

RAPPEL FONDAMENTAUX DE JAVASCRIPT ET LES PIÈGES À ÉVITER

SYNTAXE

Pour déclarer une variable avec var:

```
var a = 1;
```

PORTÉE GLOBALE OU FONCTION

La portée de var est limitée à la **fonction englobante**, mais non au bloc.

```
function example() {  
  var a = 1;  
  if (true) {  
    var a = 2;  
  }  
  console.log(a); // 2  
}
```

PROBLÈMES DE DÉCLARATION

`var` peut être redéclarée sans générer d'erreur, ce qui peut causer des problèmes.

```
var a = 1;  
var a = 2;
```


SYNTAXE

Pour déclarer une variable avec **let**:

```
let a = 1;
```

PORTÉE DU BLOC

La portée de let est limitée au **bloc englobant**.

```
{  
  let a = 1;  
}  
console.log(a); // ReferenceError
```

AVANTAGES PAR RAPPORT À var

`let` ne permet pas la **redéclaration** et limite la **portée** au bloc.

```
let a = 1;  
let a = 2; // SyntaxError
```


SYNTAXE

Pour déclarer une **constante** avec const:

```
const a = 1;
```

PORTÉE DU BLOC

La portée de const est limitée au **bloc englobant**, comme let.

```
{  
  const a = 1;  
}  
console.log(a); // ReferenceError
```

IMMUABILITÉ

`const` empêche la **réaffectation**, mais pas la modification d'objets.

```
const a = { key: 1 };
a = { key: 2 }; // TypeError
a.key = 2; // Pas d'erreur, objet modifié
```

RAPPEL FONDAMENTAUX DE JAVASCRIPT ET LES PIÈGES À ÉVITER

PRIMITIFS

Types de données atomiques, immuables et sans méthodes.

Type de données	Exemple	Description
String	"Hello, World!"	Chaine de caractères
Number	42	Nombre entier ou décimal
Boolean	true / false	Valeurs logiques vrai ou faux
BigInt	10n	Nombre entiers de taille arbitraire
Symbol	Symbol("label")	Valeur unique et immuable

NUMBER

Représentation de nombres **entiers** et **réels**.

```
const a = 42;  
const b = 3.14;
```

PROBLÈMES DE PRÉCISION

Les nombres en JavaScript sont représentés en **double précision flottante IEEE 754**. Cela peut entraîner des problèmes de précision.

```
console.log(0.1 + 0.2); // 0.3000000000000004
```

STRING

Représentation de **chaînes de caractères**.

```
const str = "Hello, World!";
```

MÉTHODES COURANTES

Méthode	Description
length	Longueur de la chaîne
charAt ()	Caractère à l'indice donné
indexOf ()	Première occurrence d'un caractère
slice ()	Sous-chaîne d'indices donnés
toUpperCase ()	Convertir en majuscules
toLowerCase ()	Convertir en minuscules

BOOLEAN

Représentation **vraie / fausse**.

```
const isTrue = true;
const isFalse = false;
```

OPÉRATEURS LOGIQUES

- `&&` (**AND**)
- `||` (**OR**)
- `!` (**NOT**)

NULL ET UNDEFINED

Valeurs spéciales représentant l'**absence de valeur**.

- null : **Intentionnelle**
- undefined : **Non définie**

```
let a = null;  
let b;  
console.log(b); // undefined
```

COMPORTEMENT PARTICULIER

- null: égal mais pas **identique** à undefined
- typeof null: retourne "**object**"
- typeof undefined: retourne "**undefined**"

FONCTIONS

DÉCLARATION DE FONCTION

Les **fonctions** sont des blocs de code **réutilisables** qui permettent d'organiser et de structurer le code.

```
function nomDeFonction(parametre1, parametre2) {  
    // Bloc de code à exécuter  
    return résultat;  
}  
  
const result = nomDeFonction(arg1, arg2);
```

SYNTAXE

```
function nom (param1, param2, ...) {  
    // corps de la fonction  
}
```

Exemple:

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

NOM ET PARAMÈTRES

Le **nom** de la fonction doit être **unique** et **décrire** son rôle. Les **paramètres** sont des variables utilisées dans la fonction.

```
function calculerSomme(a, b) {  
    return a + b;  
}  
  
let resultat = calculerSomme(3, 4);  
console.log(resultat); // Affiche 7
```

PORTÉE

Les fonctions ont une **portée locale**, et leur déclaration est **hissée en haut** du scope dans lequel elles sont définies.

EXPRESSIONS DE FONCTION

Les **expressions de fonction** sont des fonctions définies en tant que valeurs d'une variable ou comme arguments d'une autre fonction.

```
let somme = function(a, b) {  
    return a + b;  
};  
  
let resultat = somme(1, 2);  
console.log(resultat); // 3
```

Avantages	Inconvénients
Flexibilité	Difficulté de débogage
Réutilisation aisée	Moins lisible
Passer en argument	

OPÉRATEURS

OPÉRATEURS DE COMPARAISON

Les **opérateurs de comparaison** permettent de comparer deux valeurs et de renvoyer un **booléen** décrivant la relation entre elles.

- `==` : Égalité (vérifie si deux valeurs sont égales).
- `==` : Égalité stricte (vérifie si deux valeurs sont égales et de même type).
- `!=` : Inégalité (vérifie si deux valeurs sont différentes).
- `!==` : Inégalité stricte (vérifie si deux valeurs sont différentes ou de types différents).
- `>` : Supérieur à (vérifie si la valeur à gauche est supérieure à celle de droite).
- `<` : Inférieur à (vérifie si la valeur à gauche est inférieure à celle de droite).
- `>=` : Supérieur ou égal à (vérifie si la valeur à gauche est supérieure ou égale à celle de droite).
- `<=` : Inférieur ou égal à (vérifie si la valeur à gauche est inférieure ou égale à celle de droite).

ÉGALITÉ ET IDENTITÉ

Opérateur	Description	Exemple
<code>==</code>	Égalité (vérifie si les valeurs sont égales)	<code>5 == '5'</code>
<code>====</code>	Identité (vérifie si les valeurs et les types sont égaux)	<code>5 === '5'</code>

INÉGALITÉ ET NON-IDENTITÉ

Opérateur	Description	Exemple
<code>!=</code>	Inégalité	<code>5 != '5'</code>
<code>==</code>	Non-identité	<code>5 == '5'</code>

SUPÉRIEUR, INFÉRIEUR, ET LEURS VARIANTES

Opérateur	Description	Exemple
<	Inférieur	5 < 6
<=	Inférieur ou égal	5 <= 6
>	Supérieur	6 > 5
>=	Supérieur ou égal	6 >= 5

PIÈGES AVEC LA CONVERSION DE TYPE AUTOMATIQUE

Les opérateurs == et != effectuent des **conversions de type automatiques** qui peuvent entraîner des comportements inattendus lors des comparaisons.

Pour éviter ces problèmes, privilégiez l'utilisation des opérateurs === et !==.

OPÉRATEURS D'AFFECTION

Les **opérateurs d'affectation** permettent d'affecter une valeur à une variable.

