Clean Code

Jérôme Commaret - Janvier 2026

Votre intervenant

Qui suis-je ?

- 15 ans d’expérience dans le digital
- 7 ans en gestion de projet, 8 ans en temps de dev
- Travaillé pour Coca-Cola, Publicis, mais aussi en startup et en ESN
- Contributeur Open-Source sur un IDE IA
- Jeux vidéos : Trackmania, Zelda…
- Papa d’une petite fille qui a 2 ans et demi.

Vous ?

- Votre Prénom ?
- Vos passions ?
- Vous avez déjà fait des stages en dev ?
- Quel expérience du code avez vous ?
- Etes vous déjà repassés derrière du code que vous aviez fait il y a 6 mois ?

Pourquoi le clean code ?

Jérôme Commaret - Janvier 2026

Exercice

Javascript Copier

```javascript
function x(a, b) {
  let c = [];
  for (let i = 0; i < a.length; i++) {
    if (a[i] > b) c.push(a[i]);
  }
  return c;
}
```

Que fait cette fonction ?

Exercice

Javascript Copier

```javascript
function x(a, b) {
  let c = [];
  for (let i = 0; i < a.length; i++) {
    if (a[i] > b) c.push(a[i]);
  }
  return c;
}
```

Que fait cette fonction ?

Paramêtres :

a : Un **tableau de nombres** (ex: [1, 5, 10, 2]).

b : Un **nombre** servant de seuil (ex: 3).

Exercice

Que fait cette fonction ?

Paramêtres :

a : Un **tableau de nombres** (ex: [1, 5, 10, 2]).

b : Un **nombre** servant de seuil (ex: 3).

Logique :

- La fonction parcourt le tableau a

- Pour chaque élément, si sa valeur est **supérieure à b**,

elle l'ajoute à un nouveau **tableau c**

- Elle retourne le **tableau c** contenant uniquement les éléments **> b**.

C’est totalement illisible…

…Pas maintenable…

…Et on oublie la simplicité.

Refactorisation

Comment refactoriser ?

Vous avez 15 minutes

On en discute après

Comment refactoriser ?

1 - Utiliser des noms descriptifs

- Remplacer **`x`** par **`filterNumbersBelowThreshold`**
  Le nom de **fonction décrit l'action**

- Remplacer **`a`** par **`numbers`**

Indique qu'il s'agit de **`nombres`**

- Remplacer **`b`** par **`threshold`**

Indique qu'il s'agit d'un **`seuil`**

Comment refactoriser ?

1 - Utiliser des noms descriptifs

- Remplacer **`x`** par **`filterNumbersBelowThreshold`**
  Le nom de **fonction décrit l'action**

- Remplacer **`a`** par **`numbers`**

Indique qu'il s'agit de **`nombres`**

- Remplacer **`b`** par **`threshold`**

Indique qu'il s'agit d'un **`seuil`**

2 - Fonctionnalités javascript ?

- Utilisation directe de **`filter`**

pour retourner un nouveau tableau.

- supprimer le `return c`

Refacto :

```
function filterNumbersBelowThreshold(numbers, threshold) {
  return numbers.filter(number => number <= threshold);
}
```

Refacto :

C’est quoi tout ce blanc au dessus ?

```
function filterNumbersBelowThreshold(numbers, threshold) {
  return numbers.filter(number => number <= threshold);
}
```

Refacto :

C’est pour mettre un exemple d’utilisation

```
 * @example
 * // Retourne [2, 5, 3]
 * filterNumbersBelowThreshold([2, 8, 5, 3, 10], 5);
 */
function filterNumbersBelowThreshold(numbers, threshold) {
  return numbers.filter(number => number <= threshold);
}
```

Refacto :

Les paramêtres

```
 * @param {number[]} numbers - Tableau de nombres à filtrer.
 * @param {number} threshold - Valeur seuil pour la comparaison.
 * @returns {number[]} Nouveau tableau contenant uniquement les éléments <= threshold.
 *
 * @example
 * // Retourne [2, 5, 3]
 * filterNumbersBelowThreshold([2, 8, 5, 3, 10], 5);
 */
function filterNumbersBelowThreshold(numbers, threshold) {
  return numbers.filter(number => number <= threshold);
}
```

Et l’explication

Javascript Copier

```javascript
/**
 * Filtre les éléments d'un tableau qui sont inférieurs ou égaux à une valeur seuil.
 *
 * @param {number[]} numbers - Tableau de nombres à filtrer.
 * @param {number} threshold - Valeur seuil pour la comparaison.
 * @returns {number[]} Nouveau tableau contenant uniquement les éléments <= threshold.
 *
 * @example
 * // Retourne [2, 5, 3]
 * filterNumbersBelowThreshold([2, 8, 5, 3, 10], 5);
 */
function filterNumbersBelowThreshold(numbers, threshold) {
  return numbers.filter(number => number <= threshold);
}
```

Quelques stats

Les développeurs passent **42 % de leur temps** à comprendre du code existant (source : Stripe).

