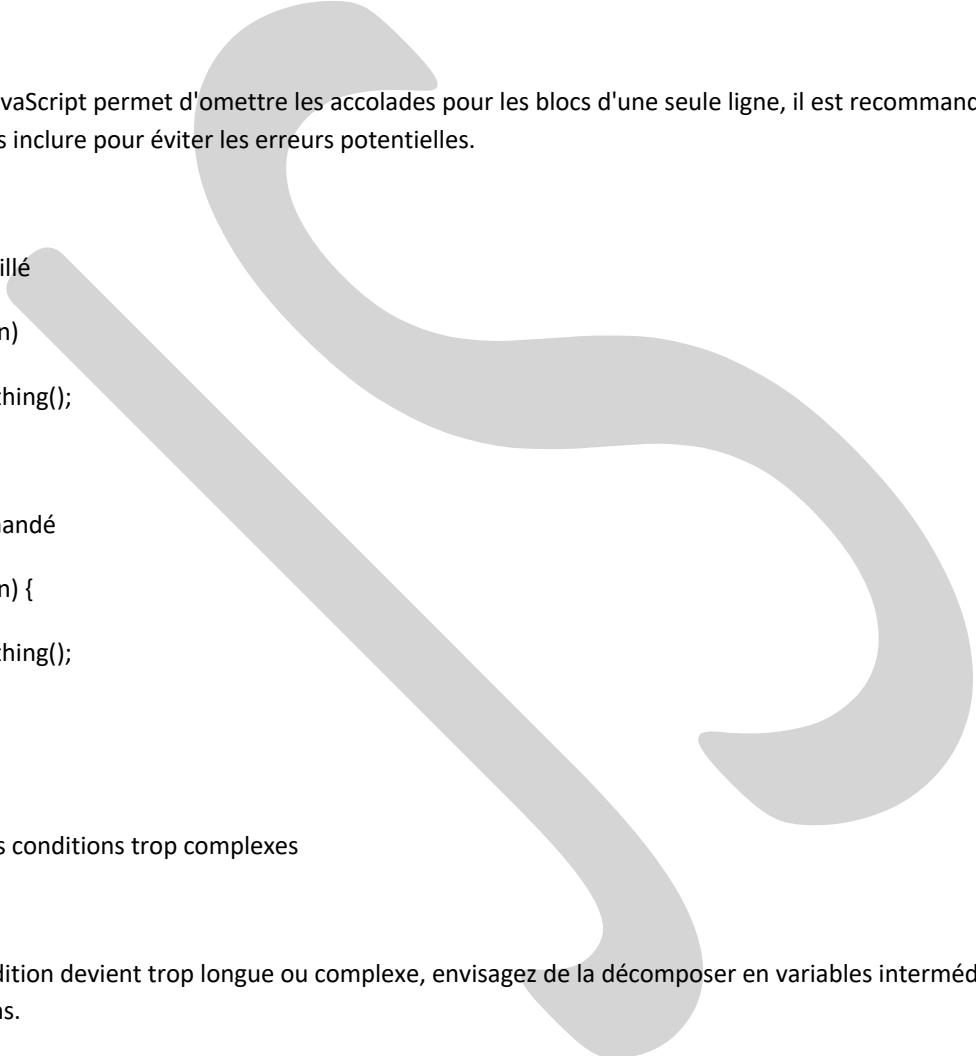
```
// Traitement principal...
return result;
}

...

```

### 1. Utilisez des accolades même pour les blocs d'une seule ligne

Même si JavaScript permet d'omettre les accolades pour les blocs d'une seule ligne, il est recommandé de toujours les inclure pour éviter les erreurs potentielles.



```
// Déconseillé
if (condition)
 doSomething();
```

```
// Recommandé
if (condition) {
 doSomething();
}
```

### 2. Évitez les conditions trop complexes

Si une condition devient trop longue ou complexe, envisagez de la décomposer en variables intermédiaires ou en fonctions.



```
// Difficile à lire
if (user.age >= 18 && user.subscription && (user.role === 'admin' || user.permissions.includes('edit'))) {
 // ...
}
```

```
// Plus lisible

const isAdult = user.age >= 18;

const hasActiveSubscription = Boolean(user.subscription);

const hasEditRights = user.role === 'admin' || user.permissions.includes('edit');

if (isAdult && hasActiveSubscription && hasEditRights) {

 // ...

}
```

### 3. Évitez les conditions imbriquées profondes

Comme nous l'avons vu, utilisez des "early returns", des conditions composées, ou restructurez votre code pour éviter les imbrications profondes.

### 4. Utilisez switch pour les comparaisons multiples d'égalité

L'instruction `switch` est plus lisible et efficace pour comparer une valeur à plusieurs cas possibles.

### 5. Utilisez l'opérateur ternaire avec modération

L'opérateur ternaire est excellent pour les conditions simples, mais peut devenir illisible s'il est imbriqué ou trop complexe.

### 6. Testez explicitement les valeurs nulles ou indéfinies

```
// Évitez de compter sur la coercition en boolean
```

```
if (variable) { ... }
```

```
// Préférez tester explicitement ce que vous recherchez
```

```
if (variable !== null && variable !== undefined) { ... }
```

```
// ou avec l'opérateur de coalescence des nuls
```

```
const value = variable ?? defaultValue;
```

## CAS D'UTILISATION COURANTS

## VALIDATION D'ENTRÉE UTILISATEUR

```
function validateLogin(username, password) {
 if (!username) {
 return { valid: false, error: "Nom d'utilisateur requis" };
 }
 if (username.length < 3) {
 return { valid: false, error: "Le nom d'utilisateur doit contenir au moins 3 caractères" };
 }
 if (!password) {
 return { valid: false, error: "Mot de passe requis" };
 }
 if (password.length < 8) {
 return { valid: false, error: "Le mot de passe doit contenir au moins 8 caractères" };
 }
 return { valid: true };
}
```

## CONTRÔLE D'ACCÈS BASÉ SUR LES RÔLES

```
function canAccessResource(user, resource) {
 // Vérification préalable
 if (!user || !resource) return false;

 // Admin a accès à tout
 if (user.role === 'admin') return true;
```

```
// Vérification des permissions spécifiques

switch (resource.type) {

 case 'document':

 return user.permissions.includes('read_documents') || resource.ownerId === user.id;

 case 'report':

 return user.permissions.includes('read_reports');

 case 'settings':

 return user.role === 'manager' || resource.ownerId === user.id;

 default:

 return false;

}

}
```

#### TRAITEMENT CONDITIONNEL DANS LES INTERFACES UTILISATEUR

```
function renderUserProfile(user) {

 let profileHTML = '<div class="profile">';

 // Info de base toujours affichée

 profileHTML += `<h2>${user.name} || 'Anonyme'</h2>`;

 // Affichage conditionnel de propriétés supplémentaires

 if (user.avatar) {

 profileHTML += ``;

 } else {

 profileHTML += `<div class="default-avatar">${user.name.charAt(0)}</div>`;

 }

 if (user.bio) {
```

```

profileHTML += `<p class="bio">${user.bio}</p>`;

}

// Options spécifiques au rôle

if (user.role === 'admin') {

 profileHTML += `Administrateur`;

} else if (user.role === 'moderator') {

 profileHTML += `Modérateur`;

}

// Actions disponibles selon l'état

if (!user.isActive) {

 profileHTML += `<div class="inactive-message">Compte inactif</div>`;

} else if (user.isPremium) {

 profileHTML += `<div class="premium-features">

 <button>Fonctionnalités premium</button>

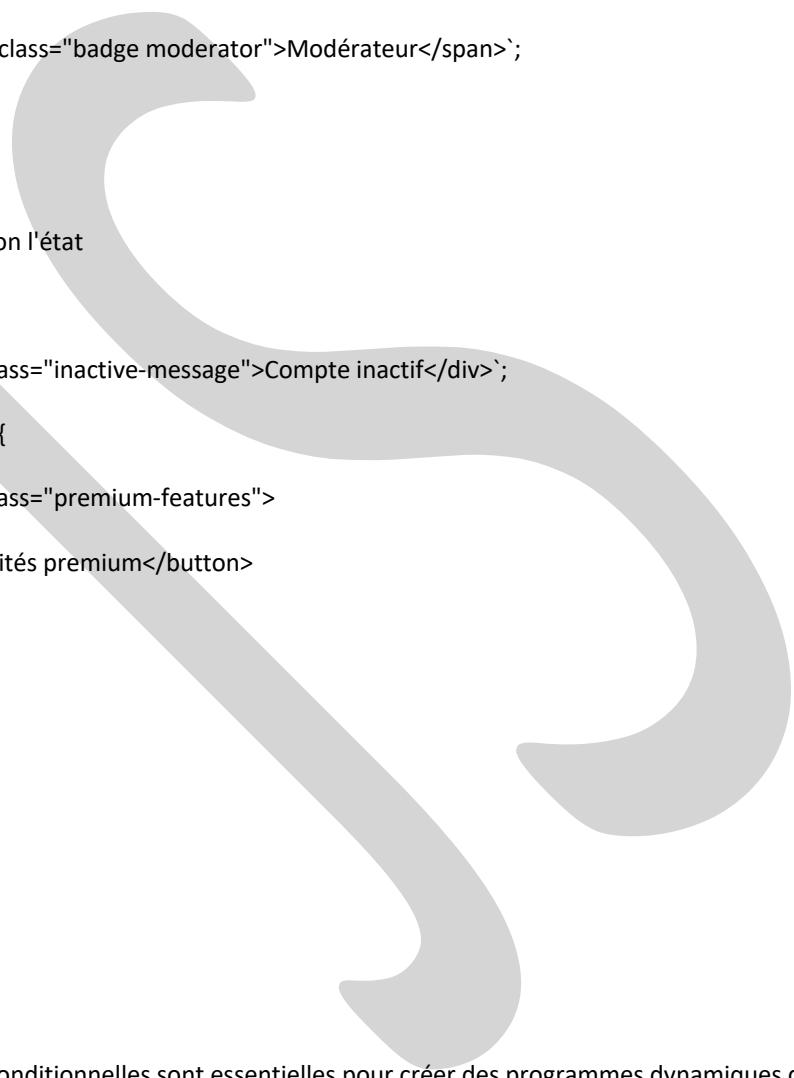
 </div>`;

}

profileHTML += `</div>`;

return profileHTML;
}

```



Les structures de contrôle conditionnelles sont essentielles pour créer des programmes dynamiques qui s'adaptent à différentes situations. JavaScript offre une variété d'options, de la structure `if` traditionnelle aux opérateurs modernes comme l'opérateur de coalescence des nuls, en passant par les expressions concises utilisant les opérateurs logiques.

Choisir la bonne structure conditionnelle pour chaque situation et suivre les bonnes pratiques vous permettra d'écrire un code plus lisible, plus maintenable et moins sujet aux erreurs. N'oubliez pas que la clarté du code est souvent plus importante que sa concision, surtout lorsqu'il s'agit de logique conditionnelle complexe.

Dans la section suivante, nous explorerons les structures de boucle, qui permettent d'exécuter du code de manière répétée en fonction de certaines conditions.

### 2.3.2. STRUCTURES DE CONTRÔLE DE BOUCLES

Les structures de boucle permettent d'exécuter un bloc de code de manière répétée tant qu'une condition spécifiée est satisfaite. Les boucles sont essentielles pour automatiser des tâches répétitives, parcourir des collections de données, et implémenter des algorithmes complexes.

Dans cette section, nous explorerons les différentes structures de boucle disponibles en JavaScript, leurs syntaxes, leurs cas d'utilisation, et les bonnes pratiques pour les utiliser efficacement.

#### LA BOUCLE FOR

La boucle for est la structure de boucle la plus couramment utilisée en JavaScript. Elle fournit un moyen concis d'initialiser une variable, de définir une condition d'exécution et de mettre à jour la variable.

#### SYNTAXE DE BASE

```
for (initialisation; condition; incrémentation) {
 // Code à exécuter à chaque itération
}
```

- Initialisation : Expression exécutée une seule fois avant la première itération
- Condition : Expression évaluée avant chaque itération, la boucle continue tant que cette condition est true
- Incrémentation : Expression exécutée à la fin de chaque itération

## EXEMPLES D'UTILISATION

```
// Boucle simple de 0 à 9

for (let i = 0; i < 10; i++) {
 console.log(i); // Affiche 0, 1, 2, ..., 9
}

// Parcourir un tableau

const fruits = ["pomme", "banane", "orange", "kiwi"];

for (let i = 0; i < fruits.length; i++) {
 console.log(fruits[i]); // Affiche chaque élément du tableau
}

// Compteur décroissant

for (let i = 10; i > 0; i--) {
 console.log(i); // Compte à rebours: 10, 9, 8, ..., 1
}

// Incrémentation personnalisée

for (let i = 0; i < 20; i += 2) {
 console.log(i); // Affiche les nombres pairs: 0, 2, 4, ..., 18
}
```

## PORTEE DES VARIABLES DANS LA BOUCLE FOR

Les variables déclarées dans l'initialisation de la boucle for avec let ou const sont limitées à la portée de la boucle.

```
// i n'est accessible que dans la boucle
for (let i = 0; i < 5; i++) {
 console.log(i);
}

// console.log(i); // Erreur: i n'est pas défini

// Avec var (déprécié), la variable est accessible en dehors de la boucle
for (var j = 0; j < 5; j++) {
 // Code...
}

console.log(j); // 5 (j est accessible)
```

### BOUCLES IMBRIQUÉES

On peut imbriquer des boucles for l'une dans l'autre, ce qui est utile pour traiter des structures de données multidimensionnelles.

```
// Crédit à et affichage d'une matrice 3x3
for (let i = 0; i < 3; i++) {
 let ligne = "";
 for (let j = 0; j < 3; j++) {
 ligne += `(${i},${j}) `;
 }
 console.log(ligne);
}

/* Résultat:
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
*/
```

```
// Parcourir un tableau bidimensionnel

const matrice = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
];

for (let i = 0; i < matrice.length; i++) {
 for (let j = 0; j < matrice[i].length; j++) {
 console.log(matrice[i][j]); // Affiche chaque élément
 }
}
```

#### PARTIES OPTIONNELLES DE LA BOUCLE FOR

Les trois parties de la boucle for sont optionnelles :

```
```javascript
// Initialisation à l'extérieur
let i = 0;
for (; i < 5; i++) {
    console.log(i);
}
```

```
// Condition dans le corps de la boucle avec break

for (let i = 0; ; i++) {
    if (i >= 5) break;
    console.log(i);
```

```
}
```

```
// Incrémentation dans le corps de la boucle
for (let i = 0; i < 5;) {
    console.log(i);
    i++;
}
```

```
// Boucle infinie (utiliser avec précaution)
// for (;;) {
//     // Code exécuté indéfiniment
//     if (condition) break; // Sortie conditionnelle nécessaire
// }
```

LA BOUCLE WHILE

La boucle while exécute un bloc de code tant qu'une condition spécifiée est true. C'est la structure de boucle la plus simple.

SYNTAXE DE BASE

```
while (condition) {
    // Code à exécuter tant que la condition est true
}
```

EXEMPLES D'UTILISATION

```
// Compteur simple
```

```

let compteur = 0;

while (compteur < 5) {
    console.log(compteur); // Affiche 0, 1, 2, 3, 4
    compteur++;
}

// Traitement conditionnel

let nombre = 1;

while (nombre < 100) {
    console.log(nombre);

    nombre *= 2; // Double le nombre à chaque itération: 1, 2, 4, 8, 16, 32, 64
}

// Attente d'une condition externe

const attendreConsentement = () => {
    while (!utilisateur.aAccepte) {
        // Attendre le consentement

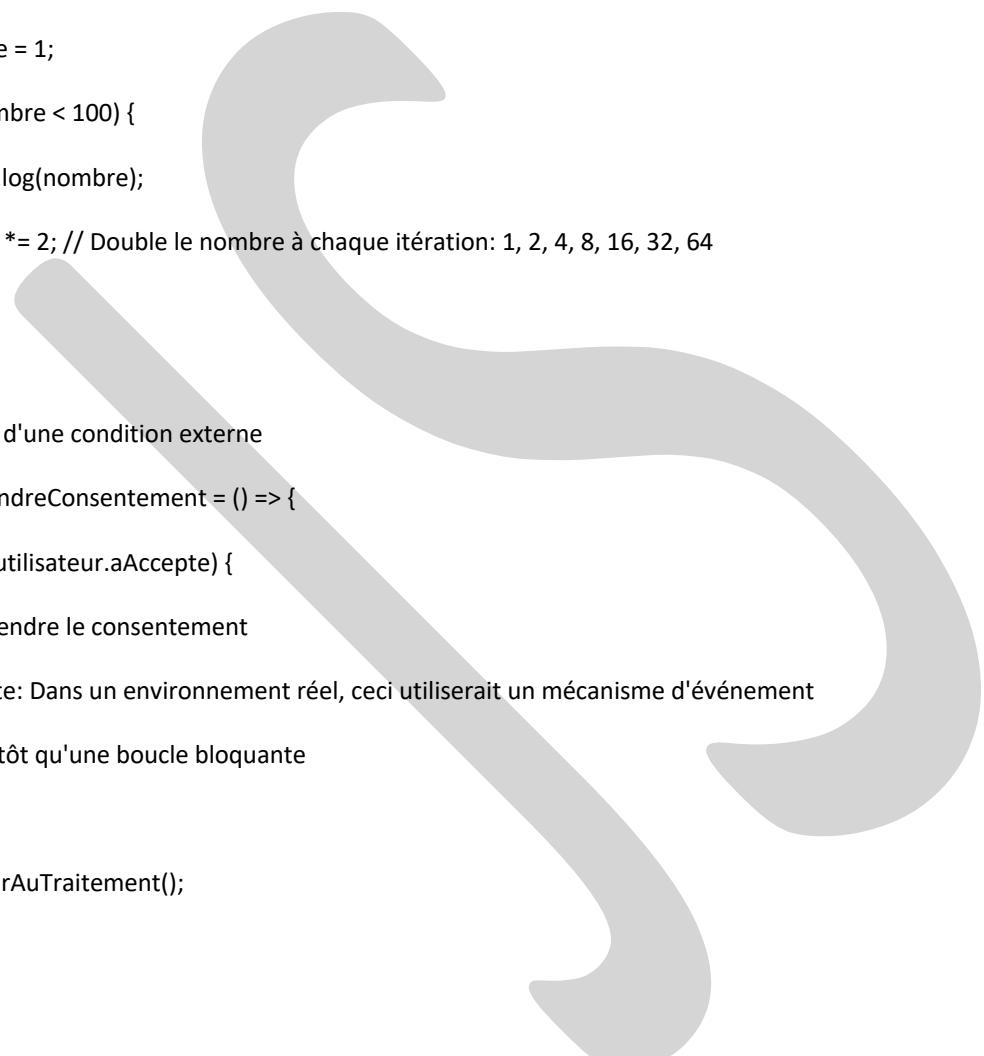
        // Note: Dans un environnement réel, ceci utiliserait un mécanisme d'événement
        // plutôt qu'une boucle bloquante
    }

    procederAuTraitement();
};

// Traiter jusqu'à la fin d'une liste

const processItems = (items) => {
    while (items.length > 0) {
        const item = items.shift(); // Retire et renvoie le premier élément
        traiterItem(item);
    }
}

```

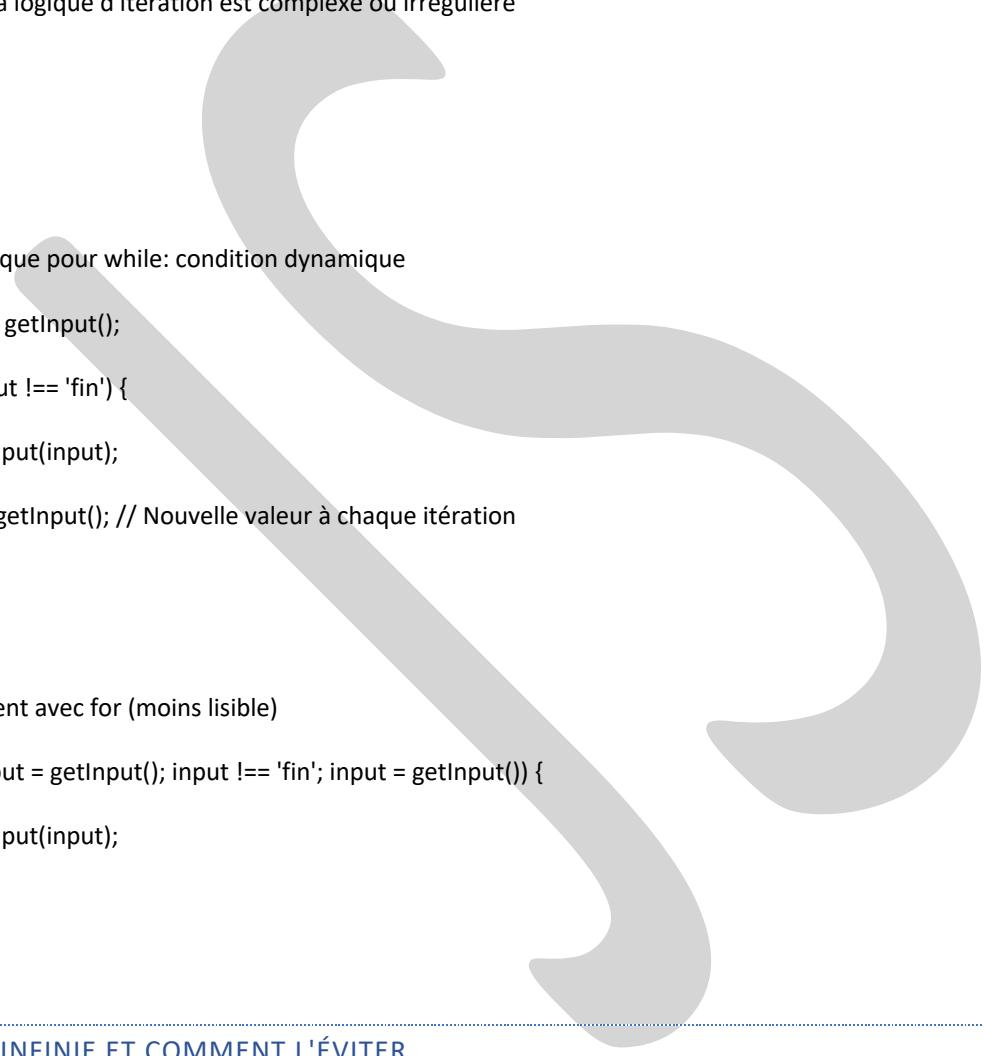


```
};
```

BOUCLE WHILE VS FOR

La boucle while est généralement préférée lorsque :

- Le nombre d'itérations n'est pas connu à l'avance
- La condition de sortie dépend de facteurs externes à la boucle
- La logique d'itération est complexe ou irrégulière



```
// Cas typique pour while: condition dynamique

let input = getInput();
while (input !== 'fin') {
    traiterInput(input);
    input = getInput(); // Nouvelle valeur à chaque itération
}
```

```
// Équivalent avec for (moins lisible)

for (let input = getInput(); input !== 'fin'; input = getInput()) {
    traiterInput(input);
}
```

BOUCLE INFINIE ET COMMENT L'ÉVITER

Si la condition d'une boucle while ne devient jamais 'false', cela crée une boucle infinie qui peut faire planter le programme.

```
// Boucle infinie - À ÉVITER

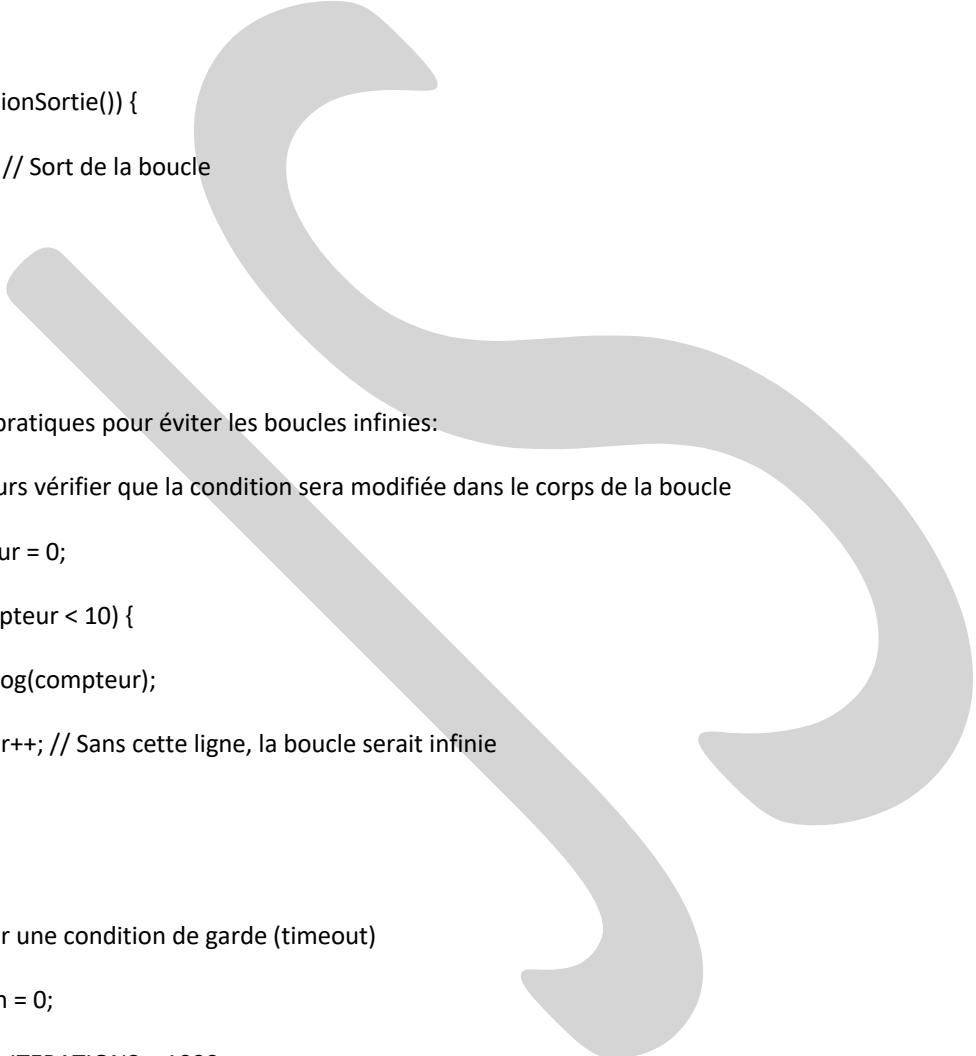
// while (true) {
```

```
// // Code exécuté indéfiniment
// }
```

```
// Boucle avec condition de sortie explicite
```

```
while (true) {
    const donnees = obtenirDonnees();
    traiterDonnees(donnees);
```

```
    if (conditionSortie()) {
        break; // Sort de la boucle
    }
}
```



```
// Bonnes pratiques pour éviter les boucles infinies:
```

// 1. Toujours vérifier que la condition sera modifiée dans le corps de la boucle

```
let compteur = 0;
while (compteur < 10) {
    console.log(compteur);
    compteur++; // Sans cette ligne, la boucle serait infinie
}
```

// 2. Utiliser une condition de garde (timeout)

```
let iteration = 0;
const MAX_ITERATIONS = 1000;

while (conditionPrincipale() && iteration < MAX_ITERATIONS) {
    // Code...
    iteration++;
}
```

```
if (iteration >= MAX_ITERATIONS) {
    console.error("Nombre maximum d'itérations atteint, possible boucle infinie");
}
```

LA BOUCLE DO...WHILE

La boucle do...while est similaire à la boucle while, mais elle exécute le bloc de code au moins une fois avant de vérifier la condition.

SYNTAXE DE BASE

```
do {
    // Code exécuté au moins une fois
} while (condition);
```

EXEMPLES D'UTILISATION

```
// Exécution garantie au moins une fois
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 5); // Affiche 0, 1, 2, 3, 4
```

```
// Même si la condition est initialement fausse
let j = 10;
do {
```

```

console.log("Cette ligne s'affiche une fois");

j++;

} while (j < 5); // La condition est fausse, mais le bloc s'exécute une fois

```

```

// Validation d'entrée utilisateur

let input;

do {

    input = promptUtilisateur("Entrez un nombre positif:");

} while (isNaN(input) || input <= 0);

```

CAS D'UTILISATION TYPIQUES

La boucle do...while est particulièrement utile dans les scénarios où :

- Vous voulez garantir qu'un bloc de code s'exécute au moins une fois
- La validation doit être effectuée après une première action
- Vous interagissez avec l'utilisateur et vous avez besoin d'une entrée initiale

```

// Menu interactif

let choix;

do {

    afficherMenu();

    choix = obtenirChoixUtilisateur();

    switch (choix) {

        case '1': fonctionnalite1(); break;

        case '2': fonctionnalite2(); break;

        case '3': fonctionnalite3(); break;

        case 'q': console.log("Au revoir!"); break;

        default: console.log("Option invalide"); break;

    }
}

```

```
} while (choix !== 'q');
```

3.1.1. INTRODUCTION AUX FONCTIONS

QU'EST-CE QU'UNE FONCTION?

Une fonction est un bloc de code réutilisable conçu pour effectuer une tâche particulière. Les fonctions sont l'un des concepts fondamentaux de JavaScript et constituent les éléments de base pour organiser, structurer et réutiliser le code.

En JavaScript, les fonctions sont des "citoyens de première classe", ce qui signifie qu'elles peuvent être :

- Assignées à des variables
- Passées en tant qu'arguments à d'autres fonctions
- Retournées par d'autres fonctions
- Stockées dans des structures de données comme les tableaux et les objets

Cette flexibilité fait des fonctions un outil extrêmement puissant dans la programmation JavaScript.

POURQUOI UTILISER DES FONCTIONS?

Les fonctions offrent de nombreux avantages dans le développement logiciel :

1. Réutilisation du code

Au lieu d'écrire plusieurs fois le même code, vous pouvez le placer dans une fonction et l'appeler autant de fois que nécessaire.

