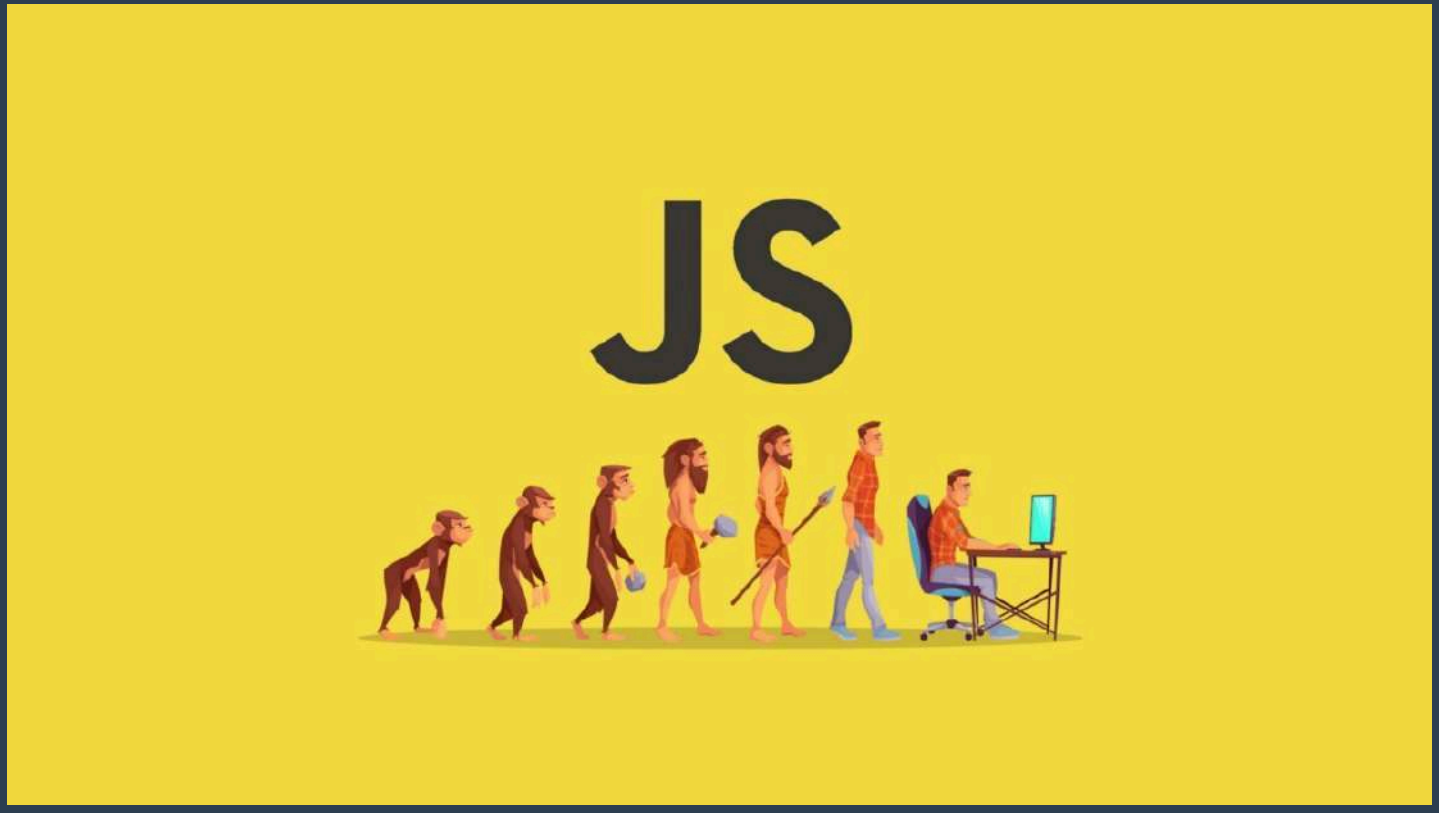
JAVASCRIPT



Steve Lebleu - 2CE-X75-A - 2024-2025

Konfer.

INTRODUCTION

Javascript est un langage de programmation de haut niveau, interprété et *orienté objet*.

Il est principalement utilisé pour rendre les pages web dynamiques et interactives.

C'est le langage le plus utilisé au monde. Il est présent dans la quasi-totalité des sites web.

Vanilla JS est un terme utilisé pour désigner le Javascript pur, sans aucun framework ou bibliothèque.

UN PEU D'HISTOIRE

- Créé en 1995 par Brendan Eich
- Initialement nommé LiveScript
- Largement adopté par les navigateurs web
- Évolution constante avec ECMAScript

FORCES

- Polyvalence
- Grosse grosse communauté
- Ecosystème badass
- Syntaxe relativement simple
- Programmation événementielle et asynchronisme
- Interactions avec le DOM

FAIBLESSES

- Typage dynamique pas ouf
- Héritage prototypal pas simple
- C'est parfois un peu n'importe quoi...

... Dans la série *"WTF Javascript"* on trouve ...

... la coercition de type du seigneur ...

```
1 console.log([] + {}); // "[object Object]"
2 console.log({} + []); // 0
```

... *null* est un objet ...

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

... mieux ... *Array* est un objet ...

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

... l'hypothétique opérateur == ...

```
1 console.log(0 == false); // true
2 console.log('' == 0);    // true
3 console.log([] == false); // true
4 console.log([] == ![]);   // true
```

... les errements du *this* contextuel ...

```
1 const obj = {
2   method: function () {
3     console.log(this);
4   },
5 };
6 obj.method(); // obj
7 const method = obj.method;
8 method(); // undefined (ou `window` dans certains contextes)
```