Un code propre réduit les bugs de **30 à 50 %** (source : IBM)

60 à 80% du coût total d'un logiciel est consacré à la maintenance (correction de bugs, évolutions, refactoring). (Source : Standish Group)

Un code mal écrit peut multiplier par **10** le temps de maintenance. (Source : IBM)

Un fail notable qui aurait pu être évité

En 1999, la sonde **Mars Climate Orbiter** (NASA) a été perdue à cause d'une **erreur d'unité** (livres vs newtons) dans le code. Coût : **327 millions de dollars**.

(Source : NASA, "Mars Climate Orbiter Failure Report", 1999)

Pause

Objectif du cours

À la fin de ce cours, vous serez capables de :

- **Comprendre** et **appliquer** les principes du Clean Code (lisibilité, simplicité, maintenabilité).
- **Structurer votre code** avec des fonctions courtes, des noms clairs et une gestion d’erreurs robuste.
- **Refactorer du code existant** pour le rendre plus propre et maintenable.
- **Écrire des tests unitaires** pour valider la qualité de leur code.
- **Travailler en équipe** en respectant des conventions de code partagées.

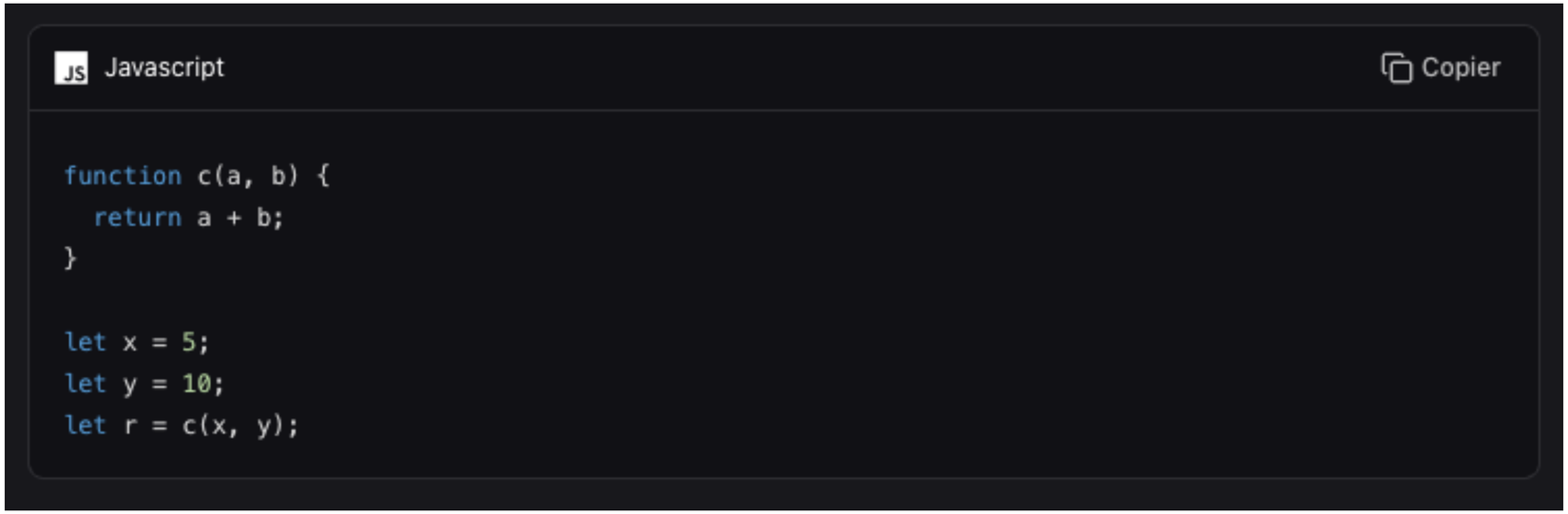
Les 5 piliers du Clean Code:

- Lisibilité
- Simplicité (KISS)
- Maintenabilité (DRY)
- Robustesse
- Testabilité

Lisibilité

Ex 1 : Auto-documentation

Nommage des fonctions et variables



Il s’agit d’une fonction qui calcule la somme de 2 nombres

Refactorisez ce code pour qu’il soit **auto-documenté** grâce au nommage des fonctions et des variables.

