

# Sommaire

- Utilité de la qualité de code
- Spécificités du front
- TP 1 (sans clean code)
- Présentation des bonnes pratiques JS/TS/Angular
- TP 2
- Refactoring
- Test Unitaire et TDD dans le clean code
- Intégration Lint, Sonar, CI/CD
- Tests e2e
- Performance et optimisation

# Utilité de la qualité de code

**Importance de la qualité du code** : La qualité du code est cruciale pour plusieurs raisons :

- **Lisibilité** : Un code propre est plus facile à lire et à comprendre, non seulement pour le développeur qui l'écrit, mais aussi pour les autres membres de l'équipe qui pourraient devoir le maintenir ou le modifier à l'avenir.
- **Maintenance** : Un code bien structuré et sans redondance est plus facile à mettre à jour, ce qui est essentiel pour les projets à long terme.
- **Débogage** : Un code de qualité est généralement moins sujet aux erreurs et plus facile à déboguer, car les problèmes potentiels sont mieux isolés.
- **Performance** : Bien que la performance ne soit pas toujours directement liée à la propreté du code, un code bien écrit tend à être plus efficace en termes de ressources.
- **Collaboration** : Lorsque plusieurs développeurs travaillent sur le même projet, avoir un code propre permet une meilleure collaboration et réduit les frictions au sein de l'équipe.

# Spécificités du front

## Contexte du développement frontend avec Angular :

- **Modularité** : Angular favorise une approche modulaire, où chaque composant, service ou module a une responsabilité claire. Cela permet de réutiliser et de maintenir facilement le code.
- **Structure des composants** : Dans Angular, la séparation des composants selon leur rôle dans l'interface utilisateur est primordiale. Cela implique de découper l'application en composants logiques et de les organiser de manière cohérente.
- **Réactivité et performance** : Angular intègre des mécanismes pour gérer la réactivité des interfaces, via l'utilisation d'observables et de la gestion de l'état. Le clean code dans ce contexte signifie que chaque partie de l'application réagit de manière efficace aux changements de données.
- **Testing** : Angular inclut des outils comme Jasmine et Karma pour faciliter les tests unitaires et d'intégration, ce qui fait partie intégrante d'une stratégie de code propre.

# Exercice : Développement d'une application de Bowling en Angular

Vous devez développer une application de bowling en Angular qui gère les scores de plusieurs joueurs.

1. **Créer un composant unique pour l'application de bowling.**

2. **Implémenter les fonctionnalités suivantes :**

- **Gestion des joueurs** : L'application doit permettre de gérer plusieurs joueurs.
- **Lancers aléatoires** : Pour chaque joueur, implémentez une fonctionnalité qui génère des lancers aléatoires.
- **Calcul du score** : Le score doit être calculé après chaque lancer, en prenant en compte les règles de base du bowling (strikes, spares).
- **Affichage du score** : Affichez le score de chaque joueur à l'écran.

# Présentation des bonnes pratiques JS/TS/Angular

## 1. Nommage clair et significatif

Le nom des variables, fonctions, et classes doit être suffisamment descriptif pour indiquer clairement leur rôle. Un bon nom est explicite et permet de comprendre immédiatement l'intention du code sans nécessiter de commentaires ou de deviner le contexte.

### Exemple :

- Mauvais : `let d: number;`
  - **Problème** : Le nom `d` est ambigu. On ne sait pas ce que cette variable représente.
- Bon : `let totalCost: number;`
  - **Pourquoi c'est mieux** : `totalCost` est explicite et décrit clairement que la variable stocke le coût total.

### Recommandations pour Angular :

- **Variables** : Utilisez des noms de variables qui reflètent leur contenu ou leur rôle.
  - Ex : `let userList = [];` au lieu de `let list = [];`
- **Fonctions** : Le nom de la fonction doit refléter l'action ou le calcul effectué.
  - Ex : `calculateTotalCost()` au lieu de `doCalculation()`.
- **Classes et Services** : Les noms doivent refléter le rôle ou l'entité qu'ils représentent.
  - Ex : `UserService` pour un service qui gère les utilisateurs.

# Présentation des bonnes pratiques JS/TS/Angular

## Démonstration :

```
calculate() {  
  let t = this.totalAmount * 0.1;  
  return t;  
}
```

```
calculateTax() {  
  let taxAmount = this.totalAmount * 0.1;  
  return taxAmount;  
}
```

# Présentation des bonnes pratiques JS/TS/Angular

## 2. Fonctions et méthodes courtes

Les fonctions doivent être aussi courtes que possible tout en accomplissant une tâche spécifique. L'objectif est de rendre chaque fonction compréhensible d'un coup d'œil. Une fonction courte est plus facile à tester et à maintenir. Si une fonction devient trop longue, elle devrait probablement être divisée en plusieurs fonctions plus petites.