```
// Sans fonction (répétition)

console.log("Bonjour Alice! Comment allez-vous?");

console.log("Bonjour Bob! Comment allez-vous?");

console.log("Bonjour Charlie! Comment allez-vous?");
```

```
// Avec fonction (réutilisable)

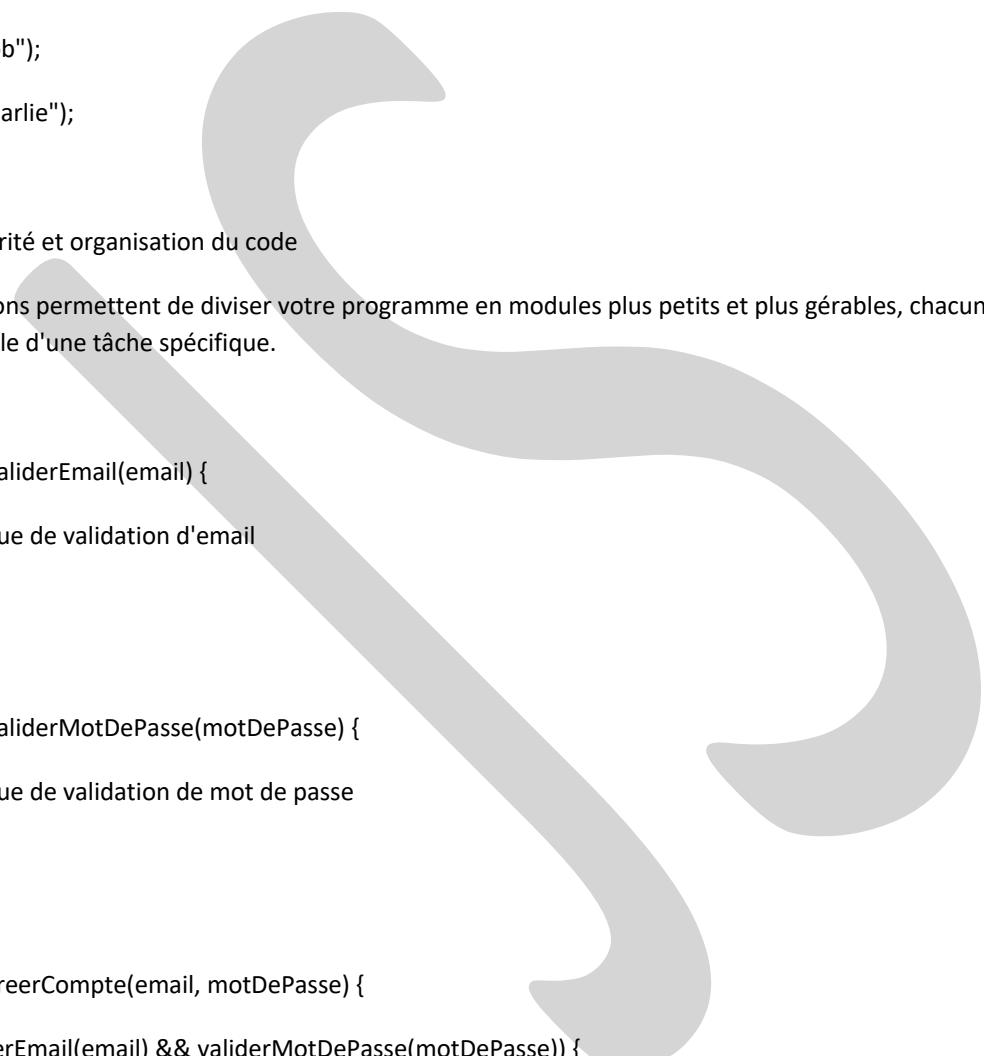
function saluer(nom) {
    console.log(`Bonjour ${nom}! Comment allez-vous?`);

}

saluer("Alice");
saluer("Bob");
saluer("Charlie");
```

2. Modularité et organisation du code

Les fonctions permettent de diviser votre programme en modules plus petits et plus gérables, chacun responsable d'une tâche spécifique.



```
function validerEmail(email) {
    // Logique de validation d'email
}

function validerMotDePasse(motDePasse) {
    // Logique de validation de mot de passe
}

function creerCompte(email, motDePasse) {
    if (validerEmail(email) && validerMotDePasse(motDePasse)) {
        // Création du compte
    }
}
```

3. Abstraction

Les fonctions permettent de masquer la complexité en exposant uniquement les informations nécessaires à l'utilisation.

```
// L'utilisateur de cette fonction n'a pas besoin de comprendre
// les détails complexes de comment les taxes sont calculées

function calculerPrixFinal(prixHT) {
    // Logique complexe de calcul de taxes

    const tauxTVA = obtenirTauxTVA();

    const taxesSpeciales = calculerTaxesSpeciales(prixHT);

    const prixTTC = prixHT * (1 + tauxTVA) + taxesSpeciales;

    return prixTTC;
}

// Utilisation simple

const prixFinal = calculerPrixFinal(99.99);
```

4. Facilité de maintenance

Lorsqu'une logique est centralisée dans une fonction, les modifications et corrections ne doivent être apportées qu'à un seul endroit.

5. Testabilité

Les fonctions bien conçues sont plus faciles à tester car elles ont une responsabilité claire et des entrées/sorties bien définies.

DÉCLARATION ET EXPRESSION DE FONCTION

JavaScript offre plusieurs façons de définir des fonctions. Les deux principales sont la déclaration de fonction et l'expression de fonction.

DÉCLARATION DE FONCTION

```
function nomDeLaFonction(param1, param2, ..., paramN) {
    // Corps de la fonction
    // Code à exécuter
    return valeur; // Optionnel
}
```

EXEMPLE :

```
function additionner(a, b) {
    return a + b;
}

const somme = additionner(5, 3); // 8
```

EXPRESSION DE FONCTION

Une expression de fonction définit une fonction comme partie d'une expression, souvent en l'assignant à une variable.

```
const nomDeLaFonction = function(param1, param2, ..., paramN) {
    // Corps de la fonction
    return valeur; // Optionnel
};
```

Exemple :

```
const multiplier = function(a, b) {
    return a * b;
};
```

```
const produit = multiplier(4, 2); // 8
```

DIFFÉRENCES ENTRE DÉCLARATION ET EXPRESSION

1. Hoisting (remontée) : Les déclarations de fonction sont "remontées" au sommet de leur portée, ce qui signifie qu'elles peuvent être appelées avant leur déclaration dans le code. Les expressions de fonction ne sont pas remontées de la même manière.

```
// Ceci fonctionne
```

```
console.log(addition(2, 3)); // 5
```

```
function addition(a, b) {
```

```
    return a + b;
```

```
}
```

```
// Ceci génère une erreur
```

```
console.log(soustraction(5, 2)); // Erreur: soustraction is not a function
```

```
const soustraction = function(a, b) {
```

```
    return a - b;
```

```
};
```

- 2.*Nommage : Les déclarations de fonction ont obligatoirement un nom, tandis que les expressions de fonction peuvent être anonymes.

3. Contexte d'utilisation : Les expressions de fonction sont plus flexibles et peuvent être utilisées dans des contextes où une expression est attendue.

FONCTIONS FLÉCHÉES (ES6)

ES6 a introduit une syntaxe plus concise pour définir des fonctions, appelée "fonctions fléchées" (arrow functions).

```
const nomDeFonction = (param1, param2, ..., paramN) => {
    // Corps de la fonction
    return valeur;
};
```

Syntaxes simplifiées :

```
// Si la fonction a un seul paramètre, les parenthèses sont optionnelles
const carre = x => {
    return x * x;
};
```

```
// Si le corps contient uniquement un return, les accolades et le mot-clé return peuvent être omis
const carre = x => x * x;
```

```
// Fonction sans paramètre
const salut = () => {
    return "Bonjour!";
};
```

```
// Fonction retournant un objet (parenthèses nécessaires pour éviter l'ambiguïté)
```

```
const creerPersonne = (nom, age) => ({ nom, age });
```

DIFFÉRENCES AVEC LES FONCTIONS TRADITIONNELLES

1. Syntaxe plus concise

2. Pas de liaison `this` : Les fonctions fléchées n'ont pas leur propre `this`. Elles héritent du `this` de leur contexte parent.

```
// Fonction traditionnelle (this dépend de comment la fonction est appelée)
```

```
const personne1 = {
    nom: "Alice",
    saluer: function() {
        console.log(`Bonjour, je suis ${this.nom}`);
    }
};
```

```
// Fonction fléchée (this est celui du contexte de définition)
```

```
const personne2 = {
    nom: "Bob",
    saluer: () => {
        console.log(`Bonjour, je suis ${this.nom}`); // this n'est pas personne2!
    }
};
```

```
personne1.saluer(); // "Bonjour, je suis Alice"
```

```
personne2.saluer(); // "Bonjour, je suis undefined" ou dépend du contexte global
```

3. Pas de arguments : Les fonctions fléchées n'ont pas leur propre objet arguments.

4. Ne peuvent pas être utilisées comme constructeurs : Les fonctions fléchées ne peuvent pas être utilisées avec « new ».

5. Pas de méthodes « prototype »

PARAMÈTRES DE FONCTION

Les paramètres sont les variables qui reçoivent les valeurs passées à une fonction lors de son appel.

PARAMÈTRES DE BASE

```
function saluer(prenom, nom) {  
    console.log(`Bonjour, ${prenom} ${nom}!`);  
}
```

```
saluer("John", "Doe"); // "Bonjour, John Doe!"
```

PARAMÈTRES PAR DÉFAUT (ES6)

Les paramètres par défaut permettent de spécifier des valeurs utilisées lorsqu'un argument n'est pas fourni.

```
function saluer(prenom, nom = "Anonyme") {  
    console.log(`Bonjour, ${prenom} ${nom}!`);  
}
```

```
saluer("John"); // "Bonjour, John Anonyme!"  
saluer("John", "Doe"); // "Bonjour, John Doe!"
```

NOMBRE VARIABLE DE PARAMÈTRES

1. L'objet arguments (méthode traditionnelle)

Chaque fonction (non fléchée) a accès à un objet spécial appelé « arguments » qui contient tous les arguments passés.

```
function somme() {
    let total = 0;
    for (let i = 0; i < arguments.length; i++) {
        total += arguments[i];
    }
    return total;
}
```

```
console.log(somme(1, 2, 3, 4)); // 10
```

2. Paramètre rest (ES6)

Le paramètre rest (`...`) permet de collecter un nombre variable d'arguments dans un tableau.

```
function somme(...nombres) {
    return nombres.reduce((total, n) => total + n, 0);
}
```

```
console.log(somme(1, 2, 3, 4)); // 10
```

Le paramètre rest doit être le dernier paramètre dans la liste des paramètres :

```
function registre(action, ...elements) {
    console.log(`Action: ${action}`);
    console.log(`Éléments: ${elements.join(', ')}`);
}
```

```
registre("AJOUTER", "pomme", "banane", "orange");
// Action: AJOUTER
// Éléments: pomme, banane, orange
```

DESTRUCTURATION DES PARAMÈTRES (ES6)

La destructuration permet d'extraire des valeurs de tableaux ou de propriétés d'objets directement dans les paramètres.



```
// Destructuration d'objet

function afficherPersonne({ nom, age, ville = "Inconnu" }) {
    console.log(`${nom}, ${age} ans, de ${ville}`);
}

const personne = { nom: "Alice", age: 30, profession: "Développeur" };
afficherPersonne(personne); // "Alice, 30 ans, de Inconnu"
```

```
// Destructuration de tableau

function afficherCoordonnees([x, y, z = 0]) {
    console.log(`x: ${x}, y: ${y}, z: ${z}`);
}

afficherCoordonnees([10, 20]); // "x: 10, y: 20, z: 0"
```

VALEUR DE RETOUR DES FONCTIONS

Une fonction peut renvoyer une valeur à l'aide de l'instruction return. Si aucune instruction return n'est présente, la fonction retourne undefined.

```
// Fonction avec retour

function multiplier(a, b) {
    return a * b;
}

const resultat = multiplier(4, 5); // 20
```

```
// Fonction sans retour explicite

function saluer(nom) {
    console.log(`Bonjour ${nom}!`);

    // Pas de return, donc retourne undefined
}

const resultatSalut = saluer("Alice"); // undefined
```

RETOUR ANTICIPÉ

L'instruction `return` termine immédiatement l'exécution de la fonction et renvoie la valeur spécifiée.

```
function estPositif(nombre) {
    if (nombre <= 0) {
        return false;
    }

    // Ce code ne s'exécute que si nombre > 0
    return true;
}
```

```
// Utilisation comme garde

function diviser(a, b) {

    if (b === 0) {

        console.error("Division par zéro!");

        return null; // Sortie anticipée

    }

    return a / b; // S'exécute uniquement si b !== 0
}
```

RETOURNER PLUSIEURS VALEURS

JavaScript ne permet pas de retourner plusieurs valeurs directement, mais il existe plusieurs approches pour simuler ce comportement :

1. Retourner un tableau

```
function calculerStatistiques(nombres) {

    const min = Math.min(...nombres);

    const max = Math.max(...nombres);

    const sum = nombres.reduce((acc, n) => acc + n, 0);

    const avg = sum / nombres.length;

    return [min, max, avg];
}
```

```
const [minimum, maximum, moyenne] = calculerStatistiques([1, 5, 3, 9, 2]);
```

2. Retourner un objet

```

function calculerStatistiques(nombres) {
    const min = Math.min(...nombres);
    const max = Math.max(...nombres);
    const sum = nombres.reduce((acc, n) => acc + n, 0);
    const avg = sum / nombres.length;

    return { min, max, moyenne: avg };
}

const stats = calculerStatistiques([1, 5, 3, 9, 2]);
console.log(stats.min, stats.max, stats.moyenne);

```

PORTEE DES VARIABLES EN JAVASCRIPT

La portée (scope) détermine la visibilité et l'accessibilité des variables dans votre code.

PORTEE GLOBALE

Les variables déclarées en dehors de toute fonction ou bloc ont une portée globale et sont accessibles partout dans le code.

```

// Variable globale
const appName = "MonApplication";

function afficherInfo() {
    console.log(appName); // Accès à la variable globale
}

```

```
afficherInfo(); // "MonApplication"
```

PORTEE DE FONCTION

Les variables déclarées à l'intérieur d'une fonction ne sont accessibles que dans cette fonction.

```
function calculer() {
    const x = 10; // Portée limitée à la fonction calculer
    console.log(x); // 10
}

calculer();
// console.log(x); // Erreur: x is not defined
```

PORTEE DE BLOC (ES6)

Les variables déclarées avec let et const ont une portée de bloc, limitée au bloc {} dans lequel elles sont déclarées.

```
if (true) {
    let x = 10; // Portée limitée au bloc if
    const y = 20; // Portée limitée au bloc if
    var z = 30; // Portée de fonction (ou globale si en dehors d'une fonction)
}

// console.log(x); // Erreur: x is not defined
// console.log(y); // Erreur: y is not defined
console.log(z); // 30 (var ignore la portée de bloc)
```

```

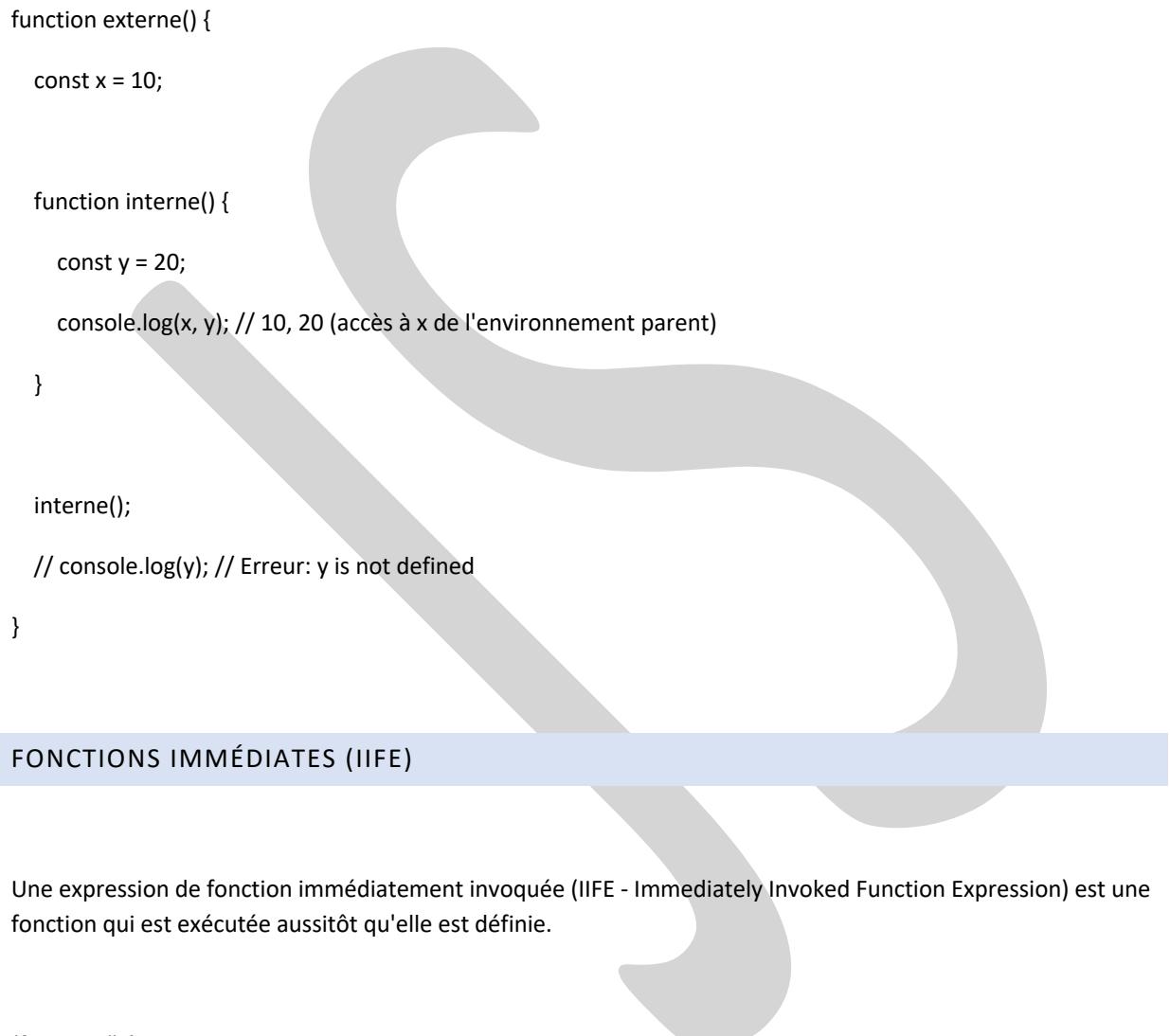
## VARIABLES ET ENVIRONNEMENTS LEXICAUX

Chaque fonction crée son propre environnement lexical, qui contient ses variables locales et a accès aux environnements parents.

```
function externe() {
 const x = 10;

 function interne() {
 const y = 20;
 console.log(x, y); // 10, 20 (accès à x de l'environnement parent)
 }

 interne();
 // console.log(y); // Erreur: y is not defined
}
```



## FONCTIONS IMMÉDIATES (IIFE)

Une expression de fonction immédiatement invoquée (IIFE - Immediately Invoked Function Expression) est une fonction qui est exécutée aussitôt qu'elle est définie.

```
(function() {
 // Code isolé
 const privateVar = "Je suis privée";
 console.log(privateVar);
})();
```

```
// console.log(privateVar); // Erreur: privateVar is not defined
```

Les IIFE sont souvent utilisées pour créer un scope isolé et éviter de polluer le scope global.

#### EXEMPLE AVEC PARAMÈTRES

```
(function(global, nom) {
 console.log('Bonjour, ${nom}!');
 global.salutation = `Salut, ${nom}`; // Crée une variable globale
})(window, "JavaScript"); // Passe window comme global et "JavaScript" comme nom

console.log(salutation); // "Salut, JavaScript!"
```

Les fonctions sont des éléments fondamentaux en JavaScript qui permettent d'organiser, de réutiliser et de structurer le code. Leur flexibilité offre de nombreuses approches pour résoudre divers problèmes de programmation.

Dans les prochaines sections, nous explorerons des concepts plus avancés liés aux fonctions, tels que les closures, les fonctions d'ordre supérieur, et les modèles de conception basés sur les fonctions.

Exercices pratiques:

1. Créez une fonction qui calcule la factorielle d'un nombre.
2. Écrivez une fonction qui accepte un nombre variable d'arguments et renvoie leur somme.
3. Créez une fonction qui prend un objet personne et affiche son nom complet, avec une valeur par défaut pour le nom de famille.
4. Réécrivez une fonction traditionnelle en utilisant la syntaxe de fonction fléchée.

#### 3.1.2. FONCTIONS AVANCÉES ET CLOSURES

## INTRODUCTION AUX CLOSURES

Une closure (ou fermeture) est l'une des caractéristiques les plus puissantes et souvent mal comprises de JavaScript. Une closure se produit lorsqu'une fonction est capable de "se souvenir" et d'accéder à son environnement lexical (portée) même lorsqu'elle est exécutée en dehors de cet environnement.

En termes simples, une closure permet à une fonction de conserver l'accès aux variables de la fonction parente même après que cette dernière ait terminé son exécution.

### DÉFINITION FORMELLE

Une closure est créée lorsqu'une fonction interne fait référence à des variables de sa fonction externe (englobante). La fonction interne "ferme" (capture) les variables de son environnement parent, d'où le terme "closure".

### EXEMPLE DE BASE

```
function creerCompteur() {
 let compteur = 0; // Variable locale à creerCompteur

 function incrementer() {
 compteur++; // Accès à la variable de la fonction parente
 return compteur;
 }

 return incrementer; // Retourne la fonction interne
}
```

```
const monCompteur = creerCompteur(); // monCompteur est maintenant une closure
```

```
console.log(monCompteur()); // 1
```

```
console.log(monCompteur()); // 2
console.log(monCompteur()); // 3
```

Dans cet exemple :

1. creerCompteur définit une variable locale compteur
2. incrementer est une fonction interne qui utilise cette variable
3. creerCompteur renvoie la fonction incrementer
4. Même après que creerCompteur ait terminé son exécution, monCompteur (qui est la fonction incrementer) continue d'avoir accès à la variable `compteur`

### POURQUOI LES CLOSURES SONT-ELLES IMPORTANTES ?

Les closures sont fondamentales en JavaScript pour plusieurs raisons :

1. Données privées : Elles permettent de créer des variables "privées" non accessibles depuis l'extérieur
2. Préservation d'état : Elles permettent aux fonctions de maintenir un état entre les appels
3. Encapsulation : Elles favorisent le principe d'encapsulation de la programmation orientée objet
4. Fonctions d'ordre supérieur : Elles sont essentielles pour les fonctions qui créent d'autres fonctions
5. Modèles de conception : Elles sont à la base de nombreux modèles de conception en JavaScript

### UTILISATIONS PRATIQUES DES CLOSURES

1. Création de données privées

```
function createBankAccount(initialBalance) {
 let balance = initialBalance; // Variable "privée"

 return {
 deposit: function(amount) {
```

```

if (amount > 0) {
 balance += amount;
 return true;
}
return false;
},
withdraw: function(amount) {
 if (amount > 0 && balance >= amount) {
 balance -= amount;
 return true;
 }
 return false;
},
getBalance: function() {
 return balance;
}
};

}

const account = createBankAccount(100);
account.deposit(50); // balance = 150
account.withdraw(30); // balance = 120
console.log(account.getBalance()); // 120

```



```

// Impossible d'accéder directement à balance
// console.log(account.balance); // undefined
...

```

## 2. Fonctions de fabrique (Factory functions)

```
function createMultiplier(factor) {
 return function(number) {
 return number * factor;
 };
}
```

```
const double = createMultiplier(2);
const triple = createMultiplier(3);
```

```
console.log(double(5)); // 10
console.log(triple(5)); // 15
```

### 3. Gestionnaires d'événements avec contexte

```
function setupButtonHandler(buttonId, message) {
 const button = document.getElementById(buttonId);

 button.addEventListener('click', function() {
 // Cette fonction de rappel est une closure qui capture la variable 'message'
 alert(message);
 });
}
```

```
setupButtonHandler('button1', 'Bonjour depuis le bouton 1');
setupButtonHandler('button2', 'Bonjour depuis le bouton 2');
```

### 4. Memoization (mise en cache)



```

function memoize(fn) {
 const cache = {};

 return function(...args) {
 const key = JSON.stringify(args);

 if (cache[key] === undefined) {
 cache[key] = fn(...args);
 }

 return cache[key];
 };
}

// Fonction coûteuse à exécuter

function calculerFibonacci(n) {
 if (n <= 1) return n;

 return calculerFibonacci(n - 1) + calculerFibonacci(n - 2);
}

// Version mémoisée

const fibonacciMemoize = memoize(function(n) {
 if (n <= 1) return n;

 return fibonacciMemoize(n - 1) + fibonacciMemoize(n - 2);
});

console.time('Sans mémoïsation');
console.log(calculerFibonacci(35));
console.timeEnd('Sans mémoïsation');

```

```
console.time('Avec mémoïsation');

console.log(fibonacciMemoize(35));

console.timeEnd('Avec mémoïsation');
```

## 5. Modules (pattern module)

```
const monModule = (function() {

 // Variables privées

 let compteur = 0;

 const valeurSecrete = "xyz123";

 // Fonction privée

 function incrementerCompteur() {

 compteur++;

 }

 // API publique

 return {

 augmenter: function() {

 incrementerCompteur();

 return compteur;

 },

 diminuer: function() {

 compteur--;

 return compteur;

 },

 getCompteur: function() {

 return compteur;

 }

 }

});
```

```

};

})();

console.log(monModule.getCompteur()); // 0
console.log(monModule.augmenter()); // 1
console.log(monModule.augmenter()); // 2
console.log(monModule.diminuer()); // 1

```

```

// Impossible d'accéder aux variables ou fonctions privées
// console.log(monModule.compteur); // undefined
// console.log(monModule.valeurSecret); // undefined
// monModule.incrementerCompteur(); // Erreur

```

## PIÈGES COURANTS AVEC LES CLOSURES

### 1. Problème de boucle et de closure

Un problème classique se produit lorsqu'on crée des closures dans une boucle :

```

// Problème

function setupButtons() {
 const buttons = document.getElementsByTagName('button');

 for (var i = 0; i < buttons.length; i++) {
 buttons[i].addEventListener('click', function() {
 console.log('Bouton ' + i + ' cliqué');
 });
 }
}

setupButtons();

```

```

 }
}
```

Solutions :

```
// Solution 1 : IIFE pour créer un nouveau scope

function setupButtons() {

 const buttons = document.getElementsByTagName('button');

 for (var i = 0; i < buttons.length; i++) {

 (function(index) {

 buttons[index].addEventListener('click', function() {

 console.log('Bouton ' + index + ' cliqué');

 });

 })(i);

 }
}
```

// Solution 2 : Utiliser let (ES6) pour créer un nouveau scope à chaque itération

```
function setupButtons() {

 const buttons = document.getElementsByTagName('button');

 for (let i = 0; i < buttons.length; i++) {

 buttons[i].addEventListener('click', function() {

 console.log('Bouton ' + i + ' cliqué');

 // Fonctionne car 'let' crée un nouveau scope pour chaque itération

 });

 }
}
```

## 2. Fermeture sur des valeurs mutables

Les closures capturent des références, pas des valeurs. Cela peut parfois conduire à des comportements inattendus :

```
function createFunctions() {
 let result = [];
 let obj = { value: 'initial' };

 result.push(function() { return obj.value; });

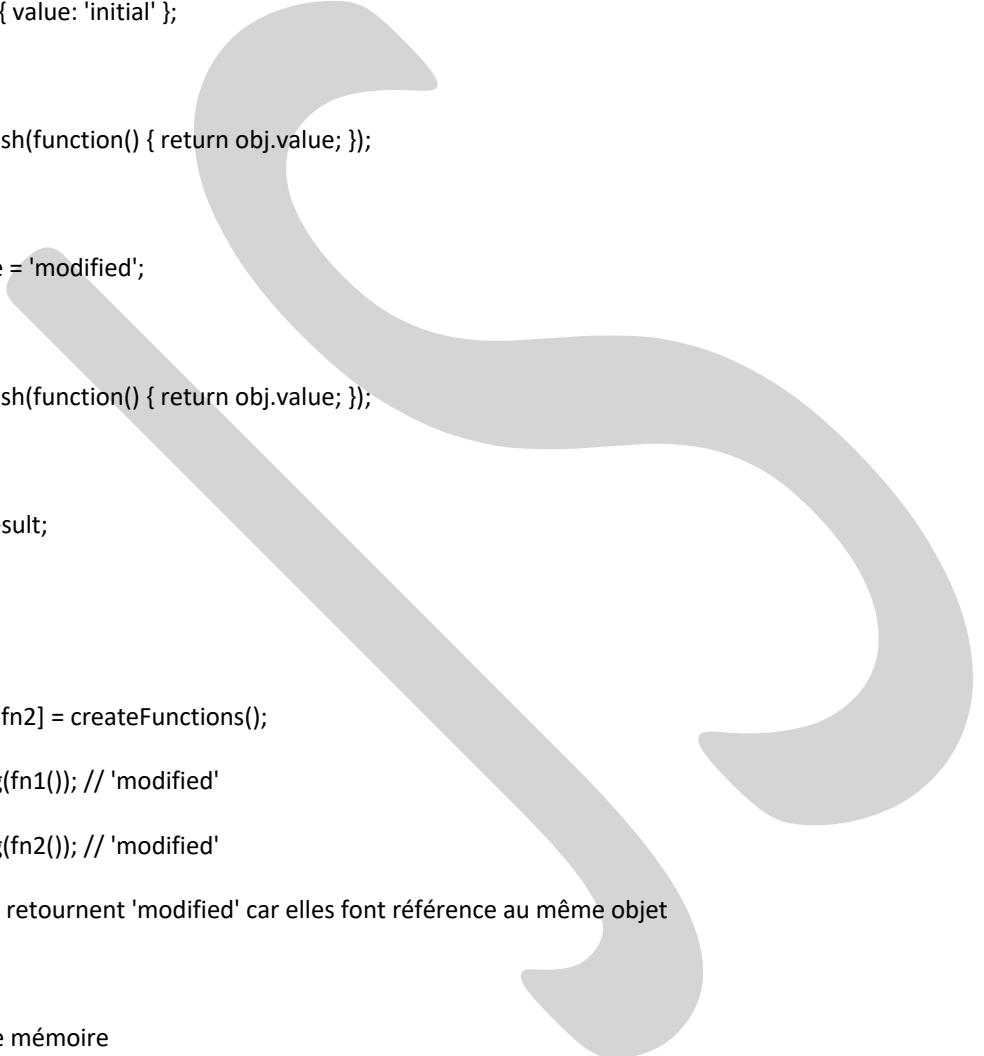
 obj.value = 'modified';

 result.push(function() { return obj.value; });

 return result;
}

const [fn1, fn2] = createFunctions();
console.log(fn1()); // 'modified'
console.log(fn2()); // 'modified'

// Les deux retournent 'modified' car elles font référence au même objet
```



## 3. Fuites de mémoire

Les closures peuvent provoquer des fuites de mémoire si elles ne sont pas gérées correctement :

```
function attachEventListener() {
 const largeData = new Array(1000000).fill('some data');
```

```

const element = document.getElementById('myButton');

element.addEventListener('click', function() {
 // Cette closure maintient une référence à largeData,
 // empêchant sa libération tant que l'écouteur existe
 console.log('Button clicked, data size:', largeData.length);
});

}

```

Solution :



```

function attachEventListener() {
 const element = document.getElementById('myButton');

 // Extrayez seulement les données nécessaires
 const dataSize = prepareData().length;

 element.addEventListener('click', function() {
 // N'utilise que dataSize, pas la référence complète
 console.log('Button clicked, data size:', dataSize);
 });
}

function prepareData() {
 return new Array(1000000).fill('some data');
}

// Ici, largeData peut être garbage collecté
}

```

## FONCTIONS D'ORDRE SUPÉRIEUR

Les fonctions d'ordre supérieur sont des fonctions qui prennent d'autres fonctions comme arguments ou qui renvoient des fonctions. Les closures sont souvent utilisées avec des fonctions d'ordre supérieur.