(20 minutes)

Nommage des fonctions et variables

Javascript Copier

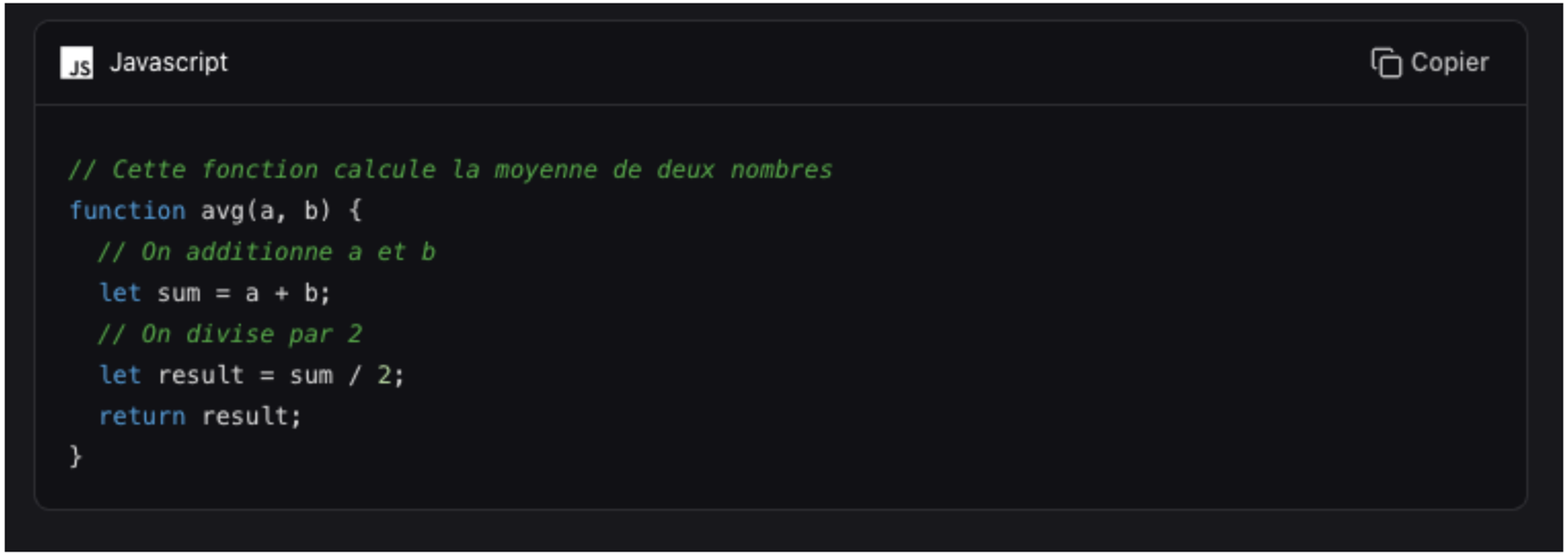
```javascript
// ✅ Solution : noms descriptifs
function calculateSum(firstNumber, secondNumber) {
  return firstNumber + secondNumber;
}

const basePrice = 5;
const taxAmount = 10;
const totalPrice = calculateSum(basePrice, taxAmount);
```

Et const car la valeur ne sera pas modifiée au moment de l’execution.

Ex 2 : Éviter les commentaires évidents

Commentaires évidents



Les commentaires sont redondants

Le nommage est trop générique

Supprimez les commentaires et améliorez le nommage pour que le code soit auto-explicatif. (20 minutes)

Commentaires évidents

Nom explicite de fonction

Nom des valeurs plus précis que A et B

Pas besoin de commentaires, le code se lit comme une phrase

Ex 3 : Fonctions courtes et focalisés

Fonction courtes et focalisés

Javascript Copier

```javascript
function processOrder(order) {
  // Vérifie si la commande est valide
  if (!order.items || order.items.length === 0) {
    throw new Error("Commande invalide");
  }

  // Calcule le total
  let total = 0;
  for (const item of order.items) {
    total += item.price * item.quantity;
  }

  // Applique une réduction si nécessaire
  if (order.customer.isPremium) {
    total *= 0.9;
  }

  // Envoie un email de confirmation
  console.log(`Confirmation envoyée à ${order.customer.email}`);

  return total;
}
```

La fonction fait trop de choses (validation, calcul, réduction, email).
Difficile à maintenir et à tester. -> a réécrire -> 30 minutes

Fonction courtes et focalisés

Javascript Copier

```javascript
// ✅ Solution : fonctions focalisées
function validateOrder(order) {
  if (!order.items || order.items.length === 0) {
    throw new Error("Commande invalide");
  }
}

function calculateTotal(order) {
  return order.items.reduce((total, item) => total + (item.price * item.quantity), 0);
}

function applyDiscount(total, customer) {
  return customer.isPremium ? total * 0.9 : total;
}

function sendConfirmationEmail(customer) {
  console.log(`Confirmation envoyée à ${customer.email}`);
}

function processOrder(order) {
  validateOrder(order);
  let total = calculateTotal(order);
  total = applyDiscount(total, order.customer);
  sendConfirmationEmail(order.customer);
  return total;
}
```

- Chaque fonction a une **responsabilité unique** (principe SRP).
- Le code est plus facile à **tester** et à **réutiliser**.
- La fonction processOrder devient une simple orchestration.