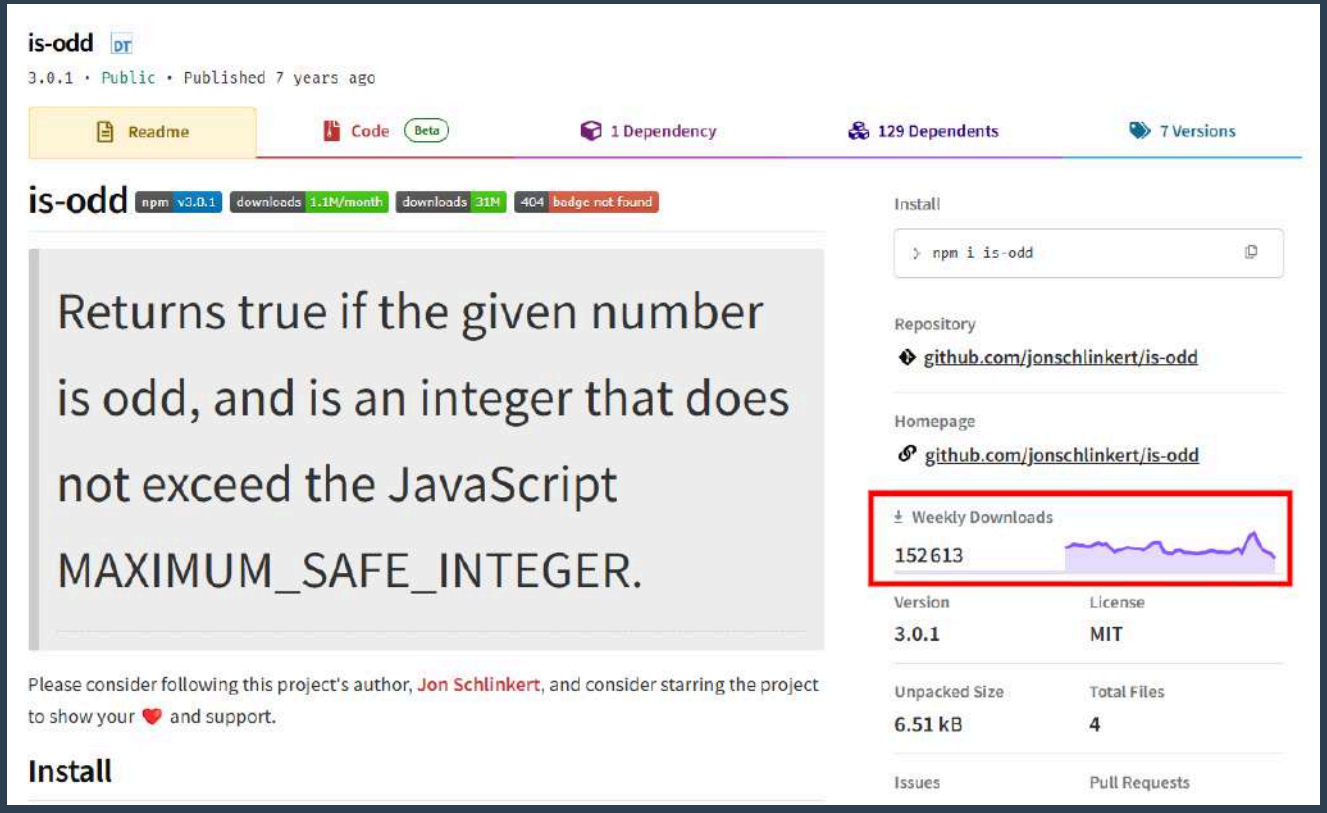
... le désormais célèbre *"callbacks hell"*...

```
1  doSomething((err, result) => {
2    if (err) {
3      handleError(err);
4    } else {
5      doSomethingElse(result, (err, anotherResult) => {
6        if (err) {
7          handleError(err);
8        } else {
9          // etc.
10       }
11     });
12   }
13 });
```

... les casseroles historiques ...

```
1 console.log(parseInt("08")); // 8
2 console.log(parseInt("09")); // 9
3 console.log(0.1 + 0.2 === 0.3); // false
```

... les modules ridicules ...


is-odd
3.0.1 • Public • Published 7 years ago
Readme
Code Beta
1 Dependency
129 Dependents
7 Versions
is-odd
npm v3.0.1
downloads 1.1M/month
downloads 31M
404 badge not found
Returns true if the given number is odd, and is an integer that does not exceed the JavaScript MAXIMUM_SAFE_INTEGER.
Please consider following this project's author, Jon Schlinkert, and consider starring the project to show your ❤ and support.
Install
Install
npm i is-odd
Repository
github.com/jonschlinkert/is-odd
Homepage
github.com/jonschlinkert/is-odd
Weekly Downloads
152613
Version
3.0.1
License
MIT
Unpacked Size
6.51 kB
Total Files
4
Issues
Pull Requests

... le cauchemar des modules avant ESM ...

... l'écosystème fragmenté ...

... l'obsolescence des librairies ...

... et bien d'autres joyeusetés...

"Javascript est comme une voiture sans freins... qui essaie de te convaincre qu'avoir des freins, c'est pour les faibles."


Why We Should Stop Using JavaScript According to Dougla...

... mais bon ...

... il y a plein de trucs cools aussi...

... par exemple Typescript ...

... et puis ...

... on peut se consoler en se disant que c'est pas PHP ...

... et puis surtout ...

... il n'y a pas mieux sur le marché ...

ENVIRONNEMENT(S)

Le développement JS moderne se fait principalement dans des environnements de type Node.js.

- Node.js + NPM (ou autre)
- Modules
- Outils de build
- Outils de formatage
- Outils de tests
- Outils de débogage

GESTIONNAIRES DE PAQUETS

- npm - Le gestionnaire par défaut
- yarn - Alternative populaire
- pnpm - Gestion optimisée de l'espace

OUTILS DE BUILD

- Webpack - Bundler le plus populaire
- Vite - Build tool moderne et rapide
- Rollup - Excellent pour les librairies
- Parcel - Configuration "zero-config"

LINTERS & FORMATTERS

- ESLint - Analyse statique du code
- Prettier - Formattage automatique
- StandardJS - Style guide avec règles prédéfinies

TESTS UNITAIRES

- Jest - Framework de test complet
- Vitest - Alternative rapide compatible Vite
- Mocha - Framework flexible

TESTS E2E

- Cypress - Tests modernes dans le navigateur
- Playwright - Alternative multi-navigateurs
- Selenium - Solution historique

DEBUGGING

- Chrome DevTools Debugger
- VSCode Debugger
- Source Maps
- console.log() (lol... ou pas)

CONSOLE

La console est ton amie.

AFFICHER DES MESSAGES

```
1 console.log('Hello, world!');
```

AFFICHER DES AVERTISSEMENTS

```
1 console.warn('Ceci est un avertissement!');
```

AFFICHER DES ERREURS

```
1 console.error('Ceci est une erreur!');
```

AFFICHER DES INFORMATIONS

```
1 console.info('Ceci est une information.');
```

AFFICHER DES OBJETS

```
1 const obj = { name: 'John', age: 30 };
2 console.log(obj);
```

GROUPES DE MESSAGES

```
1 console.group('Groupe 1');
2 console.log('Message 1');
3 console.log('Message 2');
4 console.groupEnd();
```

CHRONOMÉTRER DES OPÉRATIONS

```
1 console.time('Timer');
2 // code à chronométrer
3 console.timeEnd('Timer');
```

IDE'S

VS CODE

- Éditeur de code gratuit et open-source
- Extensions JS essentielles:
  - ESLint
  - Prettier
  - JavaScript (ES6) code snippets
  - Chrome debugger

UTILISATION

- Inline
  - Attribut `href` d'un lien
  - Attribut d'événement
- Intégré dans le HTML
- Externe via un fichier .js

ATTRIBUT href

```
1 <a href="javascript:alert('Hello, world!');">Cliquez ici</a>
```

ATTRIBUT D'ÉVÉNEMENT

```
1 <span onclick="alert('Hello, world!');">Cliquez ici</span>
```

INTÉGRATION HTML

```
<script>
  // code here
</script>
```

INTÉGRATION HTML

```
<script>
  // code here
</script>
```

CHARGEMENT DE FICHIER

```
1 <script src="./my-script.js"></script>
```

Les scripts sont exécutés dans l'ordre d'apparition dans le document.

type="module"

```
1 <script src="./my-script.js" type="module"></script>
```

La voie moderne pour charger un script JS.