Opérateur	Description	Exemple	Équivalent
=	Affectation simple	x = 5	x = 5
+=	Ajouter puis affecter	x += 5	x = x + 5
-=	Soustraire puis affecter	x -= 5	x = x - 5
*=	Multiplier puis affecter	x *= 5	x = x * 5
/=	Diviser puis affecter	x /= 5	x = x / 5
%=	Modulo puis affecter	x %= 5	x = x % 5

OPÉRATEURS COMBINÉS

Opérateur	Description	Exemple
<code>+=</code>	Addition et affectation	<code>a += 3</code>
<code>-=</code>	Soustraction et affectation	<code>a -= 3</code>
<code>*=</code>	Multiplication et affectation	<code>a *= 3</code>
<code>/=</code>	Division et affectation	<code>a /= 3</code>
<code>%=</code>	Modulo et affectation	<code>a %= 3</code>
<code>**=</code>	Puissance et affectation	<code>a **= 3</code>

EFFETS DE BORD POTENTIELS

Éviter de mélanger des opérations avec des **effets de bord** dans une même expression, car cela peut rendre le code difficile à lire et à maintenir.

PIÈGES COURANTS

COERCITION DE TYPE

La **coercition de type** est la conversion automatique de données d'un type à un autre lors de l'exécution d'opérations.

Coercition explicite	Coercition implicite
String (42)	42 + ''
Number ('42')	'42' * 1
Boolean (0)	!!0

- Coercition **explicite** : L'utilisateur convertit intentionnellement une valeur d'un type à un autre.
- Coercition **implicite** : Le langage convertit automatiquement les valeurs en fonction du contexte.

CONVERSION AUTOMATIQUE DE TYPES

JavaScript **convertit automatiquement** les types de données lors de l'évaluation d'**expressions** avec des **types mélangés**.

Opérateur	Exemple	Résultat
Égalité (==)	"5" == 5	true
Addition	"5" + 1	"51"
Soustraction	"5" - 1	4

IMPACT SUR LES COMPARAISONS

Les opérations de comparaison avec == entraînent souvent des conversions **inattendues**, préférez utiliser === pour éviter ces pièges.

```
console.log(1 == "1"); // true
console.log(1 === "1"); // false
console.log(0 == false); // true
console.log(0 === false); // false
```

UTILISATION DE "STRICT" POUR ÉVITER LES PROBLÈMES

En mode **strict**, certaines erreurs sont détectées et empêchent la **coercition** de type involontaire. Utilisez '`use strict`'; en haut de votre fichier.

```
"use strict";\n\n// exemple de code en mode strict\nfunction exemple() {\n    // code ...\n}
```

HOISTING

Le **Hoisting** est un comportement de **JavaScript** qui "élève" les déclarations de variables et de fonctions au sommet de leur **portée**.

Les fonctions déclarées

Fonction complète hoistée

Les variables déclarées avec "var"

Seul le nom de la variable est hoistée

Accessible avant sa déclaration

Valeur non accessible avant sa déclaration

```
console.log(maFonction()); // "Hello, World!"  
  
function maFonction() {  
    return "Hello, World!";  
}  
  
console.log(maVariable); // undefined  
var maVariable = "Je suis hoistée";
```

COMPORTEMENT DE DÉCLARATION

Les déclarations de variables avec `var` et de fonctions sont déplacées au sommet de la **portée**, mais les **affectations** ne le sont pas.

```
console.log(x); // undefined
var x = 3;
console.log(x); // 3

console.log(y); // ReferenceError: y is not defined
let y = 3;
```

EFFETS SUR LES VARIABLES ET FONCTIONS

Le **hoisting** peut causer des **erreurs inattendues**, notamment lorsqu'on utilise des variables qui n'ont pas encore été déclarées.

Sans hoisting Avec hoisting

```
var a = 5;
```

```
var b = 10;
```

```
var c = a + b;
```

```
a = 5;
```

```
b = 10;
```

ASTUCES POUR PRÉVENIR LES PROBLÈMES

- Utilisez `let` et `const` plutôt que `var`
- Déclarez les **variables** et les **fonctions** avant de les utiliser

IMMUABILITÉ

L'**immutabilité** est la garantie qu'un objet ne sera pas modifié une fois créé.

Avantages :

- Prévention des bugs liés à la modification involontaire des objets
- Amélioration de la lisibilité du code
- Facilité de gestion de la concurrence (pour les applications multi-threadées)

Inconvénients :

- Peut entraîner une utilisation plus importante de la mémoire, car de nouveaux objets sont créés à chaque mise à jour.

Exemples d'opérations immuables en JavaScript :

```
const array1 = [1, 2, 3];
const array2 = array1.concat(4); // Crée un nouvel array avec l'élément 4 ajouté
```

```
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // Crée un nouvel objet avec les propriétés de obj1 et la nouvelle propriété c
```

UTILISATION DE CONST

`const` garantit que la référence à une variable est **immuable**, mais l'objet lui-même peut encore être modifié.

OBJETS ET TABLEAUX EN JAVASCRIPT

Les **objets** et **tableaux** sont **mutables** et peuvent être modifiés même lorsqu'ils sont référencés par une variable const.

```
const obj = {  
    prop1: "Valeur initiale",  
};  
  
const arr = [1, 2, 3];  
  
obj.prop1 = "Valeur modifiée";  
arr[1] = 5;
```

Avant modification

obj.prop1 = 'Valeur initiale'

arr = [1, 2, 3]

Après modification

obj.prop1 = 'Valeur modifiée'

arr = [1, 5, 3]

MÉTHODES POUR ÉVITER LES MODIFICATIONS ACCIDENTELLES

- Utilisez `Object.freeze()` pour rendre un objet **immuable**
- Créez des **copies** de tableaux et d'objets avant de les modifier

Méthode	Utilisation
<code>Object.freeze()</code>	Rend l'objet immuable (non modifiable)
Copie de tableaux	<code>const newArray = [...originalArray];</code>
Copie d'objets	<code>const newObject = { ...originalObject};</code>

INTRODUCTION AU DÉBOGAGE

IMPORTANCE DU DÉBOGAGE

Le **débogage** est incontournable dans le développement logiciel. Il permet de :

- Identifier et corriger les **erreurs**
- Comprendre le fonctionnement du **code**
- Améliorer la qualité et les **performances** du logiciel

TECHNIQUES COURANTES DE DÉBOGAGE

1. `console.log()`
2. **Débogueurs intégrés** aux navigateurs et éditeurs de code
3. **Source maps**
4. Outils externes (**Chrome DevTools**, **Firefox Developer Edition**, etc.)
5. Utilisation des **bonnes pratiques** de développement (documentation, tests, etc.)

MAÎTRISER LES OUTILS DE DÉBOGAGE ET LE DÉBOGAGE

CONSOLE DU NAVIGATEUR

La console du navigateur est un **outil intégré** qui permet d'afficher les messages, d'exécuter du code **JavaScript** et de **déboguer**.