Ex 4 : Eviter les structures imbriqués

Eviter les structures imbriqués

Javascript Copier

```javascript
function getUserRole(user) {
  if (user) {
    if (user.isAdmin) {
      return "admin";
    } else {
      if (user.isEditor) {
        return "editor";
      } else {
        return "viewer";
      }
    }
  } else {
    return "guest";
  }
}
```

- Imbrication profonde (if dans if dans else).
- Difficile à suivre visuellement.

A refacto avec les « early returns » 10 minutes

Eviter les structures imbriqués

- Utilisation de **early returns** pour sortir tôt de la fonction.
- Code **plus plat** et plus facile à lire.
- Logique plus intuitive.

Pause

Ex 5 : Utilisation des Objets pour Regrouper les Paramètres

Objects pour regrouper les paramètres

Javascript Copier

```javascript
function createUser(name, age, email, isAdmin, isActive) {
  // ...
}
```

- Difficile de se souvenir de l'ordre des paramètres.
- Risque d'erreur lors de l'appel de la fonction.

Objects pour regrouper les paramètres

- **Nommage explicite** lors de l'appel de la fonction.
- **Valeurs par défaut** pour les paramètres optionnels.
- Moins de risques d'erreurs.

Recap

- Noms descriptifs pour les fonctions et les variables
- Auto-documentation
- Eviter les commentaires évidents
- Utiliser les fonctions natives
- Fonctions courtes et focalisés
- Limiter les imbrications
- Utiliser les objects pour regrouper les paramètres

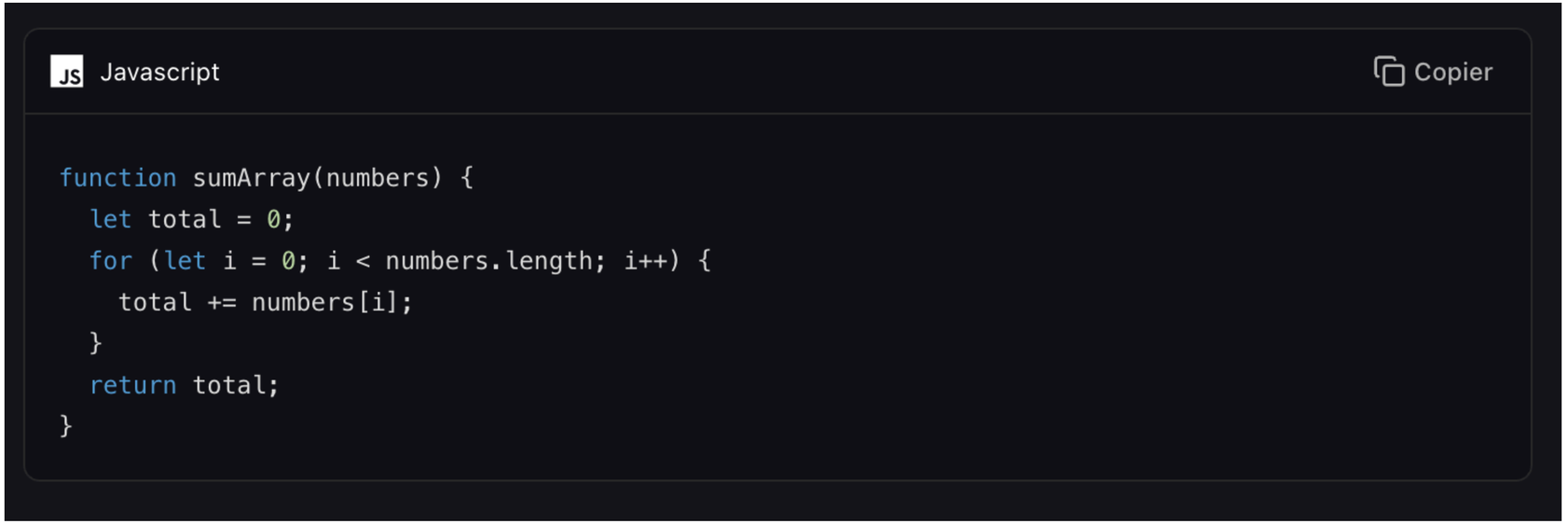
Simplicité

KISS

Keep It Simple, Stupid !