- Intrinsèqument lié aux modules (ES6)
- Différé automatiquement (aka `defer`)
- Scope limité au module
- Feature `import` / `export`
- Chargé en mode strict avec une politique CORS
- Chemin relatif obligatoire
- Supporté par tous les navigateurs modernes
- Sinon fallback avec `nomodule`

```
// utils.js
export function greet(name) {
  return `Hello, ${name}`;
}

// my-script.js
import { greet } from './utils.js';

console.log(greet("Alice"));
```

type="text/plain"

```
<script type="text/plain">
  // code here
</script>
```

- Pour inclure du code non JS
- Pratique pour gérer des templates

```
<!-- Inclusion d'un modèle HTML -->
<script id="template" type="text/plain">
  <div class="user-card">
    <h2>{{name}}</h2>
    <p>Email: {{email}}</p>
  </div>
</script>
```

```
<!-- Script pour utiliser le modèle -->
<script>
  // Récupérer le contenu du script
  const templateElement = document.getElementById('template');
  const template = templateElement.textContent;
</script>
```

type="application/json"

```
<script type="application/json">
  // code here
</script>
```

- Pour inclure des données JSON
- Facilement récupérable via Javascript

```
<script id="data-json" type="application/json">
  {
    "users": [
      { "id": 1, "name": "Alice", "email": "alice@test.com" },
      { "id": 2, "name": "Bob", "email": "bob@test.com" }
    ]
  }
</script>
```

```
<script>
  // Récupérer le contenu du script JSON
  const jsonDataElement = document.getElementById('data-json');
  const jsonData = JSON.parse(jsonDataElement.textContent);
</script>
```

ATTRIBUTS DE CHARGEMENT HISTORIQUES

defer

```
1 <script src="./my-script.js" defer=""></script>
```

- Chargement différé et ordonné des scripts
- Chargement en parallèle
- Amélioration des performances
  - Réduction du temps de blocage lié au parsing
  - Evite le *render blocking* des scripts head
- Compatible avec tous les navigateurs
- Simple, explicite et efficace

- Pas nécessaire si...
  - Les scripts sont placés en fin de page
  - Les scripts sont chargés en tant que modules

async

```
1 <script src="./my-script.js" async=""></script>
```

- Charge les scripts en arrière-plan
- Exécution immédiate après le chargement
- Utile quand les scripts n'ont pas de dépendances

Caractéristique	`async`	`defer`
Chargement	En parallèle avec le DOM	En parallèle avec le DOM
Exécution	Dès que le script est chargé	Après le parsing complet du DOM
Ordre	Non garanti	Garanti

COMPATIBILITÉ

- La majorité des features ES6+ est supportée par les browsers modernes.
- Possible d'utiliser des polyfills pour les navigateurs plus anciens.

- Caniuse
- Babel
- Browserlist

BONNES PRATIQUES

- Préférer `type="module"`
- Fichiers externes pour séparer le code de la structure.
- Concaténation, minification et obfuscation de code pour les performances.

VARIABLES

DÉCLARATION

- `var` - A l'ancienne (à éviter)
- `let` - Pour les variables qui changent
- `const` - Pour les constantes

DIFFÉRENCES PRINCIPALES

```
1  // let et const ont une portée de bloc
2  {
3    let x = 1;
4    const y = 2;
5  }
6
7  // console.log(x); // ReferenceError
8  // console.log(y); // ReferenceError
9
10 // var a une portée de fonction
11 function test() {
12   var z = 3;
13 }
14 // console.log(z); // ReferenceError
```

HOISTING

```
1 // var est "remontée"
2 console.log(x); // undefined
3 var x = 5;
4
5 // let et const ne sont pas "remontées"
6 console.log(y); // ReferenceError
7 let y = 5;
```

REDÉCLARATION

```
1 var x = 1;
2 var x = 2; // OK
3
4 let y = 1;
5 // let y = 2; // SyntaxError
6
7 const z = 1;
8 // const z = 2; // SyntaxError
```

RÉASSIGNATION

```
1 var x = 1;
2 x = 2; // OK
3
4 let y = 1;
5 y = 2; // OK
6
7 const z = 1;
8 // z = 2; // TypeError
```

CONSTANTES ET OBJETS

```
1 const user = {
2   name: 'John'
3 };
4
5 user.name = 'Jane'; // OK
6 // user = {}; // TypeError
```

TYPES DE VARIABLES

- Primitifs : string, number, boolean, null, undefined, symbol
- Objets : object, array, function, date, etc.

EXEMPLE DE TYPES

```
1  // Primitifs
2  const name = 'John';              // string
3  const age = 30;                   // number
4  const isActive = true;            // boolean
5  const empty = null;               // null
6  let undefined;                    // undefined
7  const id = Symbol('id');          // symbol
8
9  // Objets
10 const user = { name: 'John' }; // object
11 const numbers = [1, 2, 3];      // array
```

PORTÉE DES VARIABLES

```
1  const global = 'Je suis globale';
2
3  function test() {
4    const locale = 'Je suis locale';
5
6    if (true) {
7      const bloc = 'Je suis dans un bloc';
8      console.log(global);  // OK
9      console.log(locale);  // OK
10     console.log(bloc);    // OK
11   }
12   // console.log(bloc);     // ReferenceError
13 }
```

BONNES PRATIQUES

- Utiliser `const` par défaut
- Utiliser `let` si réassignation nécessaire
- Éviter `var`
- Noms explicites en camelCase
- Éviter les variables globales

FONCTIONS

DÉCLARATION DE FONCTIONS

```
1  // Déclaration classique
2  function direBonjour(nom) {
3    return `Bonjour ${nom}!`;
4  }
5
6  // Expression de fonction
7  const direAuRevoir = function(nom) {
8    return `Au revoir ${nom}!`;
9  };
10
11 // Fonction fléchée (ES6+)
12 const saluer = (nom) => `Salut ${nom}!`;
```

HOISTING

```
1  direBonjour(); // "Bonjour !"
2
3  function direBonjour() {
4    console.log("Bonjour !");
5  }
6
7  direAuRevoir(); // TypeError
8
9  const direAuRevoir = function() {
10   console.log("Au revoir !");
11 };
```

PARAMÈTRES

```
1 // Paramètres par défaut
2 function accueillir(nom = 'visiteur') {
3   return `Bienvenue ${nom}!`;
4 }
5
6 // Rest parameters
7 function somme(...nombres) {
8   return nombres.reduce((total, n) => total + n, 0);
9 }
```

PURE FUNCTIONS

```
1  // Fonction pure
2  function addition(a, b) {
3    return a + b;
4  }
5
6  // Fonction impure
7  let total = 0;
8  function ajouterAuTotal(n) {
9    total += n; // Effet de bord
10   return total;
11 }
```