### 1. Fonctions prenant des fonctions en arguments

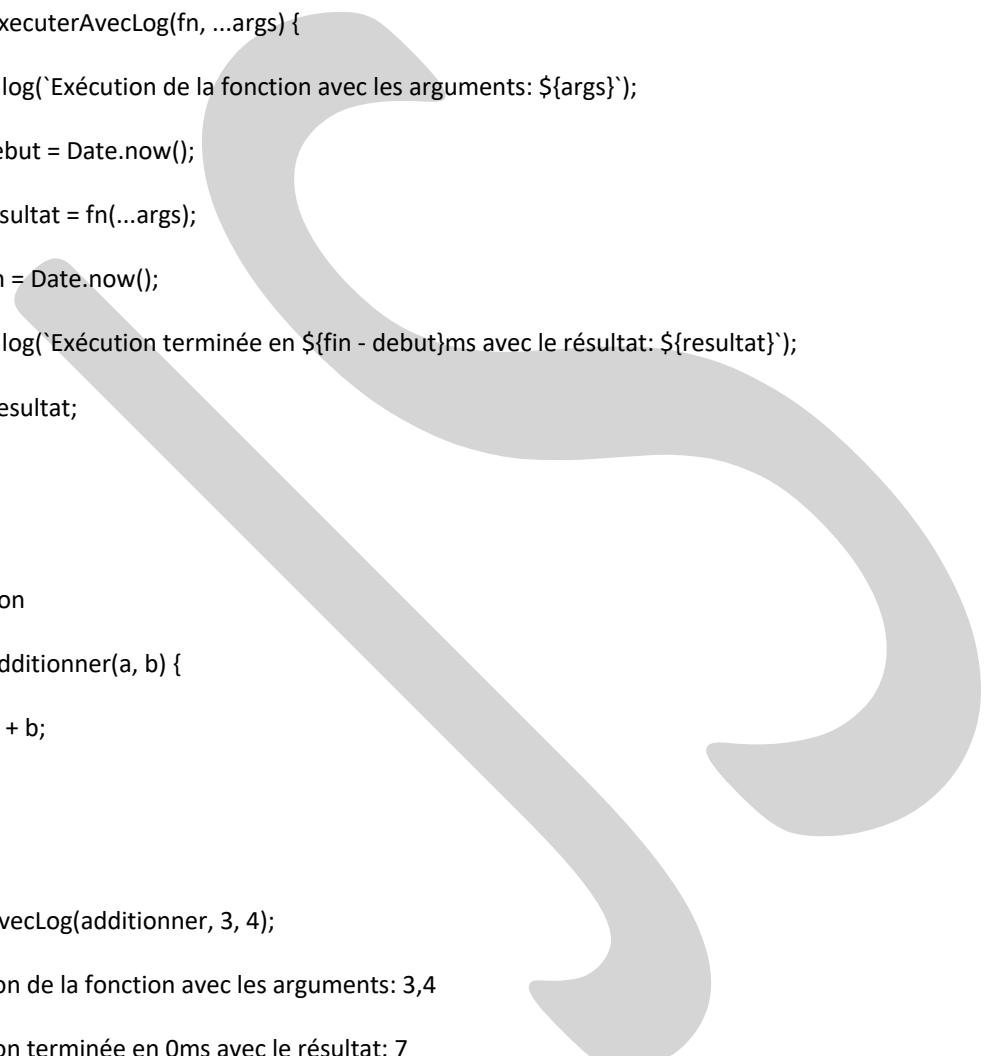
```
// Fonction d'ordre supérieur qui prend une fonction en paramètre

function executerAvecLog(fn, ...args) {
 console.log(`Exécution de la fonction avec les arguments: ${args}`);
 const debut = Date.now();
 const resultat = fn(...args);
 const fin = Date.now();
 console.log(`Exécution terminée en ${fin - debut}ms avec le résultat: ${resultat}`);
 return resultat;
}

// Utilisation

function additionner(a, b) {
 return a + b;
}

executerAvecLog(additionner, 3, 4);
// Exécution de la fonction avec les arguments: 3,4
// Exécution terminée en 0ms avec le résultat: 7
```



### 2. Fonctions retournant des fonctions

```
// Fonction d'ordre supérieur qui retourne une fonction

function createValidator(validationFn, errorMsg) {
 return function(value) {
```

```

if (!validationFn(value)) {
 return { valid: false, error: errorMsg };
}

return { valid: true };
};

}

// Création de validateurs spécifiques
const validateEmail = createValidator(
 value => /\S+@\S+\.\S+/.test(value),
 "Format d'email invalide"
);

const validatePassword = createValidator(
 value => value.length >= 8,
 "Le mot de passe doit contenir au moins 8 caractères"
);

// Utilisation
console.log(validateEmail("test@example.com")); // { valid: true }

console.log(validateEmail("invalid")); // { valid: false, error: "Format d'email invalide" }

console.log(validatePassword("short")); // { valid: false, error: "Le mot de passe doit contenir au moins 8 caractères" }

```

### 3. Curry et composition de fonctions

Le currying est une technique avancée permettant de transformer une fonction qui prend plusieurs arguments en une séquence de fonctions qui prennent un seul argument chacune.

```
// Version standard
```

```
function add(a, b) {
 return a + b;
}
```

// Version curryfiée

```
function curriedAdd(a) {
 return function(b) {
 return a + b;
 };
}
```

// Utilisation

```
console.log(add(2, 3)); // 5
console.log(curriedAdd(2)(3)); // 5
```

// Création d'une fonction spécialisée

```
const addTwo = curriedAdd(2);
console.log(addTwo(3)); // 5
console.log(addTwo(10)); // 12
```

La composition de fonctions permet de combiner plusieurs fonctions pour en créer une nouvelle :

// Fonctions de base

```
const double = x => x * 2;
const increment = x => x + 1;
const square = x => x * x;
```

// Fonction de composition

```
function compose(...fns) {
 return function(initialValue) {
```

```

 return fns.reduceRight((value, fn) => fn(value), initialValue);

 };

}

// Création d'une fonction composée

const doublePlusOneSquared = compose(square, increment, double);

// Utilisation

console.log(doublePlusOneSquared(3)); // ((3*2)+1)^2 = 49

```

## RÉCURSIVITÉ ET FONCTIONS AUTO-RÉFÉRENTES

La récursivité est une technique où une fonction s'appelle elle-même. Les closures peuvent être utilisées pour créer des fonctions récursives auto-référentes.

### RÉCURSIVITÉ DE BASE

```

function factorielle(n) {

 if (n <= 1) return 1;

 return n * factorielle(n - 1);

}

console.log(factorielle(5)); // 120 (5 * 4 * 3 * 2 * 1)

```

### RÉCURSIVITÉ AVEC FONCTION ANONYME AUTO-RÉFÉRENTE

```

// Fonction récursive anonyme qui calcule la factorielle

const fact = (function() {

```

```
const f = function(n) {
 if (n <= 1) return 1;
 return n * f(n - 1);
};

return f;
})();
```

```
console.log(fact(5)); // 120
```

## RÉCURSIVITÉ ET CLOSURES

```
// Génération d'arborescence

function createTreeProcessor(tree) {
 // Fonction interne récursive qui utilise la closure

 function process(node, depth = 0) {
 console.log(" ".repeat(depth) + node.name);

 if (node.children && node.children.length > 0) {
 for (const child of node.children) {
 process(child, depth + 1);
 }
 }
 }

 return function() {
 process(tree);
 };
}
```

```
const tree = {
 name: "Root",
 children: [
 { name: "Child 1" },
 {
 name: "Child 2",
 children: [
 { name: "Grandchild 1" },
 { name: "Grandchild 2" }
]
 }
];
};
```

```
const processTree = createTreeProcessor(tree);
processTree();
// Root
// Child 1
// Child 2
// Grandchild 1
// Grandchild 2
```

## OPTIMISATION ET BONNES PRATIQUES

### 1. Minimiser les références capturées

Capturez uniquement les variables dont vous avez besoin dans vos closures :

```
// Pas optimal - capture tout l'objet
```

```
function createPersonLogger(person) {
 return function() {
 console.log(` ${person.firstName} ${person.lastName}`);
 };
}
```

// Optimal - capture seulement les valeurs nécessaires

```
function createPersonLogger(person) {
 const fullName = `${person.firstName} ${person.lastName}`;
 return function() {
 console.log(fullName);
 };
}
```

## 2. Libérer les références

Assurez-vous de libérer les références qui ne sont plus nécessaires :

```
function setupHandler() {
 const largeData = new Array(1000000);
 const element = document.getElementById('myElement');
```

```
function clickHandler() {
 console.log('Clicked', largeData.length);
}
```

```
element.addEventListener('click', clickHandler);
```

```
// Fournir une fonction pour nettoyer
```

```

return function cleanup() {
 element.removeEventListener('click', clickHandler);
 // Supprimer la référence
 largeData = null;
};

}

```

```

const cleanup = setupHandler();
// Plus tard, quand ce n'est plus nécessaire
cleanup();

```

### 3. Éviter la sur-imbrication

Trop de niveaux d'imbrication peuvent rendre le code difficile à lire et à maintenir :

```

// Difficile à lire et à maintenir

function complexFunction() {
 return function() {
 return function() {
 return function() {
 // ...
 };
 };
 };
}

// Plus lisible

function createStepOne() {
 // ...
}

```

```
function createStepTwo(stepOne) {
 // ...
}

function createStepThree(stepTwo) {
 // ...
}

function complexFunction() {
 const stepOne = createStepOne();
 const stepTwo = createStepTwo(stepOne);
 return createStepThree(stepTwo);
}
```

Les closures sont un mécanisme fondamental en JavaScript qui permet de créer des structures de code puissantes et flexibles. Elles sont à la base de nombreux modèles de conception et fonctionnalités avancées du langage.

Maîtriser les closures vous permettra de :

- Créer des données privées et encapsulées
- Maintenir un état entre les appels de fonction
- Implémenter des modèles de conception avancés
- Optimiser vos performances avec des techniques comme la mémoïsation
- Créer des API élégantes et des DSL (Domain Specific Languages)

Les closures sont particulièrement puissantes lorsqu'elles sont combinées avec d'autres concepts avancés comme les fonctions d'ordre supérieur, le currying, et la composition de fonctions, que nous explorerons dans les sections suivantes.

Exercices pratiques :

1. Créez une fonction de compteur qui peut être incrémentée, décrémentée et réinitialisée.
2. Implémentez une fonction de mémoïsation qui met en cache les résultats d'une fonction.
3. Créez une bibliothèque de validation de formulaire utilisant des closures pour générer différents validateurs.
4. Implémentez le pattern module pour créer une calculatrice avec des fonctions publiques et privées.

### 3.2.1. CALLBACKS ET PROMESSES

#### INTRODUCTION À L'ASYNCRONISME EN JAVASCRIPT

JavaScript est un langage à thread unique qui utilise un modèle d'exécution non bloquant. Cette caractéristique permet au code de continuer à s'exécuter pendant que des opérations asynchrones (comme des requêtes réseau, des opérations de fichier, ou des minuteries) sont en cours. Pour gérer ces opérations asynchrones, JavaScript a évolué à travers différents mécanismes : d'abord les callbacks, puis les promesses, et enfin `async/await`.

Pourquoi l'asynchronisme est important

- ☞ Performance : Permet d'exécuter des opérations longues sans bloquer le thread principal
- ☞ Réactivité : Maintient une interface utilisateur réactive même pendant des opérations intensives
- ☞ Concurrence : Permet de gérer plusieurs opérations simultanément

#### LES FONCTIONS DE RAPPEL (CALLBACKS)

##### DÉFINITION ET CONCEPT

Un callback est une fonction passée en argument à une autre fonction, qui sera exécutée après que cette dernière ait terminé son traitement.

```
function executerPlustard(callback {
 // Faire quelque chose...
 console.log("Opération principale terminée");
 callback();
})
```

```
// Puis exécuter le callback
callback();
}
```

```
// Utilisation
executerPlustard(function() {
 console.log("Callback exécuté");
});
```

```
// Sortie :
// Opération principale terminée
// Callback exécuté
```

### CALLBACKS SYNCHRONES VS ASYNCHRONES

#### CALLBACKS SYNCHRONES

Exécutés immédiatement pendant l'exécution de la fonction principale :

```
// Callback synchrone
[1, 2, 3].forEach(function(element) {
 console.log(element);
});
```

#### CALLBACKS ASYNCHRONES

Exécutés après un certain délai ou lorsqu'un événement particulier se produit :

```
// Callback asynchrone

setTimeout(function() {
 console.log("Ce message s'affiche après 2 secondes");
}, 2000);
```

```
console.log("Ce message s'affiche immédiatement");
```

```
// Sortie :
// Ce message s'affiche immédiatement
// Ce message s'affiche après 2 secondes
```

### EXEMPLES D'UTILISATION DES CALLBACKS

#### 1. Lecture de fichier en Node.js

```
const fs = require('fs');

fs.readFile('monFichier.txt', 'utf8', function(err, data) {
 if (err) {
 console.error("Erreur lors de la lecture du fichier:", err);
 return;
 }
 console.log("Contenu du fichier:", data);
});
```

```
console.log("Cette ligne s'exécute avant la fin de la lecture du fichier");
```

#### 2. Requête AJAX avec XMLHttpRequest

```
function chargerDonnees(url, callback) {

 const xhr = new XMLHttpRequest();

 xhr.open('GET', url);

 xhr.onload = function() {

 if (xhr.status === 200) {

 callback(null, JSON.parse(xhr.responseText));

 } else {

 callback(new Error(`Erreur ${xhr.status}: ${xhr.statusText}`));

 }

 };

 xhr.onerror = function() {

 callback(new Error('Erreur réseau'));

 };

 xhr.send();

}

chargerDonnees('https://api.example.com/data', function(err, donnees) {

 if (err) {

 console.error(err);

 return;

 }

 console.log(donnees);

});
```

---

#### LE PROBLÈME DU "CALLBACK HELL" (L'ENFER DES CALLBACKS)

Lorsque plusieurs opérations asynchrones doivent être exécutées séquentiellement, les callbacks peuvent conduire à un code profondément imbriqué et difficile à maintenir :

```
getUtilisateur(idUtilisateur, function(utilisateur) {
 getCommandes(utilisateur.id, function(commandes) {
 getDetailsCommande(commandes[0].id, function(details) {
 getProduitsCommande(details.id, function(produits) {
 // La pyramide de callbacks continue...
 console.log(produits);
 }, function(erreur) {
 console.error("Erreur produits:", erreur);
 });
 }, function(erreur) {
 console.error("Erreur détails:", erreur);
 });
 }, function(erreur) {
 console.error("Erreur commandes:", erreur);
 });
}, function(erreur) {
 console.error("Erreur utilisateur:", erreur);
});
```

Problèmes avec cette approche :

- ☞ Lisibilité : Code difficile à lire et à comprendre
- ☞ Gestion d'erreurs : La gestion des erreurs devient répétitive et verbeuse
- ☞ Refactorisation : Difficile à modifier ou à réorganiser
- ☞ Portée : Les variables partagées entre les callbacks peuvent poser des problèmes

## LES PROMESSES (PROMISES)

## DÉFINITION ET CONCEPT

Une promesse est un objet qui représente une valeur qui pourrait être disponible maintenant, dans le futur, ou jamais. Elle possède trois états possibles :

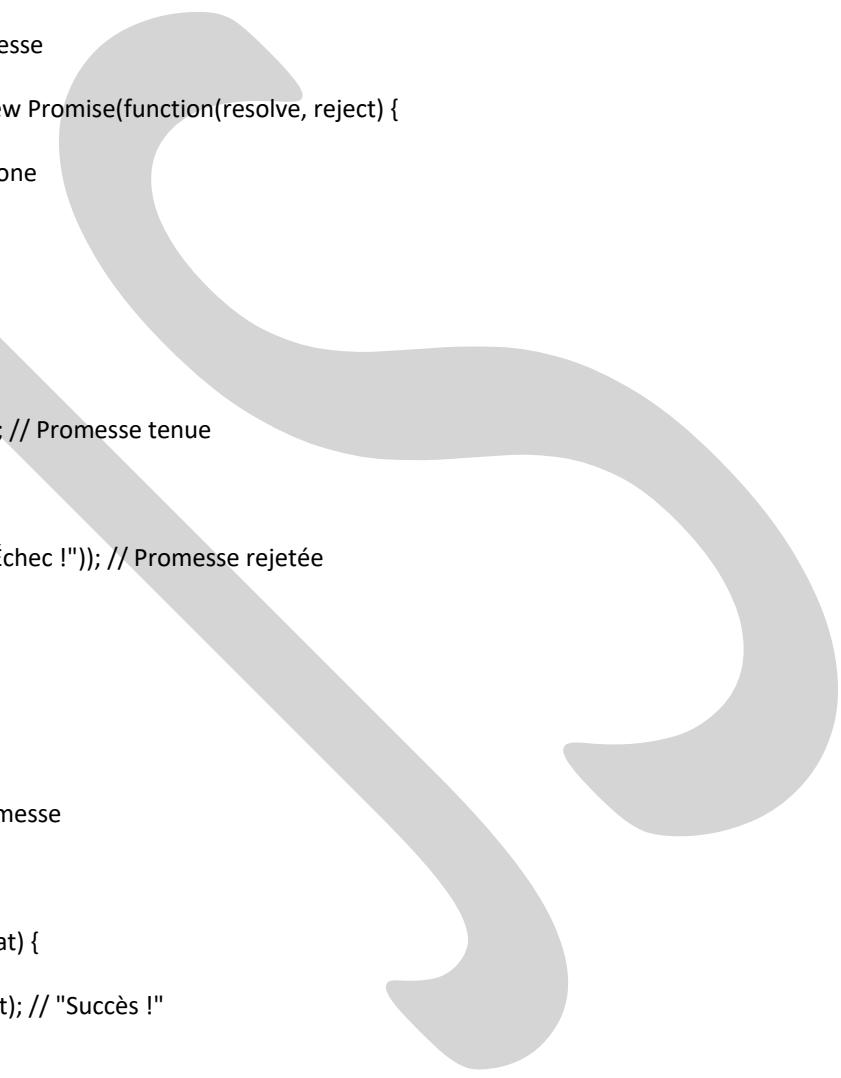
- ✉ pending (en attente) : état initial, ni terminée ni rejetée
- ✉ fulfilled (tenue) : l'opération a réussi et la promesse a une valeur
- ✉ rejected (rejetée) : l'opération a échoué et la promesse a une raison d'échec

```
// Création d'une promesse
const maPromesse = new Promise(function(resolve, reject) {
 // Opération asynchrone
 const success = true;

 if (success) {
 resolve("Succès !"); // Promesse tenue
 } else {
 reject(new Error("Échec !")); // Promesse rejetée
 }
});

// Utilisation d'une promesse
maPromesse
 .then(function(resultat) {
 console.log(resultat); // "Succès !
 })
 .catch(function(erreur) {
 console.error(erreur); // En cas d'échec
 });

```



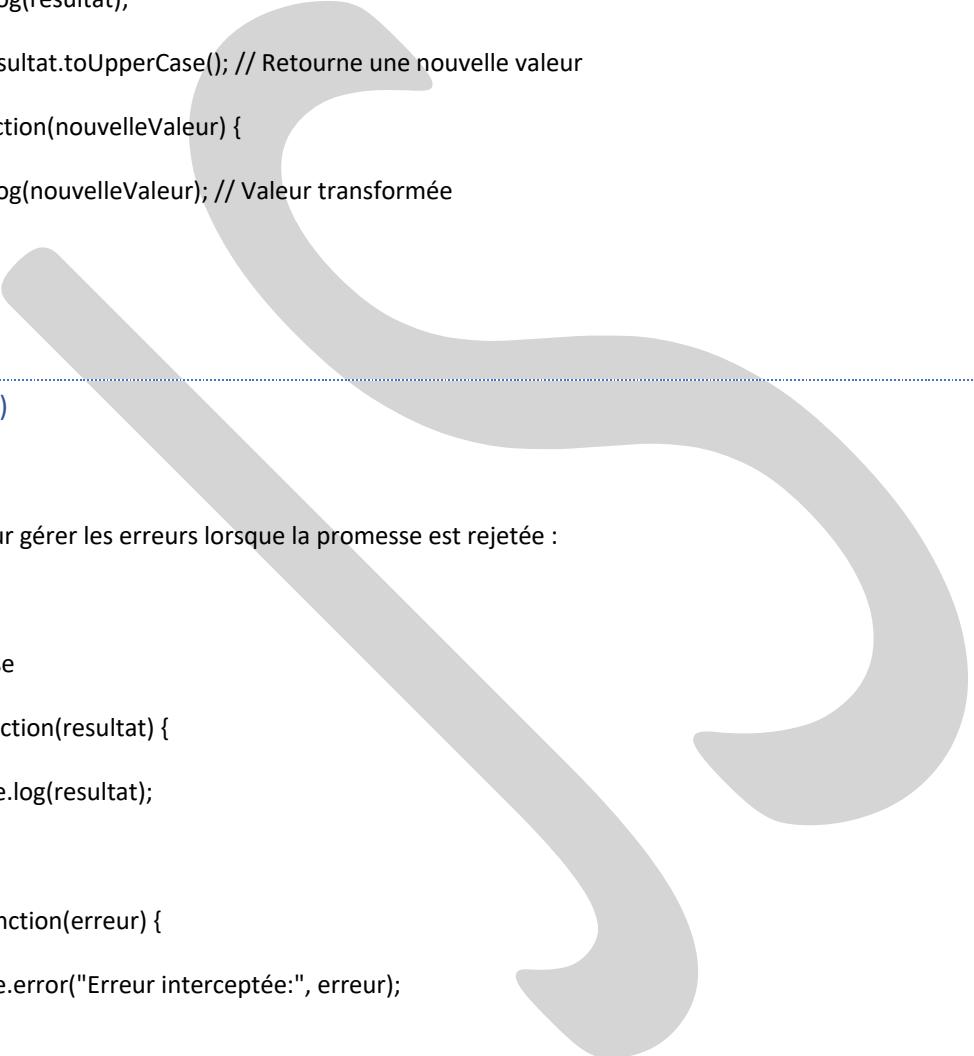
## MÉTHODES PRINCIPALES DES PROMESSES

---

### 1-THEN()

Utilisée pour spécifier ce qui se passe lorsque la promesse est tenue :

```
maPromesse.then(function(resultat) {
 console.log(resultat);
 return resultat.toUpperCase(); // Retourne une nouvelle valeur
}).then(function(nouvelleValeur) {
 console.log(nouvelleValeur); // Valeur transformée
});
```




---

### 2-CATCH()

Utilisée pour gérer les erreurs lorsque la promesse est rejetée :

```
maPromesse
 .then(function(resultat) {
 console.log(resultat);
 })
 .catch(function(erreur) {
 console.error("Erreur interceptée:", erreur);
 });

```




---

### 3-FINALLY()

Exécutée que la promesse soit tenue ou rejetée :

```
maPromesse
```

```
.then(function(resultat) {
 console.log(resultat);
})
.catch(function(erreur) {
 console.error(erreur);
})
.finally(function() {
 console.log("Nettoyage final, quoi qu'il arrive");
});
```

## CHAÎNAGE DES PROMESSES

L'un des principaux avantages des promesses est la capacité à enchaîner les opérations asynchrones de manière lisible :

```
getUtilisateur(idUtilisateur)
 .then(function(utilisateur) {
 return getCommandes(utilisateur.id);
 })
 .then(function(commandes) {
 return getDetailsCommande(commandes[0].id);
 })
 .then(function(details) {
 return getProduitsCommande(details.id);
 })
 .then(function(produits) {
 console.log(produits);
 })
```

```
.catch(function(erreur) {
 console.error("Une erreur s'est produite:", erreur);
});
```

Avantages :

- Flux de code : Le code suit une séquence logique de haut en bas
- Gestion d'erreurs centralisée : Un seul bloc catch peut gérer les erreurs de toute la chaîne
- Transparence : Chaque étape a un rôle clair et distinct

## MÉTHODES STATIQUES DES PROMESSES

### PROMISE.RESOLVE() ET PROMISE.REJECT()

Créent des promesses déjà résolues ou rejetées :

```
// Promesse immédiatement résolue
const promesseResolue = Promise.resolve("Déjà prêt");
```

```
// Promesse immédiatement rejetée
const promesseRejetee = Promise.reject(new Error("Échec immédiat"));
```

```
promesseResolue.then(console.log); // "Déjà prêt"
promesseRejetee.catch(console.error); // Error: Échec immédiat
```

### PROMISE.ALL()

Attend que toutes les promesses soient tenues ou qu'une seule soit rejetée :

```
const promesse1 = fetch('/api/utilisateurs');
```

```

const promesse2 = fetch('/api/commandes');

const promesse3 = fetch('/api/produits');

Promise.all([promesse1, promesse2, promesse3])

.then(function(resultats) {

 // resultats est un tableau contenant les résultats des 3 promesses

 const [utilisateurs, commandes, produits] = resultats;

 console.log(utilisateurs, commandes, produits);

})

.catch(function(erreur) {

 // S'exécute si l'une des promesses est rejetée

 console.error(erreur);

});

```

#### PROMISE.RACE()

Retourne la première promesse qui est soit tenue, soit rejetée :

```

const promesseRapide = new Promise(resolve => setTimeout(() => resolve("Rapide"), 100));

const promesseLente = new Promise(resolve => setTimeout(() => resolve("Lente"), 500));

```

```

Promise.race([promesseRapide, promesseLente])

.then(console.log); // "Rapide"

```

#### PROMISE.ALLSETTLED() (ES2020)

Attend que toutes les promesses soient réglées (tenues ou rejetées) :

```
const promesses = [
```

```

Promise.resolve(1),
Promise.reject(new Error("Erreur")),
Promise.resolve(3)

];

Promise.allSettled(promesses)
.then(resultats => {
 // resultats est un tableau d'objets décrivant l'état de chaque promesse
 resultats.forEach(resultat => {
 if (resultat.status === 'fulfilled') {
 console.log('Valeur:', resultat.value);
 } else {
 console.log('Raison du rejet:', resultat.reason);
 }
 });
});

```

#### PROMISE.ANY() (ES2021)

Retourne la première promesse qui est tenue, ou rejette avec un AggregateError si toutes sont rejetées :

```

const promesses = [
 Promise.reject(new Error("Erreur 1")),
 Promise.resolve("Succès"),
 Promise.reject(new Error("Erreur 2"))
];

```

```

Promise.any(promesses)
.then(console.log) // "Succès"

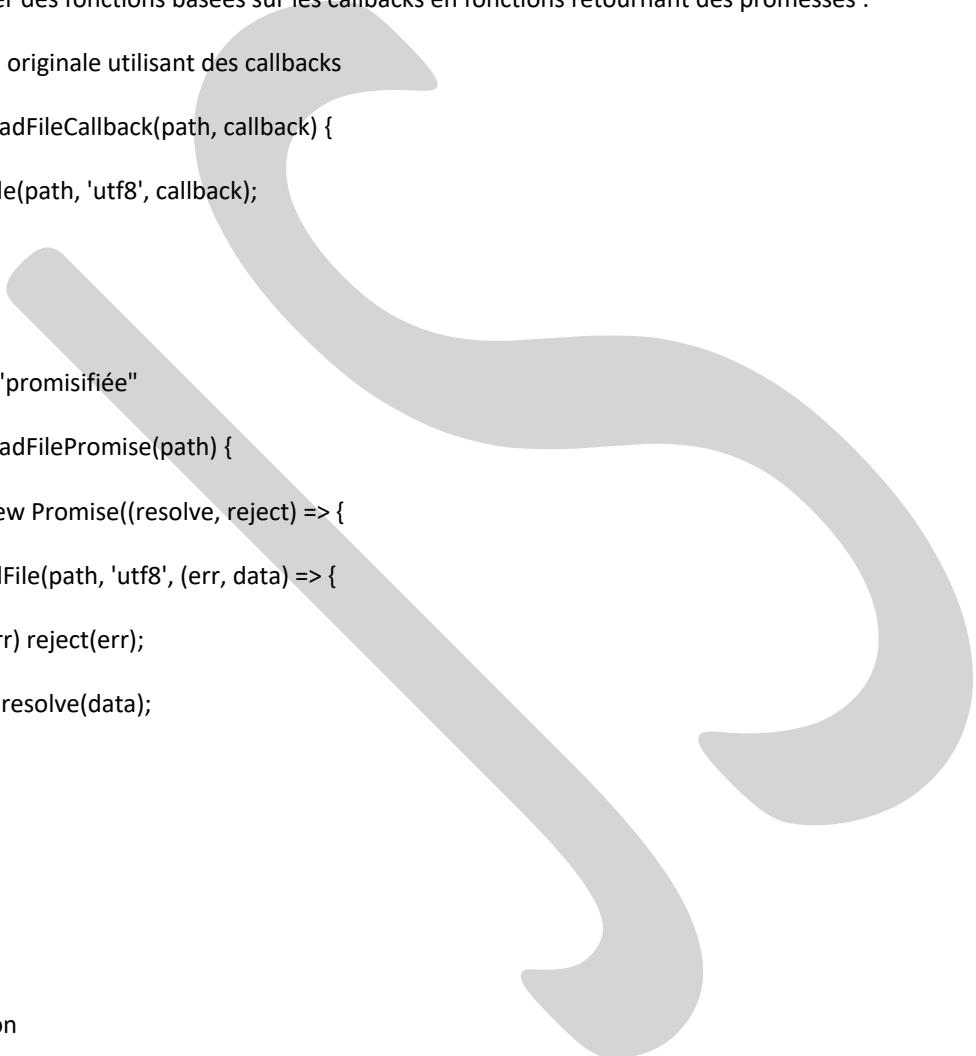
```

```
.catch(erreur => {
 console.error("Toutes les promesses ont été rejetées:", erreur);
});
```

## PROMISIFICATION : TRANSFORMER DES CALLBACKS EN PROMESSES

Transformer des fonctions basées sur les callbacks en fonctions retournant des promesses :

```
// Fonction originale utilisant des callbacks
function readFileCallback(path, callback) {
 fs.readFile(path, 'utf8', callback);
}
```



```
// Version "promisifiée"
function readFilePromise(path) {
 return new Promise((resolve, reject) => {
 fs.readFile(path, 'utf8', (err, data) => {
 if (err) reject(err);
 else resolve(data);
 });
 });
}
```

```
// Utilisation
readFilePromise('monFichier.txt')
 .then(data => console.log(data))
 .catch(err => console.error(err));
```

## FONCTION UTILITAIRE DE PROMISIFICATION

```

function promisify(fn) {
 return function(...args) {
 return new Promise((resolve, reject) => {
 fn(...args, (err, result) => {
 if (err) reject(err);
 else resolve(result);
 });
 });
 };
}

// Utilisation
const readFileSync = promisify(fs.readFileSync);
readFileSync('monFichier.txt', 'utf8')
 .then(data => console.log(data))
 .catch(err => console.error(err));