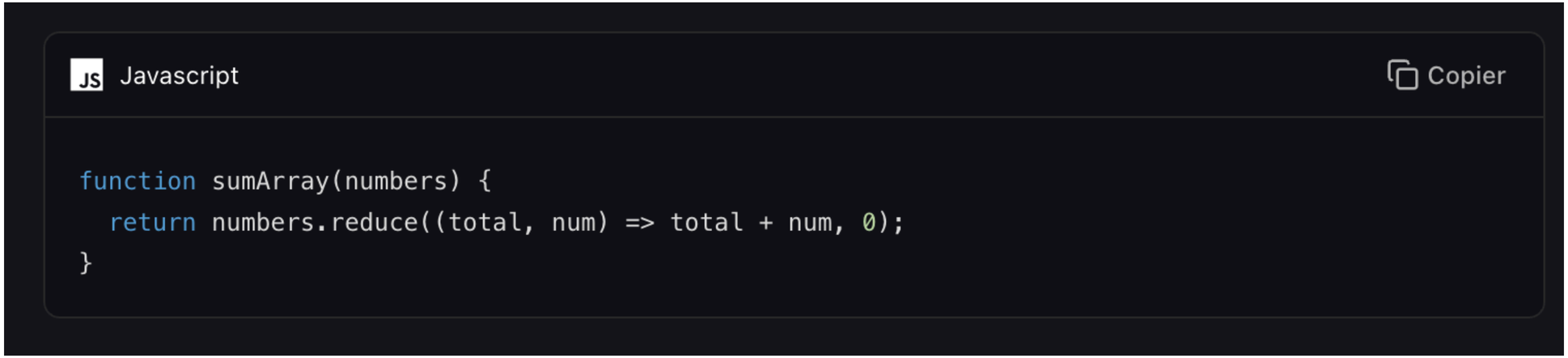
EX 1

Utiliser les fonctionnalités natives



20 minutes

Utiliser les fonctionnalités natives



- reduce est une méthode native optimisée pour les accumulations.
- Moins de code, plus lisible, et moins de risques d’erreurs (ex : oublier d’initialiser total).

EX 2

Simplicité du code ?

JS Javascript Copier

```javascript
function calculateTotal(items) {
  return items.reduce((acc, item) => {
    const price = item.price * (1 + (item.taxRate || 0) / 100);
    const discount = item.discount ? price * (item.discount / 100) : 0;
    return acc + (price - discount);
  }, 0);
}
```

A simplifier

Problèmes :

- Logique imbriquée difficile à suivre.
- Calculs complexes en une seule ligne.

Simplifié

Javascript Copier

```javascript
function calculateItemPrice(item) {
  const priceWithTax = item.price * (1 + (item.taxRate || 0) / 100);
  const discountAmount = item.discount ? priceWithTax * (item.discount / 100) : 0;
  return priceWithTax - discountAmount;
}

function calculateTotal(items) {
  return items.reduce((total, item) => total + calculateItemPrice(item), 0);
}
```

Explications

- Séparation des responsabilités (calculateItemPrice).
- Noms de variables explicites.
- Logique linéaire et facile à déboguer.