CLOSURE (FERMETURE)

```
 1 function createCompteur() {
 2   let count = 0;
 3   return {
 4     increment: () => ++count,
 5     getCount: () => count
 6   };
 7 }
 8
 9 const compteur = createCompteur();
10 compteur.increment(); // 1
11 compteur.increment(); // 2
```

this ET CONTEXTE

```
const personne = {
  nom: 'Alice',
  direBonjour: function() {
    return `Bonjour, je suis ${this.nom}`;
  },
  direBonjourFleche: () => {
    // this ne fonctionne pas ici!
    return `Bonjour, je suis ${this.nom}`;
  }
};
```

MÉTHODES DE BINDING

```
1 const personne = {
2   nom: 'Alice',
3   direBonjour: function() {
4     return `Bonjour, je suis ${this.nom}`;
5   }
6 };
7
8 const direBonjour = personne.direBonjour;
9 const direBonjourLie = personne.direBonjour.bind(personne);
```

FONCTIONS D'ORDRE SUPÉRIEUR

```
1  // Fonction qui prend une fonction en paramètre
2  function executerDeuxFois(fn) {
3    fn();
4    fn();
5  }
6
7  // Fonction qui retourne une fonction
8  function multiplierPar(n) {
9    return function(x) {
10     return x * n;
11   };
12 }
13
14 const multiplierParDeux = multiplierPar(2);
```

CURRIED FUNCTIONS

```
1  // Version normale
2  function addition(a, b, c) {
3    return a + b + c;
4  }
5
6  // Version currifiée
7  function additionCurry(a) {
8    return function(b) {
9      return function(c) {
10       return a + b + c;
11     }
12   }
13 }
14
15 // Version currifiée et fléchée
```

AVANTAGES DU CURRYING

```
1  // Création de fonctions spécialisées
2  const ajouterDix = additionCurry(10);
3  const ajouterQuinze = ajouterDix(5);
4
5  console.log(ajouterDix(5)(3));     // 18
6  console.log(ajouterQuinze(3));     // 18
7
8  // Utile pour la composition
9  const multiplierPar = x => y => x * y;
10 const ajouterA = x => y => x + y;
11
12 const calculComplexe = x => ajouterA(5)(multiplierPar(2)(x));
13
14 console.log(calculComplexe(3)); // 11
```

FONCTIONS RÉCURSIVES

```
1 function factorielle(n) {
2   if (n <= 1) return 1;
3   return n * factorielle(n - 1);
4 }
5
6 function fibonacci(n) {
7   if (n <= 1) return n;
8   return fibonacci(n - 1) + fibonacci(n - 2);
9 }
```

BONNES PRATIQUES

- Préférer les fonctions pures
- Une fonction = une responsabilité
- Noms explicites en camelCase
- Éviter les effets de bord
- Limiter le nombre de paramètres
- Documenter les fonctions complexes

OPÉRATEURS

OPÉRATEURS ARITHMÉTIQUES

```
1 let a = 5;
2 let b = 2;
3
4 console.log(a + b);  // Addition: 7
5 console.log(a - b);  // Soustraction: 3
6 console.log(a * b);  // Multiplication: 10
7 console.log(a / b);  // Division: 2.5
8 console.log(a % b);  // Modulo: 1
9 console.log(a ** b); // Puissance: 25
```

OPÉRATEURS D'AFFECTATION

```
let x = 5;
x += 3;  // x = x + 3
x -= 2;  // x = x - 2
x *= 4;  // x = x * 4
x /= 2;  // x = x / 2
x %= 3;  // x = x % 3
x **= 2; // x = x ** 2
```

OPÉRATEURS DE COMPARAISON

```
1 console.log(5 == "5");   // true (égalité simple)
2 console.log(5 === "5");  // false (égalité stricte)
3 console.log(5 != "5");   // false
4 console.log(5 !== "5");  // true
5 console.log(5 > 3);      // true
6 console.log(5 >= 5);     // true
7 console.log(3 < 5);      // true
8 console.log(5 <= 5);     // true
```

OPÉRATEURS LOGIQUES

```
1  // AND (&&)
2  console.log(true && true);   // true
3  console.log(true && false);  // false
4
5  // OR (||)
6  console.log(true || false);  // true
7  console.log(false || false); // false
8
9  // NOT (!)
10 console.log(!true);          // false
11 console.log(!false);         // true
```

OPÉRATEUR NULLISH COALESCING

```
1 const valeur = null;
2 const defaut = "valeur par défaut";
3
4 console.log(valeur ?? defaut);  // "valeur par défaut"
5 console.log(0 ?? defaut);       // 0
6 console.log("" ?? defaut);      // ""
7 console.log(false ?? defaut);   // false
```

OPÉRATEUR DE CHAÎNAGE OPTIONNEL

```
1 const user = {
2   address: {
3     street: "123 rue principale"
4   }
5 };
6
7 console.log(user?.address?.street);  // "123 rue principale"
8 console.log(user?.contact?.email);   // undefined
```

OPÉRATEURS UNAIRES

```
let x = 5;
console.log(+x);     // Conversion en nombre: 5
console.log(-x);     // Négation: -5
console.log(++x);    // Incrémentation: 6
console.log(--x);    // Décrémentation: 5
console.log(typeof x); // "number"
```

OPÉRATEUR SPREAD (...)

```
1 const array1 = [1, 2, 3];
2 const array2 = [...array1, 4, 5];  // [1, 2, 3, 4, 5]
3
4 const obj1 = { x: 1, y: 2 };
5 const obj2 = { ...obj1, z: 3 };    // { x: 1, y: 2, z: 3 }
```

OPÉRATEUR TERNAIRE

```
1  const age = 20;
2  const statut = age >= 18 ? "majeur" : "mineur";
3
4  // Équivalent à:
5  let statut2;
6  if (age >= 18) {
7    statut2 = "majeur";
8  } else {
9    statut2 = "mineur";
10 }
```

BONNES PRATIQUES

- Préférer === à ==
- Attention aux conversions implicites
- Utiliser les parenthèses pour la lisibilité
- Éviter les opérateurs unaires en cascade (++, --)
- Préférer les opérateurs logiques courts

BOUCLES

BOUCLE FOR

```
1 for (let i = 0; i < 5; i++) {
2   console.log(i);
3 }
```

BOUCLE FOR...OF

```
1 const fruits = ['pomme', 'banane', 'orange'];
2 for (const fruit of fruits) {
3   console.log(fruit);
4 }
```

BOUCLE FOR...IN

```
1 const user = {
2   name: 'Alice',
3   age: 30
4 };
5 for (const key in user) {
6   console.log(`${key}: ${user[key]}`);
7 }
```

BOUCLE WHILE

```
1 let i = 0;
2 while (i < 5) {
3   console.log(i);
4   i++;
5 }
```

BOUCLE DO...WHILE

```
1 let i = 0;
2 do {
3   console.log(i);
4   i++;
5 } while (i < 5);
```

INSTRUCTIONS DE CONTRÔLE

```
1  // break
2  for (let i = 0; i < 5; i++) {
3    if (i === 3) break;
4    console.log(i);
5  }
6
7  // continue
8  for (let i = 0; i < 5; i++) {
9    if (i === 3) continue;
10   console.log(i);
11 }
```

ÉTIQUETTES DE BOUCLE

```
1 externe: for (let i = 0; i < 3; i++) {
2   for (let j = 0; j < 3; j++) {
3     if (i === 1 && j === 1) break externe;
4     console.log(i, j);
5   }
6 }
```

BONNES PRATIQUES

- Préférer les boucles natives (for, while) pour les performances
- Préférer For...of à For...in
- Éviter les boucles infinies
- Utiliser les instructions de contrôle avec parcimonie
- Éviter les étiquettes de boucle

OBJETS

En JS, à part les types primitifs, presque tout est objet.