```

#### 4. ERREURS SYNCHRONES DANS LES HANDLERS DE PROMESSES

Les erreurs synchrones dans les fonctions `then()` ou `catch()` sont automatiquement capturées et transformées en rejets de promesse :

```

```javascript
Promise.resolve('donnees')
  .then(data => {
    // Cette erreur sera capturée par le .catch()
    throw new Error('Erreur de traitement');
  })
  .catch(error => {
    console.error(error.message);
  });

```

```

        return "Ceci ne sera jamais atteint";

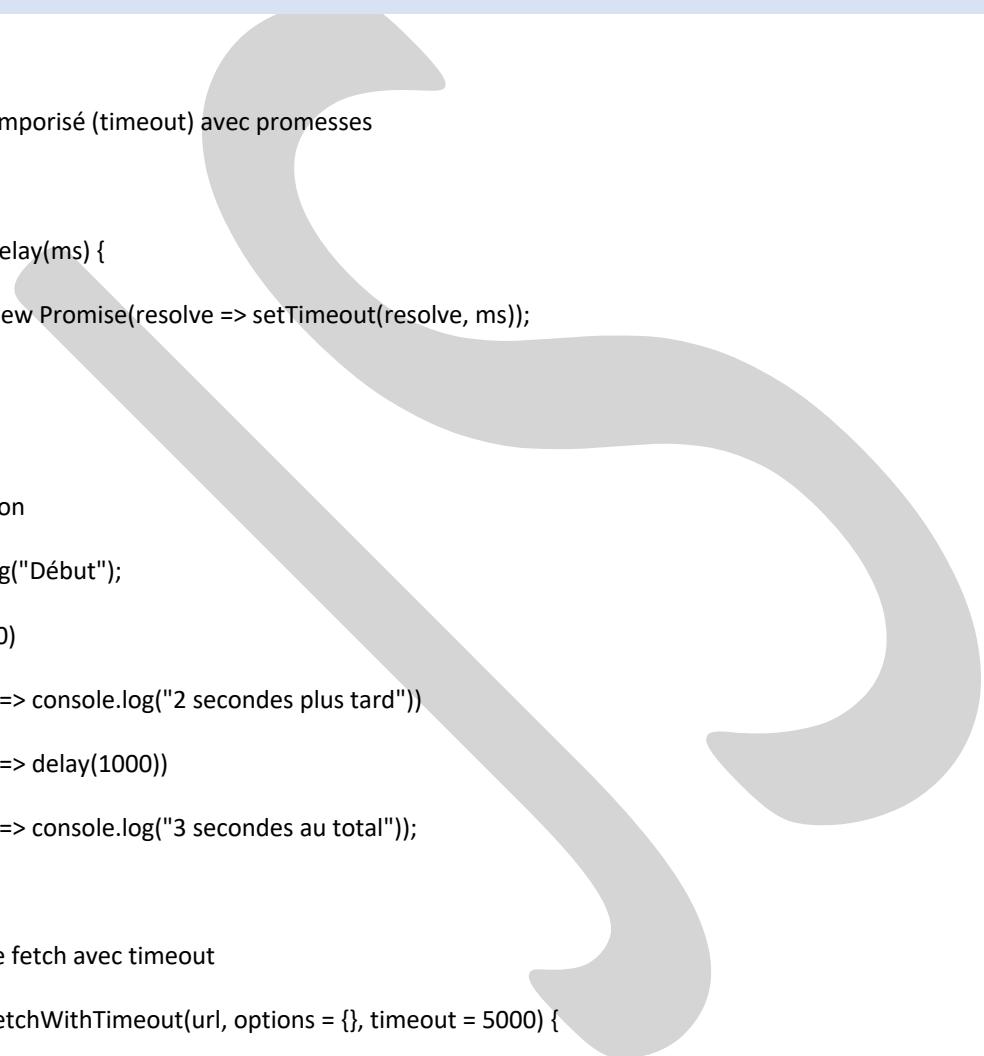
    })

    .catch(error => {
        console.error(error); // Error: Erreur de traitement
    });

```

APPLICATIONS PRATIQUES DES PROMESSES

1. Délai temporisé (timeout) avec promesses



```

function delay(ms) {

    return new Promise(resolve => setTimeout(resolve, ms));
}

```

```

// Utilisation

console.log("Début");

delay(2000)

    .then(() => console.log("2 secondes plus tard"))

    .then(() => delay(1000))

    .then(() => console.log("3 secondes au total"));

```

2. Requête fetch avec timeout

```

function fetchWithTimeout(url, options = {}, timeout = 5000) {

    return Promise.race([
        fetch(url, options),
        new Promise(_ , reject) =>
            setTimeout(() => reject(new Error('Timeout')), timeout)
    ])
};

```

```

}

fetchWithTimeout('https://api.example.com/data', {}, 3000)

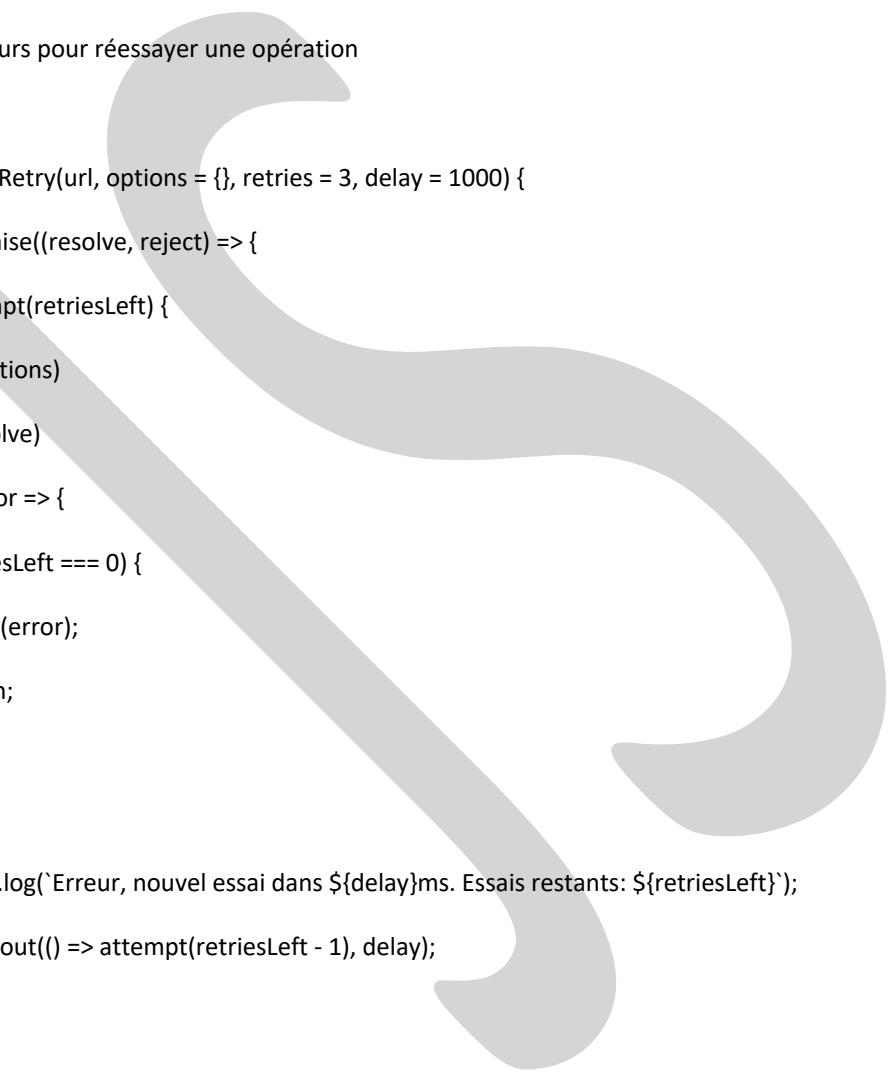
.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Erreur ou timeout:', error));

```

3. Rétention d'erreurs pour réessayer une opération



```

function fetchWithRetry(url, options = {}, retries = 3, delay = 1000) {

  return new Promise((resolve, reject) => {

    function attempt(retriesLeft) {
      fetch(url, options)
        .then(resolve)
        .catch(error => {
          if (retriesLeft === 0) {
            reject(error);
            return;
          }
          console.log(`Erreur, nouvel essai dans ${delay}ms. Essais restants: ${retriesLeft}`);
          setTimeout(() => attempt(retriesLeft - 1), delay);
        });
    }

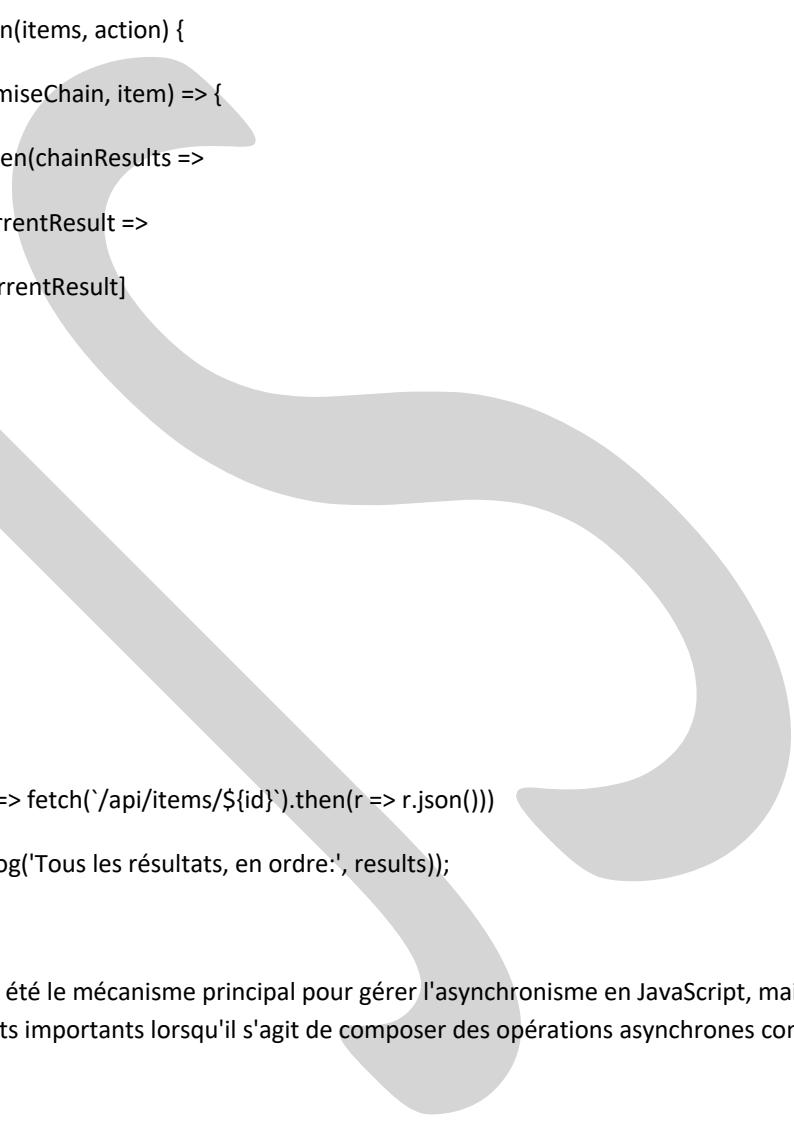
    attempt(retries);
  });
}

fetchWithRetry('https://api.example.com/data', {}, 3, 1000)

```

```
.then(response => response.json())
.then(data => console.log('Données récupérées:', data))
.catch(error => console.error('Échec après plusieurs tentatives:', error));
```

4. Exécution séquentielle de promesses sur un tableau



```
function sequentialExecution(items, action) {
  return items.reduce((promiseChain, item) => {
    return promiseChain.then(chainResults =>
      action(item).then(currentResult =>
        [...chainResults, currentResult]
      )
    );
  }, Promise.resolve([]));
}

// Utilisation
const ids = [1, 2, 3, 4, 5];
sequentialExecution(ids, id => fetch('/api/items/${id}').then(r => r.json()))
  .then(results => console.log('Tous les résultats, en ordre:', results));
```

Les callbacks ont longtemps été le mécanisme principal pour gérer l'asynchronisme en JavaScript, mais ils présentent des inconvénients importants lorsqu'il s'agit de composer des opérations asynchrones complexes.

Les promesses représentent une amélioration significative, offrant :

- Un flux de code plus linéaire et lisible
- Une meilleure gestion des erreurs
- Des mécanismes puissants pour composer des opérations asynchrones

Dans la prochaine section, nous explorerons comment les fonctionnalités `async/await` de JavaScript modernisent encore davantage la gestion de l'asynchronisme en s'appuyant sur les promesses tout en offrant une syntaxe encore plus propre et intuitive.

EXERCICES PRATIQUES

1. Exercice de base : Convertir une fonction utilisant des callbacks en une fonction retournant une promesse.

2. Exercice intermédiaire : Implémenter une fonction `delay` qui retourne une promesse se résolvant après un temps spécifié, puis l'utiliser pour créer une séquence de messages s'affichant à intervalles réguliers.

3. Exercice avancé : Créer une fonction `fetchWithTimeout` qui combine `fetch` avec un mécanisme de timeout, rejetant la promesse si la requête prend trop de temps.

4. Défi : Implémenter un mécanisme de cache pour les requêtes API utilisant les promesses, qui évite de refaire des requêtes identiques dans un intervalle de temps donné.

3.2.2. ASYNC/AWAIT

Async/await est une syntaxe introduite en ES2017 (ES8) qui permet d'écrire du code asynchrone comme s'il était synchrone. Cette fonctionnalité est construite sur les promesses et offre une méthode plus intuitive et lisible pour gérer les opérations asynchrones.

RELATION AVEC LES PROMESSES

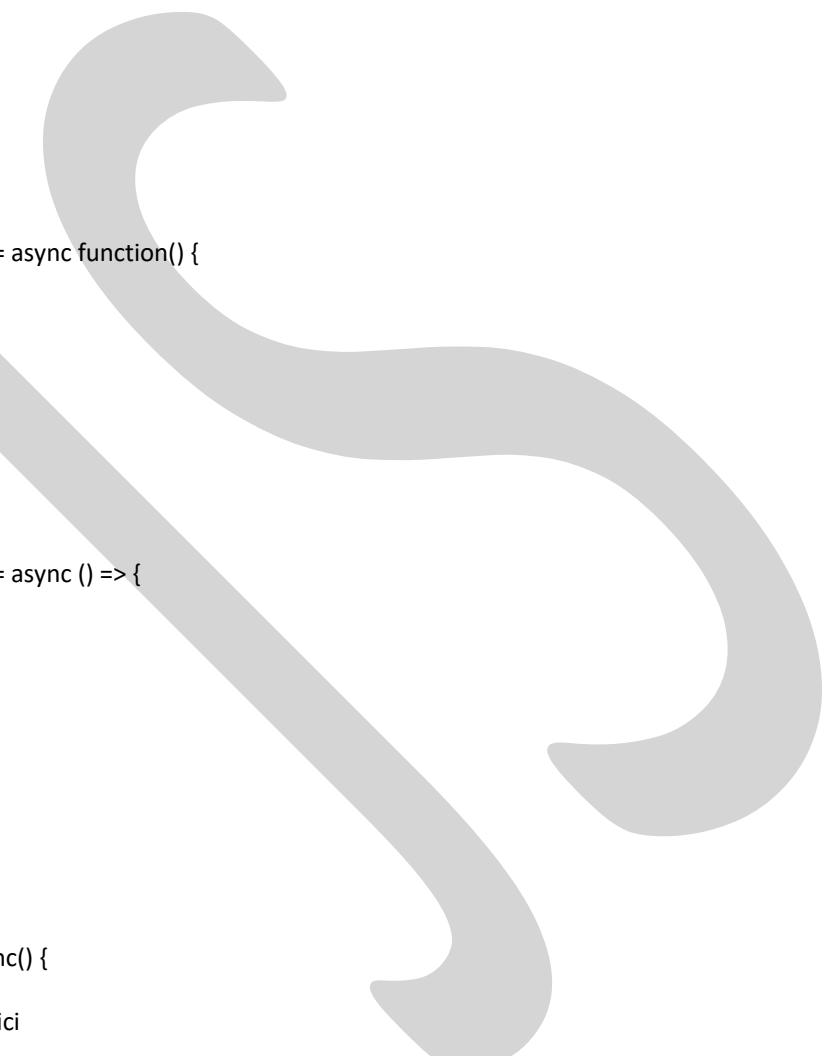
Async/await n'est pas un remplacement des promesses, mais plutôt une couche syntaxique construite au-dessus des promesses. Cela signifie que :

- ☞ Toute fonction `async` renvoie une promesse
- ☞ `Await` ne peut être utilisé qu'à l'intérieur des fonctions `async`
- ☞ `Await` ne fonctionne qu'avec les promesses (ou des objets avec une méthode `then` compatible)

LA DÉCLARATION ASYNC

Une fonction async est déclarée en ajoutant le mot-clé `async` avant la définition de la fonction :

```
// Déclaration de fonction  
async function maFonctionAsync() {  
    // Code asynchrone ici  
}
```



```
// Expression de fonction  
const maFonctionAsync = async function() {  
    // Code asynchrone ici  
};
```

```
// Fonction fléchée  
const maFonctionAsync = async () => {  
    // Code asynchrone ici  
};
```

```
// Méthode de classe  
class MaClasse {  
    async maMethodeAsync() {  
        // Code asynchrone ici  
    }  
}
```

VALEUR DE RETOUR

Une fonction async renvoie toujours une promesse. Si la fonction retourne une valeur explicitement, cette valeur sera encapsulée dans une promesse résolue. Si une exception est levée dans la fonction, la promesse sera rejetée avec cette exception.

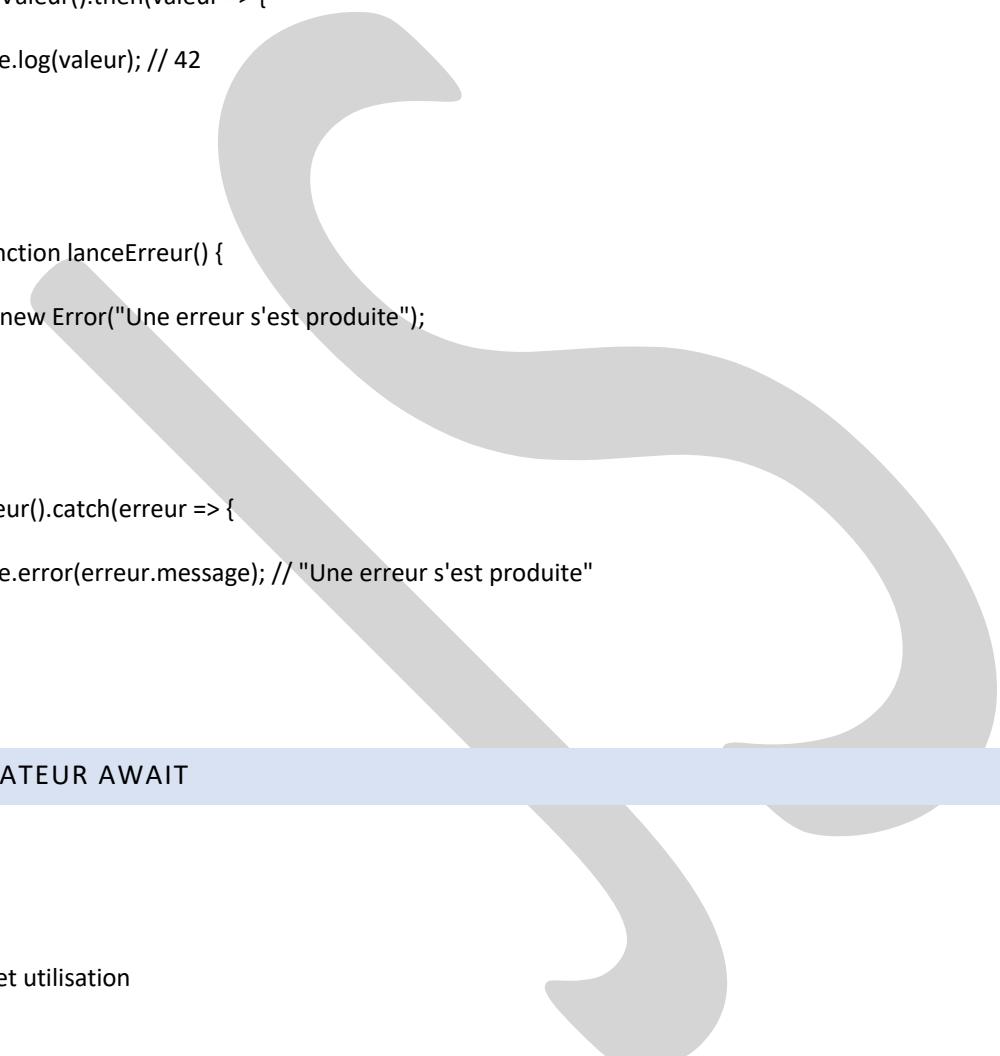
```
async function retourneValeur() {
    return 42;
}

retourneValeur().then(valeur => {
    console.log(valeur); // 42
});

async function lanceErreur() {
    throw new Error("Une erreur s'est produite");
}

lanceErreur().catch(erreur => {
    console.error(erreur.message); // "Une erreur s'est produite"
});
```

L'OPÉRATEUR AWAIT



Syntaxe et utilisation

L'opérateur await est utilisé pour attendre qu'une promesse soit résolue ou rejetée, et il ne peut être utilisé qu'à l'intérieur d'une fonction async :

```
async function exempleAwait() {
    const resultat = await maPromesse;
    console.log(resultat);
```

```
}
```

L'exécution de la fonction est "suspendue" à la ligne contenant await jusqu'à ce que la promesse soit résolue ou rejetée. Pendant cette suspension, le thread principal reste libre d'exécuter d'autres tâches.

Gestion des valeurs résolues

Lorsqu'une promesse est résolue, await renvoie la valeur avec laquelle elle a été résolue :

```
async function exempleValeurResolue() {
    // Ces deux lignes font la même chose
    const resultat1 = await Promise.resolve("Bonjour");
    const resultat2 = await "Bonjour"; // Conversion automatique en promesse

    console.log(resultat1, resultat2); // "Bonjour" "Bonjour"
}
```

Gestion des erreurs

Quand une promesse est rejetée, await "jette" la raison de rejet comme une exception, qui peut être capturée avec try/catch :

```
async function exempleGestionErreur() {
    try {
        const resultat = await Promise.reject(new Error("Échec"));

        console.log(resultat); // Cette ligne ne sera jamais atteinte
    } catch (error) {
        console.error("Erreur capturée:", error.message); // "Erreur capturée: Échec"
    }
}
```

```
}
```

COMPARAISON : PROMESSES VS ASYNC/AWAIT

La même logique de code, écrite avec des promesses puis avec `async/await` :

AVEC PROMESSES

```
function obtenirDonnees(id) {
    return fetch(`https://api.example.com/data/${id}`)
        .then(response => {
            if (!response.ok) {
                throw new Error(`Erreur HTTP ${response.status}`);
            }
            return response.json();
        })
        .then(data => {
            return traiterDonnees(data);
        })
        .then(resultat => {
            console.log("Résultat final:", resultat);
            return resultat;
        })
        .catch(error => {
            console.error("Erreur lors de l'opération:", error);
            throw error;
        });
}
```

AVEC ASYNC/AWAIT

```
async function obtenirDonnees(id) {
  try {
    const response = await fetch(`https://api.example.com/data/${id}`);

    if (!response.ok) {
      throw new Error(`Erreur HTTP ${response.status}`);
    }

    const data = await response.json();
    const resultat = await traiterDonnees(data);

    console.log("Résultat final:", resultat);
    return resultat;
  } catch (error) {
    console.error("Erreur lors de l'opération:", error);
    throw error;
  }
}
```

AVANTAGES D'ASYNC/AWAIT

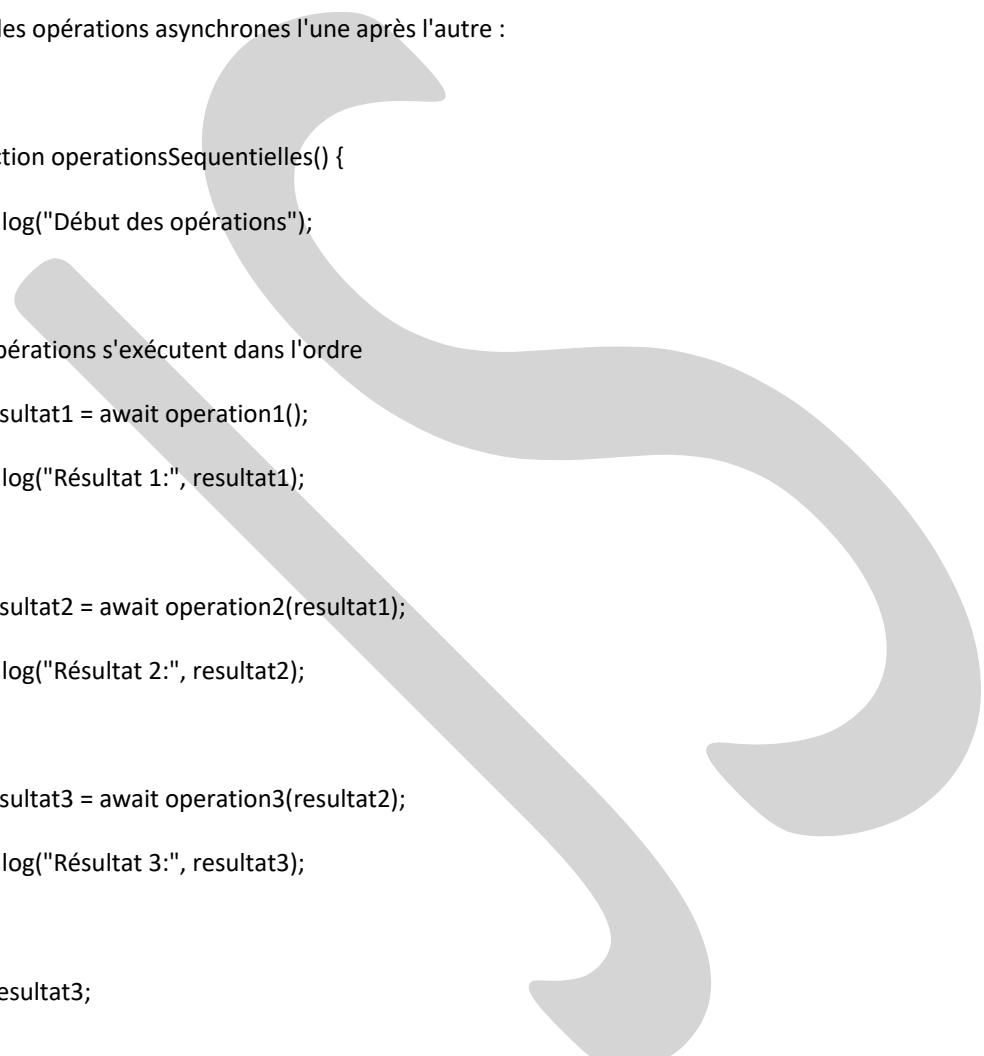
1. Lisibilité : Le code se lit comme du code synchrone, ce qui est plus intuitif
2. Débogage : Les stack traces sont plus claires et les points d'arrêt fonctionnent mieux
3. Gestion d'erreurs : Le mécanisme standard try/catch est plus familier
4. Variables dans la portée : Évite les problèmes de portée des variables entre les callbacks .then()

5. Combinaison d'opérations : Facilite le mélange de code synchrone et asynchrone

MODÈLES D'UTILISATION D'ASYNC/AWAIT

EXÉCUTION SÉQUENTIELLE

Exécuter des opérations asynchrones l'une après l'autre :



```
async function operationsSequentielles() {  
  console.log("Début des opérations");  
  
  // Les opérations s'exécutent dans l'ordre  
  const resultat1 = await operation1();  
  console.log("Résultat 1:", resultat1);  
  
  const resultat2 = await operation2(resultat1);  
  console.log("Résultat 2:", resultat2);  
  
  const resultat3 = await operation3(resultat2);  
  console.log("Résultat 3:", resultat3);  
  
  return resultat3;  
}
```

EXÉCUTION PARALLÈLE

Exécuter plusieurs opérations asynchrones en même temps :

```
async function operationsParalleles() {  
    console.log("Début des opérations parallèles");  
  
    // Démarrer toutes les opérations sans attendre  
    const promesse1 = operation1();  
    const promesse2 = operation2();  
    const promesse3 = operation3();  
  
    // Attendre les résultats  
    const resultat1 = await promesse1;  
    const resultat2 = await promesse2;  
    const resultat3 = await promesse3;  
  
    return [resultat1, resultat2, resultat3];  
}  
  
// Ou plus simplement avec Promise.all  
async function operationsParallelesAvecPromiseAll() {  
    console.log("Début des opérations parallèles");  
  
    const resultats = await Promise.all([  
        operation1(),  
        operation2(),  
        operation3()  
    ]);  
  
    return resultats;  
}
```

ATTENTE CONDITIONNELLE

Attendre conditionnellement une promesse en fonction de certains critères :

```
async function attenteConditionnelle(condition) {
    let resultat;

    if (condition) {
        // Attendre seulement si la condition est vraie
        resultat = await operationLongue();
    } else {
        resultat = valeurParDefaut;
    }

    return resultat;
}
```

BOUCLE AVEC AWAIT

Traiter une série d'éléments de manière asynchrone :

```
async function traiterElements(elements) {
    const resultats = [];

    // Traitement séquentiel (un après l'autre)
    for (const element of elements) {
        const resultat = await traiterElement(element);
        resultats.push(resultat);
    }
}
```

```

    return resultats;

}

async function traiterElementsEnParallele(elements) {
    // Transformation en promesses

    const promesses = elements.map(element => traiterElement(element));

    // Attendre que toutes les promesses soient résolues

    const resultats = await Promise.all(promesses);

    return resultats;
}

```

PIÈGES COMMUNS ET SOLUTIONS

1. Oublier d'utiliser await

Un piège fréquent est d'appeler une fonction `async` sans utiliser `await` :

```

// Problème : la promesse n'est pas attendue

async function probleme() {

    const data = getData(); // getData() renvoie une promesse, mais nous ne l'attendons pas

    console.log(data); // Affiche un objet Promise, pas les données

}

```

```

// Solution

async function solution() {

    const data = await getData(); // Attendre que la promesse soit résolue

```

```
    console.log(data); // Affiche les données
}
```

2. await hors d'une fonction async

L'opérateur await ne peut être utilisé qu'à l'intérieur d'une fonction async :

```
// Erreur de syntaxe
const data = await getData(); // SyntaxError: await n'est pas valide ici
```

// Solution 1: Envelopper dans une fonction async

```
async function getData() {
  const data = await fetchData();
  return data;
}
```

// Solution 2: Utiliser une IIFE async

```
(async () => {
  const data = await getData();
  console.log(data);
})();
```

3. Gestion des erreurs oubliée

Ne pas gérer les erreurs avec try/catch :