EX 3

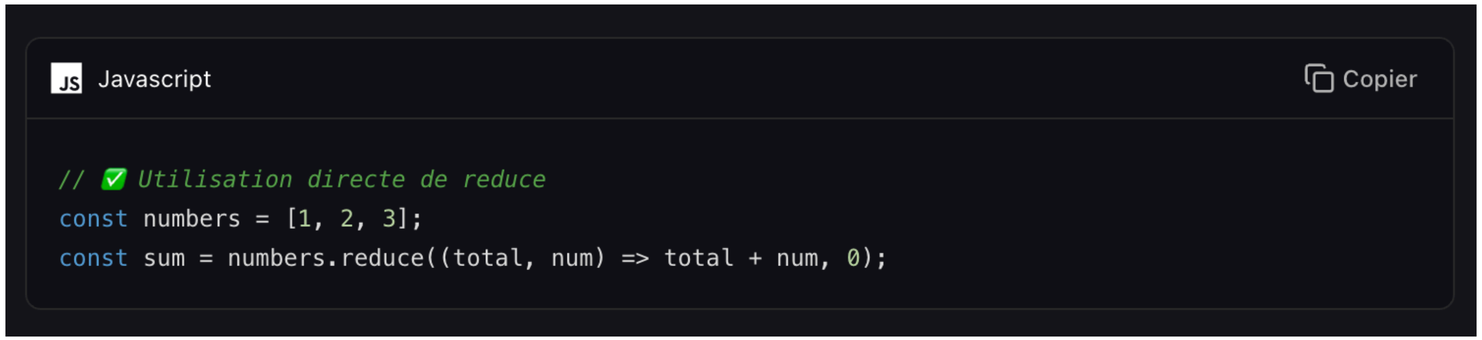
Eviter les abstractions inutiles

Javascript Copier

```javascript
// ❌ Fonction trop générique et complexe
function processCollection(collection, operation, initialValue) {
  return collection.reduce(
    (accumulator, current) => operation(accumulator, current),
    initialValue
  );
}

// Utilisation
const sum = processCollection([1, 2, 3], (a, b) => a + b, 0);
```

- La fonction `processCollection` est une réinvention de `reduce`, déjà intégrée en JavaScript.
- Le code est moins lisible qu'une utilisation directe de `reduce`.



- Utilise les fonctionnalités natives de JavaScript.
- Plus simple et plus lisible.

DRY

Don’t Repeat Yourself

Duplication de logique

Javascript Copier

```javascript
function calculerPerimetreCarre(cote) {
    return 4 * cote;
}

function calculerPerimetreRectangle(longueur, largeur) {
    return 2 * (longueur + largeur);
}

// Utilisation
console.log(calculerPerimetreCarre(5)); // 20
console.log(calculerPerimetreRectangle(5, 3)); // 16
```

Problème : La logique de calcul de périmètre est répétée, même si elle est légèrement différente.

En DRY

Javascript Copier

```javascript
function calculerPerimetre(...cotes) {
    if (cotes.length === 1) {
        return 4 * cotes[0]; // Carré
    } else if (cotes.length === 2) {
        return 2 * (cotes[0] + cotes[1]); // Rectangle
    }
    throw new Error("Nombre de côtés non supporté.");
}

// Utilisation
console.log(calculerPerimetre(5)); // 20
console.log(calculerPerimetre(5, 3)); // 16
```

Avantage : Une seule fonction gère tous les cas, et la logique est centralisée.

Messages

Javascript Copier

```javascript
function afficherErreur(message) {
    console.error(`[ERREUR] ${message}`);
}

function afficherAvertissement(message) {
    console.warn(`[AVERTISSEMENT] ${message}`);
}

function afficherInfo(message) {
    console.log(`[INFO] ${message}`);
}

// Utilisation
afficherErreur("Fichier introuvable");
afficherAvertissement("Attention, stockage plein");
afficherInfo("Tâche terminée");
```

Problème : La structure des messages est répétée, seule la méthode de log change.

Messages (En DRY)

Javascript Copier

```javascript
function afficherMessage(niveau, message) {
    const niveaux = {
        erreur: { methode: console.error, prefixe: "[ERREUR]" },
        avertissement: { methode: console.warn, prefixe: "[AVERTISSEMENT]" },
        info: { methode: console.log, prefixe: "[INFO]" },
    };
    const { methode, prefixe } = niveaux[niveau];
    methode(`${prefixe} ${message}`);
}

// Utilisation
afficherMessage("erreur", "Fichier introuvable");
afficherMessage("avertissement", "Attention, stockage plein");
afficherMessage("info", "Tâche terminée");
```

Avantage : Une seule fonction gère tous les types de messages, et il est facile d'ajouter un nouveau niveau.

Formulaires

Javascript Copier

```javascript
function validerNom(nom) {
    if (nom.length < 2) {
        throw new Error("Le nom doit faire au moins 2 caractères.");
    }
    return true;
}

function validerEmail(email) {
    if (!email.includes("@")) {
        throw new Error("L'email doit contenir un @.");
    }
    return true;
}

// Utilisation
try {
    validerNom("Jérôme");
    validerEmail("jerome@example.com");
} catch (error) {
    console.error(error.message);
}
```

Problème : La structure de validation et de gestion d’erreur est répétée.

Formulaire

Javascript Copier

```javascript
function validerChamp(valeur, regle, messageErreur) {
    if (!regle(valeur)) {
        throw new Error(messageErreur);
    }
    return true;
}

// Utilisation
try {
    validerChamp("Jérôme", (v) => v.length >= 2, "Le nom doit faire au moins 2 caractères.");
    validerChamp("jerome@example.com", (v) => v.includes("@"), "L'email doit contenir un @.");
} catch (error) {
    console.error(error.message);
}
```

Avantage : La logique de validation est générique et réutilisable pour n’importe quel champ.