### Exemple :

- Mauvais :

```
calculateTotal() {  
  let subtotal = 0;  
  for (let i = 0; i < this.items.length; i++) {  
    subtotal += this.items[i].price * this.items[i].quantity;  
  }  
  let tax = subtotal * 0.1;  
  let discount = this.getDiscount();  
  let total = subtotal + tax - discount;  
  return total;  
}
```

# Présentation des bonnes pratiques JS/TS/Angular

- Bon :

```
calculateTotal() {  
  const subtotal = this.calculateSubtotal();  
  const tax = this.calculateTax(subtotal);  
  const discount = this.getDiscount();  
  return subtotal + tax - discount;  
}  
  
private calculateSubtotal(): number {  
  return this.items.reduce((sum, item) => sum + item.price * item.quantity, 0);  
}  
  
private calculateTax(subtotal: number): number {  
  return subtotal * 0.1;  
}
```



# Présentation des bonnes pratiques JS/TS/Angular

## 3. Un Seul Niveau d'Abstraction par Fonction

Une fonction ne doit pas mélanger différents niveaux d'abstraction. Par exemple, une fonction qui orchestre des actions de haut niveau ne devrait pas s'encombrer de détails de bas niveau. Cela aide à maintenir la clarté de la logique de la fonction et facilite sa compréhension.

### Exemple :

- Mauvais :

```
renderPage() {  
  this.fetchData();  
  this.renderHeader();  
  this.showSpinner();  
}
```

- Bon :

```
renderPage() {  
  this.showLoadingState();  
  this.fetchData().then(() => {  
    this.hideLoadingState();  
    this.renderContent();  
  });  
}  
  
private showLoadingState() {  
  this.renderHeader();  
  this.showSpinner();  
}
```

# Présentation des bonnes pratiques JS/TS/Angular

## 4. Code DRY (Don't Repeat Yourself)

Le principe DRY consiste à éviter les duplications de code. Le code dupliqué augmente la charge de maintenance car il peut entraîner des incohérences et nécessite des modifications multiples en cas de changement. L'objectif est de centraliser les logiques communes dans des fonctions ou des services réutilisables.

### Exemple :

- Mauvais :

```
this.saveUser(user);  
this.logger.log('User saved successfully.');
```

```
this.notificationService.notify('User has been saved.');
```

- Bon :

```
this.saveUser(user);  
this.handlePostSave('User saved successfully.', 'User has been saved.');
```

```
private handlePostSave(logMessage: string, notificationMessage: string) {  
  this.logger.log(logMessage);  
  this.notificationService.notify(notificationMessage);  
}
```

# Présentation des bonnes pratiques JS/TS/Angular

## 5. Utilisation de l'Égalité faible vs. Égalité stricte

- **Égalité Faible (==)** : L'égalité faible compare deux valeurs en effectuant une conversion implicite de type, ce qui peut entraîner des résultats inattendus. Par exemple, `0 == false` retourne `true`, car JavaScript convertit `false` en `0` avant de faire la comparaison.
- **Égalité Stricte (===)** : L'égalité stricte compare les valeurs sans conversion de type. Si les deux valeurs sont de types différents, la comparaison retourne `false`. Par exemple, `0 === false` retourne `false` parce que `0` est un nombre et `false` est un booléen.
- **Prévisibilité** : L'égalité stricte garantit que seules des valeurs du même type et de la même valeur seront considérées comme égales. Cela évite les comportements inattendus qui peuvent survenir avec l'égalité faible.
- **Clarté** : En utilisant l'égalité stricte, le code devient plus lisible et compréhensible. Les développeurs peuvent être sûrs que la comparaison est faite sans conversion implicite de type, ce qui rend le code plus clair et plus facile à comprendre.
- **Réduction des bugs** : Les conversions implicites peuvent introduire des bugs subtils difficiles à détecter. En utilisant l'égalité stricte, vous réduisez le risque de telles erreurs, ce qui améliore la qualité globale du code.

# Présentation des bonnes pratiques JS/TS/Angular

## Mauvais Exemple :

```
const userAge = '18'; // string

if (userAge == 18) {
  console.log('User is an adult.');
```

## Bon Exemple :

```
const userAge = '18'; // string

if (userAge === 18) {
  console.log('User is an adult.');
```

# Présentation des bonnes pratiques JS/TS/Angular

## 6. Gestion des Erreurs

La gestion des erreurs dans Angular (et JavaScript en général) consiste à capturer, traiter et éventuellement signaler les erreurs qui peuvent survenir lors de l'exécution de l'application. Une bonne gestion des erreurs est essentielle pour créer des applications robustes et résilientes, conformes aux principes de Clean Code.

- **Anticipation des erreurs** : Un code propre anticipe les erreurs possibles et les gère de manière appropriée. Cela signifie prévoir des cas où les entrées peuvent être invalides, où des requêtes réseau peuvent échouer, etc.
- **Gérer les erreurs localement** : Les erreurs doivent être capturées et gérées aussi près que possible de l'endroit où elles se produisent, ce qui rend le code plus robuste et maintenable.
- **Clarté dans le traitement des erreurs** : Les blocs de gestion des erreurs (`try...catch`) doivent être clairs et ne pas cacher les erreurs. Il est essentiel d'avoir une stratégie claire pour gérer chaque type d'erreur.
- **Ne pas masquer les erreurs** : Évitez d'ignorer les erreurs ou de les masquer sans raison. Chaque erreur doit être logiquement traitée ou, si elle ne peut pas être gérée localement, propagée de manière appropriée pour être traitée ailleurs.