```
// Problème : pas de gestion d'erreur
async function probleme() {
  const data = await riskyOperation(); // Si ça échoue, l'erreur n'est pas gérée
```

```

    return processData(data);
}

// Solution

async function solution() {
  try {
    const data = await riskyOperation();
    return processData(data);
  } catch (error) {
    console.error("Erreur lors de l'opération:", error);
    throw error; // Ou gérer l'erreur d'une autre manière
  }
}

```

4. Blocage des opérations parallèles

Utiliser await dans une boucle lorsque les opérations pourraient être exécutées en parallèle :

```

// Problème : exécution séquentielle inutile

async function probleme(ids) {
  const resultats = [];
  for (const id of ids) {
    const data = await fetchData(id); // Attend chaque requête une par une
    resultats.push(data);
  }
  return resultats;
}

// Solution : exécution parallèle

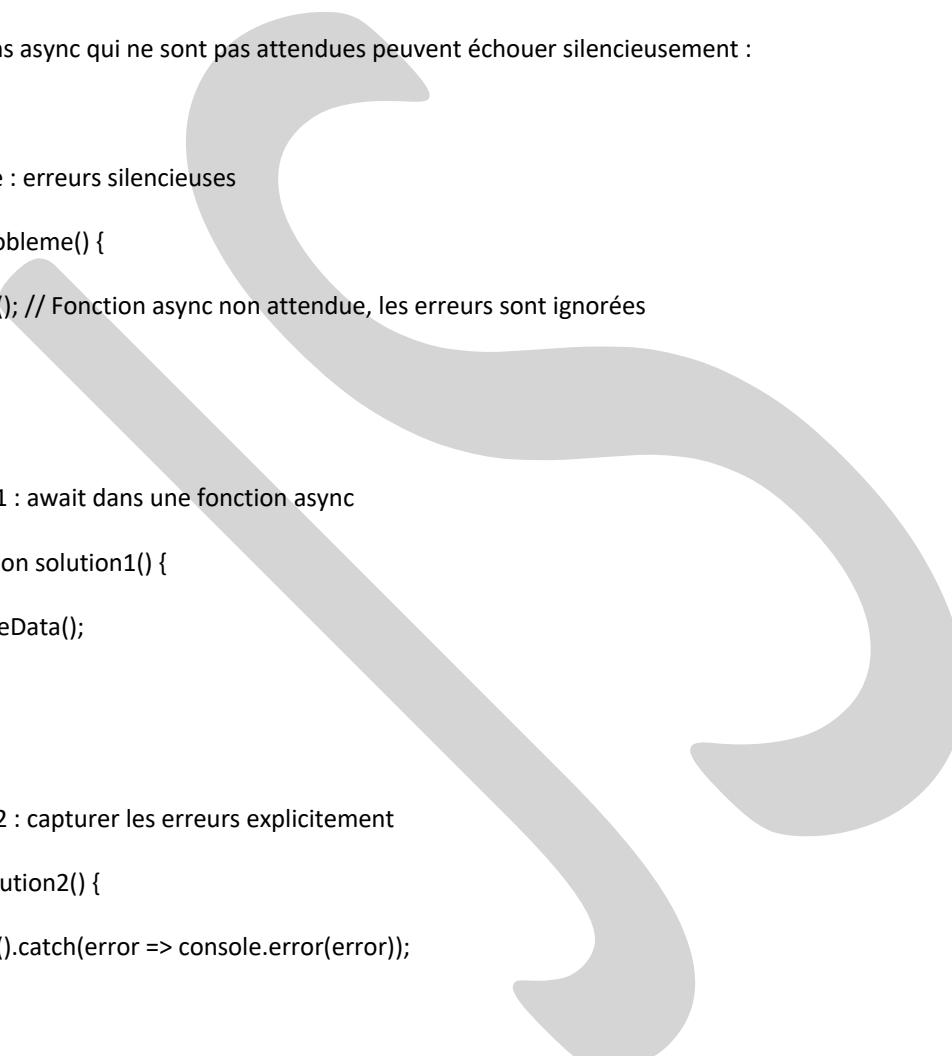
```

```
async function solution(ids) {
  const promesses = ids.map(id => fetchData(id));
  return Promise.all(promesses);
}
```

5. Erreurs silencieuses

Les fonctions async qui ne sont pas attendues peuvent échouer silencieusement :

```
// Problème : erreurs silencieuses
function probleme() {
  saveData(); // Fonction async non attendue, les erreurs sont ignorées
}
```



```
// Solution 1 : await dans une fonction async
async function solution1() {
  await saveData();
}
```

```
// Solution 2 : capturer les erreurs explicitement
function solution2() {
  saveData().catch(error => console.error(error));
}
```

TECHNIQUES AVANCÉES AVEC ASYNC/AWAIT

1. Timeout pour les opérations asynchrones

Ajouter un timeout à une opération asynchrone :

```

async function fetchWithTimeout(url, timeout = 5000) {

    const controller = new AbortController();

    const id = setTimeout(() => controller.abort(), timeout);

    try {

        const response = await fetch(url, { signal: controller.signal });

        clearTimeout(id);

        return response.json();

    } catch (error) {

        clearTimeout(id);

        if (error.name === 'AbortError') {

            throw new Error(`Requête expirée après ${timeout}ms`);

        }

        throw error;

    }

}

```

2. Retrying (réessayer) les opérations échouées

Implémenter un mécanisme de retry pour les opérations qui peuvent échouer temporairement :

```

async function fetchWithRetry(url, options = {}, retries = 3, backoff = 300) {

    let lastError;

    for (let i = 0; i < retries; i++) {

        try {

            return await fetch(url, options);

        } catch (error) {

```

```

        console.log(`Tentative ${i + 1} échouée. Nouvel essai dans ${backoff}ms...`);

        lastError = error;

        // Attendre avant de réessayer (backoff exponentiel)

        await new Promise(resolve => setTimeout(resolve, backoff));

        backoff *= 2; // Doubler le temps d'attente pour chaque essai

    }

}

throw lastError;
}

```

3. Pattern décorateur avec async/await

Créer des décorateurs pour les fonctions async :

```

function withLogging(asyncFn) {
    return async function(...args) {
        console.log(`Appel de ${asyncFn.name} avec args:`, args);
        try {
            const result = await asyncFn(...args);
            console.log(`${asyncFn.name} a réussi:`, result);
            return result;
        } catch (error) {
            console.error(`${asyncFn.name} a échoué:`, error);
            throw error;
        }
    };
}

```

```
// Utilisation

async function fetchUser(id) {
  const response = await fetch(`/api/users/${id}`);
  return response.json();
}
```

```
const fetchUserWithLogging = withLogging(fetchUser);
const user = await fetchUserWithLogging(123);
```

4. Annulation des opérations asynchrones

L'API AbortController permet d'annuler des opérations fetch en cours :

```
async function fetchCancellable(url) {
  const controller = new AbortController();
  const { signal } = controller;

  // Exposer une méthode pour annuler la requête
  const cancel = () => controller.abort();

  // Créer la promesse
  const promise = (async () => {
    try {
      const response = await fetch(url, { signal });
      return await response.json();
    } catch (error) {
      if (error.name === 'AbortError') {
        return { cancelled: true };
      }
    }
  })();
}
```

```

        }

        throw error;

    }

})();

// Retourner à la fois la promesse et la fonction d'annulation

return { promise, cancel };

}

// Utilisation

const { promise, cancel } = fetchCancellable('/api/data');

// Pour annuler après un certain temps

setTimeout(cancel, 2000); // Annule après 2 secondes

try {

    const result = await promise;

    if (result.cancelled) {

        console.log('La requête a été annulée');

    } else {

        console.log('Données reçues:', result);

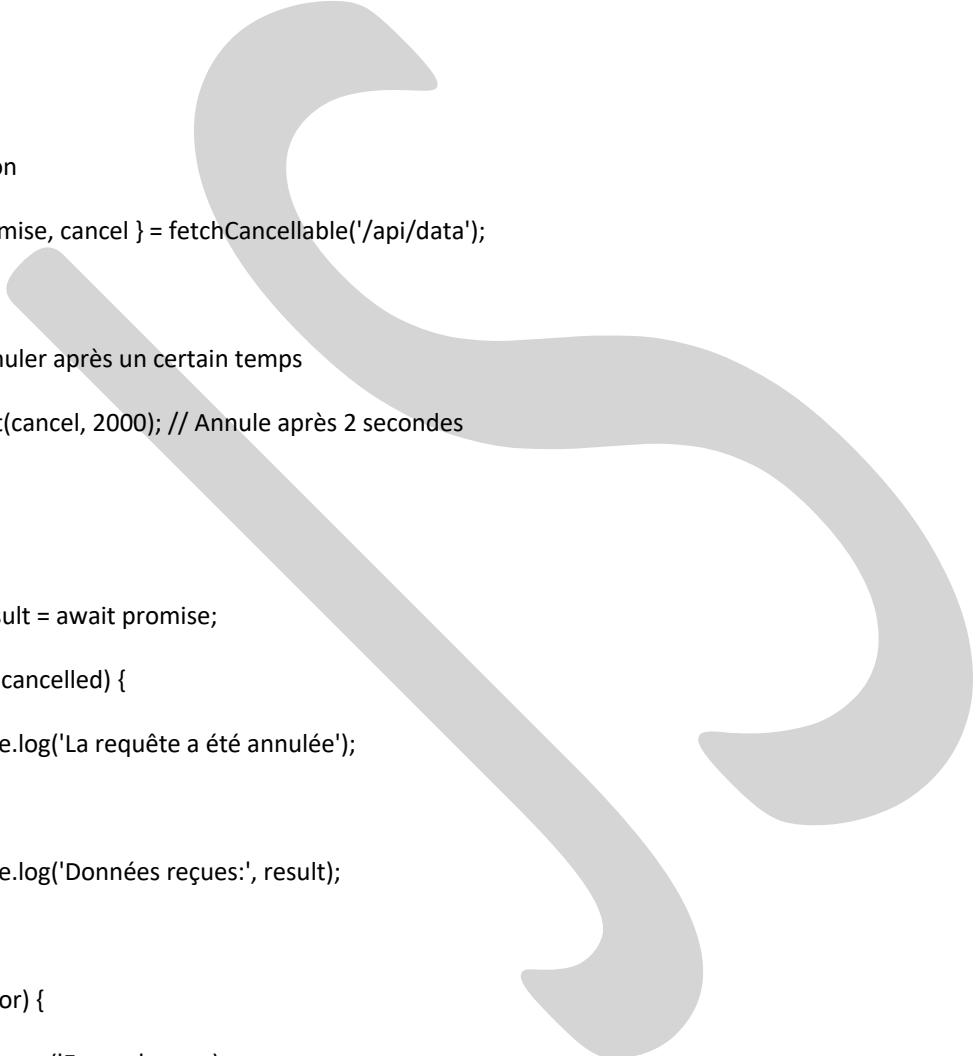
    }

} catch (error) {

    console.error('Erreur:', error);

}

```



ASYNC/AWAIT ET L'API FETCH

L'API fetch est souvent utilisée avec async/await pour réaliser des requêtes HTTP :

```
async function fetchAPI(url, options = {}) {  
  try {  
    const response = await fetch(url, options);  
  
    if (!response.ok) {  
      throw new Error(`Erreur HTTP: ${response.status}`);  
    }  
  
    return await response.json();  
  } catch (error) {  
    console.error('Erreur de fetch:', error);  
    throw error;  
  }  
}  
  
// GET request  
async function getData(url) {  
  return fetchAPI(url);  
}  
  
// POST request  
async function postData(url, data) {  
  return fetchAPI(url, {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json'  
    },  
    body: JSON.stringify(data)  
  });  
}
```

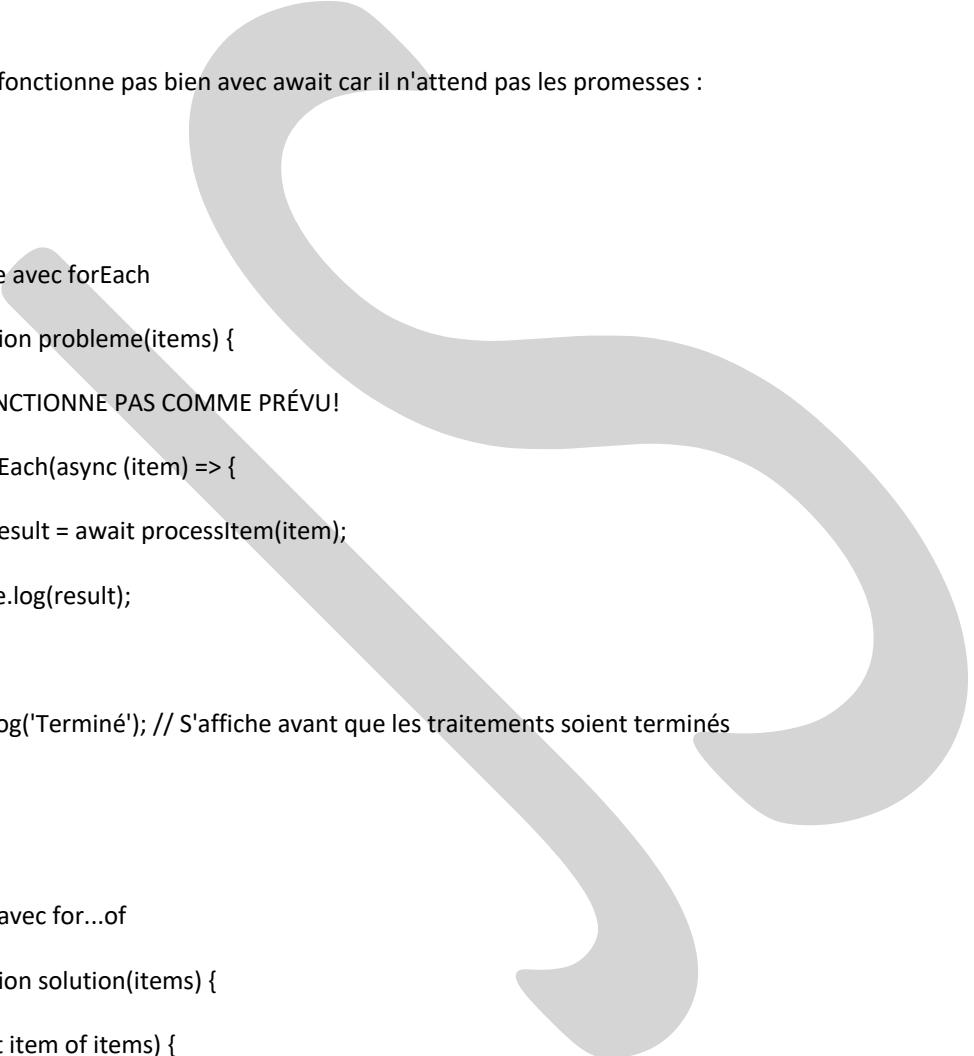
```
});  
}
```

ASYNC/AWAIT DANS LES BOUCLES ET MAPS

FOREACH VS FOR...OF

forEach ne fonctionne pas bien avec await car il n'attend pas les promesses :

```
// Problème avec forEach  
  
async function probleme(items) {  
    // NE FONCTIONNE PAS COMME PRÉVU!  
  
    items.forEach(async (item) => {  
  
        const result = await processItem(item);  
  
        console.log(result);  
  
    });  
  
    console.log('Terminé'); // S'affiche avant que les traitements soient terminés  
  
}
```



```
// Solution avec for...of  
  
async function solution(items) {  
  
    for (const item of items) {  
  
        const result = await processItem(item);  
  
        console.log(result);  
  
    }  
  
    console.log('Terminé'); // S'affiche après que tous les traitements soient terminés  
  
}
```

MAP AVEC ASYNC/AWAIT

Utiliser map avec des fonctions async :

```
// Génère un tableau de promesses (pas de valeurs)
const promesses = [1, 2, 3].map(async (num) => {
    const result = await processItem(num);
    return result;
});
```

```
// Pour obtenir les résultats, il faut attendre toutes les promesses
const resultats = await Promise.all(promesses);
```

ASYNC/AWAIT AU NIVEAU DU MODULE (TOP-LEVEL AWAIT)

Depuis ES2022, await peut être utilisé au niveau du module sans être dans une fonction async :

```
// Dans un module ES (avec type="module" dans le HTML ou .mjs)
// Avant ES2022, ceci n'était pas possible
const response = await fetch('/api/data');
const data = await response.json();

console.log(data);
```

```
// Exporter les données chargées de manière asynchrone
export { data };
```

Cela permet de charger des données de manière asynchrone avant que le module ne soit utilisé.

Async/await représente une évolution majeure dans la façon dont JavaScript gère l'asynchronisme. Cette syntaxe offre une approche plus intuitive et lisible pour travailler avec les opérations asynchrones, tout en s'appuyant sur la puissance des promesses.

Les avantages principaux sont :

- ☞ Un code plus lisible qui ressemble à du code synchrone
- ☞ Une meilleure gestion des erreurs grâce à try/catch
- ☞ Une structuration du code plus claire et plus maintenable
- ☞ Des opérations asynchrones plus faciles à raisonner et à déboguer

Bien que async/await simplifie considérablement le code asynchrone, il est important de comprendre les promesses qui les sous-tendent et d'être conscient des pièges courants.

Avec la maîtrise de async/await, vous pouvez écrire un code asynchrone élégant et efficace qui reste lisible et maintenable, même pour des flux de travail asynchrones complexes.

EXERCICES PRATIQUES

1. Exercice de base : Convertir une fonction utilisant des promesses en une fonction utilisant async/await.
2. Exercice intermédiaire : Créer une fonction qui récupère des données utilisateur à partir d'une API, puis utilise ces données pour faire une autre requête à une API différente.
3. Exercice avancé : Implémenter une fonction qui exécute plusieurs requêtes en parallèle, mais avec une limite sur le nombre de requêtes simultanées.
4. Défi : Créer une fonction générique de "retry" qui réessaie une fonction asynchrone un certain nombre de fois, avec un délai croissant entre les tentatives, et qui gère correctement les erreurs.

3.3.1. ÉVÉNEMENTS JAVASCRIPT

Les événements sont des actions ou des occurrences qui se produisent dans le système que vous programmez et auxquelles le système peut réagir. En JavaScript, particulièrement dans le contexte du navigateur, les événements sont une partie fondamentale de l'interaction avec les utilisateurs, permettant de créer des expériences web dynamiques et réactives.

Qu'est-ce qu'un événement ?

Un événement peut être déclenché par :

- ☞ Les actions de l'utilisateur (clic, survol, frappe au clavier, etc.)
- ☞ Les changements dans le navigateur (chargement de page, redimensionnement)
- ☞ Les modifications dans le DOM (changements de contenu, d'attributs)
- ☞ Les communications réseau (réception de données)
- ☞ Les minuteries (timeouts, intervalles)

MODÈLE D'ÉVÉNEMENTS JAVASCRIPT

Le modèle d'événements en JavaScript fonctionne sur trois concepts principaux :

- ☞ Événements : Les actions qui se produisent
- ☞ Écouteurs d'événements (event listeners) : Les fonctions qui "écoutent" et réagissent aux événements
- ☞ Gestionnaires d'événements (event handlers) : Les fonctions exécutées lorsqu'un événement est détecté

TYPES D'ÉVÉNEMENTS COURANTS

ÉVÉNEMENTS DU DOCUMENT/FENÊTRE

```
// Chargement du document

window.addEventListener('load', function() {
    console.log('La page est entièrement chargée');

});
```

```
// Lorsque le DOM est prêt (avant les images/CSS)
document.addEventListener('DOMContentLoaded', function() {
    console.log('DOM chargé, mais pas forcément les ressources externes');
});

// Redimensionnement de la fenêtre
window.addEventListener('resize', function() {
    console.log('Fenêtre redimensionnée');
});

// Défilement (scroll)
window.addEventListener('scroll', function() {
    console.log('Défilement de la page');
});
```

ÉVÉNEMENTS DE SOURIS

```
const element = document.getElementById('monElement');

// Clic
element.addEventListener('click', function() {
    console.log('Élément cliqué');
});

// Double clic
element.addEventListener('dblclick', function() {
    console.log('Double clic sur l\'élément');
});

// Survol (entrée)
```

```

element.addEventListener('mouseenter', function() {
    console.log('Souris entre dans l\'élément');
});

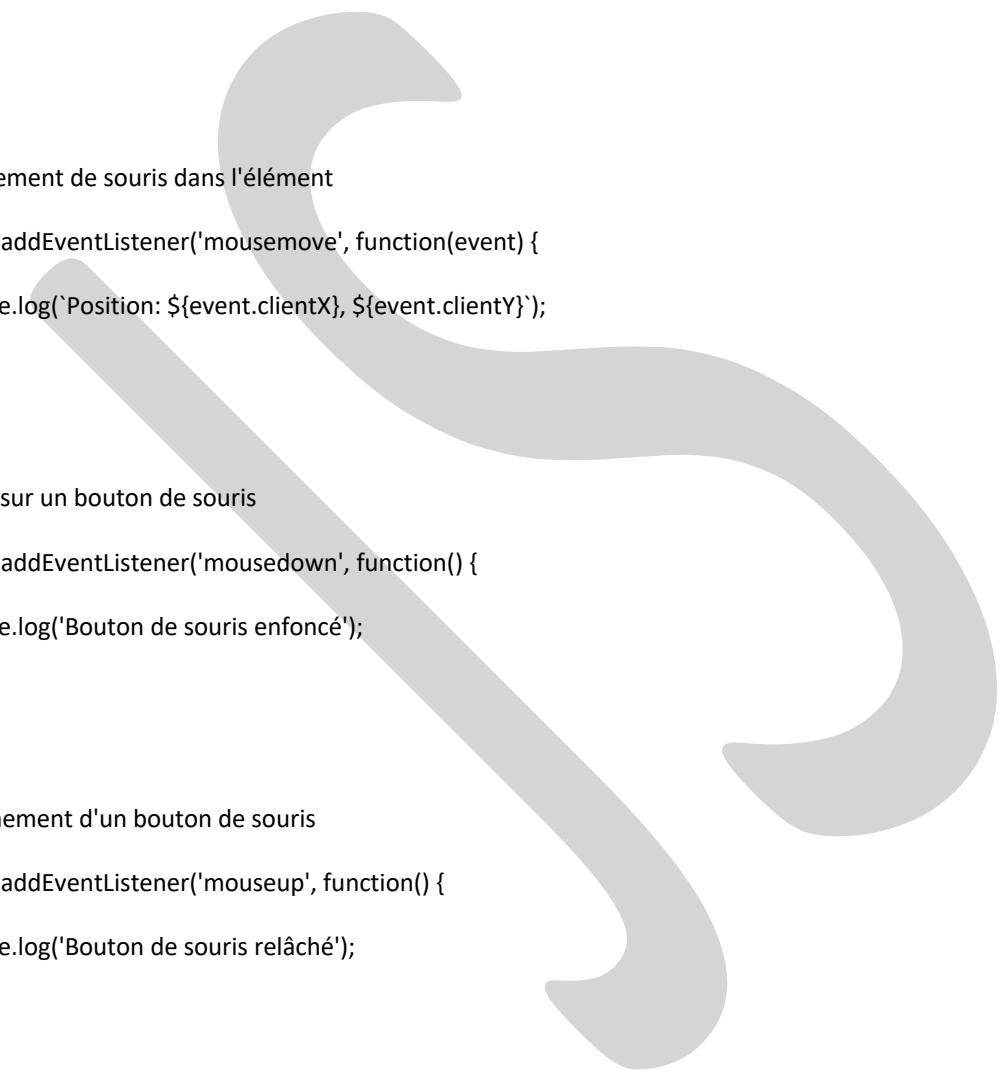
// Survol (sortie)
element.addEventListener('mouseleave', function() {
    console.log('Souris quitte l\'élément');
});

// Mouvement de souris dans l'élément
element.addEventListener('mousemove', function(event) {
    console.log(`Position: ${event.clientX}, ${event.clientY}`);
});

// Appui sur un bouton de souris
element.addEventListener('mousedown', function() {
    console.log('Bouton de souris enfoncé');
});

// Relâchement d'un bouton de souris
element.addEventListener('mouseup', function() {
    console.log('Bouton de souris relâché');
});

```



ÉVÉNEMENTS DE CLAVIER

```

// Appui sur une touche
document.addEventListener('keydown', function(event) {
    console.log(`Touche enfoncée: ${event.key}`);
});

```

```
});

// Relâchement d'une touche

document.addEventListener('keyup', function(event) {

    console.log(`Touche relâchée: ${event.key}`);

});


```

```
// Touche pressée (combinaison de keydown et keyup)

document.addEventListener('keypress', function(event) {

    console.log(`Caractère saisi: ${event.key}`);

});
```

ÉVÉNEMENTS DE FORMULAIRE

```
const formulaire = document.getElementById('monFormulaire');

const champTexte = document.getElementById('champTexte');
```

```
// Soumission du formulaire

formulaire.addEventListener('submit', function(event) {

    event.preventDefault(); // Empêche la soumission standard

    console.log('Formulaire soumis');

});
```

```
// Changement de valeur (quand l'élément perd le focus)

champTexte.addEventListener('change', function() {

    console.log('Valeur changée');

});
```

```
// Saisie en temps réel
```

```
champTexte.addEventListener('input', function() {
    console.log('Saisie en cours:', this.value);
});
```

// Focus sur un élément

```
champTexte.addEventListener('focus', function() {
    console.log('Champ sélectionné');
});
```

// Perte de focus

```
champTexte.addEventListener('blur', function() {
    console.log('Champ désélectionné');
});
```

ÉVÉNEMENTS TACTILES

```
const elementTactile = document.getElementById('elementTactile');
```

// Début du toucher

```
elementTactile.addEventListener('touchstart', function(event) {
    console.log('Toucher commencé');
});
```

// Déplacement pendant le toucher

```
elementTactile.addEventListener('touchmove', function(event) {
    console.log('Toucher déplacé');
});
```

// Fin du toucher

```

elementTactile.addEventListener('touchend', function(event) {
    console.log('Toucher terminé');

});

// Annulation du toucher

elementTactile.addEventListener('touchcancel', function(event) {
    console.log('Toucher annulé');

});

```

GESTION DES ÉVÉNEMENTS

MÉTHODES D'ATTACHEMENT D'ÉVÉNEMENTS

Il existe trois principales façons d'attacher des gestionnaires d'événements en JavaScript :

1. ATTRIBUTS HTML (DÉCONSEILLÉ)

```
<button onclick="console.log('Cliqué')">Cliquez-moi</button>
```

2. PROPRIÉTÉS D'ÉVÉNEMENT DU DOM

```

const bouton = document.getElementById('monBouton');

bouton.onclick = function() {
    console.log('Bouton cliqué');
};

```

Inconvénient : On ne peut assigner qu'un seul gestionnaire par type d'événement et par élément.

3. ADDEVENTLISTENER (MÉTHODE RECOMMANDÉE)

```
const bouton = document.getElementById('monBouton');

bouton.addEventListener('click', function() {
    console.log('Premier gestionnaire');
});

bouton.addEventListener('click', function() {
    console.log('Deuxième gestionnaire');
});
```

Avantages :

- Plusieurs gestionnaires pour le même événement
- Plus grand contrôle grâce aux options
- Séparation claire entre HTML et JavaScript

DÉTACHEMENT DES GESTIONNAIRES D'ÉVÉNEMENTS

Pour retirer un gestionnaire, il faut utiliser `removeEventListener` avec exactement la même fonction :

```
function gestionnaire() {
    console.log('Gestionnaire exécuté');
}

const bouton = document.getElementById('monBouton');

bouton.addEventListener('click', gestionnaire);

// Plus tard, pour retirer le gestionnaire

bouton.removeEventListener('click', gestionnaire);
```

Remarque : Les fonctions anonymes ne peuvent pas être détachées directement :

```
// Problème : impossible de retirer ce gestionnaire
bouton.addEventListener('click', function() {
    console.log('Clic');
});
```

```
// Solution : utiliser une fonction nommée ou stockée dans une variable
const fonctionGestionnaire = function() {
    console.log('Clic');
};

bouton.addEventListener('click', fonctionGestionnaire);

// Plus tard

bouton.removeEventListener('click', fonctionGestionnaire);
```

L'OBJET EVENT

Chaque gestionnaire d'événement reçoit un objet Event qui contient des informations sur l'événement déclenché.