RACCOURCIS CLAVIER

- **Google Chrome**: Ctrl + Shift + J (Windows/Linux) ou Cmd + Opt + J (Mac)
- **Firefox**: Ctrl + Shift + K (Windows/Linux) ou Cmd + Opt + K (Mac)
- **Safari**: Cmd + Opt + C (Mac)

ACCÈS VIA LES MENUS

- **Google Chrome** : Menu (3 points verticaux) -> Plus d'outils -> Outils de développement
- **Firefox** : Menu (3 barres horizontales) -> Développeur Web -> Console Web
- **Safari** : Menu (Safari) -> Préférences -> Avancé -> Afficher le menu Développement -> Développer -> Afficher la console d'erreur

CONSOLE.LOG()

Affiche des messages dans la **console** :

```
console.log("Hello, World!");
```

CONSOLE.ERROR()

Affiche des **erreurs** dans la console :

```
console.error("Erreur!");
```

CONSOLE.WARN()

Affiche des **avertissements** dans la console:

```
console.warn("Attention!");
```

CONSOLE.TABLE()

Affiche un tableau dans la console:

```
const objArray = [
  { a: 1, b: 2 },
  { a: 3, b: 4 },
];
console.table(objArray);
```


CONSOLE.GROUP()

Crée un groupe de messages dans la console :

```
console.group("Groupe 1");
console.log("Message 1");
console.log("Message 2");
console.groupEnd();
```

CONSOLE.GROUPEND()

Ferme le **groupe** de messages en cours.

CONSOLE.TIME()

Démarre un nouveau **chronomètre**.

```
console.time("Mon chronomètre");
```

CONSOLE.TIMEEND()

Arrête le **chronomètre** et affiche le **temps écoulé**.

```
console.timeEnd("Mon chronomètre");
```

DÉBOGAGE DANS L'ÉDITEUR DE CODE

CONFIGURATION ET CHOIX DE L'ÉDITEUR DE CODE

Pour déboguer dans un éditeur de code, il est important de choisir un éditeur adapté à vos besoins. Certains des exemples d'éditeurs populaires incluent:

- **Visual Studio Code**
- **Atom**
- **WebStorm**

TYPESCRIPT ET INTELLISENSE

- TypeScript peut aider au **débogage** en ajoutant des **types** aux variables.
- Intellisense facilite la détection d'erreurs en suggérant des **méthodes** et **propriétés** disponibles.

POINTS D'ARRÊT (BREAKPOINTS)

Les **points d'arrêt** vous permettent d'**arrêter l'exécution du code** à un moment précis pour faciliter le **débogage**.

CRÉATION ET GESTION DES POINTS D'ARRÊT

Pour créer un point d'arrêt, cliquez sur la **marge** à gauche du **numéro de ligne** dans votre éditeur.

POINTS D'ARRÊT CONDITIONNELS

Possibilité de définir des **conditions** pour décider si l'exécution doit être interrompue au point d'arrêt.

Situation

Vérifier la valeur d'une variable

Vérifier si une condition complexe est remplie

Solution

Utiliser un point d'arrêt normal et vérifier manuellement la valeur

Utiliser un point d'arrêt conditionnel en définissant la condition

Exemple d'utilisation d'un point d'arrêt conditionnel dans la console du navigateur :

```
for (let i = 0; i < 10; i++) {  
    console.log(`Valeur de i: ${i}`);  
    if (i === 5) {  
        console.log("La valeur de i est maintenant 5");  
    }  
}
```

Ajouter un point d'arrêt conditionnel dans la console sur la ligne du `if` en précisant la condition `i === 5`.

EXÉCUTION PAS À PAS

L'**exécution pas à pas** vous permet de suivre l'exécution du code **ligne par ligne**.

MARCHE À SUIVRE (STEP INTO)

Permet de suivre l'**exécution** à l'intérieur d'une **fonction**.

```
function operation(a, b) {  
  const somme = a + b;  
  const diff = a - b;  
  
  return [somme, diff];  
}  
  
const resultat = operation(10, 5);  
console.log(resultat);
```

MARCHE AU-DESSUS (STEP OVER)

Permet de suivre l'exécution en **sautant** (ne rentre pas à l'intérieur) les fonctions appelées.

SORTIE (STEP OUT)

Permet de **sortir** d'une fonction et de revenir à son appel lorsque l'exécution s'est effectuée à l'intérieur.

```
function calcul(param1, param2) {  
    const somme = param1 + param2;  
    return somme;  
}  
  
const resultat = calcul(1, 2);  
console.log(resultat);
```

INSPECTION DES VARIABLES

Lors du débogage, il est important de vérifier la **portée** et les **valeurs** des variables pour détecter les erreurs.

- Utilisation de `console.log(variable)` pour afficher la valeur d'une variable
- Utilisation de `debugger;` pour lancer les outils de débogage dans le navigateur

PORTEE ET VALEURS DES VARIABLES

- Accédez à la **fenêtre des variables** dans votre éditeur pour voir la **portée des variables**.
- Surveillez les changements de valeur pour détecter les erreurs.

MODIFICATION DES VARIABLES EN TEMPS RÉEL

- Il est possible de **modifier la valeur des variables en temps réel** lors du **débogage** pour tester différentes situations.
- Utilisez la **fenêtre des variables** pour modifier les valeurs selon vos besoins.

SOURCE MAPS

COMPRÉHENSION DES SOURCE MAPS

Les **source maps** permettent de relier le code de production (**minifié, transpilé**) avec le code source d'origine afin de faciliter le **débogage**.

Avantages

Facilite le débogage

Améliore la lisibilité du code

Simplifie le suivi des erreurs

Exemples d'utilisation

En cas d'erreur dans le code de production

Lors de l'inspection du code via les outils de développement du navigateur

En cas de rapport d'erreur envoyé automatiquement par les utilisateurs

GÉNÉRATION ET UTILISATION

Les **source maps** sont généralement générées automatiquement lors de la **compilation** ou du **build** et associées au fichier de production.

```
//# sourceMappingURL=my-code.min.js.map
```

ASPECTS DE PERFORMANCE

L'utilisation de **source maps** peut légèrement ralentir le chargement des pages. Il est recommandé de les utiliser uniquement en **développement**.

DÉBOGAGE DU CODE TRANSPILÉ

Source maps facilitent le débogage du code **transpilé** (TypeScript, Babel) en montrant le code source d'origine plutôt que le code transpilé dans les outils de débogage.

CONFIGURATION DU DÉBOGAGE

Pour configurer le **débogage** avec **source maps**, suivez les étapes spécifiques à l'éditeur de code, au compilateur ou à l'outil de build utilisé, comme `tsconfig.json` pour **TypeScript** ou `.babelrc` pour **Babel**.

OUTILS EXTERNES DE DÉBOGAGE

CHROME DEVTOOLS

Chrome DevTools est un ensemble d'outils de **débogage** intégrés dans le navigateur **Google Chrome**.

NETWORK TAB

Analysez les **requêtes réseau**, chronométrez les événements et examinez les détails des requêtes et des **réponses HTTP**.

Actions	Description
Analyser les requêtes réseau	Identifier les appels HTTP effectués
Chronométrier les événements	Mesurer le temps de chargement des ressources
Examiner les détails	Visualiser les en-têtes et le contenu des requêtes et réponses

PERFORMANCE TAB

Profilez la **performance** de votre site web (calcul, **FPS**, utilisation de la mémoire) et enregistrez des **instantanés** de l'activité des pages.

MEMORY TAB

Inspectez et optimisez l'utilisation de la mémoire de votre **application**.