- Objet global `window`
- Objets natifs (`Date`, `Math`, etc.)
- Prototypes
- Objets littéraux
- Instances de classes

window

Le coeur des API Javascript dans le navigateur.

```
1 console.log(window); // Enjoy Window { ... }
```

- Objet global
- Propriétés et méthodes globales

TOUT Javascript est accessible via l'objet `window`.

OBJETS NATIFS

Objets fournis par le langage Javascript.

- `Date`
- `Math`
- `Array`
- `String`
- `Number`
- `Boolean`
- `Function`
- `Object`

PROTOTYPES

- Ancêtres historiques du modèle objet JS
- Chaque objet a un prototype
- Permet l'héritage
- Permet de partager des propriétés et méthodes

CRÉATION D'UN OBJET

```
1  function Personne(nom) {
2    this.nom = nom;
3  }
4
5  Personne.prototype.direBonjour = function() {
6    return `Bonjour, je suis ${this.nom}`;
7  };
8
9  const alice = new Personne('Alice');
10
11 console.log(alice.direBonjour()); // "Bonjour, je suis Alice"
```

CRÉATION D'UN OBJET

```
1  function Personne(nom) {
2    this.nom = nom;
3  }
4
5  Personne.prototype.direBonjour = function() {
6    return `Bonjour, je suis ${this.nom}`;
7  };
8
9  const alice = new Personne('Alice');
10
11 console.log(alice.direBonjour()); // "Bonjour, je suis Alice"
```

CRÉATION D'UN OBJET

```
1  function Personne(nom) {
2    this.nom = nom;
3  }
4
5  Personne.prototype.direBonjour = function() {
6    return `Bonjour, je suis ${this.nom}`;
7  };
8
9  const alice = new Personne('Alice');
10
11 console.log(alice.direBonjour()); // "Bonjour, je suis Alice"
```

MÉTHODES DE PROTOTYPES

```
1  // Vérifier si une propriété est héritée
2  console.log(personne.hasOwnProperty('nom')); // true
3  console.log(personne.propertyIsEnumerable('nom')); // true
4
5  // Obtenir le prototype
6  console.log(Object.getPrototypeOf(personne)); // {}
7
8  // Convertir en chaîne
9  console.log(personne.toString()); // '[object Object]'
10 console.log(personne.valueOf()); // { nom: 'Alice' }
11
12 // Tester l'appartenance
13 console.log(personne instanceof Object); // true
```

- Peu visibles dans le contexte Javascript
- Pourtant omniprésents
- Ex. Un objet JS hérite du prototype `Object`
- Déconseillé de surcharger un prototype natif

Object

- Prototype de tous les objets Javascript
- Méthodes statiques pour manipuler les objets

CRÉATION D'UN OBJET

```
1 // Constructeur Object
2 const personne = new Object();
3 personne.nom = 'Martin';
4
5 // Methode statique Object.create
6 const user = Object.create(personne);
7 user.age = 30;
```

Le constructeur `Object` et la méthode statique `Object.create` sont très peu utilisés.

Les méthodes statiques le sont beaucoup plus.

AJOUT D'UNE PROPRIÉTÉ

```
const personne = new Object();
personne.nom = 'Martin';

// Définir une propriété avec un descripteur
Object.defineProperty(personne, 'age', {
  value: 30,
  writable: false,      // Ne peut pas être modifié
  enumerable: true,     // Apparaît dans les boucles
  configurable: false   // Ne peut pas être supprimé
});

// Obtenir le descripteur
console.log(Object.getOwnPropertyDescriptor(personne, 'age'));
```

MANIPULATION DE PROPRIÉTÉS

```
1  const personne = new Object();
2  personne.nom = 'Martin';
3  personne.age = 30;
4
5  // Vérifier l'existence d'une propriété
6  console.log(personne.hasOwnProperty('nom')); // true
7  console.log('nom' in personne); // true
8
9  // Lister les propriétés
10 console.log(Object.keys(personne)); // ['nom', 'age']
11 console.log(Object.values(personne)); // ['Dupont', 30]
12 console.log(Object.entries(personne)); // [['nom', 'Dupont'], ['
```

COPIE D'OBJETS

```
const personne = new Object();
personne.nom = 'Martin';
personne.age = 30;

// Copier un objet
const copie = Object.assign({}, personne);

// Copie profonde (warning)
const copieSuperficielle = { ...personne };
const copieProfonde = JSON.parse(JSON.stringify(personne));
```

COPIE SUPERFICIELLE (SHALLOW COPY)

- Valeurs primitives copiées par **valeur**
- Valeurs complexes copiées par **référence**

COPIE PROFONDE (DEEP COPY)

- Aucune référence partagée avec l'objet original
- Convertit les objets `Date` en `String`
- Ne gère pas les fonctions
- Explose sur les références circulaires

FUSION D'OBJETS

```
1 // Fusion d'objets
2 const obj1 = { a: 1 };
3 const obj2 = { b: 2 };
4 const fusion = Object.assign({}, obj1, obj2);
```

RESTRICTIONS SUR LES OBJETS

```
1  const personne = new Object();
2  personne.nom = 'Martin';
3  personne.age = 30;
4
5  // Empêcher les modifications
6  Object.freeze(personne); // Gel complet
7  Object.seal(personne); // Permet modification mais pas ajout/supp
8  Object.preventExtensions(personne); // Empêche l'ajout de propri
9
10 // Vérifier le statut
11 console.log(Object.isFrozen(personne));
12 console.log(Object.isSealed(personne));
13 console.log(Object.isExtensible(personne));
```

OBJETS LITTÉRAUX

Notation la plus courante, la plus simple et la plus performante pour créer des objets en Javascript.