```
document.addEventListener('click', function(event) {
    console.log('Type d\'événement:', event.type); // 'click'
    console.log('Élément cible:', event.target); // L'élément qui a déclenché l'événement
    console.log('Position X:', event.clientX); // Position horizontale du clic
    console.log('Position Y:', event.clientY); // Position verticale du clic

    // Arrêter la propagation de l'événement
    event.stopPropagation();
```

```
// Empêcher le comportement par défaut
event.preventDefault();
});

});
```

PROPRIÉTÉS COMMUNES DE L'OBJET EVENT

Propriété	Description
type	Type de l'événement (ex: 'click', 'keydown')
target	Élément qui a déclenché l'événement
currentTarget	Élément auquel le gestionnaire est attaché
timeStamp	Moment où l'événement a été créé
bubbles	Indique si l'événement remonte dans le DOM
cancelable	Indique si preventDefault() peut être appelé

PROPRIÉTÉS SPÉCIFIQUES SELON LE TYPE D'ÉVÉNEMENT

ÉVÉNEMENTS DE SOURIS

- ☞ `clientX`, `clientY` : Position relative à la fenêtre visible
- ☞ `pageX`, `pageY` : Position relative au document entier
- ☞ `button` : Bouton de souris utilisé

ÉVÉNEMENTS DE CLAVIER

- ☞ `key` : Valeur de la touche pressée
- ☞ `code` : Code physique de la touche
- ☞ `altKey`, `ctrlKey`, `shiftKey`, `metaKey` : État des touches de modification

ÉVÉNEMENTS TACTILES

- ☞ `touches` : Liste de tous les points de contact
- ☞ `changedTouches` : Points de contact impliqués dans l'événement

PROPAGATION DES ÉVÉNEMENTS

Les événements en JavaScript se propagent en trois phases :

1. Phase de capture: De la racine du document vers l'élément cible
2. Phase de cible : Sur l'élément cible lui-même
3. Phase de remontée (bubbling) : De l'élément cible vers la racine du document

DÉLÉGATION D'ÉVÉNEMENTS

La délégation d'événements est une technique qui tire parti de la propagation des événements pour gérer efficacement de nombreux éléments similaires :

```
// Au lieu d'attacher un gestionnaire à chaque élément d'une liste
const liste = document.getElementById('maListe');

liste.addEventListener('click', function(event) {
  // Vérifier si l'élément cliqué (ou un de ses parents) est un <li>
  const elementLi = event.target.closest('li');

  if (elementLi && liste.contains(elementLi)) {
    console.log('Élément de liste cliqué:', elementLi.textContent);
  }
})
```

```
});
```

Avantages de la délégation :

- Moins de gestionnaires d'événements = meilleures performances
- Fonctionne automatiquement pour les éléments ajoutés dynamiquement
- Moins de code à maintenir

ÉVÉNEMENTS PERSONNALISÉS

JavaScript permet également de créer et déclencher des événements personnalisés :

```
// Création d'un événement personnalisé
const evenementPersonnalise = new CustomEvent('monEvenement', {
  detail: {
    message: 'Bonjour depuis un événement personnalisé',
    timestamp: Date.now()
  },
  bubbles: true, // L'événement remonte dans le DOM
  cancelable: true // L'événement peut être annulé
});
```

```
// Élément sur lequel déclencher l'événement
const element = document.getElementById('monElement');

// Écouter l'événement personnalisé
element.addEventListener('monEvenement', function(event) {
  console.log('Événement personnalisé reçu:', event.detail.message);
  console.log('Timestamp:', new Date(event.detail.timestamp));
});
```

```
// Déclencher l'événement
element.dispatchEvent(eventPersonnalise);
```

EXEMPLE COMPLET AVEC COMMUNICATION ENTRE COMPOSANTS

```
// Composant Panier
class Panier {

    constructor() {
        this.element = document.getElementById('panier');
        this.produitsCount = 0;

        // Création d'un événement personnalisé
        this.changementPanierEvent = new CustomEvent('changementPanier', {
            bubbles: true,
            detail: { count: this.produitsCount }
        });
    }

    ajouterProduit() {
        this.produitsCount++;
        this.mettreAJourUI();
        this.notifierChangement();
    }

    mettreAJourUI() {
        this.element.textContent = `Panier: ${this.produitsCount} produit(s)`;
    }

    notifierChangement() {
```



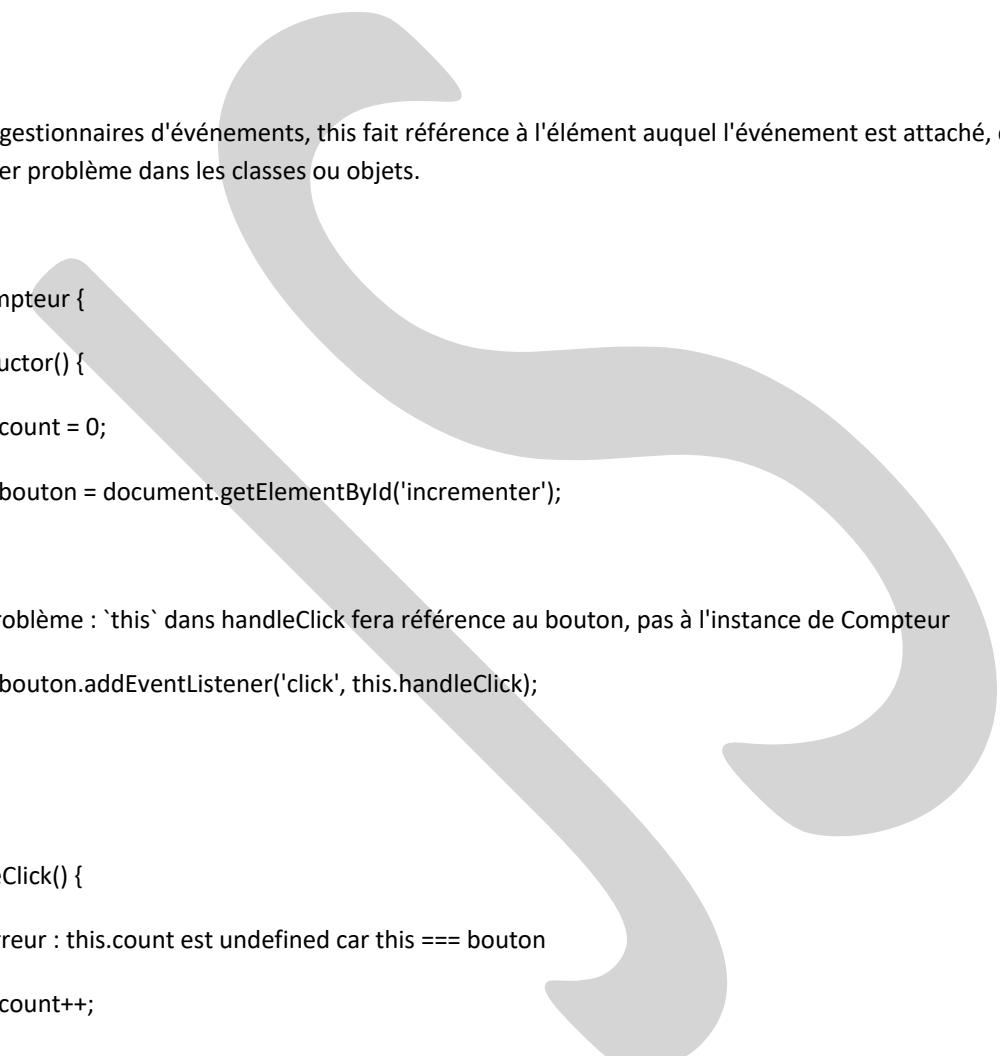
```
// Mettre à jour les détails de l'événement  
this.changementPanierEvent.detail.count = this.produitsCount;  
  
// Déclencher l'événement  
this.element.dispatchEvent(this.changementPanierEvent);  
}  
}  
  
// Composant BadgePanier qui réagit aux changements du panier  
class BadgePanier {  
    constructor() {  
        this.element = document.getElementById('badgePanier');  
  
        // Écouter l'événement personnalisé  
        document.addEventListener('changementPanier', this.mettreAJour.bind(this));  
    }  
  
    mettreAJour(event) {  
        const count = event.detail.count;  
        this.element.textContent = count;  
        this.element.style.display = count > 0 ? 'block' : 'none';  
    }  
}  
  
// Initialisation  
const panier = new Panier();  
const badge = new BadgePanier();  
  
// Bouton d'ajout de produit
```

```
const boutonAjouter = document.getElementById('ajouterProduit');

boutonAjouter.addEventListener('click', () => panier.ajouterProduit());
```

ÉVÉNEMENTS ET LE CONTEXTE THIS

Dans les gestionnaires d'événements, this fait référence à l'élément auquel l'événement est attaché, ce qui peut poser problème dans les classes ou objets.



```
class Compteur {

    constructor() {
        this.count = 0;

        this.bouton = document.getElementById('incrementer');

        // Problème : `this` dans handleClick fera référence au bouton, pas à l'instance de Compteur
        this.bouton.addEventListener('click', this.handleClick);

    }

    handleClick() {
        // Erreur : this.count est undefined car this === bouton
        this.count++;
        console.log(this.count);
    }
}
```

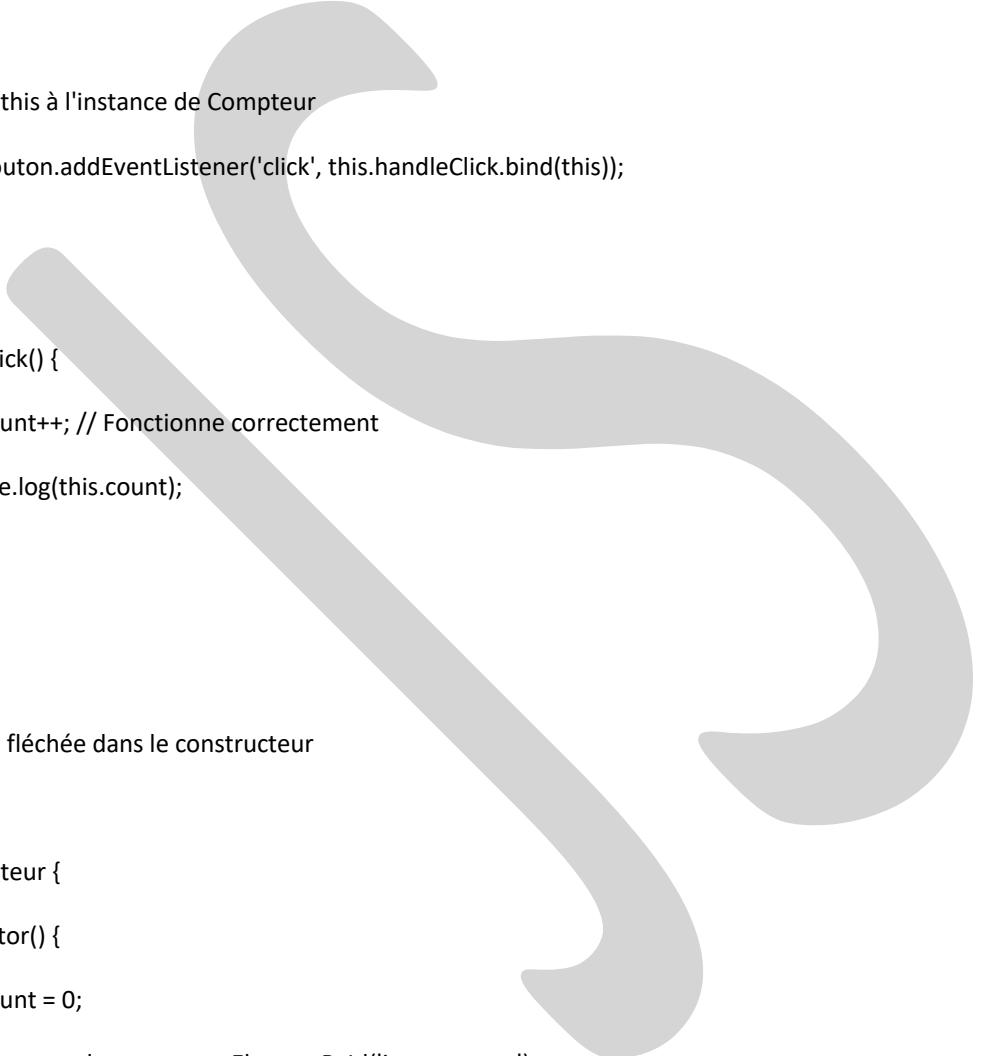
SOLUTIONS

1. Liaison avec bind()

```
class Compteur {
  constructor() {
    this.count = 0;
    this.bouton = document.getElementById('incrementer');

    // Lier this à l'instance de Compteur
    this.bouton.addEventListener('click', this.handleClick.bind(this));
  }

  handleClick() {
    this.count++; // Fonctionne correctement
    console.log(this.count);
  }
}
```



2. Fonction fléchée dans le constructeur

```
class Compteur {
  constructor() {
    this.count = 0;
    this.bouton = document.getElementById('incrementer');

    // Les fonctions fléchées maintiennent le this lexical
    this.bouton.addEventListener('click', () => {
      this.count++;
      console.log(this.count);
    });
  }
}
```

```

    });
}

}

```

3. Définir la méthode comme une propriété avec une fonction fléchée

```

class Compteur {

  constructor() {
    this.count = 0;

    this.bouton = document.getElementById('incrementer');

    this.bouton.addEventListener('click', this.handleClick);
  }

  // Définir la méthode comme une propriété de classe avec une fonction fléchée

  handleClick = () => {
    this.count++;
    console.log(this.count);
  };
}

```

Les événements sont le fondement de l'interactivité en JavaScript, permettant aux développeurs de créer des applications web réactives. Une bonne compréhension du modèle d'événements, de la propagation et des bonnes pratiques est essentielle pour développer des applications efficaces et sans bugs.

Les points clés à retenir :

- Utilisez `addEventListener` pour attacher des gestionnaires d'événements
- Tirez parti de la délégation d'événements pour améliorer les performances
- Soyez conscient du contexte `this` dans les gestionnaires d'événements

- Nettoyez les gestionnaires d'événements pour éviter les fuites de mémoire
- Utilisez les événements personnalisés pour une communication propre entre composants

EXERCICES PRATIQUES

1. Exercice de base : Créer un compteur qui s'incrémente chaque fois qu'un bouton est cliqué.
2. Exercice intermédiaire : Implémenter une liste de tâches où les tâches peuvent être ajoutées, marquées comme terminées ou supprimées en utilisant la délégation d'événements.
3. Exercice avancé : Créer un système de communication entre composants en utilisant des événements personnalisés. Par exemple, un panier d'achat qui notifie d'autres composants lorsque des articles sont ajoutés ou supprimés.
4. Défi : Développer un mini-éditeur de texte avec des raccourcis clavier personnalisés, en utilisant les événements de clavier et en gérant correctement la propagation des événements.

3.3.2. MANIPULATION DU DOM (DOCUMENT OBJECT MODEL)

Le Document Object Model (DOM) est une interface de programmation qui représente les documents HTML et XML sous forme d'une structure d'arbre, où chaque noeud représente une partie du document. Le DOM permet aux programmes JavaScript d'accéder et de modifier dynamiquement le contenu, la structure et le style d'une page web.

QU'EST-CE QUE LE DOM ?

- Une représentation structurée du document HTML/XML en mémoire
- Une API permettant de modifier le contenu, la structure et le style du document
- Indépendant du langage (bien que JavaScript soit le plus utilisé)

- Un pont entre la page web et les scripts qui interagissent avec elle

STRUCTURE ARBORESCENTE DU DOM

Le DOM organise le document comme un arbre avec des nœuds parents et enfants :

Document

```
└── html
    ├── head
    |   ├── title
    |   |   └── "Ma page web"
    |   └── meta
    └── body
        ├── header
        |   └── h1
        |       └── "Mon titre"
        ├── main
        |   ├── p
        |   |   └── "Contenu"
        |   └── div
        |       ├── img
        |       └── span
        |           └── "Légende"
        └── footer
```



TYPES DE NŒUDS COURANTS

1. Document : Le point d'entrée dans l'arbre DOM
2. Element : Représente un élément HTML (div, p, a, etc.)
3. Text : Contient le texte des éléments
4. Attribute : Représente les attributs des éléments
5. Comment : Représente les commentaires HTML
6. DocumentFragment : Un conteneur léger pour stocker des nœuds DOM

SÉLECTION D'ÉLÉMENTS DU DOM

MÉTHODES DE SÉLECTION D'ÉLÉMENTS UNIQUES

```
// Par ID (renvoie un élément unique ou null)
const header = document.getElementById('header');

// Premier élément correspondant au sélecteur CSS
const firstParagraph = document.querySelector('p');
const firstItemInList = document.querySelector('ul li');
const activeElement = document.querySelector('.active');
```

MÉTHODES DE SÉLECTION DE PLUSIEURS ÉLÉMENTS

```
// Par nom de balise (renvoie une HTMLCollection "vivante")
const allParagraphs = document.getElementsByTagName('p');

// Par nom de classe (renvoie une HTMLCollection "vivante")
const menuItems = document.getElementsByClassName('menu-item');

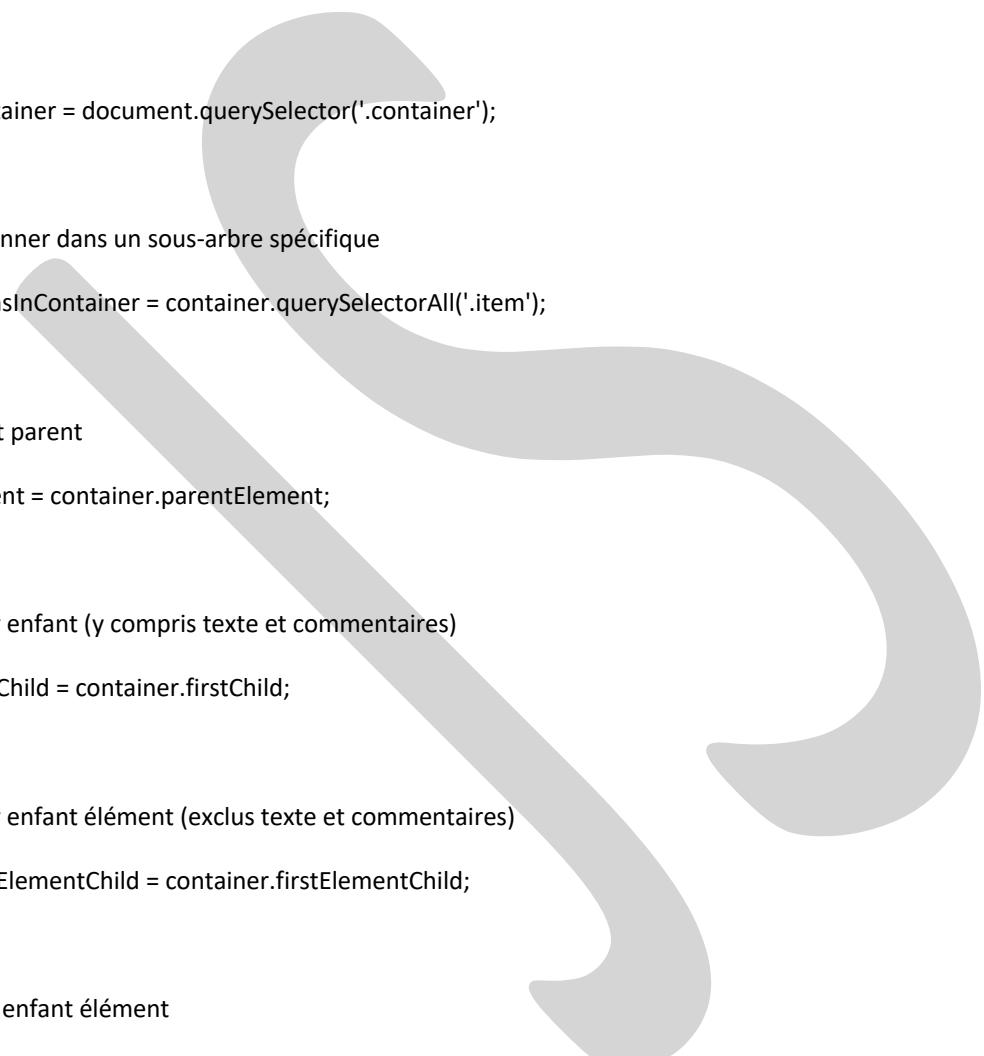
// Par sélecteur CSS (renvoie une NodeList statique)
```

```
const allLinks = document.querySelectorAll('a');

const importantElements = document.querySelectorAll('.important');
```

Note : Une HTMLCollection est "vivante", c'est-à-dire qu'elle se met à jour automatiquement lorsque le DOM change. Une NodeList est généralement statique (sauf exceptions comme childNodes).

SÉLECTION RELATIVE À UN ÉLÉMENT



```
const container = document.querySelector('.container');

// Sélectionner dans un sous-arbre spécifique
const itemsInContainer = container.querySelectorAll('.item');

// Élément parent
const parent = container.parentElement;

// Premier enfant (y compris texte et commentaires)
const firstChild = container.firstChild;

// Premier enfant élément (exclus texte et commentaires)
const firstElementChild = container.firstElementChild;

// Dernier enfant élément
const lastElementChild = container.lastElementChild;

// Tous les enfants (y compris texte et commentaires)
const allChildren = container.childNodes;

// Tous les enfants éléments (exclus texte et commentaires)
```

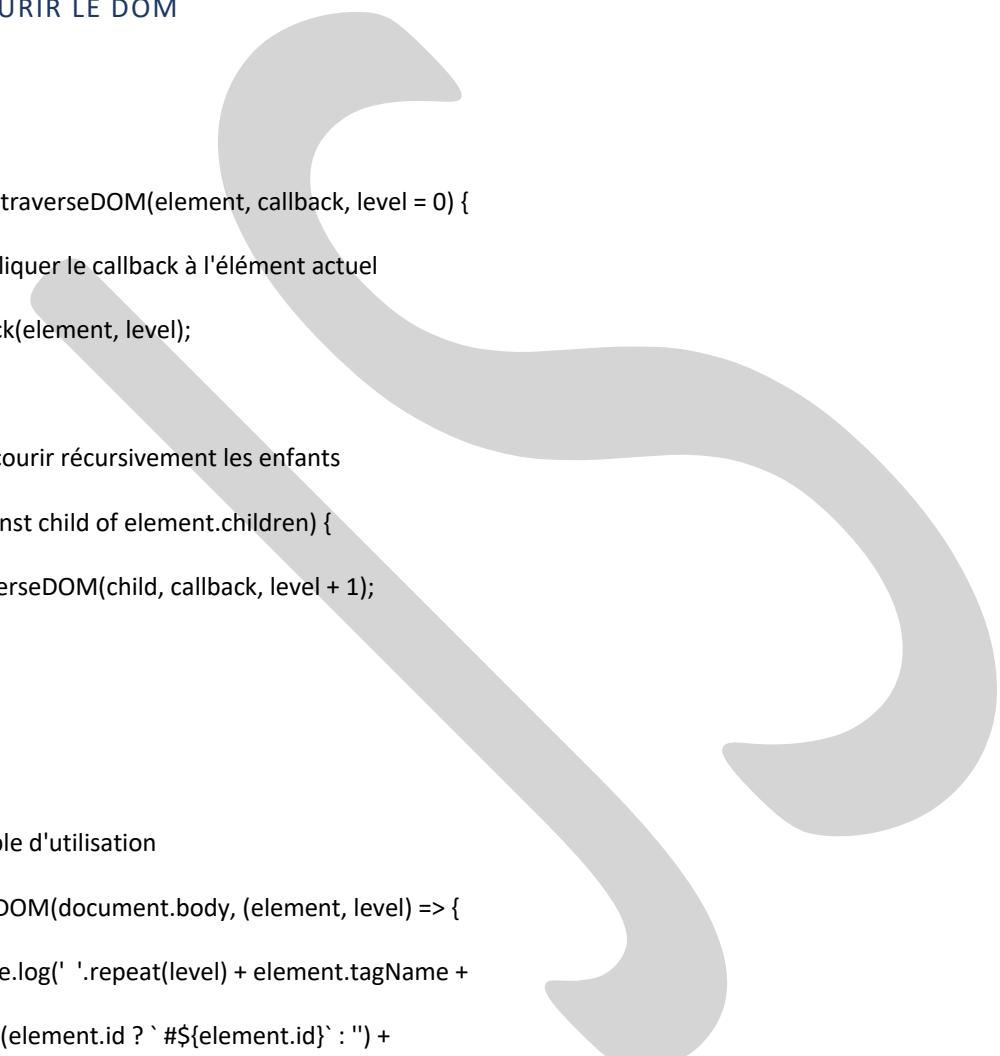
```
const allElementChildren = container.children;

// Frère précédent/suivant (élément)

const previousSibling = container.previousElementSibling;

const nextSibling = container.nextElementSibling;
```

PARCOURIR LE DOM



```
function traverseDOM(element, callback, level = 0) {
    // Appliquer le callback à l'élément actuel
    callback(element, level);

    // Parcourir récursivement les enfants
    for (const child of element.children) {
        traverseDOM(child, callback, level + 1);
    }
}

// Exemple d'utilisation
traverseDOM(document.body, (element, level) => {
    console.log(' '.repeat(level) + element.tagName +
        (element.id ? `#${element.id}` : '') +
        (element.className ? `.${element.className.replace(' ', '.')}` : ""));
});
```

MODIFICATION DU CONTENU

Manipulation du texte et du HTML

```
const element = document.getElementById('example');
```

```
// Définir/obtenir le contenu texte (échappe le HTML)
```

```
element.textContent = 'Nouveau texte';
```

```
const text = element.textContent;
```

```
// Définir/obtenir le HTML interne
```

```
element.innerHTML = '<strong>Texte en gras</strong>';
```

```
const html = element.innerHTML;
```

```
// Définir/obtenir le HTML externe (incluant l'élément lui-même)
```

```
element.outerHTML = '<div class="new">Remplace complètement</div>';
```

```
const outerHtml = element.outerHTML;
```

CRÉATION ET INSERTION D'ÉLÉMENTS

```
// Créer un nouvel élément
```

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

```
newParagraph.textContent = 'Paragraphe créé dynamiquement';
```

```
newParagraph.className = 'important';
```

```
// Créer un nœud de texte
```

```
const textNode = document.createTextNode('Du texte simple');
```

```
// Insérer à la fin d'un élément parent
```

```
const container = document.querySelector('.container');
```

```
container.appendChild(newParagraph);
```

```
// Insérer avant un élément spécifique

const referenceElement = document.getElementById('reference');

container.insertBefore(newParagraph, referenceElement);

// Méthodes modernes d'insertion

container.append(newParagraph, textNode); // Insère à la fin, accepte plusieurs nœuds
container.prepend(newParagraph); // Insère au début
referenceElement.before(newParagraph); // Insère avant
referenceElement.after(newParagraph); // Insère après
referenceElement.replaceWith(newParagraph); // Remplace
```

DOCUMENTFRAGMENT POUR LES INSERTIONS MULTIPLES

```
// Créer un fragment (conteneur pour plusieurs éléments)

const fragment = document.createDocumentFragment();

// Ajouter des éléments au fragment

for (let i = 0; i < 10; i++) {
    const li = document.createElement('li');
    li.textContent = `Item ${i}`;
    fragment.appendChild(li);
}

// Ajouter le fragment au DOM (une seule opération de reflow)

const list = document.querySelector('ul');

list.appendChild(fragment);
```

CLONAGE D'ÉLÉMENTS

```
const original = document.getElementById('original');

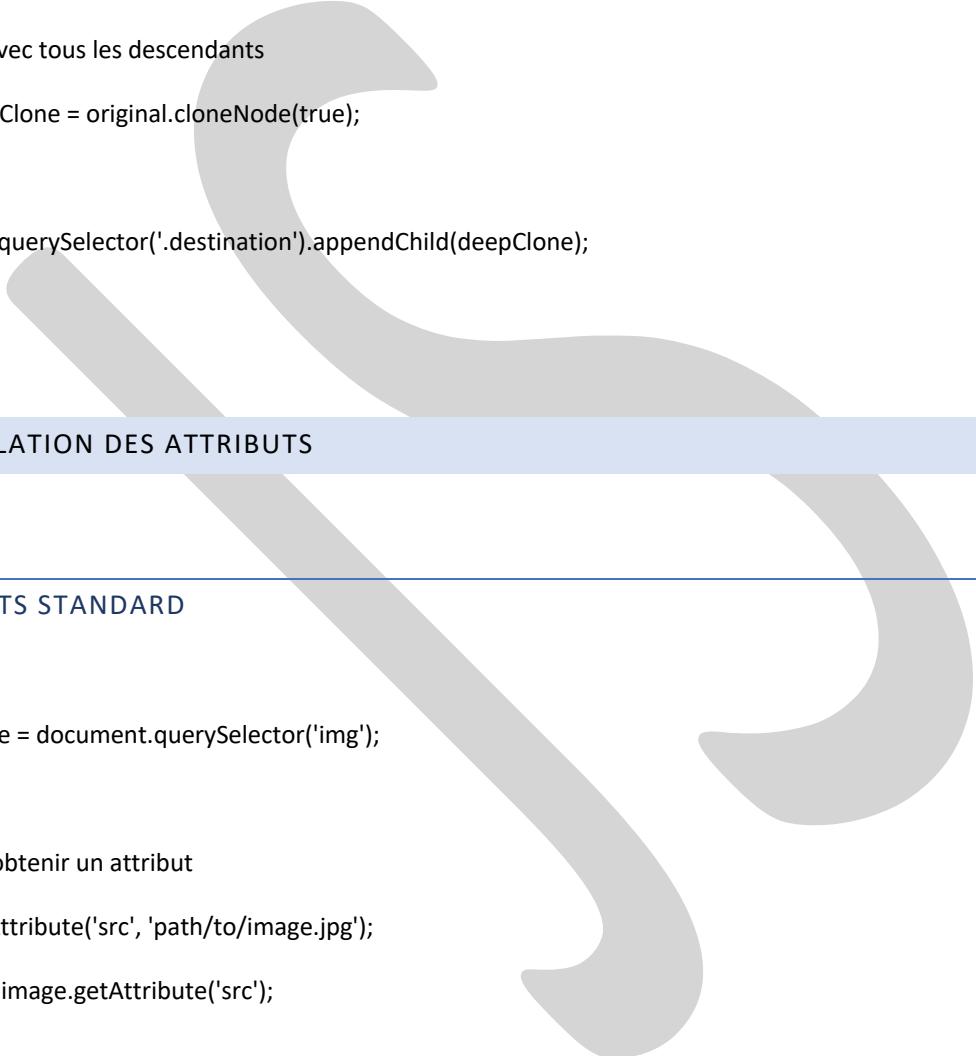
// Cloner sans les enfants

const shallowClone = original.cloneNode(false);

// Cloner avec tous les descendants

const deepClone = original.cloneNode(true);

document.querySelector('.destination').appendChild(deepClone);
```



MANIPULATION DES ATTRIBUTS

ATTRIBUTS STANDARD

```
const image = document.querySelector('img');

// Définir/obtenir un attribut

image.setAttribute('src', 'path/to/image.jpg');

const src = image.getAttribute('src');

// Vérifier si un attribut existe

const hasAlt = image.hasAttribute('alt');

// Supprimer un attribut

image.removeAttribute('title');
```

```
// Accès direct aux attributs courants  
  
image.src = 'path/to/new-image.jpg';  
  
image.alt = 'Description de l\'image';  
  
image.id = 'main-image';
```

ATTRIBUTS DE DONNÉES PERSONNALISÉS (DATA-*)

```
const element = document.querySelector('.product');  
  
// Définir via setAttribute  
  
element.setAttribute('data-product-id', '123');  
  
// Accéder via getAttribute  
  
const productId = element.getAttribute('data-product-id');  
  
// Utiliser l'API dataset (camelCase)  
  
element.dataset.productId = '456';  
  
element.dataset.inStock = 'true';  
  
element.dataset.lastUpdated = '2023-05-15';  
  
const isInStock = element.dataset.inStock;  
  
console.log(Object.keys(element.dataset)); // ['productId', 'inStock', 'lastUpdated']
```

MANIPULATION DES CLASSES ET STYLES

MANIPULATION DES CLASSES

```
const element = document.getElementById('myElement');

// Ajouter une classe
element.classList.add('active');

// Supprimer une classe
element.classList.remove('hidden');

// Basculer une classe (ajouter si absente, retirer si présente)
element.classList.toggle('expanded');

// Vérifier si une classe existe
const isHighlighted = element.classList.contains('highlight');

// Remplacer une classe par une autre
element.classList.replace('old-class', 'new-class');

// Définir plusieurs classes en remplaçant toutes les existantes
element.className = 'btn btn-primary large';
```

MANIPULATION DES STYLES INLINE

```
const box = document.querySelector('.box');

// Définir des styles individuels
box.style.backgroundColor = 'blue';
box.style.width = '200px';
box.style.marginTop = '20px';
box.style.border = '1px solid black';
```

```
// Définir plusieurs styles à la fois  
  
Object.assign(box.style, {  
  
    padding: '10px',  
  
    color: 'white',  
  
    fontWeight: 'bold',  
  
    transition: 'all 0.3s ease'  
  
});
```

```
// Obtenir un style inline spécifique  
  
const bgColor = box.style.backgroundColor;  
  
// Noter que box.style ne donne accès qu'aux styles inline,  
// pas aux styles des feuilles de style
```

OBTENIR LES STYLES CALCULÉS

```
const element = document.getElementById('myElement');  
  
// Obtenir tous les styles calculés  
  
const computedStyles = window.getComputedStyle(element);  
  
// Obtenir une propriété spécifique  
  
const fontSize = computedStyles.fontSize;  
  
const opacity = computedStyles.getPropertyValue('opacity');  
  