FIREFOX DEVELOPER EDITION

Firefox Developer Edition est une version de Firefox spécialement conçue pour les **développeurs** avec des **outils de débogage améliorés**.

NETWORK MONITOR

Visualisez les **requêtes réseau** en détail, inspectez les **en-têtes**, les **réponses**, les **cookies** et les **données envoyées et reçues**.

Tableau des types de données inspectées :

Types	Description
En-têtes	Les métadonnées fournies
Réponses	Datos reçus du serveur
Cookies	Informations de session
Données envoyées	Datos envoyés au serveur

PERFORMANCE TAB

Mesurez l'impact des **fonctionnalités** et des **changements de code** sur la performance globale de votre site web.

Colonne 1	Colonne 2
Audit du site	Identifiez les problèmes de performance
Optimisation	Apportez des améliorations au code
Surveillance	Suivez les changements au fil du temps
Analyse des données	Interprétez les résultats pour prendre des décisions éclairées

MEMORY TAB

Analysez l'utilisation de la mémoire de votre application pour identifier les problèmes de **performance** et les **fuites de mémoire**.

- Utilisation de la mémoire
- Allocation de mémoire
- Désallocation de mémoire
- Profilage de mémoire

ID	Description de l'action	Utilisation mémoire (avant)	Utilisation mémoire (après)
1	Chargement de l'application	50 Mo	80 Mo
2	Exécution d'une fonction spécifique	80 Mo	120 Mo
3	Suppression d'un composant / libération	120 Mo	70 Mo

NODE.JS DEBUGGING

Node.js propose des **options de débogage intégrées** pour faciliter le développement d'applications.

- Utilisation de l'option `--inspect`
- Utilisation de la fonction `debugger;`
- Utilisation des DevTools de Chrome pour le débogage

Option	Utilisation
<code>node --inspect script.js</code>	Permet d'activer le débogage pour le fichier <code>script.js</code>
<code>node --inspect-brk script.js</code>	Active le débogage et met en pause l'exécution de <code>script.js</code> avant d'exécuter la première instruction
<code>debugger;</code>	Insérer cette instruction dans le script pour créer un point d'arrêt

UTILISATION DE L'INSPECTEUR INTÉGRÉ

Démarrez **Node.js** avec l'option `--inspect` pour activer l'**inspecteur intégré** et déboguez facilement vos applications **Node.js**.

ÉDITION À DISTANCE VIA L'EXTENSION CHROME DEVTOOLS

Utilisez l'extension **Chrome DevTools** pour **Node.js** pour connecter et **déboguer** des applications Node.js à distance directement dans votre navigateur.

BONNES PRATIQUES DE DÉBOGAGE

COMMENT DÉBOGUER EFFICACEMENT

Voici quelques étapes pour vous aider à déboguer efficacement:

- 1. Comprendre le problème**
- 2. Reproduire le problème**
- 3. Utiliser les outils de débogage appropriés**
- 4. Isoler la cause du problème**
- 5. Tester la solution**

IMPORTANCE DE LA DOCUMENTATION ET DES COMMENTAIRES

Une **documentation** claire et des **commentaires** appropriés dans le code facilitent la compréhension du fonctionnement du code et la résolution des problèmes.

- Documenter les **fonctions**, les **paramètres** et les **résultats attendus**
- Expliquer les parties complexes du code avec des commentaires

RÉSOLUTION DES PROBLÈMES COURANTS

Voici quelques problèmes courants et comment les résoudre:

- **SyntaxError:** Vérifiez les erreurs de syntaxe (manque de points-virgules, erreurs de parenthèses, etc.)
- **TypeError:** Vérifiez que les types des variables sont corrects
- **Logique:** Revoyez les conditions, les boucles et les opérations pour identifier les erreurs de logique

DEBUG VS RELEASE MODE

Il existe deux modes pour exécuter votre code: **Debug** et **Release**.

Mode	Description	Utilisation
Debug	Facilite le débogage avec des informations supplémentaires et des optimisations désactivées	Pendant le développement et le débogage
Release	Optimise le code pour la performance et supprime les informations de débogage inutiles	Pour la version finale du code

DÉFINITION

Un **objet** en JavaScript est une collection de **propriétés**. En JS, **tout** est un objet : fonctions, tableaux, dates, regex...

PROPRIÉTÉS

Les **propriétés** sont des valeurs associées à un objet. On y accède en utilisant la notation pointée :
objet.propriété.

Exemples de propriétés:

```
let voiture = {  
    marque: "Toyota",  
    modele: "Corolla",  
    annee: 2020,  
};  
  
console.log(voiture.marque); // Affiche 'Toyota'  
console.log(voiture.modele); // Affiche 'Corolla'  
console.log(voiture.annee); // Affiche 2020
```

MÉTHODES

Les **méthodes** sont des fonctions associées à un **objet**. On les appelle avec la notation pointée :
objet.methode () .

```
let voiture = {  
    marque: "Tesla",  
    modele: "Model 3",  
    afficherDetails: function () {  
        console.log("Marque : " + this.marque + ", Modèle : " + this.modele);  
    },  
};  
  
voiture.afficherDetails(); // Affiche "Marque : Tesla, Modèle : Model 3"
```


DÉFINITION

Une **classe** est un modèle pour créer des **objets** avec des **propriétés** et des **méthodes** particulières.

CONSTRUCTEURS

Les **constructeurs** définissent la manière dont les objets d'une classe sont créés et initialisés. On les écrit dans la classe avec `constructor ()`.

```
class Personne {  
    constructor(nom, prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}  
  
const personne1 = new Personne("Dupont", "Jean");  
console.log(personne1.nom, personne1.prenom); // "Dupont Jean"
```


DÉFINITION

`**this**` représente le **contexte** de l'appel d'une méthode ou l'**objet** qui l'appelle.

UTILISATION DANS LES OBJETS ET MÉTHODES

`this` permet d'accéder aux **propriétés** et **méthodes** de l'objet courant.

```
class Exemple {
    constructor() {
        this.propriete = "valeur";
    }
    methode() {
        console.log(this.propriete);
    }
}
```

PORTÉE

La portée de `**this**` dépend de la manière dont la fonction est appelée. Il peut être modifié avec `**.bind()**`, `**.call()**` ou `**.apply()**`.

PROGRAMMATION OBJET ET MODULAIRE EN JAVASCRIPT

DÉFINITION ET AVANTAGES

Les **modules** permettent d'**organiser** et de **découper** votre code en différents fichiers. Les avantages sont :

- **Réutilisation** du code
- Facilité de **maintenance**
- Simplification du **déboggage**
- **Encapsulation** et gestion des **dépendances**

SYNTAXE

Le **Module pattern** est un design pattern permettant d'encapsuler le code en utilisant une **fonction anonyme immédiatement invoquée (IIFE)**.

```
const monModule = (function () {
    // Variables et fonctions privées
    let privateVar = "Secret";

    function privateFunc() {
        return privateVar;
    }

    // Exposer des variables et fonctions publiques
    return {
        publicVar: "Accès libre",
        publicFunc: function () {
            return privateFunc();
        },
    };
});
```


SYNTAXE

La syntaxe `import` et `export` permet de partager des éléments entre plusieurs fichiers.