CRÉATION D'OBJETS

```
1 // Notation littérale
2 const personne = {
3   nom: 'Dupont',
4   age: 30,
5   ville: 'Paris'
6 };
```

ACCÈS AUX PROPRIÉTÉS

```
1 // Notation point
2 console.log(personne.nom); // 'Dupont'
3
4 // Notation crochet
5 console.log(personne['age']); // 30
6
7 // Destructuration
8 const { nom, age } = personne;
```

MÉTHODES D'OBJET

```
1 const utilisateur = {
2   nom: 'Alice',
3   direBonjour() {
4     return `Bonjour, je suis ${this.nom}`;
5   },
6   changerNom(nouveauNom) {
7     this.nom = nouveauNom;
8   }
9 };
```

INSTANCES DE CLASSES ES6+

CRÉATION D'OBJETS

```
1  class Personne {
2    constructor(nom) {
3      this.nom = nom;
4    }
5
6    direBonjour() {
7      return `Bonjour, je suis ${this.nom}`;
8    }
9  }
10
11 const personne = new Personne('Alice');
```

HÉRITAGE

```
1  class Employe extends Personne {
2    constructor(nom, poste) {
3      super(nom);
4      this.poste = poste;
5    }
6
7    sePresenter() {
8      return `${this.direBonjour()}, je suis ${this.poste}`;
9    }
10 }
```

GETTERS ET SETTERS

```
1  const compte = {
2    _solde: 0,
3    get solde() {
4      return `${this._solde}€`;
5    },
6    set solde(montant) {
7      if (montant >= 0) {
8        this._solde = montant;
9      }
10   }
11 };
```

PRIVATE FIELDS (ES2022+)

```
1  class CompteBancaire {
2    #solde = 0;
3
4    deposer(montant) {
5      this.#solde += montant;
6    }
7
8    get solde() {
9      return this.#solde;
10   }
11 }
```

BONNES PRATIQUES

- Préférer la notation littérale
- Utiliser les classes pour les structures complexes
- Éviter de surcharger les prototypes natifs
- Éviter la mutation directe des objets
- Encapsuler les données sensibles
- Utiliser la destructuration
- Documenter les structures complexes

TABLEAUX

CRÉATION DE TABLEAUX

```
1  // Notation littérale
2  const fruits = ['pomme', 'banane', 'orange'];
3
4  // Constructeur Array
5  const nombres = new Array(1, 2, 3);
6
7  // Array.from
8  const lettres = Array.from('hello'); // ['h', 'e', 'l', 'l', 'o']
9
10 // Spread operator
11 const copie = [...fruits];
```

ACCÈS AUX ÉLÉMENTS

```
1 const fruits = ['pomme', 'banane', 'orange'];
2
3 console.log(fruits[0]);     // 'pomme'
4 console.log(fruits.at(-1)); // 'orange' (dernier élément)
5
6 // Destructuration
7 const [premier, deuxieme, ...reste] = fruits;
```

MÉTHODES DE BASE

```
const fruits = ['pomme'];

// Ajout/Suppression
fruits.push('banane');      // Ajoute à la fin
fruits.unshift('orange');   // Ajoute au début
fruits.pop();               // Retire le dernier
fruits.shift();             // Retire le premier

// Longueur
console.log(fruits.length);
```

MÉTHODES DE RECHERCHE

```
1 const nombres = [1, 2, 3, 2, 4];
2
3 // Recherche
4 console.log(nombres.indexOf(2));       // 1
5 console.log(nombres.lastIndexOf(2));   // 3
6 console.log(nombres.includes(4));      // true
7 console.log(nombres.find(n => n > 2)); // 3
8 console.log(nombres.findIndex(n => n > 2)); // 2
```

MÉTHODES DE TRANSFORMATION

```
const nombres = [1, 2, 3, 4, 5];

// Map : transformer chaque élément
const doubles = nombres.map(n => n * 2);

// Filter : filtrer les éléments
const pairs = nombres.filter(n => n % 2 === 0);

// Reduce : réduire à une seule valeur
const somme = nombres.reduce((acc, n) => acc + n, 0);
```

MÉTHODES DE TRI

```
1  const fruits = ['banane', 'pomme', 'orange'];
2  const nombres = [3, 1, 4, 1, 5];
3
4  // Tri simple
5  fruits.sort();
6  nombres.sort((a, b) => a - b);
7
8  // Inverse
9  fruits.reverse();
10
11 // Tri personnalisé
12 const personnes = [{nom: 'Alice', age: 30}, {nom: 'Bob', age: 25
13 personnes.sort((a, b) => a.age - b.age);
```

MÉTHODES DE MODIFICATION

```
1  const fruits = ['pomme', 'banane', 'orange', 'kiwi'];
2
3  // Slice : extraire une portion
4  const portion = fruits.slice(1, 3);  // ['banane', 'orange']
5
6  // Splice : modifier le tableau
7  fruits.splice(1, 1, 'fraise');  // Remplace 'banane' par 'fraise
8
9  // Join : convertir en chaîne
10 console.log(fruits.join(', '));  // "pomme, fraise, orange, kiwi
```

MÉTHODES DE TEST

```
1  const nombres = [1, 2, 3, 4, 5];
2
3  // Some : au moins un élément satisfait la condition
4  console.log(nombres.some(n => n > 4));  // true
5
6  // Every : tous les éléments satisfont la condition
7  console.log(nombres.every(n => n > 0)); // true
8
9  // IsArray : vérifier si c'est un tableau
10 console.log(Array.isArray(nombres));    // true
```

MÉTHODES ES2023+

```
1  const nombres = [1, 2, 3, 4, 5];
2
3  // toSorted : tri sans mutation
4  const triés = nombres.toSorted();
5
6  // toReversed : inverse sans mutation
7  const inversés = nombres.toReversed();
8
9  // with : remplace un élément sans mutation
10 const modifiés = nombres.with(2, 10);
```

BONNES PRATIQUES

- Préférer les méthodes immutables
- Utiliser les méthodes fonctionnelles (`map`, `filter` `reduce`)
- Éviter de modifier la longueur pendant une itération
- Attention aux références lors de la copie
- Utiliser les méthodes ES2023+ quand possible

INTERACTIONS DOM

SÉLECTION D'ÉLÉMENTS

```
1 // Sélecteur unique
2 const titre = document.querySelector('h1');
3 const menu = document.getElementById('menu');
4
5 // Sélecteurs multiples
6 const paragraphes = document.querySelectorAll('p');
7 const boutons = document.getElementsByClassName('btn');
8 const items = document.getElementsByTagName('li');
```

MANIPULATION DU CONTENU

```
1  // Modifier le texte
2  element.textContent = 'Nouveau texte';
3  element.innerText = 'Texte visible';
4
5  // Modifier le HTML
6  element.innerHTML = '<span>Du HTML</span>';
7
8  // Attributs
9  element.setAttribute('class', 'active');
10 element.getAttribute('id');
11 element.removeAttribute('style');
```

MANIPULATION DES CLASSES

```
1  // Ajouter/Supprimer des classes
2  element.classList.add('active');
3  element.classList.remove('hidden');
4  element.classList.toggle('visible');
5  element.classList.replace('old', 'new');
6
7  // Vérifier
8  if (element.classList.contains('active')) {
9    // ...
10 }
```

MANIPULATION DU STYLE

```
1  // Style direct
2  element.style.backgroundColor = 'red';
3  element.style.marginTop = '20px';
4
5  // Récupérer le style calculé
6  const style = window.getComputedStyle(element);
7  console.log(style.getPropertyValue('color'));
8
9  // Variables CSS
10 element.style.setProperty('--main-color', 'blue');
```

