# Optimiser le script JavaScript

## Programmation défensive

La programmation défensive consiste à anticiper les erreurs et les problèmes qui pourraient survenir dans un programme. En JavaScript, il est possible de vérifier les types de données, les valeurs de retour des fonctions, les erreurs de syntaxe, etc.

Ex: si vous faites une sélection avec querySelector et qu'il y a une chance que l'élément n'existe pas, vous pouvez vérifier si l'élément existe avant de continuer.

```
let elementHTML = document.querySelector("section");

if (elementHTML !== null) {
    elementHTML.classList.add("invisible");
}
```

Un autre façon de faire de la programmation défensive est de gérer les exceptions au début d'une fonction complexe et de quitter la fonction si une erreur survient en utilisant return ou throw.

```
function calculerSomme(a, b) {
    if (typeof a !== "number" || typeof b !== "number") {
        return "Les paramètres doivent être des nombres.";
    }

    if (a === null || b === null) {
        return "Les paramètres ne doivent pas être nuls.";
    }

    return a + b;
}
```

### Lancer une exception ou une erreur

Il est possible de lancer une exception ou une erreur en utilisant le mot-clé `throw`. Cela permet de signaler une erreur et d'arrêter l'exécution du programme. Vous pouvez lancer une exception avec un message d'erreur pour informer les autres développeurs du problème. Généralement, `throw` est utilisé conjointement avec les instructions `try...catch` pour gérer les erreurs.

On peut lancer une exception avec un message d'erreur personnalisé en utilisant le mot-clé `throw` suivi d'un objet Error (avec new) avec un message d'erreur.

```
```

```
function calculerSomme(a, b) {
    if (typeof a !== "number" || typeof b !== "number") {
        throw new Error("Les paramètres doivent être des nombres.");
    }

    if (a === null || b === null) {
        throw new Error("Les paramètres ne doivent pas être nuls.");
    }

    return a + b;
}

try {
    calculerSomme(5, null);
} catch (error) {
    console.error(error.message);
}
```

### Try...catch

Le bloc try...catch permet de gérer les erreurs dans un programme. Vous pouvez mettre le code qui risque de générer une erreur dans le bloc try et gérer l'erreur dans le bloc catch.

Le bloc try contient le code qui risque de générer une erreur. Si une erreur se produit, le programme passe au bloc catch. L'erreur ne sera jamais affichée et le programme continuera à s'exécuter normalement.

Le bloc catch contient le code qui gère l'erreur. Il prend en paramètre l'objet Error qui contient des informations sur l'erreur. Vous pouvez afficher le message d'erreur avec error.message.

```
try {
    // Code qui risque de générer une erreur
} catch (error) {
    console.error(error.message);
    // Gestion de l'erreur
}
```

### Gestion des erreurs

Il est important de gérer les erreurs dans un programme pour éviter les plantages et les comportements inattendus. Minimalement, vous pouvez afficher un message d'erreur dans la console pour informer les autres développeurs du problème.

### Conversion de types lorsque nécessaire

En JavaScript, il est fréquent de récupérer une valeur sous forme de chaine de caractères. Cependant, il est parfois nécessaire de convertir cette valeur en nombre pour effectuer des opérations mathématiques.

Pour convertir une chaine de caractères en nombre, vous pouvez utiliser les fonctions parseInt ou parseFloat. La fonction parseInt convertit une chaine de caractères en entier, tandis que la fonction parseFloat convertit une chaine de caractères en nombre à virgule flottante.

## Éviter les variables globales

Les variables globales sont accessibles partout dans le code. Cela peut entraîner des problèmes de lisibilité, de maintenabilité et de sécurité. Il est préférable d'utiliser des variables locales ou des propriétés d'objets pour limiter la portée des variables.