- Export :

```
// fichier utils.js
export function utilA() { ... }
export function utilB() { ... }
```

- Import :

```
// fichier main.js
import { utilA, utilB } from "./utils.js";

utilA();
utilB();
```

EXEMPLES D'UTILISATION

- **export par défaut :**

```
// fichier utils.js
export default function utilA() { ... }

// fichier main.js
import utilA from './utils.js';

utilA();
```

- **export nommé :**

```
// fichier utils.js
export function utilA() { ... }

// fichier main.js
import { utilA } from './utils.js';

utilA();
```

BUNDLERS ET BUILD TOOLS

Bundlers et build tools permettent d'optimiser, transformer et empaqueter le code pour un déploiement en production.

- **Webpack** : Un bundler populaire qui permet de gérer les dépendances, les transformations et l'empaquetage du code.
- **Rollup** : Un bundler axé sur la rapidité et la modularité.
- **Parcel** : Un bundler simple et facile à configurer, idéal pour les petits projets ou ceux qui débutent avec les bundlers.

FONCTIONS DÉCLARÉES (FUNCTION DECLARATIONS)

FONCTIONS DÉCLARÉES (FUNCTION DECLARATIONS)

Les **fonctions déclarées** sont des fonctions définies avec le mot-clé "**function**", un **nom** et un **corps de fonction**.

```
function maFonction(param1, param2) {  
    // Corps de la fonction  
    // Traitement des paramètres et autres opérations  
}
```

SYNTAXE

Pour déclarer une **fonction**, utilisez la syntaxe suivante :

```
function nomDeLaFonction(param1, param2, ...) {  
    // Corps de la fonction  
}
```


HOISTING

Les fonctions **déclarées** sont **hissées** (hoisted) en haut de leur portée, ce qui permet de les appeler avant de les déclarer.

```
console.log(carre(5)); // 25

function carre(x) {
    return x * x;
}
```

LISIBILITÉ DU CODE

Les **fonctions déclarées** sont explicitement nommées, ce qui facilite la **compréhension** et le **débogage** du code.

FONCTIONS EXPRIMÉES (FUNCTION EXPRESSIONS)

SYNTAXE

Les **fonctions exprimées** sont des **fonctions anonymes** assignées à des variables ou utilisées comme arguments dans d'autres fonctions.

```
const maFonction = function (parametre) {
    // Corps de la fonction exprimée
};

function fonctionParente(callback) {
    // Corps de la fonction parente
    callback();
}

fonctionParente(maFonction);
```


CRÉATION DE FONCTIONS ANONYMES

```
setTimeout(function () {  
    console.log("Exécuté après 1 seconde");  
, 1000);
```

Fonctions anonymes : fonctions sans nom, souvent utilisées comme fonctions de rappel.

FONCTIONS FLÉCHÉES (Arrow Functions)

FONCTIONS FLÉCHÉES (Arrow Functions)

Les **fonctions fléchées** sont une syntaxe plus concise pour écrire des fonctions en **JavaScript**.

COMPARAISON DE LA SYNTAXE

Fonction traditionnelle :

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

Fonction fléchée :

```
const addition = (a, b) => a + b;
```

AVANTAGES DES FONCTIONS FLÉCHÉES

- Syntaxe plus concise
- Pas de liaison de `this`, arguments et `super`
- Non constructible (ne peut pas être utilisée avec `new`)

SYNTAXE

La syntaxe pour les **fonctions fléchées** est:

```
(param1, param2, ..., paramN) => { instructions }
(param1, param2, ..., paramN) => expression
```

- Utilisation du symbole "**=>**"
- Syntaxe **concise** pour les fonctions à une seule instruction
- Parenthèses **optionnelles** pour un seul paramètre

AVANTAGES

- **Syntaxe plus courte**
- Résolution du problème de "**this**"
- Utilisation dans les **expressions fonctionnelles**

Arrow Function Fonction traditionnelle

() => { }	function () { }
------------	-----------------

(x) => x * 2	function(x) {return x * 2;}
--------------	-----------------------------

SYNTAXE PLUS COURTE

Les **fonctions fléchées** ont une syntaxe plus **concise**, ce qui rend le code plus **lisible** et **facile à maintenir**.

```
// Syntaxe de fonction standard
function addition(a, b) {
    return a + b;
}

// Syntaxe de fonction fléchée
const addition = (a, b) => a + b;
```

RÉSOLUTION DU PROBLÈME DE "THIS"

Les **fonctions fléchées** héritent du contexte `this` de leur portée englobante, ce qui permet d'éviter les problèmes liés à la liaison de `this`.

```
class ExempleClasse {
    constructor() {
        this.variableClasse = "Variable de classe";
    }

    method() {
        setTimeout(() => {
            console.log(this.variableClasse);
        }, 1000);
    }
}

const exemple = new ExempleClasse();
exemple.method();
```

UTILISATION DANS LES EXPRESSIONS FONCTIONNELLES

Les **fonctions fléchées** sont particulièrement utiles dans le cadre de la **programmation fonctionnelle**, car elles facilitent la création de fonctions **anonymes** à utiliser comme argument pour d'autres fonctions.

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

// Utilisation d'une fonction fléchée comme argument d'une fonction de traitement
const doubled = numbers.map((num) => num * 2);

console.log(doubled); // [2, 4, 6, 8, 10]
```

FONCTIONS AUTO-INVOQUÉES (IMMEDIATELY INVOKED FUNCTION EXPRESSIONS, IIFE)

SYNTAXE

Pour créer une **fonction auto-invoquée**, entourez la fonction exprimée de parenthèses et ajoutez une autre paire de parenthèses pour l'invoquer immédiatement après la déclaration.

```
(function () {  
    // Code à exécuter  
})();
```


ENCAPSULATION DES VARIABLES

Les variables déclarées dans une **IIFE** ne sont pas accessibles à l'extérieur de la fonction, ce qui permet d'éviter les **conflits de noms** et de protéger les **données sensibles**.

```
(function () {
    var variableCachee = "Je suis cachée à l'extérieur";
})();

console.log(variableCachee); // Erreur : variableCachee n'est pas définie
```

EXÉCUTION UNE FOIS SANS POLLUER LA PORTÉE GLOBALE

Les **IIFE** (Immediately Invoked Function Expression) sont exécutées une seule fois au moment de leur déclaration et ne polluent pas la portée globale. Elles sont utiles pour initier des **modules** ou des **fonctionnalités** sans exposer de variables ou de fonctions globales.

```
(function () {  
    // Code à l'intérieur de l'IIFE  
    console.log("Exécuté une fois sans polluer la portée globale");  
})();
```

FONCTIONS GÉNÉRATRICES (Generator Functions)

FONCTIONS GÉNÉRATRICES (Generator Functions)

Les **fonctions génératrices** sont des fonctions spéciales qui permettent de créer des **générateurs** pour simplifier le code **asynchrone** et la gestion des états.