CRÉATION D'ÉLÉMENTS

```
1  // Créer un élément
2  const div = document.createElement('div');
3  div.textContent = 'Nouveau div';
4
5  // Ajouter dans le DOM
6  parent.appendChild(div);
7  parent.insertBefore(div, referenceNode);
8  parent.replaceChild(div, oldChild);
9
10 // Supprimer
11 element.remove();
12 parent.removeChild(element);
```

NAVIGATION DANS LE DOM

```
1 // Relations parent/enfant
2 const parent = element.parentNode;
3 const enfants = element.children;
4 const premier = element.firstElementChild;
5 const dernier = element.lastElementChild;
6
7 // Relations frères/sœurs
8 const suivant = element.nextElementSibling;
9 const precedent = element.previousElementSibling;
```

DIMENSIONS ET POSITION

```
1  // Dimensions de l'élément
2  const rect = element.getBoundingClientRect();
3  console.log(rect.width, rect.height);
4  console.log(rect.top, rect.left);
5
6  // Dimensions avec padding/border
7  console.log(element.offsetWidth);
8  console.log(element.offsetHeight);
9
10 // Position relative au parent
11 console.log(element.offsetLeft);
12 console.log(element.offsetTop);
```

MANIPULATION DE FORMULAIRES

```
1  const form = document.querySelector('form');
2  const input = document.querySelector('input');
3
4  // Valeurs
5  console.log(input.value);
6  input.value = 'Nouveau texte';
7
8  // Validation
9  input.checkValidity();
10 input.setCustomValidity('Message d\'erreur');
11 input.reportValidity();
12
13 // Focus
14 input.focus();
15 input.blur();
```

FRAGMENTS ET TEMPLATES

```
1  // Fragment pour optimisation
2  const fragment = document.createDocumentFragment();
3  for (let i = 0; i < 100; i++) {
4    const div = document.createElement('div');
5    fragment.appendChild(div);
6  }
7  document.body.appendChild(fragment);
8
9  // Template
10 const template = document.querySelector('#monTemplate');
11 const clone = template.content.cloneNode(true);
```

BONNES PRATIQUES

- Minimiser les manipulations directes du DOM
- Utiliser les fragments pour les insertions multiples
- Préférer les sélecteurs modernes (querySelector)
- Mettre en cache les sélections fréquentes
- Éviter les reflows multiples
- Utiliser requestAnimationFrame pour les animations

EVÉNEMENTS

Signaux envoyés par le navigateur pour indiquer qu'une action s'est produite.

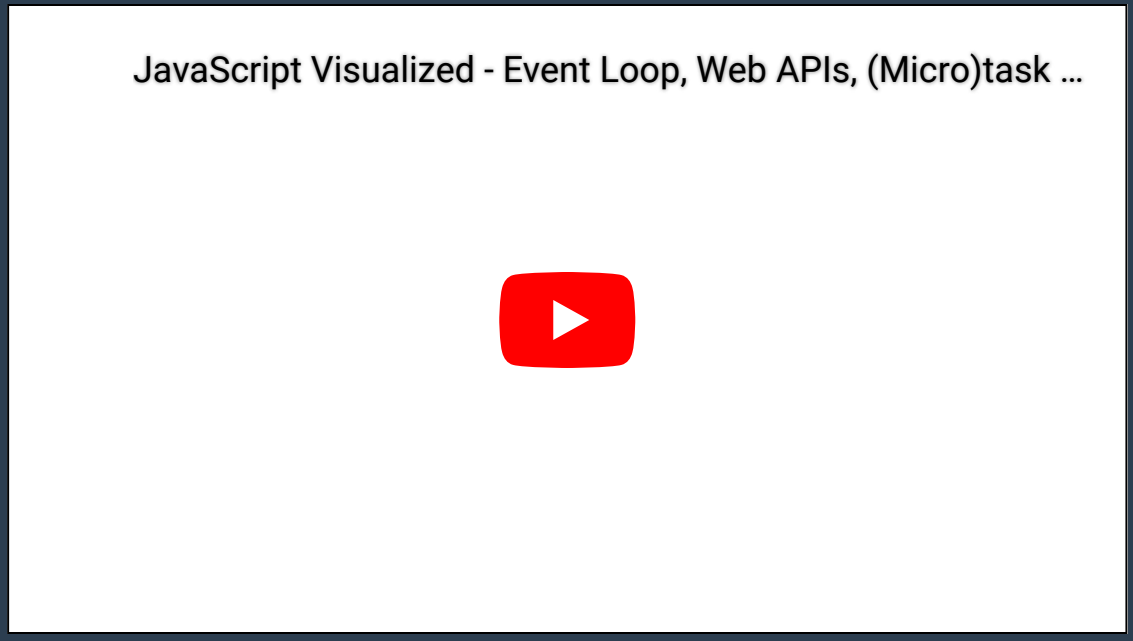
Les événements peuvent être liés à...

- L'objet `window` (fenêtre)
- L'objet `document` (arbre DOM)
- Des éléments HTML (arbre DOM)
- Tout autre objet Javascript

Les événements sont déclenchés par une action de l'utilisateur ou par le navigateur lui-même.

Ils sont au coeur de la programmation Javascript.

EVENT LOOP


JavaScript Visualized - Event Loop, Web APIs, (Micro)task ...

GESTIONNAIRES D'ÉVÉNEMENTS

```
1  // Méthode addEventListener
2  element.addEventListener('click', function(event) {
3    console.log('Cliqué!');
4  });
5
6  // Propriété on-event
7  element.onclick = function(event) {
8    console.log('Cliqué aussi!');
9  };
10
11 // Dans le HTML (à éviter)
12 // <button onclick="maFonction()">Cliquer</button>
```

L'OBJET EVENT

```
1 element.addEventListener('click', function(event) {
2   console.log(event.type); // 'click'
3   console.log(event.target); // élément cliqué
4   console.log(event.currentTarget); // élément avec le listener
5   console.log(event.clientX); // position X de la souris
6   console.log(event.clientY); // position Y de la souris
7 });
```

TYPES D'ÉVÉNEMENTS COURANTS

- **Souris**: `click`, `dblclick`, `mouseover`, ...
- **Clavier**: `keydown`, `keyup`, `keypress`
- **Formulaire**: `submit`, `change`, `input`, `focus`, `blur`
- **`document`**: `DOMContentLoaded`, `load`, ...
- **`window`**: `resize`, `scroll`, ...