API

Javascript Copier

```javascript
async function recupererUtilisateurs() {
    const response = await fetch("https://api.example.com/utilisateurs");
    if (!response.ok) {
        throw new Error("Erreur lors de la récupération des utilisateurs.");
    }
    return response.json();
}

async function recupererProduits() {
    const response = await fetch("https://api.example.com/produits");
    if (!response.ok) {
        throw new Error("Erreur lors de la récupération des produits.");
    }
    return response.json();
}
```

Problème : La logique de gestion des erreurs et de parsing de la réponse est répétée.

API

Javascript Copier

```javascript
async function recupererDonnees(url) {
    const response = await fetch(url);
    if (!response.ok) {
        throw new Error(`Erreur lors de la récupération des données depuis ${url}.`);
    }
    return response.json();
}

// Utilisation
async function afficherDonnees() {
    try {
        const utilisateurs = await recupererDonnees("https://api.example.com/utilisateurs");
        const produits = await recupererDonnees("https://api.example.com/produits");
        console.log(utilisateurs, produits);
    } catch (error) {
        console.error(error.message);
    }
}
```

Avantage : Une seule fonction gère toutes les requêtes API, et le code est plus maintenable.

Pause

Robustesse

Pourquoi c’est indispensable ?

Un code robuste :

- **Résiste aux entrées invalides** (ex : formulaires mal remplis).
- **Gère les erreurs de manière élégante** (ex : appels API échoués).
- **Facilite la maintenance** et réduit les bugs en production.

Exemple : formulaire d’inscription

Javascript Copier

```javascript
function validerInscription(nom, email, age) {
    const utilisateur = { nom, email, age };
    console.log("Utilisateur inscrit :", utilisateur);
    return true;
}

// Utilisation
validerInscription("", "jerome@example", -5); // ❌ Aucune erreur détectée
```

- Quels sont les problèmes potentiels avec ce code ?
- Que se passe-t-il si nom est une chaîne vide ?
- Comment gérer un age négatif ou non numérique ?
- Comment valider le format de l’email ?

A refactoriser : 30 minutes

Solution

Javascript Copier

```javascript
function validerInscription(nom, email, age) {
    // Validation du nom
    if (typeof nom !== "string" || nom.trim() === "") {
        throw new Error("Le nom est obligatoire et doit être une chaîne non vide.");
    }

    // Validation de l'email
    if (typeof email !== "string" || !email.includes("@") || !email.includes(".")) {
        throw new Error("L'email doit contenir un @ et un point.");
    }

    // Validation de l'âge
    if (typeof age !== "number" || age <= 0 || age > 120) {
        throw new Error("L'âge doit être un nombre valide entre 1 et 120.");
    }

    const utilisateur = { nom, email, age };
    console.log("Utilisateur inscrit :", utilisateur);
    return true;
}

// Utilisation
try {
    validerInscription("Jérôme", "jerome@example.com", 30); // ✅ Succès
    validerInscription("", "jerome@example", -5); // ❌ Lève une erreur
} catch (error) {
    console.error("Erreur :", error.message);
}
```

Calcul de total d’un panier

Javascript Copier

```javascript
function calculerTotal(panier) {
    return panier.reduce((total, article) => total + article.prix, 0);
}

// Utilisation
const panier = [
    { nom: "Livre", prix: 20 },
    { nom: "Stylo", prix: "5" }, // ❌ `prix` est une chaîne
    { nom: "Cahier", prix: -10 }, // ❌ Prix négatif
];
console.log(calculerTotal(panier)); // NaN ou résultat incorrect
```

- Comment gérer les articles avec un prix non numérique ?
- Que faire si panier est null ou undefined ?

Utiliser les types et verifier que le nombre est positif

Calcul de total d’un panier

Javascript Copier

```javascript
function calculerTotal(panier) {
    if (!Array.isArray(panier)) {
        throw new Error("Le panier doit être un tableau.");
    }

    let total = 0;
    panier.forEach((article, index) => {
        if (typeof article.prix !== "number" || article.prix < 0) {
            console.warn(`Article invalide à l'index ${index} : prix = ${article.prix}`);
            return; // Ignore l'article invalide
        }
        total += article.prix;
    });

    return total;
}

// Utilisation
const panier = [
    { nom: "Livre", prix: 20 },
    { nom: "Stylo", prix: "5" }, // ⚠️ Avertissement logged
    { nom: "Cahier", prix: -10 }, // ⚠️ Avertissement logged
];
console.log(calculerTotal(panier)); // 20 (seul le livre est compté)
```

API (suite)

Javascript Copier

```javascript
async function recupererUtilisateur(id) {
    const response = await fetch(`https://api.example.com/utilisateurs/${id}`);
    const data = await response.json(); // ❌ Risque si la réponse n'est pas du JSON
    return data;
}

// Utilisation
recupererUtilisateur(1).then(data => console.log(data));
recupererUtilisateur(999).then(data => console.log(data)); // ❌ 404 non géré
```

- Que se passe-t-il si l'API retourne une erreur 404 ?
- Comment gérer une réponse qui n'est pas du JSON ?
- Comment tester cette fonction sans dépendre de l'API réelle ?
- Que faire si la requête échoue à cause d'un problème réseau ?