```
function* maFonctionGeneratrice() {
  yield 1;
  yield 2;
  yield 3;
}

const generateur = maFonctionGeneratrice();

console.log(generateur.next().value); // 1
console.log(generateur.next().value); // 2
console.log(generateur.next().value); // 3
```

SYNTAXE

Pour déclarer une **fonction génératrice** en JavaScript, il faut utiliser le mot-clé `function*` et l'instruction `yield`:

```
function* generatorFunction() {  
    yield value1;  
    yield value2;  
}
```

EXEMPLES D'UTILISATION

```
function* idGenerator() {
  let id = 1;
  while (true) {
    yield id++;
  }
}

const gen = idGenerator();

console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

AVANTAGES

Les **fonctions génératrices** ont plusieurs avantages:

- **Pause** et **reprise** de l'exécution de la fonction
- Simplification du **code asynchrone** et de la **gestion des états**

Avantages	Explication
Pause et reprise d'exécution	Le code s'exécute de manière séquentielle
Simplification code asynchrone	Simplifie la gestion des opérations non bloquantes

FONCTIONS DE RAPPEL (CALLBACK FUNCTIONS)

DÉFINITION

Les **fonctions de rappel** sont des fonctions passées en argument à une autre fonction et sont exécutées ultérieurement par la fonction appelante.

Avantages

Gestion de l'asynchronisme

Inconvénients

Peut devenir difficile à lire (Callback Hell)

Modularité

Gestion d'erreur moins intuitive

- Exemple :

```
function fonctionParente(callback) {
    // Code de la fonction parente
    callback();
}

fonctionParente(function () {
    console.log("Fonction de rappel appelée");
});
```

EXEMPLES D'UTILISATION

```
function calculate(num1, num2, operation) {
    return operation(num1, num2);
}

function sum(a, b) {
    return a + b;
}

calculate(4, 2, sum); // 6
```

Utilisation de fonctions de rappel pour les calculs

Note : Cet exemple montre comment passer une fonction en tant que paramètre et l'appeler à l'intérieur d'une autre fonction.

```
setTimeout(function () {
    console.log("Exécution après 3 secondes");
}, 3000);
```

Utilisation de fonctions de rappel pour des timeouts

Note : Cet exemple illustre comment les fonctions de rappel sont utilisées pour gérer les événements asynchrones, tels que les timeouts.

AVANTAGES

1. **Gestion des événements** et des **opérations asynchrones**
2. **Personnalisation du comportement** des fonctions existantes
3. **Contrôle** du moment et de la manière dont la fonction de rappel est exécutée

INTRODUCTION À JQUERY

QU'EST-CE QUE JQUERY ?

jQuery est une **bibliothèque JavaScript** open-source, légère et rapide qui simplifie la manipulation du **HTML DOM**, la gestion des **événements**, l'**animation** et l'utilisation d'**AJAX**.

Avantages de jQuery Exemples d'utilisation

Simplifie le code

Manipulation du DOM

Léger et rapide

Gestion des événements

Open-source

Animation

Compatible

AJAX

AVANTAGES DE L'UTILISATION DE JQUERY

- **Facilité d'utilisation**
- **Grande communauté** et support
- **Compatibilité entre navigateurs**
- Large gamme de **plugins**
- **Animation** et **effets simplifiés**

INCLUSION DE LA BIBLIOTHÈQUE JQUERY

Pour inclure **jQuery**, ajoutez cette ligne dans la balise `<head>` de votre fichier HTML :

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
```

Ou en téléchargeant le fichier et en l'incluant **localemement** :

```
<script src="jquery-3.6.0.min.js"></script>
```

JQUERY : SÉLECTEURS

INTRODUCTION AUX SÉLECTEURS JQUERY

Les sélecteurs jQuery permettent de **sélectionner** et **manipuler** les éléments HTML de manière simple et efficace.

- Exemple de sélection d'un élément par son ID :

```
$( "#monElement" );
```

- Exemple de sélection d'éléments par leur classe :

```
$( ".maClasse" );
```

- Exemple de sélection d'éléments par leur type :

```
$( "input[type='text']" );
```

SÉLECTEURS BASÉS SUR LES ÉLÉMENTS

Ils permettent de sélectionner des éléments HTML en fonction de leur **nom de balise**.

Exemple :

```
$( "p" ); // Sélectionne tous les éléments p  
$( "div" ); // Sélectionne tous les éléments div
```

SÉLECTEURS BASÉS SUR LES ATTRIBUTS

Ils permettent de sélectionner des éléments **HTML** en fonction de leurs **attributs**.

Exemple :

```
$( "input[type='text']" ); // Sélectionne tous les éléments input avec type text  
$( "a[href='#']" ); // Sélectionne tous les éléments a avec href="#"
```

SÉLECTEURS BASÉS SUR LES CLASSES ET LES ID

Ils permettent de sélectionner des **éléments HTML** en fonction de leur **classe** ou de leur **ID**.

Exemple :

```
$(".my-class"); // Sélectionne tous les éléments avec la classe my-class  
$("#my-id"); // Sélectionne l'élément avec l'ID my-id
```

SÉLECTEURS AVANCÉS

Ils permettent de sélectionner des éléments HTML avec des critères **plus spécifiques**.

Exemple :

```
$( "p:first"); // Sélectionne le premier élément p  
$( "div:last-child"); // Sélectionne les éléments div qui sont le dernier enfant
```

Sélecteur	Description
:first	Sélectionne le premier élément du type indiqué.
:last-child	Sélectionne l'élément qui est le dernier enfant.
:even	Retourne les éléments de rang pair.
:odd	Retourne les éléments de rang impair.
`[attribute=value]	Sélectionne des éléments en fonction de l'attribut et sa valeur

FILTRAGE ET RECHERCHE D'ÉLÉMENTS

Utilisez les méthodes jQuery pour **filtrer** et **rechercher** des éléments dans la sélection actuelle.

Exemple :

```
$( "p" ).first(); // Sélectionne le premier élément p  
$( "div" ).eq(2); // Sélectionne le troisième élément div
```

MANIPULATION DU DOM

MÉTHODES DE TRAVERSÉE DU DOM

jQuery offre plusieurs méthodes pour naviguer dans le **DOM** :

- **parent()**
- **children()**
- **siblings()**
- **next()**
- **prev()**

PARENT()

Sélectionne l'élément **parent** immédiat.

```
$( "span" ).parent(); // sélectionne les parents des éléments "span"
```

CHILDREN()

Sélectionne les **enfants directs** d'un élément.

```
$( "#elem" ).children(); // sélectionne les enfants directs de l'élément #elem
```

SIBLINGS()

Sélectionne les éléments frères et sœurs.

```
$( "#elem" ).siblings(); // sélectionne les éléments frères et sœurs de l'élément #elem
```

NEXT()

Sélectionne l'élément immédiatement **suivant**.

```
$( "#elem" ).next(); // sélectionne l'élément suivant #elem
```

PREV()

Sélectionne l'élément précédent.

```
$( "#elem" ).prev(); // sélectionne l'élément précédent #elem
```

MODIFICATION DES ÉLÉMENTS

Méthodes pour changer le contenu et les attributs des éléments :

- **html()**
- **text()**
- **attr()**
- **css()**
- **Ajout/suppression de classes**

Méthode	Utilisation	Exemple
html()	Pour modifier le contenu HTML d'un élément	<code>\$('p').html('Nouveau texte');</code>
text()	Pour modifier le texte uniquement d'un élément	<code>\$('p').text('Nouveau texte');</code>
attr()	Pour modifier les attributs d'un élément (premier argument) avec une valeur donnée	<code>\$('<a>').attr('href', 'https://...');</code>
css()	Pour modifier les propriétés CSS d'un élément (premier argument) avec une valeur donnée	<code>\$('p').css('color', 'red');</code>

HTML()

Change le contenu **HTML** d'un élément.