PROPAGATION DES ÉVÉNEMENTS

```
1  // Phase de capture
2  parent.addEventListener('click', function(e) {
3    console.log('Capture!');
4  }, true);
5
6  // Phase de bouillonnement (bubbling)
7  child.addEventListener('click', function(e) {
8    console.log('Bubbling!');
9    e.stopPropagation(); // Arrête la propagation
10 });
```

DÉLÉGATION D'ÉVÉNEMENTS

```
1 // Au lieu d'attacher un événement à chaque bouton
2 document.querySelector('#container').addEventListener('click', fu
3   if (e.target.matches('button')) {
4     console.log('Bouton cliqué:', e.target.textContent);
5   }
6 });
```

ÉVÉNEMENTS PERSONNALISÉS

```
1  // Créer un événement personnalisé
2  const monEvent = new CustomEvent('monEvenement', {
3    detail: { message: 'Bonjour!' }
4  });
5
6  // Déclencher l'événement
7  element.dispatchEvent(monEvent);
8
9  // Écouter l'événement
10 element.addEventListener('monEvenement', function(e) {
11   console.log(e.detail.message);
12 });
```

GESTION DES FORMULAIRES

```
1  const form = document.querySelector('form');
2
3  form.addEventListener('submit', function(e) {
4    e.preventDefault(); // Empêche l'envoi du formulaire
5
6    const formData = new FormData(form);
7    console.log(formData.get('username'));
8  });
9
10 // Validation en temps réel
11 input.addEventListener('input', function(e) {
12   if (e.target.value.length < 3) {
13     e.target.setCustomValidity('Trop court!');
14   } else {
15     e.target.setCustomValidity('');
```

ÉVÉNEMENTS ET ASYNCHRONE

```
1  // Attendre le chargement du DOM
2  document.addEventListener('DOMContentLoaded', function() {
3    console.log('DOM chargé!');
4  });
5
6  // Attendre le chargement complet
7  window.addEventListener('load', function() {
8    console.log('Page entièrement chargée!');
9  });
10
11 // Débounce d'événements
12 function debounce(fn, delay) {
13   let timeoutId;
14   return function(...args) {
15     clearTimeout(timeoutId);
```

BONNES PRATIQUES

- Utilisez `addEventListener`.
- Supprimez les écouteurs lorsqu'ils ne sont plus nécessaires.
- Utilisez la délégation pour gérer les événements sur plusieurs éléments enfants.
- Gardez les fonctions de gestion des événements petites et concentrées sur une seule tâche.

- Utilisez `event.preventDefault()` et `event.stopPropagation()` pour contrôler le flux des événements.
- Prenez en compte l'accessibilité et assurez-vous que les gestionnaires d'événements fonctionnent avec les interactions clavier.
- Débouncez ou throttlez les événements qui se déclenchent fréquemment.
- Testez les gestionnaires d'événements de manière approfondie.

GESTION DES ERREURS

TRY...CATCH DE BASE

```
try {
  // Code qui peut générer une erreur
  const data = JSON.parse('invalid json');
} catch (error) {
  console.error('Type:', error.name); // SyntaxError
  console.error('Message:', error.message);  // Unexpected token
  console.error('Stack:', error.stack); // Stack trace complet
} finally {
  // Toujours exécuté
  console.log('Nettoyage...');
}
```

TYPES D'ERREURS NATIFS

```
1 // Erreurs communes
2 throw new Error('Erreur générique');
3 throw new SyntaxError('Erreur de syntaxe');
4 throw new TypeError('Type incorrect');
5 throw new ReferenceError('Variable non définie');
6 throw new RangeError('Valeur hors limites');
7 throw new URIError('URI malformé');
```

ERREURS PERSONNALISÉES

```
class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
  }
}

try {
  throw new ValidationError('Email invalide', 'email');
} catch (error) {
  if (error instanceof ValidationError) {
    console.log(`Erreur sur ${error.field}: ${error.message}`);
  }
}
```

GESTION ASYNCHRONE

```
1  // Avec Promises
2  fetchData()
3    .then(data => processData(data))
4    .catch(error => {
5      if (error.name === 'NetworkError') {
6        // Gérer erreur réseau
7      } else {
8        // Autres erreurs
9      }
10   });
11
12 // Avec Async/Await
13 async function getData() {
14   try {
15     const data = await fetchData();
```

ERREURS NON ATTRAPÉES

```
1  // Gestionnaire global
2  window.onerror = function(message, source, line, column, error) {
3    console.error('Erreur globale:', {
4      message,
5      source,
6      line,
7      column,
8      error
9    });
10   return false; // Permet la propagation
11 };
12
13 // Pour les Promesses non attrapées
14 window.onunhandledrejection = function(event) {
15   console.error('Promesse rejetée:', event.reason);
```

VALIDATION ET ASSERTIONS

```
1  function validateUser(user) {
2    // Vérifications préalables
3    if (!user) {
4        throw new Error('User requis');
5    }
6
7    if (typeof user.age !== 'number') {
8        throw new TypeError('Age doit être un nombre');
9    }
10 }
```

GESTION AVANCÉE

```
function tryCatch(fn) {
  return function(...args) {
    try {
      return fn.apply(this, args);
    } catch (error) {
      console.error(error); // Logger l'erreur
      return null; // Renvoyer une valeur par défaut
    }
  };
}

const secureFn = tryCatch(function() {
  // Code dangereux
});
```

CHAÎNAGE D'ERREURS

```
async function processData() {
  try {
    const data = await fetchData();
    return transformData(data);
  } catch (error) {
    throw new Error('Erreur de traitement', { cause: error });
  }
}

try {
  await processData();
} catch (error) {
  console.error('Erreur:', error);
  console.error('Cause:', error.cause);
}
```

POINTS IMPORTANTS

- Les erreurs non gérées peuvent crasher l'application
- Différencier les erreurs attendues et inattendues
- Prévoir des fallbacks appropriés
- Tester les cas d'erreur
- Documenter les erreurs possibles
- Monitorer les erreurs en production

BONNES PRATIQUES

- Utiliser `try...catch` pour gérer les erreurs
- Erreurs personnalisées pour les cas spécifiques
- Logger les erreurs pour faciliter le débogage
- Ne pas exposer les détails des erreurs aux utilisateurs finaux
- Utiliser `finally` pour les opérations de nettoyage

BONNES PRATIQUES

- Gérer les erreurs async avec `async/await` et `catch`
- Centraliser la gestion des erreurs
- Prévoir des mécanismes de fallback en cas d'erreur
- Tester les scénarios d'erreur
- Documenter les erreurs possibles et leur gestion

CONCLUSION

JAVASCRIPT EN 2025

- Langage incontournable du web
- Écosystème riche et mature
- Évolution constante (ES6+)
- Communauté très active
- Utilisé côté client ET serveur

POINTS CLÉS

- Maîtriser les fondamentaux
- Comprendre l'asynchrone
- Sécuriser ses applications
- Optimiser les performances
- Utiliser les bonnes pratiques
- Tester son code

BONNES PRATIQUES HABITUELLES

- Code lisible et maintenable
- Documentation claire
- Tests automatisés
- Sécurité dès la conception
- Performance comme priorité
- Veille technologique continue

OUTILS ESSENTIELS

- Environnement moderne (Node.js)
- Gestionnaire de paquets (npm/yarn)
- Outils de build (Webpack/Vite)
- Linters et formatters
- Framework de test
- Outils de débogage

POUR ALLER PLUS LOIN

- TypeScript
- Frameworks modernes (React, Vue, Angular)
- WebAssembly
- Progressive Web Apps
- Serverless
- Web Components

"Javascript a ses défauts, mais c'est aussi ce qui fait sa force. Sa flexibilité permet de créer des solutions innovantes pour répondre à des problèmes complexes."