Pour cela, nous avons utiliser la modularisation du code qui consiste à diviser un programme en plusieurs modules ou classe. Chaque module est responsable d'une tâche spécifique. Cela permet de rendre le code plus lisible, plus facile à maintenir et à déboguer. Dans un module, les variables et les fonctions sont encapsulées, c'est-à-dire qu'elles ne sont pas accessibles depuis l'extérieur du module.

Cependant, le fichier principal lié à la page HTML est toujours un fichier global. Pour éviter les variables globales, vous pouvez encapsuler le code dans une fonction anonyme auto-exécutée (IIFE).

### IIFE (Immediately Invoked Function Expression)

Une fonction anonyme auto-exécutée est une fonction qui est définie et appelée immédiatement. Les variables et les fonctions déclarées à l'intérieur de la fonction sont encapsulées et ne sont pas accessibles depuis l'extérieur de la fonction. Cela permet d'éviter les variables globales.

Pour se souvenir de la syntaxe, commencez par écrire deux paires de parenthèses, puis définissez une fonction anonyme à l'intérieur de la première.

Vous n'avez pas besoin d'utiliser une IIFE pour chaque fichier, mais seulement pour les fichiers qui contiennent du code global.

```
(function () {
    // Code encapsulé
})();
```

## Stratégies de contrôle de la qualité du code

### Use strict

La directive "use strict" permet de déclarer un code en mode strict. En mode strict, certaines erreurs courantes sont détectées et signalées par le moteur JavaScript. Cela permet d'éviter les erreurs silencieuses et les comportements inattendus. Il est recommandé d'utiliser "use strict" pour tous les fichiers JavaScript.

Il faut simplement ajouter la directive "use strict" sur la première ligne du fichier JavaScript.

```
"use strict";
```

### Linter

Un linter est un outil qui analyse le code source pour signaler les erreurs de syntaxe, les erreurs de style et les erreurs logiques. Il permet de détecter les erreurs avant l'exécution du programme et de les corriger rapidement.

ESLint est un linter pour JavaScript qui permet de détecter les erreurs de syntaxe, les erreurs de style et les erreurs logiques. Il permet de rendre le code plus lisible et plus cohérent en appliquant des conventions de style.

### Formatter

Il est d'utiliser des outils de formatage de code pour uniformiser le style du code et le rendre plus lisible.

Prettier est un outil de formatage de code qui permet de formater automatiquement le code source selon des règles prédéfinies. Il permet de rendre le code plus lisible et plus cohérent en appliquant des conventions de style.

EsLint et prettier sont souvent utilisés ensemble pour formater le code et détecter les erreurs de syntaxe mais peuvent être en conflit.

Voici une vidéo expliquant comment les configurer ensemble: https://www.youtube.com/watch?v=SydnKbGc7W8

## Documentation du code

### JSdoc

JSdoc est un outil de documentation de code qui permet de générer une documentation à partir des commentaires du code source. Il permet de documenter les fonctions, les classes, les variables, etc. en ajoutant des balises spéciales aux commentaires. Ces balises sont comprises par l'éditeur de code et peuvent être utilisées pour générer une documentation automatique.

**Chaque fonction doit être documentée avec JSdoc.**

Pour utiliser la norme JSdoc, vous devez ajouter des commentaires au-dessus de la fonction que vous souhaitez documenter. Les commentaires doivent commencer par "/*" *et se terminer par* "/".

Vous pouvez ajouter des balises JSdoc pour décrire les paramètres, la valeur de retour, la description, les exemples, etc. On met le type de données entre accolades suivit du nom de la variable et de la description.