// Style calculé pour un pseudo-élément  
  
const beforeContent = window.getComputedStyle(element, '::before').content;
```

DIMENSIONS ET POSITION DES ÉLÉMENTS

DIMENSIONS

```
const element = document.querySelector('.box');

// Dimensions du contenu uniquement

const width = element.clientWidth; // largeur sans bordures ni marges

const height = element.clientHeight; // hauteur sans bordures ni marges

// Dimensions incluant le padding et les bordures

const fullWidth = element.offsetWidth;

const fullHeight = element.offsetHeight;

// Dimensions du contenu, y compris le contenu caché (si scrollable)

const scrollWidth = element.scrollWidth;

const scrollHeight = element.scrollHeight;
```

POSITION ET DÉFILEMENT

```
// Position relative au parent positionné

const offsetLeft = element.offsetLeft;

const offsetTop = element.offsetTop;

// Position relative au viewport (partie visible de la fenêtre)

const rect = element.getBoundingClientRect();

console.log(rect.top, rect.right, rect.bottom, rect.left, rect.width, rect.height);
```

```
// Position de défilement

const scrollTop = element.scrollTop; // pixels défilés depuis le haut

const scrollLeft = element.scrollLeft; // pixels défilés depuis la gauche
```

```
// Faire défiler un élément

element.scrollTo(0, 100); // défiler à une position spécifique

element.scrollBy(0, 10); // défiler de manière relative
```

```
// Faire défiler un élément en vue

element.scrollIntoView();

element.scrollIntoView({ behavior: 'smooth', block: 'center' });
```

POSITION AU NIVEAU DE LA FENÊTRE

```
// Dimensions de la fenêtre (viewport)

const viewportWidth = window.innerWidth;

const viewportHeight = window.innerHeight;

// Dimensions du document entier

const pageWidth = document.documentElement.scrollWidth;

const pageHeight = document.documentElement.scrollHeight;

// Position de défilement de la page

const pageScrollTop = window.pageYOffset || document.documentElement.scrollTop;

const pageScrollLeft = window.pageXOffset || document.documentElement.scrollLeft;

// Faire défiler la page

window.scrollTo(0, 0); // haut de page

window.scrollBy(0, 100); // défiler relativement
```

```
window.scrollTo({  
    top: 1000,  
    behavior: 'smooth'  
}); // défilement fluide
```

MANIPULATION AVANCÉE DU DOM

SUPPRESSION D'ÉLÉMENTS

```
const element = document.getElementById('toRemove');  
  
// Méthode 1 : via le parent  
if (element.parentNode) {  
    element.parentNode.removeChild(element);  
}  
  
// Méthode 2 : directement (plus moderne)  
element.remove();
```

DÉPLACEMENT D'ÉLÉMENTS

```
const element = document.getElementById('toMove');  
const newParent = document.querySelector('.destination');  
  
// Déplacer un élément (il est d'abord retiré puis ajouté ailleurs)  
newParent.appendChild(element);
```

REEMPLACEMENT D'ÉLÉMENTS

```

const oldElement = document.getElementById('old');

const newElement = document.createElement('div');

newElement.textContent = 'Nouvel élément';

// Méthode 1 : via le parent

if (oldElement.parentNode) {

    oldElement.parentNode.replaceChild(newElement, oldElement);

}

// Méthode 2 : directement (plus moderne)

oldElement.replaceWith(newElement);

```

INSERTION À DES POSITIONS SPÉCIFIQUES

```

const parent = document.querySelector('.container');

const newElement = document.createElement('p');

newElement.textContent = 'Inséré à une position spécifique';

// Insérer à une position spécifique (ici, en 3e position)

const position = 2; // 0-indexed

const children = parent.children;

if (position >= children.length) {

    parent.appendChild(newElement);

} else {

    parent.insertBefore(newElement, children[position]);

}

```

```
// Méthode moderne avec insertAdjacentElement

parent.insertAdjacentElement('afterbegin', newElement); // Premier enfant

parent.insertAdjacentElement('beforeend', newElement); // Dernier enfant
```

INSERTADJACENTHTML ET INSERTADJACENTTEXT



```
const element = document.getElementById('target');

// Insérer du HTML à des positions spécifiques

element.insertAdjacentHTML('beforebegin', '<p>Avant l\'élément</p>'); // Avant l'élément
element.insertAdjacentHTML('afterbegin', '<p>Premier enfant</p>'); // Premier enfant
element.insertAdjacentHTML('beforeend', '<p>Dernier enfant</p>'); // Dernier enfant
element.insertAdjacentHTML('afterend', '<p>Après l\'élément</p>'); // Après l'élément

// Insérer du texte de la même manière

element.insertAdjacentText('beforeend', 'Texte ajouté à la fin');
```

ÉVÉNEMENTS ET LE DOM

DÉLÉGATION D'ÉVÉNEMENTS AMÉLIORÉE



```
document.querySelector('.todo-list').addEventListener('click', function(event) {

    const target = event.target;

    // Vérifier si un bouton de suppression a été cliqué

    if (target.matches('.delete-btn, .delete-btn *)) {

        const todoItem = target.closest('.todo-item');
```

```

if (todoItem) {
    todoItem.remove();
    event.preventDefault(); // Empêcher le comportement par défaut des liens
}
}

// Vérifier si une case à cocher a été cliquée
if (target.matches('.todo-checkbox')) {
    const todoItem = target.closest('.todo-item');
    if (todoItem) {
        todoItem.classList.toggle('completed', target.checked);
    }
}
});


```

MUTATION OBSERVER

Observer les changements dans le DOM :

```

// Créer un observateur
const observer = new MutationObserver((mutations) => {
    mutations.forEach(mutation => {
        if (mutation.type === 'childList') {
            console.log('Éléments ajoutés:', mutation.addedNodes);
            console.log('Éléments supprimés:', mutation.removedNodes);
        } else if (mutation.type === 'attributes') {
            console.log(`L'attribut ${mutation.attributeName} de ${mutation.target} a changé`);
        }
    });
});

```

```

});
```

// Configurer l'observateur

```

const config = {

  childList: true, // Observer les ajouts/suppressions d'enfants

  attributes: true, // Observer les changements d'attributs

  characterData: true, // Observer les changements de données de caractères

  subtree: true, // Observer également les descendants

  attributeOldValue: true, // Enregistrer les valeurs précédentes des attributs

  characterDataOldValue: true // Enregistrer les valeurs précédentes des données de caractères
};
```

// Démarrer l'observation

```

const targetNode = document.querySelector('#observed');

observer.observe(targetNode, config);

// Plus tard, arrêter l'observation

observer.disconnect();
```

INTERSECTION OBSERVER

Observer quand des éléments deviennent visibles dans le viewport :

```

const observer = new IntersectionObserver(entries, observer => {

  entries.forEach(entry => {

    if (entry.isIntersecting) {

      // L'élément est visible

      console.log('Element visible:', entry.target);
    }
  });
});
```

```

// Exemple : charger une image paresseuse

if (entry.target.dataset.src) {

    entry.target.src = entry.target.dataset.src;

    delete entry.target.dataset.src;

    // Optionnel : arrêter d'observer une fois chargé
    observer.unobserve(entry.target);
}

});

}, {

    root: null, // viewport
    rootMargin: '0px', // marge autour du viewport
    threshold: 0.1 // déclenchement lorsque 10% de l'élément est visible
});

// Observer plusieurs éléments

document.querySelectorAll('.lazy-image').forEach(img => {
    observer.observe(img);
});

```

RESIZE OBSERVER

Observer les changements de taille d'un élément :

```

const observer = new ResizeObserver(entries => {

    for (const entry of entries) {

        const { width, height } = entry.contentRect;
        console.log(`Élément redimensionné à ${width}x${height}`);
    }
});

```

```
// Réagir au redimensionnement

if (width < 600) {
    entry.target.classList.add('compact');
} else {
    entry.target.classList.remove('compact');
}

});

// Observer un élément

observer.observe(document.querySelector('.responsive-element'));
```

BONNES PRATIQUES POUR LA MANIPULATION DU DOM

1. Minimiser l'accès au DOM

- ☞ Stocker les références aux éléments DOM dans des variables
- ☞ Éviter les requêtes répétées pour le même élément
- ☞ Préférer querySelectorAll() suivi d'une boucle plutôt que des appels répétés à querySelector()

2. Utiliser les bonnes méthodes de sélection

- ☞ getElementById() est plus rapide que querySelector()
- ☞ Limiter la portée des requêtes avec les méthodes d'élément
- ☞ Être aussi spécifique que nécessaire, mais pas plus
- ☞ Utiliser des attributs id pour les éléments uniques importants

3. Gérer efficacement les modifications multiples

- ☞ Utiliser DocumentFragment
- ☞ Détailler, modifier, puis réattacher pour les modifications complexes
- ☞ Construire du HTML en chaîne pour les grandes insertions
- ☞ Considérer le remplacement complet via innerHTML pour des changements massifs

4. Utiliser des classes CSS plutôt que des styles inline

- ☞ Définir les styles dans des classes CSS
- ☞ Manipuler les classes avec classList plutôt que modifier les styles directement
- ☞ Utiliser des transitions/animations CSS plutôt que des animations JavaScript quand possible

5. Être attentif au reflow et au repaint

- ☞ Grouper les modifications de style
- ☞ Modifier les classes plutôt que les styles individuels
- ☞ Utiliser position: absolute ou fixed pour sortir les éléments du flux normal pendant l'animation
- ☞ Utiliser transform et opacity pour les animations

6. Déléguer les événements

- ☞ Attacher les gestionnaires aux éléments parents plutôt qu'aux enfants individuels
- ☞ Utiliser la méthode `matches()` pour filtrer les cibles
- ☞ Particulièrement utile pour les éléments générés dynamiquement

7. Nettoyer après soi

- ☞ Supprimer les gestionnaires d'événements quand ils ne sont plus nécessaires
- ☞ Arrêter les observateurs quand ils ne sont plus nécessaires
- ☞ Supprimer les références aux éléments DOM pour permettre la collecte des déchets

La manipulation du DOM est au cœur de la création d'applications web interactives. Bien que les frameworks modernes abstraient souvent ces opérations, comprendre comment le DOM fonctionne et comment le manipuler efficacement est une compétence fondamentale pour tout développeur JavaScript.

Maîtriser ces techniques permet de :

- ☞ Créer des applications plus réactives et performantes
- ☞ Déboguer efficacement les problèmes liés au DOM
- ☞ Comprendre comment fonctionnent les frameworks sous le capot
- ☞ Écrire du code qui s'exécute de manière optimale, même dans des environnements contraints

En appliquant les bonnes pratiques et en comprenant les implications de performance des manipulations du DOM, vous pouvez créer des expériences utilisateur fluides et réactives, quelle que soit la complexité de votre application.

EXERCICES PRATIQUES

1. Exercice de base : Créer une liste dynamique où les utilisateurs peuvent ajouter, supprimer et modifier des éléments.

2. Exercice intermédiaire : Implémenter un carrousel d'images avec des transitions fluides, en utilisant les techniques de manipulation du DOM efficace.

3. Exercice avancé : Créer un éditeur de texte riche simple qui permet de mettre du texte en gras, en italique et de changer les couleurs, en manipulant directement le DOM.

4. Défi : Développer une interface de glisser-déposer (drag and drop) pour réorganiser des éléments, avec des animations fluides et une gestion efficace du DOM.

4.1.1. INTRODUCTION AUX OBJETS

QU'EST-CE QU'UN OBJET EN JAVASCRIPT ?

En JavaScript, un objet est une collection de paires clé-valeur où les clés sont des chaînes de caractères (ou Symbols) et les valeurs peuvent être de n'importe quel type, y compris d'autres objets. Les objets sont l'un des types de données les plus fondamentaux et polyvalents en JavaScript, servant de base à presque tous les aspects avancés du langage.

Les objets JavaScript peuvent être considérés comme :

- Des collections de propriétés
- Des dictionnaires de données associatives
- Des entités qui encapsulent des données et des comportements
- Les blocs de construction fondamentaux pour les modèles de conception orientés objet

CARACTÉRISTIQUES FONDAMENTALES DES OBJETS

- Mutabilité : Les objets peuvent être modifiés après leur création
- Référence : Les variables d'objet stockent des références, et non des copies

- ☞ Dynamisme : Les propriétés peuvent être ajoutées, modifiées ou supprimées à tout moment
- ☞ Prototypes : Les objets peuvent hériter de propriétés d'autres objets via des chaînes de prototypes

CRÉATION D'OBJETS

SYNTAXE LITTÉRALE D'OBJET

La façon la plus courante et la plus simple de créer un objet est d'utiliser la notation littérale :

```
const personne = {
    nom: "Dupont",
    prenom: "Jean",
    age: 30,
    presentation: function() {
        return `Je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`;
    }
};
```

CONSTRUCTEUR OBJECT()

On peut également utiliser le constructeur « Object() » :

```
const personne = new Object();
personne.nom = "Dupont";
personne.prenom = "Jean";
personne.age = 30;
personne.presentation = function() {
```

```
return `Je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`;
};
```

FONCTION CONSTRUCTEUR

Pour créer plusieurs objets avec la même structure, on peut définir une fonction constructeur :

```
function Personne(nom, prenom, age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
    this.presentation = function() {
        return `Je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`;
    };
}
```

```
const jean = new Personne("Dupont", "Jean", 30);
const marie = new Personne("Martin", "Marie", 25);
```

MÉTHODE OBJECT.CREATE()

`Object.create()` permet de créer un nouvel objet avec un prototype spécifié :

```
const personnePrototype = {
    presentation: function() {
        return `Je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`;
    }
};
```

```
const jean = Object.create(personnePrototype);

jean.nom = "Dupont";
jean.prenom = "Jean";
jean.age = 30;
```

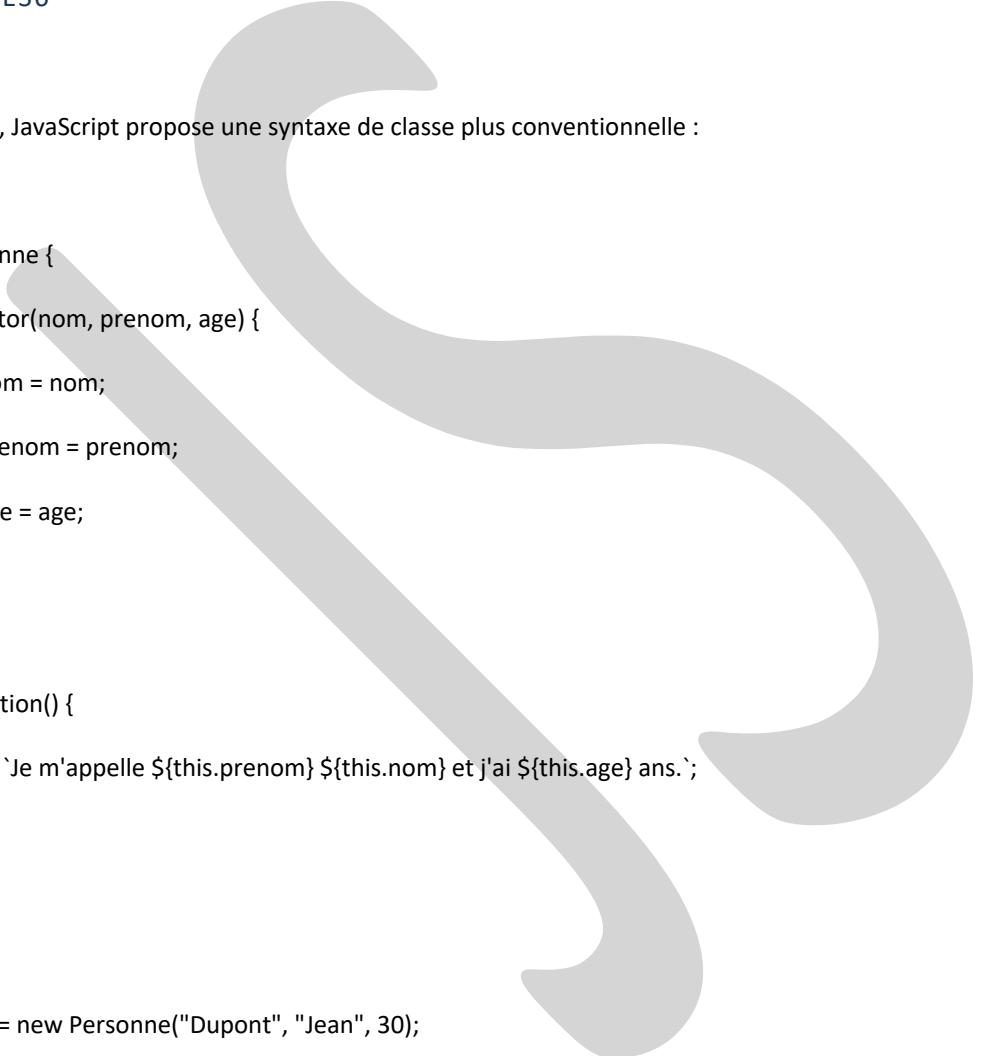
CLASSES ES6

Depuis ES6, JavaScript propose une syntaxe de classe plus conventionnelle :

```
class Personne {
  constructor(nom, prenom, age) {
    this.nom = nom;
    this.prenom = prenom;
    this.age = age;
  }

  presentation() {
    return `Je m'appelle ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`;
  }
}

const jean = new Personne("Dupont", "Jean", 30);
```



ACCÈS AUX PROPRIÉTÉS

Il existe deux façons d'accéder aux propriétés d'un objet :

NOTATION AVEC POINT

```
const nom = personne.nom;
personne.age = 31;
```

NOTATION AVEC CROCHETS

```
const nom = personne["nom"];
personne["age"] = 31;
```

// Avantage : permet d'utiliser des noms de propriétés dynamiques

```
const propriété = "nom";
const valeur = personne[propriété]; // Équivalent à personne.nom
```

PROPRIÉTÉS ET MÉTHODES D'OBJET

PROPRIÉTÉS D'OBJET

Une propriété associe une clé (un nom) à une valeur :

```
const voiture = {
  marque: "Toyota",
  modèle: "Corolla",
  année: 2020,
  couleur: "bleue"
};
```

PROPRIÉTÉS CALCULÉES (ES6)

```
const prefixe = "info";
const utilisateur = {
  [`${prefixe}_nom`]: "Dupont",
  [`${prefixe}_age`]: 30
};
// Résultat : { info_nom: "Dupont", info_age: 30 }
```

MÉTHODES D'OBJET

Une méthode est une propriété dont la valeur est une fonction :

```
const calculatrice = {
  valeur: 0,
  ajouter: function(x) {
    this.valeur += x;
    return this.valeur;
  },
  soustraire: function(x) {
    this.valeur -= x;
    return this.valeur;
  },
  // Syntaxe raccourcie (ES6)
  multiplier(x) {
    this.valeur *= x;
    return this.valeur;
  },
  diviser(x) {
    if (x === 0) throw new Error("Division par zéro");
  }
};
```

```

        this.valeur /= x;
        return this.valeur;
    }

};

calculatrice.ajouter(5); // valeur = 5
calculatrice.multiplier(2); // valeur = 10

```

LE MOT-CLÉ « THIS »

Dans les méthodes d'objet, this fait référence à l'objet sur lequel la méthode est appelée :

```

const personne = {
    nom: "Dupont",
    saluer() {
        console.log(`Bonjour, je m'appelle ${this.nom}`);
    }
};

personne.saluer(); // "Bonjour, je m'appelle Dupont"

```

ATTENTION AU CONTEXTE DE THIS

Le contexte de this peut changer selon la façon dont une fonction est appelée :

```

const personne = {
    nom: "Dupont",
    saluer() {

```

```

        console.log(`Bonjour, je m'appelle ${this.nom}`);
    }

};

const salutation = personne.saluer;

salutation(); // "Bonjour, je m'appelle undefined" (this n'est pas personne)

```

```

// Solutions

const salutationLiee = personne.saluer.bind(personne);

salutationLiee(); // "Bonjour, je m'appelle Dupont"

```

```

// Ou avec une fonction fléchée qui capture this

personne.saluerFleche = () => {

    console.log(`Bonjour, je m'appelle ${personne.nom}`);
}

```

VÉRIFICATION ET MANIPULATION DES PROPRIÉTÉS

VÉRIFIER L'EXISTENCE D'UNE PROPRIÉTÉ

```

const personne = { nom: "Dupont", age: 30 };

// Méthode 1 : opérateur in

console.log("nom" in personne); // true
console.log("adresse" in personne); // false

// Méthode 2 : hasOwnProperty (ignore les propriétés héritées)

console.log(personne.hasOwnProperty("nom")); // true

```

```
console.log(personne.hasOwnProperty("toString")); // false (héritée)
```

```
// Méthode 3 : accès direct (attention aux valeurs falsy)
```

```
console.log(personne.nom !== undefined); // true
```

```
console.log(personne.adresse !== undefined); // false
```

ÉNUMÉRATION DES PROPRIÉTÉS

```
const voiture = {
    marque: "Toyota",
    modele: "Corolla",
    annee: 2020
};
```

```
// for...in (parcourt aussi les propriétés héritées énumérables)
```

```
for (const proprietee in voiture) {
    console.log(`$${proprietee}: ${voiture[proprietee]}`);
}
```

```
// Object.keys() (uniquement les propriétés propres énumérables)
```

```
const cles = Object.keys(voiture);
console.log(cles); // ["marque", "modele", "annee"]
```

```
// Object.values() (uniquement les valeurs des propriétés propres énumérables)
```

```
const valeurs = Object.values(voiture);
console.log(valeurs); // ["Toyota", "Corolla", 2020]
```

```
// Object.entries() (paires clé-valeur des propriétés propres énumérables)
```

```
const entrees = Object.entries(voiture);
```

```
console.log(entrees); // [["marque", "Toyota"], ["modele", "Corolla"], ["annee", 2020]]
```

AJOUT ET SUPPRESSION DE PROPRIÉTÉS

Les objets JavaScript sont dynamiques, on peut donc ajouter ou supprimer des propriétés à tout moment :

```
const personne = { nom: "Dupont" };

// Ajout de propriétés
personne.prenom = "Jean";
personne["age"] = 30;

// Suppression de propriétés
delete personne.age;

console.log(personne); // { nom: "Dupont", prenom: "Jean" }
```

COPIE ET FUSION D'OBJETS

COPIE SUPERFICIELLE (SHALLOW COPY)

```
const original = { a: 1, b: 2, c: { d: 3 } };

// Méthode 1 : Object.assign()
const copie1 = Object.assign({}, original);

// Méthode 2 : Opérateur de décomposition (spread)
const copie2 = { ...original };
```

```
// Attention : les objets imbriqués sont copiés par référence
copie1.c.d = 4;
console.log(original.c.d); // 4 (modifié car c'est la même référence)
```

COPIE PROFONDE (DEEP COPY)

// Méthode 1 : JSON (avec limitations pour les fonctions, dates, etc.)

```
const copieJSON = JSON.parse(JSON.stringify(original));
```

// Méthode 2 : Bibliothèques comme lodash

```
// const copieProfonde = _.cloneDeep(original);
```

// Méthode 3 : Implémentation personnalisée

```
function copierProfondement(obj) {
    if (obj === null || typeof obj !== 'object') {
        return obj;
    }
```

```
const copie = Array.isArray(obj) ? [] : {};
```

```
for (const cle in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, cle)) {
        copie[cle] = copierProfondement(obj[cle]);
    }
}
```

```
return copie;
}
```

FUSION D'OBJETS

```
const objetA = { a: 1, b: 2 };

const objetB = { b: 3, c: 4 }; // Notez que b est présent dans les deux objets
```

// Méthode 1 : Object.assign()

```
const fusion1 = Object.assign({}, objetA, objetB);

console.log(fusion1); // { a: 1, b: 3, c: 4 } (b de objetB écrase b de objetA)
```

// Méthode 2 : Opérateur de décomposition

```
const fusion2 = { ...objetA, ...objetB };

console.log(fusion2); // { a: 1, b: 3, c: 4 }
```

// Fusion personnalisée avec logique de résolution de conflits

```
const fusion3 = {

  ...objetA,
  ...objetB,
  b: `${objetA.b} et ${objetB.b}` // Logique personnalisée pour la propriété b
};

console.log(fusion3); // { a: 1, b: "2 et 3", c: 4 }
```

DESCRIPTEURS DE PROPRIÉTÉS

JavaScript permet de contrôler finement le comportement des propriétés avec des descripteurs :

```
const personne = {};

Object.defineProperty(personne, 'nom', {
  value: 'Dupont',
  writable: true, // Peut être modifiée
```

```

enumerable: true, // Apparaît dans les boucles
configurable: true // Peut être supprimée ou reconfigurée
});

// Propriété en lecture seule
Object.defineProperty(personne, 'id', {
  value: 12345,
  writable: false, // Ne peut pas être modifiée
  enumerable: true,
  configurable: false // Ne peut pas être supprimée
});

// Tentative de modification de la propriété en lecture seule
personne.id = 67890; // En mode strict: erreur, sinon: ignoré silencieusement
console.log(personne.id); // 12345 (inchangé)

```

ACCESSEURS (GETTERS ET SETTERS)

```

const thermometre = {
  _temperature: 0, // Convention: propriété "privée"

  // Getter
  get temperature() {
    return this._temperature;
  }

  // Setter avec validation
  set temperature(valeur) {
    if (typeof valeur !== 'number') {

```

```

        throw new TypeError('La température doit être un nombre');

    }

    this._temperature = valeur;

},


// Getter avec transformation

get temperatureF() {

    return (this._temperature * 9/5) + 32;
},


// Setter avec transformation

set temperatureF(valeur) {

    if (typeof valeur !== 'number') {

        throw new TypeError('La température doit être un nombre');

    }

    this._temperature = (valeur - 32) * 5/9;

}

};

thermometre.temperature = 25;

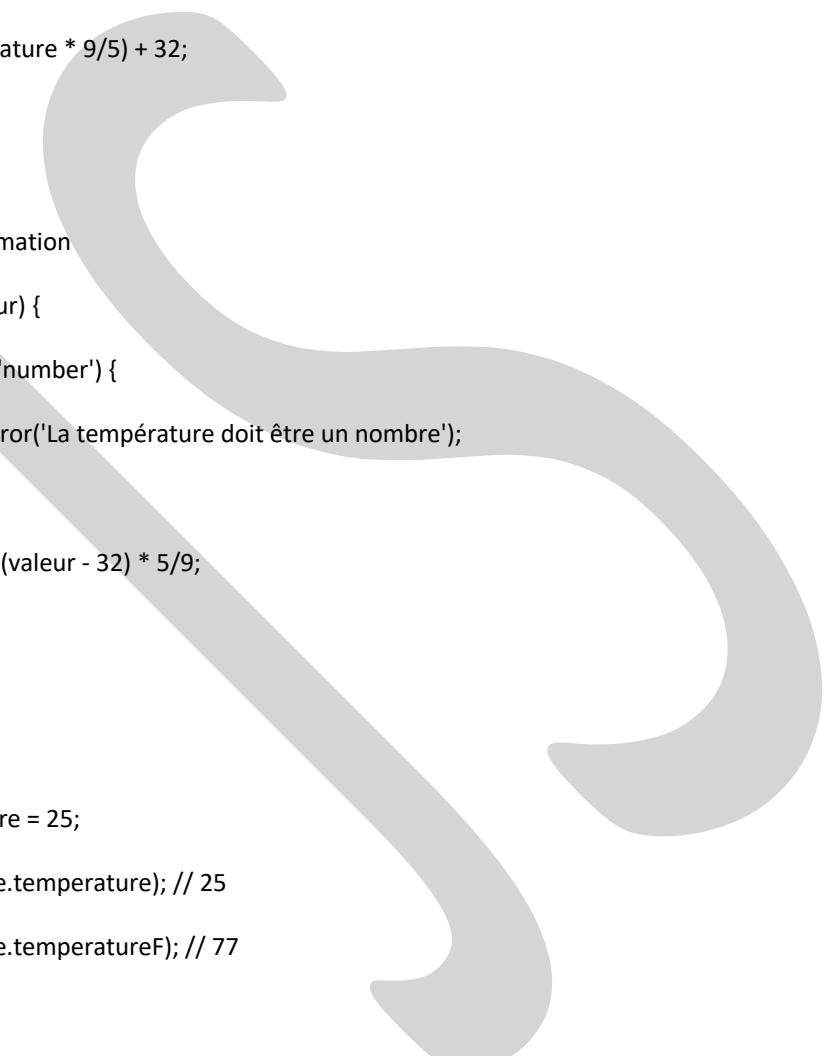
console.log(thermometre.temperature); // 25

console.log(thermometre.temperatureF); // 77


thermometre.temperatureF = 68;

console.log(thermometre.temperature); // 20

```



OBJETS IMMUABLES

Il peut être utile de créer des objets qu'on ne peut pas modifier pour éviter des effets de bord indésirables :

PROPRIÉTÉS NON MODIFIABLES

```
const config = {
  API_URL: 'https://api.example.com',
  MAX_RETRIES: 3,
  TIMEOUT: 5000
};

// Rend chaque propriété non modifiable
Object.freeze(config);

// Tentative de modification
config.TIMEOUT = 10000; // Ignoré (avec une erreur en mode strict)
console.log(config.TIMEOUT); // 5000 (inchangé)

// Vérification
console.log(Object.isFrozen(config)); // true
```

DIFFÉRENTS NIVEAUX D'IMMUTABILITÉ

```
const utilisateur = {
  nom: 'Dupont',
  adresse: {
    rue: '123 rue Principale',
    ville: 'Paris'
  }
};
```

```
// Empêche l'ajout/suppression de propriétés mais permet la modification
Object.preventExtensions(utilisateur);
console.log(Object.isExtensible(utilisateur)); // false

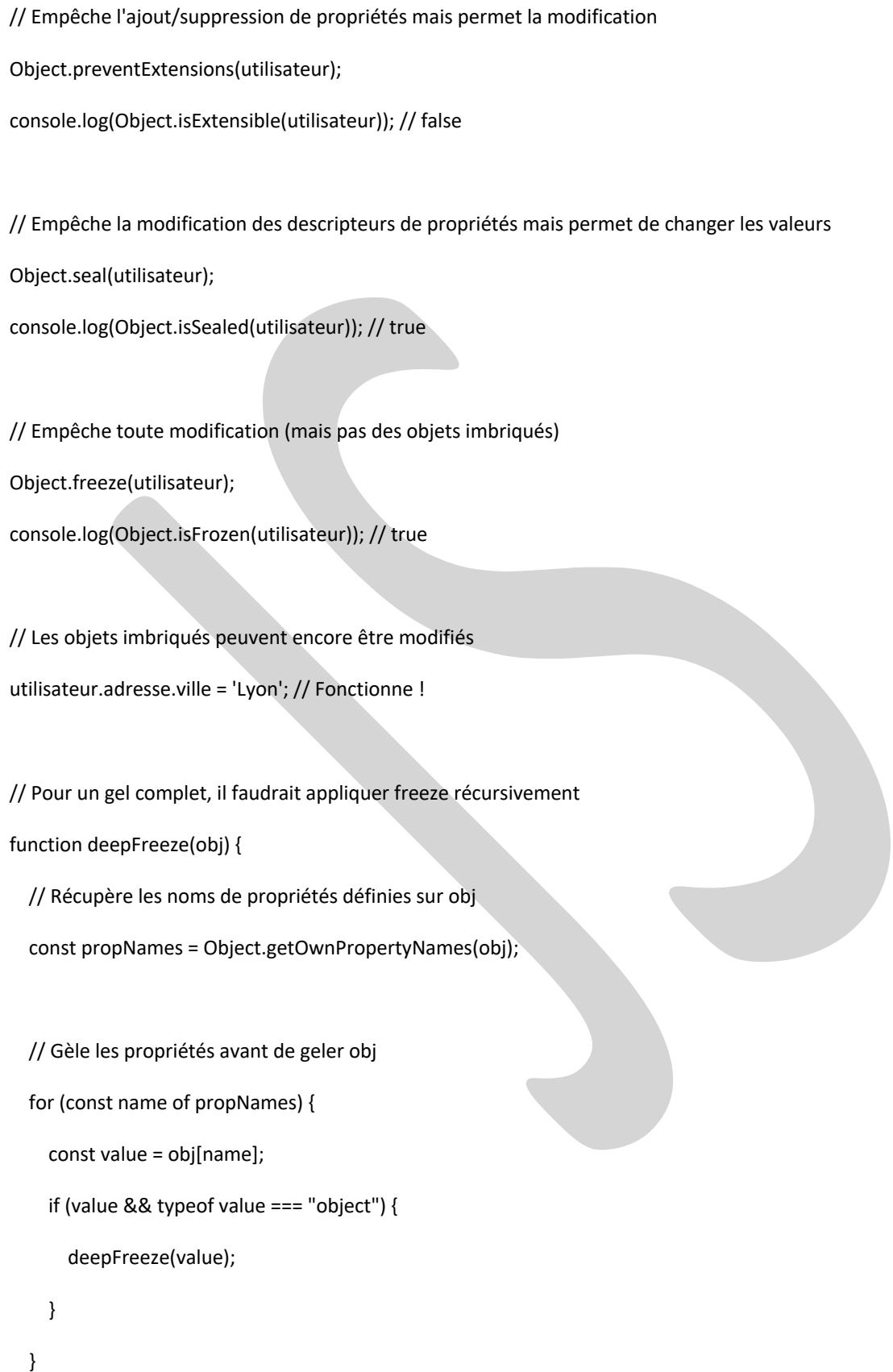
// Empêche la modification des descripteurs de propriétés mais permet de changer les valeurs
Object.seal(utilisateur);
console.log(Object.isSealed(utilisateur)); // true

// Empêche toute modification (mais pas des objets imbriqués)
Object.freeze(utilisateur);
console.log(Object.isFrozen(utilisateur)); // true

// Les objets imbriqués peuvent encore être modifiés
utilisateur.adresse.ville = 'Lyon'; // Fonctionne !