API (suite)

```javascript
async function recupererUtilisateur(id, fetchFn = fetch) {
    try {
        const response = await fetchFn(`https://api.example.com/utilisateurs/${id}`);
        if (!response.ok) {
            throw new Error(`Erreur HTTP : ${response.status}`);
        }
        const data = await response.json();
        if (!data.id) { // Vérification minimale de la structure
            throw new Error("Format de réponse inattendu.");
        }
        return data;
    } catch (error) {
        console.error("Erreur lors de la récupération :", error.message);
        throw error; // Permet à l'appelant de gérer l'erreur
    }
}
```

```javascript
// Utilisation avec mock pour les tests
async function test() {
    // Mock pour un utilisateur valide
    global.fetch = jest.fn(() =>
        Promise.resolve({
            ok: true,
            json: () => Promise.resolve({ id: 1, nom: "Jérôme" }),
        })
    );
    const utilisateur = await recupererUtilisateur(1);
    console.log(utilisateur); // { id: 1, nom: "Jérôme" }

    // Mock pour un utilisateur non trouvé
    global.fetch = jest.fn(() =>
        Promise.resolve({
            ok: false,
            status: 404,
        })
    );
    try {
        await recupererUtilisateur(999);
    } catch (error) {
        console.error(error.message); // "Erreur HTTP : 404"
    }
}

test();
```

Testabilité

Pourquoi c’est indispensable ?

- **Détecter les bugs tôt** : Éviter les régressions lors des mises à jour.
- **Améliorer la qualité du code** : Un code testable est souvent mieux structuré.
- **Faciliter la maintenance** : Les tests servent de documentation vivante.
- **Gagner en confiance** : Refactoriser sans crainte de casser l’existant.

Comment tester (React) ?

- **Découpler la logique métier**
- **Utiliser des props et des dépendances injectables**
- **Jest**

Découpler la logique metier

Javascript Copier

```javascript
// ❌ Logique métier dans le composant
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`/api/users/${userId}`)
      .then(res => res.json())
      .then(data => setUser(data));
  }, [userId]);

  return <div>{user?.name}</div>;
}

// ✅ Logique extraite dans un hook
function useUser(userId) {
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetch(`/api/users/${userId}`)
      .then(res => res.json())
      .then(data => setUser(data));
  }, [userId]);
  return user;
}

function UserProfile({ userId }) {
  const user = useUser(userId);
  return <div>{user?.name}</div>;
}
```

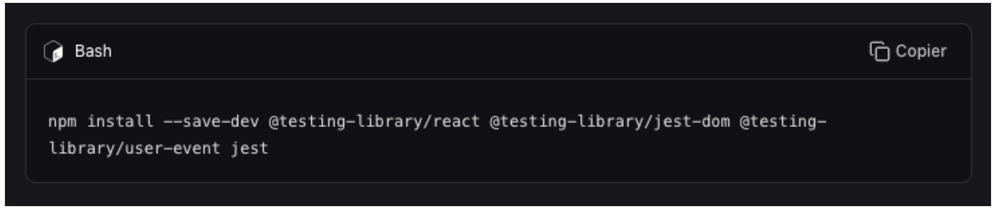
Utiliser les props et dépendances injectables

Javascript Copier

```javascript
function useUser(userId, fetchFn = fetch) {
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetchFn(`/api/users/${userId}`)
      .then(res => res.json())
      .then(data => setUser(data));
  }, [userId, fetchFn]);
  return user;
}
```

Jest

- Créer une application vite js
- Installer les dépendances



Config

https://dev.to/dangkhoado43/clean-code-principles-code-conventions-for-react-typescript-3n7d

Projet

- Créer une landing page pour un produit (peut-importe le produit)
- Mettre en place un formulaire qui permet de s’inscrire avec quand le produit n’est dispo.
- Nom / prénom / email / date de naissance, message de validation d’inscription personnalisé avec « Merci {prénom} »
- Vérifier que tout s’affiche correctement sur la page et que le le message de validat affiche bien le bon prénom.
- Tests soient fonctionnels et qu’ils soient validés.