Voici un exemple de commentaires JSdoc pour documenter une fonction :

```
/**
 * Vérifier si deux nombres sont égaux.
 * @param {number} a - Le premier nombre.
 * @param {number} b - Le deuxième nombre.
```

```
 * @returns {boolean} - Vrai si les deux nombres sont égaux, faux sinon.
 *
 * @see
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Op%C3
%A9rateurs_de_comparaison
 * @todo Ajouter une vérification pour les nombres flottants.
 */
function estEgal(a, b) {
    // Code de la fonction
    return a === b;
}
```

**Balises JSdoc**

- `@param` : Définit les paramètres de la fonction.
- `@returns` : Définit la valeur de retour de la fonction.
- `@example` : Définit un exemple d'utilisation de la fonction.
- `@see` : Définit un lien vers une autre documentation.
- `@todo` : Définit une tâche à réaliser.
- `@deprecated` : Indique que la fonction est obsolète.
- `@author` : Indique l'auteur de la fonction.

# Normes de codage

## Conventions de nommage

Les conventions de nommage permettent de rendre le code plus lisible et plus cohérent. Il est important de choisir des noms de variables, de fonctions, de classes, etc. significatifs et descriptifs. Voici quelques règles de base pour les conventions de nommage en JavaScript :

- Les noms de variables, de fonctions et de classes doivent être en camelCase. Ex: `nomDeVariable`.
- Les noms de constantes doivent être en majuscules avec des underscores. Ex: `NOM_DE_CONSTANTE`.
- Les noms de classes doivent commencer par une majuscule. Ex: `NomDeClasse`.
- Les noms de fonctions doivent être des verbes ou des phrases verbales. Ex: `calculerSomme`.

## Indentation

L'indentation permet de structurer le code de manière visuelle. Il est recommandé d'indenter le code avec des espaces pour rendre le code plus lisible. Il est recommandé d'utiliser 2 ou 4 espaces pour l'indentation.

## Longueur des lignes

Il est recommandé de limiter la longueur des lignes de code à 80 caractères pour faciliter la lecture du code. Si une ligne de code est trop longue, vous pouvez la diviser en plusieurs lignes pour la rendre plus lisible.

## Commentaires

Les commentaires permettent d'expliquer le code et de le rendre plus compréhensible. Il est recommandé d'ajouter des commentaires pour expliquer le fonctionnement du code, les choix de conception, les algorithmes, etc. Pour les longs commentaires, ajoutez le commentaire au-dessus de la ligne de code. Pour les commentaires courts, ajoutez-les à la fin de la ligne de code.

### Espaces verticaux

Les espaces verticaux permettent de séparer visuellement les différentes parties du code. Il est recommandé d'ajouter des espaces verticaux entre les fonctions, les classes, les boucles, les conditions, etc. Regroupez également les déclarations de variables et les fonctions connexes ainsi que les lignes de code qui font partie d'une même tâche.

### Espaces horizontaux

Les espaces horizontaux permettent de séparer visuellement les différents éléments du code. Il est recommandé d'ajouter **un espace** horizontal autour des opérateurs, des accolades, des parenthèses, etc. pour rendre le code plus lisible. Souvent, l'outil de formatage de code ajoute automatiquement les espaces horizontaux.

### Exemple de code optimisé

Voici un exemple de code optimisé qui respecte les normes de codage :

```
"use strict"; // Mode strict

// Fonction anonyme auto-exécutée pour éviter les variables globales
(function () {
    /**
     * Ajouter deux nombres.
     * @param {number} a - Le premier nombre.
     * @param {number} b - Le deuxième nombre.
     * @returns {number} - La somme des deux nombres.
     */
    function ajouter(a, b) {
        if (typeof a !== "number" || typeof b !== "number") {
            throw new Error("Les paramètres doivent être des nombres.");
        }

        // Retourne la somme des deux nombres
        return a + b;
    }

    /**
     * Multiplier deux nombres.
     * @param {number} a - Le premier nombre.
     * @param {number} b - Le deuxième nombre.
     * @returns {number} - Le produit des deux nombres.
     */
    function multiplier(a, b) {
```

```
        if (typeof a !== "number" || typeof b !== "number") {
            throw new Error("Les paramètres doivent être des nombres.");
        }

        // Retourne le produit des deux nombres
        return a * b;
    }
})();
```