```
$( "#elem" ).html( "<strong>Nouveau contenu</strong>" );
```

TEXT()

Change le contenu texte d'un élément.

```
$( "#elem" ).text( "Nouveau contenu" );
```

ATTR()

Change les **attributs** d'un élément.

```
$( "#elem" ).attr( "src", "nouvelle-image.jpg" );
```

CSS()

Change les **styles** d'un élément.

```
$( "#elem" ).css( "color", "blue" );
```

AJOUT/SUPPRESSION DE CLASSES

Ajoute ou supprime des **classes**.

```
$( "#elem" ).addClass( "selected" );
$( "#elem" ).removeClass("selected");
```

AJOUT/SUPPRESSION D'ÉLÉMENTS

Méthodes pour ajouter ou supprimer des éléments :

- **append()**
- **prepend()**
- **after()**
- **before()**
- **remove()**
- **empty()**

APPEND()

Insère du contenu à la **fin** d'un élément.

```
$( "#elem" ).append( "<p>Paragraph ajouté</p>" );
```

Méthode	Description
append()	Ajoute du contenu à la fin d'un élément
prepend()	Ajoute du contenu au début d'un élément
after()	Insère du contenu après un élément spécifié
before()	Insère du contenu avant un élément spécifié

PREPEND()

Insère du contenu au début d'un **élément**.

```
$( "#elem" ).prepend( "<p>Paragraphe ajouté</p>" );
```

AFTER()

Insère du contenu **après** un élément.

```
$( "#elem" ).after( "<p>Paragraphe ajouté</p>" );
```

BEFORE()

Insère du contenu **avant** un élément.

```
$( "#elem" ).before( "<p>Paragraphe ajouté</p>" );
```

REMOVE()

Supprime un élément.

```
$( "#elem" ).remove();
```


GESTION DES ÉVÉNEMENTS

GESTION DES ÉVÉNEMENTS

jQuery simplifie la **gestion des événements** en fournissant des méthodes pour attacher/détacher des **gestionnaires d'événements** aux éléments du **DOM**.

Méthode	Description
.on(event, handler)	Attache un gestionnaire à un événement
.off(event, handler)	Détache un gestionnaire d'un événement
.trigger(event)	Déclenche un événement manuellement

Exemple :

```
$("#button").on("click", function () {
    alert("Bouton cliqué");
});

$("#button").off("click");

$("#button").trigger("click");
```

ÉVÉNEMENTS DE BASE

jQuery fournit des méthodes pour gérer les **événements de base** tels que le clic, les entrées clavier et les événements de la souris.

Événement	Description
click	L'utilisateur clique sur un élément
keyup	L'utilisateur relâche une touche du clavier
keydown	L'utilisateur appuie sur une touche du clavier
hover	L'utilisateur place la souris sur un élément

DÉLÉGATION D'ÉVÉNEMENTS

La **délégation d'événements** permet d'attacher des **gestionnaires d'événements** aux éléments présents et futurs qui correspondent à un sélecteur donné.

Avantages

Économie de mémoire

Événements gérés pour les éléments ajoutés dynamiquement

Inconvénients

Dépend du contexte d'utilisation

Peut générer des événements indésirables

```
document.addEventListener("click", function (event) {
  if (event.target.matches(".monElement")) {
    console.log("L'élément a été cliqué");
  }
});
```


OFF()

```
$( "body" ).off( "click", "button" );
```

La méthode `off()` permet de détacher un **gestionnaire d'événements** précédemment attaché avec `on()`.

EFFETS ET ANIMATIONS

EFFETS DE BASE

jQuery propose des **effets visuels simples** pour donner vie à vos pages web.

```
.show()      // Affiche un élément
.hide()     // Cache un élément
.toggle()   // Affiche/cache un élément alternativement
.fadeIn()    // Fait apparaître un élément en estompant
.fadeOut()   // Fait disparaître un élément en estompant
.slideDown() // Fait apparaître un élément avec un effet de glissement
.slideUp()   // Fait disparaître un élément avec un effet de glissement
```

EFFETS DE BASE (SUITE)

Exemples d'utilisation :

```
$( "#monElement" ).show();
$( "#monElement" ).hide();
$( "#monElement" ).toggle();
$( "#monElement" ).fadeIn("slow");
$( "#monElement" ).fadeOut(2000);
$( "#monElement" ).slideDown();
$( "#monElement" ).slideUp();
```

Ces méthodes prennent en **option** une **durée** (en ms) ou l'une des chaînes : "**slow**" ou "**fast**".

ANIMATIONS PERSONNALISÉES

La méthode `.animate()` permet de créer des animations personnalisées en modifiant les **propriétés CSS** d'un élément.

```
$( "#monElement" ).animate(
{
    opacity: 0.5,
    left: "+=100px",
},
1000
);
```

Dans cet exemple, l'élément devient **semi-transparent** et se **déplace** de 100 pixels à droite.

ANIMATIONS PERSONNALISÉES (SUITE)

La méthode `.animate()` peut être chaînée pour créer des **séquences d'animations** :

```
$( "#monElement" )
  .animate({ opacity: 0.5 }, 1000)
  .animate({ left: "+=100px" }, 1000)
  .animate({ opacity: 1 }, 1000);
```

Cette séquence rend l'élément **semi-transparent**, le déplace de **100 pixels à droite**, puis le rend à nouveau **opaque**.

AJAX AVEC JQUERY

INTRODUCTION

jQuery simplifie l'utilisation d'**AJAX** (Asynchronous JavaScript and XML) pour effectuer des **requêtes HTTP asynchrones** dans les applications web.

LOAD()

La méthode `load()` permet de charger du contenu **HTML** depuis un fichier ou une **URL** et de l'insérer dans un élément DOM.

```
$( "#content" ).load("example.html");
```


GET()

La méthode get () permet d'effectuer une **requête HTTP GET asynchrone** et de traiter les données reçues.

```
$.get("example.php", function (data) {  
    $("#content").html(data);  
});
```

POST()

La méthode post () permet d'effectuer une **requête HTTP POST asynchrone** et de traiter les données reçues.

```
$.post("example.php", { data: "example" }, function (response) {
    $("#content").html(response);
});
```

AJAX()

La méthode `ajax()` permet de contrôler précisément les **requêtes HTTP asynchrones**.

```
$.ajax({
  url: "example.php",
  method: "GET",
  data: { data: "example" },
  dataType: "json",
  success: function (response) {
    $("#content").html(response);
  },
  error: function () {
    alert("Erreur");
  },
});
```


UTILISATION DE CHAÎNES

Chainer les méthodes **jQuery** permet de réduire le nombre de sélections et de manipulations du **DOM**.

```
$( "#element" ).find(".classe").css("color", "blue").text("Exemple");
```

RÉUTILISATION D'OBJETS JQUERY

Réutiliser les objets jQuery permet de réduire le **temps de calcul** en évitant de répéter les **sélections**.