// Pour un gel complet, il faudrait appliquer freeze récursivement
function deepFreeze(obj) {
    // Récupère les noms de propriétés définies sur obj
    const propNames = Object.getOwnPropertyNames(obj);

    // Gèle les propriétés avant de geler obj
    for (const name of propNames) {
        const value = obj[name];
        if (value && typeof value === "object") {
            deepFreeze(value);
        }
    }
}
```



```
return Object.freeze(obj);  
}  
  
deepFreeze(utilisateur);  
  
// Maintenant, même les objets imbriqués sont gelés  
  
utilisateur.adresse.ville = 'Marseille'; // Ignoré
```

Les objets JavaScript sont des structures de données extrêmement flexibles et puissantes qui constituent la base de nombreux aspects du langage. Comprendre comment créer, manipuler et optimiser les objets est essentiel pour tout développeur JavaScript.

Les concepts clés à retenir :

- Les objets sont des collections de paires clé-valeur
- Ils peuvent être créés et modifiés dynamiquement
- Les méthodes sont des fonctions stockées comme propriétés d'objets
- Le mot-clé `this` fait référence à l'objet courant dans les méthodes
- JavaScript offre diverses façons de manipuler et contrôler les propriétés des objets

Dans les sections suivantes, nous explorerons des concepts plus avancés comme les prototypes, l'héritage, et des modèles de conception orientés objet en JavaScript.

EXERCICES PRATIQUES

1. Exercice de base : Créez un objet "livre" avec les propriétés titre, auteur, année, et une méthode qui renvoie une description du livre.
2. Exercice intermédiaire : Créez un objet "panier" qui permet d'ajouter des produits, de calculer le total, et d'appliquer des réductions.
3. Exercice avancé : Implémentez un système de gestion d'utilisateurs avec des méthodes pour ajouter, modifier et supprimer des utilisateurs, en utilisant des propriétés privées et des accesseurs pour la validation.

4. Défi : Créez une structure d'objet immuable profonde qui garantit qu'aucune partie de l'objet ne peut être modifiée après sa création.

4.1.2. INTRODUCTION AUX TABLEAUX

QU'EST-CE QU'UN TABLEAU EN JAVASCRIPT ?

Un tableau (array) en JavaScript est une structure de données qui permet de stocker et d'organiser plusieurs valeurs dans une seule variable. Contrairement à d'autres langages de programmation, les tableaux JavaScript sont dynamiques (leur taille peut changer) et hétérogènes (ils peuvent contenir des valeurs de différents types).

Les tableaux JavaScript :

- Sont des objets spéciaux avec des méthodes et propriétés dédiées
- Utilisent des indices numériques commençant à 0
- Peuvent contenir n'importe quel type de données (nombres, chaînes, objets, fonctions, et même d'autres tableaux)
- Sont redimensionnables dynamiquement (peuvent croître ou rétrécir)
- Sont transmis par référence (et non par valeur)

CRÉATION DE TABLEAUX

LITTÉRAL DE TABLEAU

La façon la plus courante de créer un tableau est d'utiliser la syntaxe littérale avec crochets :

```
// Tableau vide
```

```
const tableauVide = [];
```

```
// Tableau avec des valeurs
```

```
const nombres = [1, 2, 3, 4, 5];
```

```
// Tableau avec différents types de données
```

```
const mixte = [42, "texte", true, { nom: "Objet" }, [1, 2, 3], function() { return "Hello"; }];

// Tableau avec des "trous" (éviter cette pratique)

const avecTrous = [1, , 3]; // L'élément à l'indice 1 est undefined
```

CONSTRUCTEUR ARRAY

On peut également utiliser le constructeur Array() :

```
// Tableau vide

const tableau1 = new Array();

// Tableau avec des valeurs

const tableau2 = new Array(1, 2, 3, 4, 5);

// Attention : comportement spécial avec un seul argument numérique

const tableau3 = new Array(5); // Crée un tableau de 5 emplacements vides (longueur 5)
```

MÉTHODES DE CRÉATION SPÉCIALES

```
// Array.of() - crée un tableau à partir des arguments

const tableau4 = Array.of(5); // [5] (un seul élément, contrairement à new Array(5))

const tableau5 = Array.of(1, 2, 3); // [1, 2, 3]
```

```
// Array.from() - crée un tableau à partir d'un itérable ou array-like

const tableau6 = Array.from("hello"); // ["h", "e", "l", "l", "o"]

const tableau7 = Array.from(new Set([1, 2, 2, 3, 3])); // [1, 2, 3]
```

```
// Array.from() avec une fonction de mapping
const tableau8 = Array.from([1, 2, 3], x => x * 2); // [2, 4, 6]
```

ACCÈS AUX ÉLÉMENTS DU TABLEAU

Les éléments d'un tableau sont indexés à partir de 0 :

```
const fruits = ["pomme", "banane", "orange", "kiwi"];
```

// Accès par indice

```
const premierFruit = fruits[0]; // "pomme"
```

```
const dernierFruit = fruits[3]; // "kiwi"
```

// Indice négatif (non supporté nativement)

// fruits[-1] ne fonctionne pas comme dans certains langages

// Accès au dernier élément

```
const dernier = fruits[fruits.length - 1]; // "kiwi"
```

// Avec la méthode at() (ES2022)

```
const dernierFruitAt = fruits.at(-1); // "kiwi"
```

```
const premierFruitAt = fruits.at(0); // "pomme"
```

PROPRIÉTÉS ET MÉTADONNÉES

LONGUEUR DU TABLEAU

```
const nombres = [10, 20, 30, 40, 50];
```

```
console.log(nombres.length); // 5
```

```
// Modifier la longueur (tronquer le tableau)
```

```
nombres.length = 3;
```

```
console.log(nombres); // [10, 20, 30]
```

```
// Augmenter la longueur (crée des "trous")
```

```
nombres.length = 5;
```

```
console.log(nombres); // [10, 20, 30, undefined, undefined]
```

VÉRIFIER SI UNE VARIABLE EST UN TABLEAU

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

```
const obj = { a: 1, b: 2 };
```

```
console.log(Array.isArray(arr)); // true
```

```
console.log(Array.isArray(obj)); // false
```

MODIFICATION DES TABLEAUX

AJOUT ET SUPPRESSION D'ÉLÉMENTS

```
const fruits = ["pomme", "orange"];
```

```
// Ajouter à la fin
```

```
fruits.push("banane", "kiwi"); // Retourne la nouvelle longueur: 4
```

```
console.log(fruits); // ["pomme", "orange", "banane", "kiwi"]
```

```
// Supprimer le dernier élément
const dernier = fruits.pop(); // "kiwi"
console.log(fruits); // ["pomme", "orange", "banane"]

// Ajouter au début
fruits.unshift("fraise"); // Retourne la nouvelle longueur: 4
console.log(fruits); // ["fraise", "pomme", "orange", "banane"]
```

```
// Supprimer le premier élément
const premier = fruits.shift(); // "fraise"
console.log(fruits); // ["pomme", "orange", "banane"]
```

MODIFICATION À UN INDICE SPÉCIFIQUE

```
const nombres = [10, 20, 30, 40, 50];
```

```
// Modification par indice
nombres[2] = 35;
console.log(nombres); // [10, 20, 35, 40, 50]
```

```
// Affectation à un indice au-delà de la longueur (crée des "trous")
nombres[7] = 80;
console.log(nombres); // [10, 20, 35, 40, 50, undefined, undefined, 80]
console.log(nombres.length); // 8
```

MÉTHODES SPLICE ET SLICE

```
// splice() - modifie le tableau original

const mois = ["Jan", "Mars", "Avril", "Juin"];

// Insérer à l'indice 1

mois.splice(1, 0, "Fév");

console.log(mois); // ["Jan", "Fév", "Mars", "Avril", "Juin"]
```

```
// Remplacer à l'indice 4

mois.splice(4, 1, "Mai");

console.log(mois); // ["Jan", "Fév", "Mars", "Avril", "Mai"]
```

```
// Supprimer 2 éléments à partir de l'indice 2

const supprimés = mois.splice(2, 2);

console.log(supprimés); // ["Mars", "Avril"]

console.log(mois); // ["Jan", "Fév", "Mai"]
```

```
// slice() - crée une copie superficielle (ne modifie pas l'original)

const nombres = [10, 20, 30, 40, 50];

const partie = nombres.slice(1, 4); // De l'indice 1 (inclus) à 4 (exclus)

console.log(partie); // [20, 30, 40]

console.log(nombres); // [10, 20, 30, 40, 50] (non modifié)
```

```
// Copie complète avec slice()

const copie = nombres.slice();

console.log(copie); // [10, 20, 30, 40, 50] (nouvelle référence)
```

RECHERCHE DANS LES TABLEAUX

VÉRIFIER LA PRÉSENCE D'UN ÉLÉMENT

```
const fruits = ["pomme", "banane", "orange", "kiwi"];
```

// includes() - vérification d'existence (ES7)

```
console.log(fruits.includes("banane")); // true
```

```
console.log(fruits.includes("fraise")); // false
```

// includes() avec position de départ

```
console.log(fruits.includes("banane", 2)); // false (commence à chercher depuis l'indice 2)
```

// indexOf() - trouver la position (ou -1 si non trouvé)

```
console.log(fruits.indexOf("orange")); // 2
```

```
console.log(fruits.indexOf("fraise")); // -1
```

// lastIndexOf() - trouver la dernière occurrence

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

```
console.log(nombres.lastIndexOf(2)); // 3
```

RECHERCHE AVEC CONDITION

```
const personnes = [
```

```
  { nom: "Alice", age: 25 },
```

```
  { nom: "Bob", age: 30 },
```

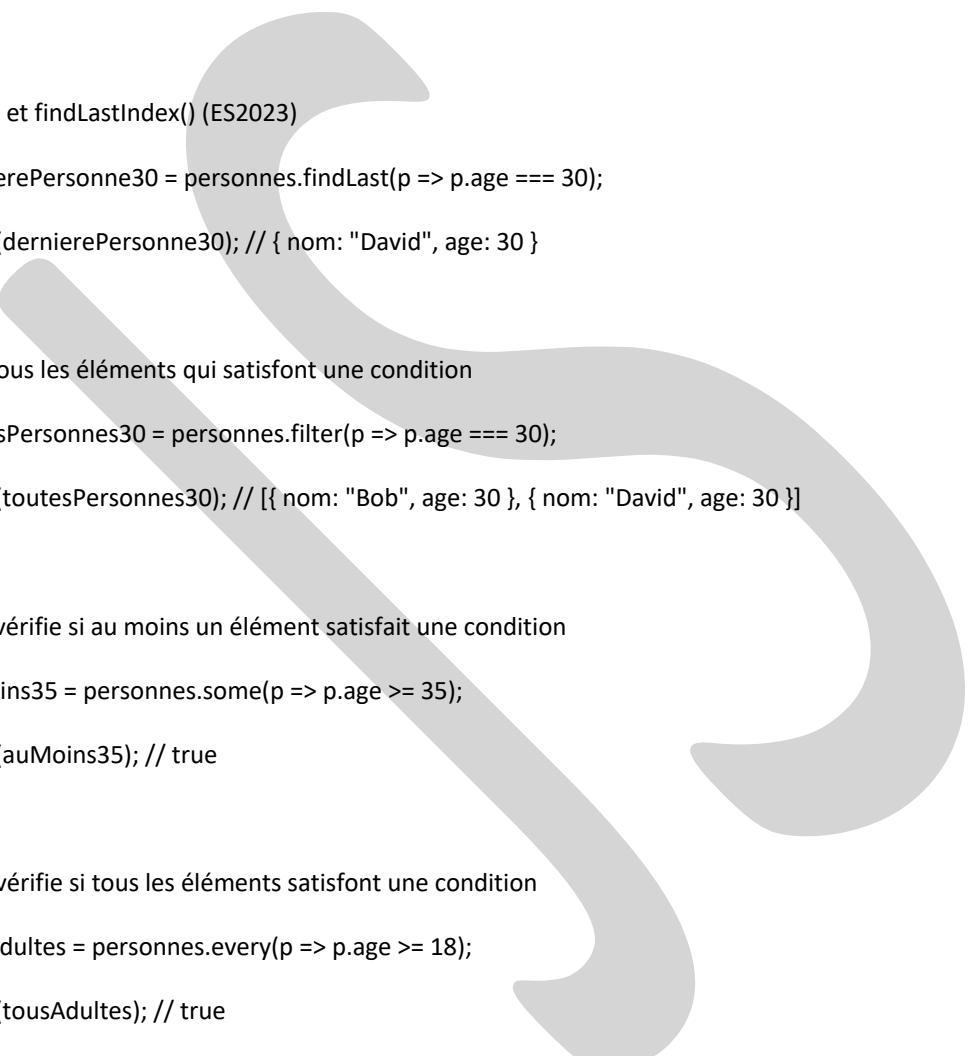
```
  { nom: "Charlie", age: 35 },
```

```
  { nom: "David", age: 30 }
```

```
];
```

```
// find() - premier élément qui satisfait une condition
const personne30 = personnes.find(p => p.age === 30);
console.log(personne30); // { nom: "Bob", age: 30 }
```

```
// findIndex() - indice du premier élément qui satisfait une condition
const indice30 = personnes.findIndex(p => p.age === 30);
console.log(indice30); // 1
```



```
// findLast() et findLastIndex() (ES2023)
const dernierePersonne30 = personnes.findLast(p => p.age === 30);
console.log(dernierePersonne30); // { nom: "David", age: 30 }
```

```
// filter() - tous les éléments qui satisfont une condition
const toutesPersonnes30 = personnes.filter(p => p.age === 30);
console.log(toutesPersonnes30); // [{ nom: "Bob", age: 30 }, { nom: "David", age: 30 }]
```

```
// some() - vérifie si au moins un élément satisfait une condition
const auMoins35 = personnes.some(p => p.age >= 35);
console.log(auMoins35); // true
```

```
// every() - vérifie si tous les éléments satisfont une condition
const tousAdultes = personnes.every(p => p.age >= 18);
console.log(tousAdultes); // true
```

TRANSFORMATION DE TABLEAUX

MAPPING ET TRANSFORMATION

```
const nombres = [1, 2, 3, 4, 5];

// map() - créer un nouveau tableau en appliquant une fonction à chaque élément

const doubles = nombres.map(x => x * 2);

console.log(doubles); // [2, 4, 6, 8, 10]
```

```
// map() avec indice

const avecIndices = nombres.map((valeur, indice) => `${indice}: ${valeur}`);

console.log(avecIndices); // ["0: 1", "1: 2", "2: 3", "3: 4", "4: 5"]
```

RÉDUCTION (AGGREGATION)

```
const nombres = [1, 2, 3, 4, 5];

// reduce() - réduire le tableau à une seule valeur

const somme = nombres.reduce((acc, valeur) => acc + valeur, 0);

console.log(somme); // 15
```

```
// reduce() pour construire un objet

const personnes = [
  { id: 1, nom: "Alice" },
  { id: 2, nom: "Bob" },
  { id: 3, nom: "Charlie" }
];

const parId = personnes.reduce((acc, personne) => {
  acc[personne.id] = personne;
  return acc;
}, {});

console.log(parId); // { 1: {id: 1, nom: "Alice"}, 2: {id: 2, nom: "Bob"}, 3: {id: 3, nom: "Charlie"} }
```

```
// reduceRight() - comme reduce() mais de droite à gauche

const mots = ["le", "chat", "est", "noir"];

const phrase = mots.reduceRight((acc, mot) => `${acc} ${mot}`, "");

console.log(phrase.trim()); // "noir est chat le"
```

PLANAGE (FLATTENING)

```
const imbriqué = [1, [2, 3], [4, [5, 6]]];
```

```
// flat() - aplatis un tableau imbriqué (ES2019)

const plat1 = imbriqué.flat();

console.log(plat1); // [1, 2, 3, 4, [5, 6]]
```

```
// flat() avec profondeur

const plat2 = imbriqué.flat(2);

console.log(plat2); // [1, 2, 3, 4, 5, 6]
```

```
// flatMap() - map suivi de flat(1)

const commandes = [
  { id: 1, produits: ["pomme", "orange"] },
  { id: 2, produits: ["banane"] },
  { id: 3, produits: ["kiwi", "fraise", "ananas"] }
];

const tousLesProduits = commandes.flatMap(cmd => cmd.produits);

console.log(tousLesProduits); // ["pomme", "orange", "banane", "kiwi", "fraise", "ananas"]
```

TRI ET ORDRE DES TABLEAUX

MÉTHODES DE TRI

```
const fruits = ["banane", "pomme", "Orange", "kiwi"];

// sort() - tri lexicographique (attention: modifie le tableau original)
fruits.sort();

console.log(fruits); // ["Orange", "banane", "kiwi", "pomme"] (tri ASCII, majuscules avant minuscules)

// sort() avec fonction de comparaison

const nombres = [10, 5, 8, 1, 30];

nombres.sort((a, b) => a - b); // Tri numérique croissant

console.log(nombres); // [1, 5, 8, 10, 30]

// Tri décroissant

nombres.sort((a, b) => b - a);

console.log(nombres); // [30, 10, 8, 5, 1]

// Tri d'objets

const personnes = [
  { nom: "Alice", age: 25 },
  { nom: "Bob", age: 30 },
  { nom: "Charlie", age: 20 }
];

personnes.sort((a, b) => a.age - b.age);

console.log(personnes); // [Charlie, Alice, Bob] (triés par âge)

// reverse() - inverser l'ordre

const lettres = ["a", "b", "c", "d"];
```

```
lettres.reverse();

console.log(lettres); // ["d", "c", "b", "a"]
```

TRI SANS MODIFIER L'ORIGINAL

```
const nombres = [10, 5, 8, 1, 30];

// Tri sans modifier le tableau original

const triés = [...nombres].sort((a, b) => a - b);

console.log(triés); // [1, 5, 8, 10, 30]

console.log(nombres); // [10, 5, 8, 1, 30] (non modifié)
```

// Autre méthode: slice() puis sort()

```
const autreTriés = nombres.slice().sort((a, b) => a - b);
```

TRI STABLE (STABLE SORT)

```
const etudiants = [
  { nom: "Alice", classe: "A", note: 15 },
  { nom: "Bob", classe: "B", note: 12 },
  { nom: "Charlie", classe: "A", note: 12 },
  { nom: "David", classe: "B", note: 15 }
];
```

// Trier d'abord par note (tri secondaire)

```
etudiants.sort((a, b) => b.note - a.note);
```

// Puis par classe (tri primaire)

```
// Le tri est stable, donc l'ordre relatif des notes est préservé
etudiants.sort((a, b) => {
  if (a.classe < b.classe) return -1;
  if (a.classe > b.classe) return 1;
  return 0;
});
```

```
console.log(etudiants);
// Résultat:
// [
//   { nom: "Alice", classe: "A", note: 15 },
//   { nom: "Charlie", classe: "A", note: 12 },
//   { nom: "David", classe: "B", note: 15 },
//   { nom: "Bob", classe: "B", note: 12 }
// ]
```

ITÉRATION SUR LES TABLEAUX

MÉTHODES D'ITÉRATION

```
const nombres = [1, 2, 3, 4, 5];

// forEach() - exécute une fonction pour chaque élément
nombres.forEach((valeur, indice) => {
  console.log(`nombres[${indice}] = ${valeur}`);
});

// for...of - itération sur les valeurs
```

```

for (const nombre of nombres) {
  console.log(nombre);
}

// for...in - itération sur les indices (déconseillé pour les tableaux)
for (const indice in nombres) {
  console.log(`indice: ${indice}, valeur: ${nombres[indice]}`);
}

// Boucle for classique
for (let i = 0; i < nombres.length; i++) {
  console.log(nombres[i]);
}

```

FONCTIONS QUI RETOURNENT UN ITÉRATEUR

```

const fruits = ["pomme", "banane", "orange"];

// entries() - itérateur sur [indice, valeur]
for (const [indice, fruit] of fruits.entries()) {
  console.log(`fruits[${indice}] = ${fruit}`);
}

// keys() - itérateur sur les indices
for (const indice of fruits.keys()) {
  console.log(indice); // 0, 1, 2
}

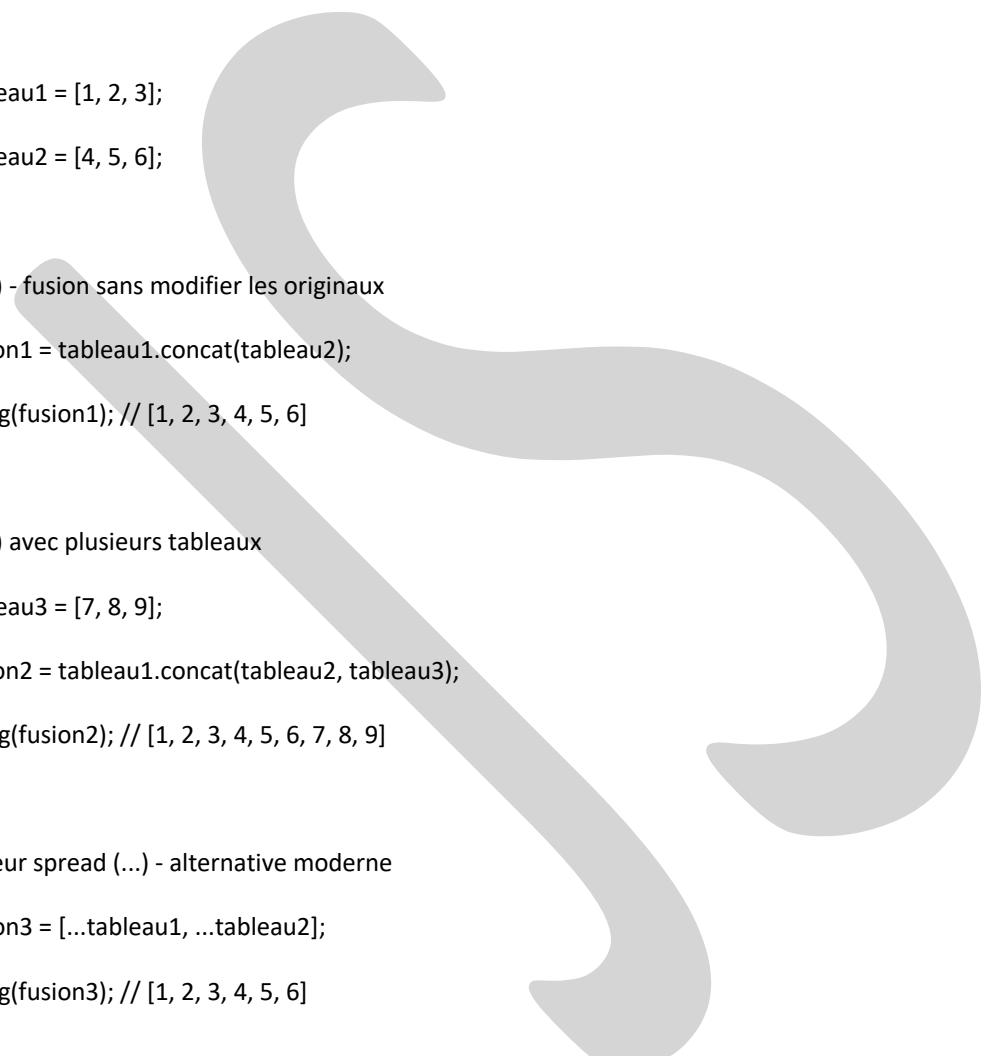
// values() - itérateur sur les valeurs

```

```
for (const fruit of fruits.values()) {
    console.log(fruit); // "pomme", "banane", "orange"
}
```

OPÉRATIONS AVANCÉES SUR LES TABLEAUX

FUSION DE TABLEAUX



```
const tableau1 = [1, 2, 3];
const tableau2 = [4, 5, 6];

// concat() - fusion sans modifier les originaux
const fusion1 = tableau1.concat(tableau2);
console.log(fusion1); // [1, 2, 3, 4, 5, 6]
```

```
// concat() avec plusieurs tableaux
const tableau3 = [7, 8, 9];
const fusion2 = tableau1.concat(tableau2, tableau3);
console.log(fusion2); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
// Opérateur spread (...) - alternative moderne
const fusion3 = [...tableau1, ...tableau2];
console.log(fusion3); // [1, 2, 3, 4, 5, 6]
```

REMPLISSAGE DE TABLEAU

```
// fill() - remplit un tableau avec une valeur statique
const tableau = [1, 2, 3, 4, 5];
tableau.fill(0);
```

```
console.log(tableau); // [0, 0, 0, 0]

// fill() avec début et fin
tableau.fill(1, 2, 4); // Remplit avec 1 de l'indice 2 (inclus) à 4 (exclus)
console.log(tableau); // [0, 0, 1, 1, 0]
```

// Array(n).fill() pour créer un tableau initialisé

```
const zeros = Array(5).fill(0);
```

```
console.log(zeros); // [0, 0, 0, 0, 0]
```

CONVERSION ENTRE TABLEAUX ET CHAÎNES

```
const fruits = ["pomme", "banane", "orange"];
```

// join() - convertir un tableau en chaîne

```
const texte = fruits.join(", ");
```

```
console.log(texte); // "pomme, banane, orange"
```

// split() - convertir une chaîne en tableau

```
const chaîne = "rouge,vert,bleu";
```

```
const couleurs = chaîne.split(",");
```

```
console.log(couleurs); // ["rouge", "vert", "bleu"]
```

// toString() - convertir en chaîne (séparateur = virgule)

```
const chaîne2 = fruits.toString();
```

```
console.log(chaîne2); // "pomme,banane,orange"
```

TABLEAUX TYPÉS (TYPEDARRAYS)

JavaScript propose également des tableaux typés pour manipuler des données binaires :

```
// Int8Array - tableau d'entiers signés 8 bits
const int8 = new Int8Array(4);
int8[0] = 127;
int8[1] = -128;
console.log(int8); // Int8Array [127, -128, 0, 0]
```

```
// Uint8Array - tableau d'entiers non signés 8 bits
const uint8 = new Uint8Array([1, 2, 3, 4]);
console.log(uint8); // Uint8Array [1, 2, 3, 4]
```

```
// Float32Array - tableau de nombres à virgule flottante 32 bits
const float32 = new Float32Array(3);
float32[0] = 1.5;
console.log(float32); // Float32Array [1.5, 0, 0]
```

```
// Autres types disponibles:
// Int16Array, Uint16Array, Int32Array, Uint32Array, Float64Array, BigInt64Array, BigUint64Array
```

Les tableaux sont l'une des structures de données les plus fondamentales en JavaScript et sont utilisés dans presque tous les programmes. Leur polyvalence et les nombreuses méthodes intégrées en font un outil puissant pour manipuler et transformer des collections de données.

Les points clés à retenir :

- Les tableaux JavaScript sont dynamiques et peuvent contenir des éléments de différents types
- JavaScript offre une riche API pour manipuler les tableaux: recherche, tri, filtrage, transformation
- De nombreuses opérations sur les tableaux peuvent être chaînées pour des transformations élégantes

- ☞ Les méthodes modernes comme map, filter, reduce permettent un code plus expressif et fonctionnel
- ☞ Il est important de comprendre quelles méthodes modifient le tableau d'origine et lesquelles créent une copie

Dans les sections suivantes, nous explorerons des structures de données plus spécialisées comme les ensembles (Set) et les maps (Map), ainsi que des modèles avancés pour travailler avec des collections de données en JavaScript.

EXERCICES PRATIQUES

1. Exercice de base : Créez une fonction qui prend un tableau de nombres et retourne un nouveau tableau contenant uniquement les nombres pairs, multipliés par deux.
2. Exercice intermédiaire : Implémentez une fonction qui prend un tableau d'objets avec des propriétés "nom" et "valeur", et retourne un objet où les noms sont les clés et les valeurs sont les valeurs associées.
3. Exercice avancé : Créez une fonction qui prend un tableau d'objets avec une propriété "catégorie" et retourne un objet où les clés sont les catégories et les valeurs sont des tableaux d'objets appartenant à cette catégorie.
4. Défi : Implémentez une fonction groupBy qui prend un tableau et une fonction clé, et retourne un objet où les clés sont les résultats de la fonction appliquée à chaque élément, et les valeurs sont des tableaux d'éléments correspondant à cette clé.