```
var $element = $("#element");
$element.hide();
// ... Autres opérations
$element.show();
```

CHOIX DE SÉLECTEURS EFFICACES

Utiliser des sélecteurs **ID** ou des sélecteurs **spécifiques** pour une performance optimale.

```
$( "#element" ).find( ".classe" );
```

UTILISATION DE VALEURS MISES EN CACHE

Mettre en cache les **valeurs réutilisées** permet de réduire le temps de calcul.

```
var $element = $("#element");
var width = $element.width();
```

OPTIMISATION ET BONNES PRATIQUES

UTILISATION DE CHAÎNES

Chaque fois que possible, enchaînez les **méthodes** pour limiter la recherche d'éléments dans le **DOM** et optimiser la **performance** :

```
$( "div" ).css( "color", "blue" ).slideUp( "slow" ).slideDown( "fast" );
```

RÉUTILISATION D'OBJETS JQUERY

Réutilisez les objets **jQuery** en les stockant dans des **variables** plutôt que de les rechercher à plusieurs reprises :

```
var div = $("div");
div.css("color", "blue");
div.slideUp("slow");
```

CHOIX DE SÉLECTEURS EFFICACES

Sélectionnez des éléments de manière optimale en :

- Utilisant les sélecteurs **ID** et **classes**
- N'utilisant pas de sélecteurs universels (*)
- En évitant les sélecteurs d'attributs **complexes**

UTILISATION DE VALEURS MISES EN CACHE

Mettez en cache les éléments déjà recherchés et les résultats de fonctions **coûteuses** pour éviter des calculs et recherches inutiles:

```
var windowWidth = $(window).width();  
  
if (windowWidth > 768) {  
    // ...  
}
```

DÉTACHEMENT TEMPORAIRE D'ÉLÉMENTS POUR UNE MANIPULATION EN MASSE

Pour modifier plusieurs éléments à la fois, détachez-les temporairement du **DOM** pour réduire les opérations de **repaint** :

```
var items = $("li");
items.detach();
items.sort(sortFunction);
$("ul").append(items);
```

STRUCTURATION ET MODULARITÉ

AVANTAGES DE LA MODULARITÉ

- **Lisibilité**
- **Maintenance**
- **Réutilisabilité**

LISIBILITÉ

La **modularité** permet de séparer le code en **unités logiques**, facilitant sa compréhension.

MAINTENANCE

Les **modules indépendants** facilitent la résolution des problèmes et la **mise à jour du code**.

RÉUTILISABILITÉ

Les **modules** peuvent être réutilisés dans plusieurs projets, évitant la duplication du code.

TECHNIQUES D'ORGANISATION DE CODE

- **IIFE** (Immediately Invoked Function Expression)
- **Modules ES6** (import/export)
- **Namespace**

IIFE (IMMEDIATELY INVOKED FUNCTION EXPRESSION)

```
(function () {  
    // instructions  
})();
```

Permet de créer une **portée privée** pour les variables et les fonctions.

MODULES ES6 (IMPORT/EXPORT)

```
// module.js
export const maVariable = "Hello!";
export function maFonction() {}

// main.js
import { maVariable, maFonction } from "./module";
```

Permet d'**importer** et **exporter** des éléments d'un module.

Import	Export
import ... from ...	export const/let ...
import ...	export function ...

NAMESPACE

```
const MonNamespace = {  
    maVariable: "Hello!",  
    maFonction: function () {},  
};
```

Permet de regrouper les éléments d'un **module** sous un **objet global**.

AMD (ASYNCHRONOUS MODULE DEFINITION)

PRINCIPE

AMD (Asynchronous Module Definition) est une spécification pour charger et gérer des modules **JavaScript** de manière **asynchrone**, ce qui permet un chargement rapide et efficace des applications sur le Web.

COMPATIBILITÉ

AMD est compatible avec tous les **navigateurs modernes** et peut également fonctionner avec des environnements tels que **Node.js**.

AVANTAGES ET INCONVÉNIENTS

Avantages

Chargement asynchrone des modules

Chargement **rapide** des applications

Facilité d'intégration avec d'autres **bibliothèques / API**

Inconvénients

Plus complexe que les approches synchrones

Moins populaire que d'autres systèmes de modules

FONCTIONNEMENT DE AMD

- `define ()`: Permet de créer des modules en définissant les dépendances et le code du module
- `require ()`: Utilisé pour charger des modules et exécuter du code après leur chargement

COMPARAISON AVEC D'AUTRES SYSTÈMES DE MODULES

Système	Compatibilité	Syntaxe	Avantages	Inconvénients
AMD	Navigateur	<code>define()</code> , <code>require()</code>	Asynchrone, modulaire, rapide	Plus complexe, moins populaire
CommonJS	Serveur	<code>module.exports</code> , <code>require()</code>	Simple, synchronisé, populaire	Pas adapté aux navigateurs, moins performant
UMD	Universel	Compatibilité AMD / CommonJS	Flexibilité, polyvalence	Plus complexe, moins performant que l'ES6

REQUIRE.JS

BUT

Require.js est une bibliothèque **JavaScript** qui implémente le standard **AMD** et facilite le chargement **asynchrone** et la gestion des **dépendances** entre les modules.

FONCTIONNALITÉS

- **Chargement asynchrone** de modules
- Gestion des **dépendances**
- Optimisation et **concaténation** des fichiers

COMPATIBILITÉ

Require.js est compatible avec la plupart des **navigateurs modernes** et des environnements tels que **Node.js**.

TÉLÉCHARGEMENT

Téléchargez **Require.js** depuis le site officiel: <https://requirejs.org/>

CONFIGURATION

Définissez les options de configuration dans un fichier séparé ou dans une balise script:

```
require.config({
  baseUrl: "js",
  paths: {
    jquery: "libs/jquery-3.6.0.min",
    myModule: "modules/my-module",
  },
  shim: {
    jquery: {
      exports: "$",
    },
  },
});
```

UTILISATION DANS UN PROJET

Incluez **Require.js** dans votre fichier **HTML**:

```
<script data-main="app" src="js/libs/require.js"></script>
```


SYNTAXE

Utilisez la méthode `define` pour créer un **module** avec ses **dépendances**:

```
define(["jquery", "myModule"], function ($, myModule) {  
    // code du module  
});
```

DÉPENDANCES

Déclarez les **dépendances** du module dans un tableau en argument de `define` et utilisez les objets passés à la fonction de rappel:

```
define(["jquery", "myModule"], function ($, myModule) {
    // utilisation de $ et myModule
});
```


R.JS (OUTIL D'OPTIMISATION)

Utilisez l'outil d'optimisation "**r.js**" pour **concaténer** et **minifier** les fichiers de votre projet:
<https://requirejs.org/docs/optimization.html>

GESTION DES ERREURS ET DES DÉPENDANCES MANQUANTES

Utilisez la méthode `**require.onError**` pour gérer les erreurs:

```
require.onError = function (error) {
  console.log("Error:", error);
};
```

PRATIQUES RECOMMANDÉES

- Utilisez des noms de fichiers et des identifiants de modules **cohérents**
- Séparez la **configuration** et le **code** de votre application
- Évitez de surcharger la **configuration globale**
- Utilisez des **chemins relatifs** dans les dépendances de modules
- Optimisez votre projet pour la **production** avec **r.js**