# Présentation des bonnes pratiques JS/TS/Angular

## Mauvais Exemple :

```
getData() {  
  this.http.get('/api/data')  
    .subscribe(data => {  
      console.log(data);  
    });  
}
```

## Bon Exemple :

```
getData() {  
  this.http.get('/api/data')  
    .subscribe({  
      next: (data) => {  
        console.log(data);  
      },  
      error: (error) => {  
        console.error('An error occurred:', error);  
        this.handleError(error);  
      }  
    });  
}  
  
handleError(error: any) {  
  // Logique pour traiter l'erreur (affichage d'un message, etc.)  
  alert('Failed to load data. Please try again later.');
```

# Présentation des bonnes pratiques JS/TS/Angular

## 7. Commentaires

- **Le Principe de Base : Un Code Auto-Explicatif**

Le principe fondamental de Clean Code est que le code lui-même doit être suffisamment clair pour que les commentaires ne soient pas nécessaires pour comprendre ce qu'il fait. Cela signifie que les noms des variables, fonctions, classes, et la structure du code doivent être explicites et bien choisis.

- **Lisibilité améliorée** : Un code bien nommé et structuré est plus facile à lire et à comprendre, ce qui réduit le besoin de commentaires explicatifs.
- **Éviter la redondance** : Les commentaires qui décrivent ce que fait le code plutôt que pourquoi il le fait sont souvent redondants et peuvent devenir obsolètes si le code change mais pas les commentaires.
- **Réduction des erreurs** : Les commentaires peuvent devenir des sources de confusion s'ils ne sont pas maintenus à jour. Un commentaire incorrect peut induire en erreur, contrairement à un code clair et précis.

### Mauvais Exemple :

```
// Calculer le coût total en ajoutant la taxe au sous-total  
let totalCost = subtotal + (subtotal * taxRate);
```

### Bon Exemple :

```
let totalCost = calculateTotalCost(subtotal, taxRate);
```

# Présentation des bonnes pratiques JS/TS/Angular

## Quand utiliser des commentaires

Bien que l'objectif soit de minimiser les commentaires, il existe des situations où les commentaires sont nécessaires et bénéfiques. Ces commentaires doivent être utilisés pour expliquer le **pourquoi** d'une décision de conception ou pour fournir des informations contextuelles qui ne sont pas évidentes dans le code lui-même.

### Cas où les commentaires sont justifiés :

- **Explication des décisions complexes** : Lorsque vous avez pris une décision de conception non triviale qui pourrait ne pas être immédiatement claire pour un autre développeur.
- **Avertissements** : Indiquer les conséquences potentielles de modifier une certaine partie du code.
- **Documentation des API publiques** : Si vous développez des API ou des services publics, il est important de documenter leur utilisation, leurs paramètres, et leurs comportements.
- **TODOs et FIXMEs** : Utilisez des commentaires pour indiquer des tâches inachevées ou des parties du code qui nécessitent une attention future.



# Présentation des bonnes pratiques JS/TS/Angular

## Bon Exemple de Commentaire :

```
// Contournement du bug #1234 dans la librairie X. Ce code doit être supprimé après la mise à jour de la librairie X à la version 2.0.  
fixLibraryBug() {  
  // Code de contournement ici  
}
```

# Présentation des bonnes pratiques JS/TS/Angular

## 3. Types de commentaires à éviter

### Types de commentaires à éviter :

- **Commentaires évidents** : Des commentaires qui répètent simplement ce que fait le code.
- **Commentaires obsolètes** : Des commentaires qui ne sont pas mis à jour lorsque le code change, créant une source potentielle de confusion.
- **Commentaires de "remplissage"** : Des commentaires ajoutés pour respecter une convention de style ou de documentation sans réelle valeur ajoutée.

### Mauvais Exemple :

```
// Incrémenter le compteur  
counter += 1;
```

# Présentation des bonnes pratiques JS/TS/Angular

## 4. Documentation des API et Services

La documentation des API et des services est essentielle lorsque vous exposez des fonctionnalités à d'autres parties de votre application ou à des développeurs externes. Ces commentaires doivent être clairs, concis, et fournir toutes les informations nécessaires pour utiliser les API correctement.

- **Utiliser JSDoc ou TypeDoc** : Pour documenter les méthodes publiques, les paramètres, et les retours de fonction.
- **Expliquer les comportements inattendus ou les effets secondaires** : Informer les utilisateurs de l'API sur tout comportement non trivial ou effets secondaires.

# Présentation des bonnes pratiques JS/TS/Angular

Bon Exemple :

```
/**
 * Crée un nouvel utilisateur dans le système.
 * @param user - Les informations de l'utilisateur à créer.
 * @returns Observable<User> - Un observable qui émet l'utilisateur créé.
 * @throws UserAlreadyExistsError - Si un utilisateur avec le même email existe déjà.
 */
createUser(user: User): Observable<User> {
  // Code pour créer un utilisateur
}
```

# Présentation des bonnes pratiques JS/TS/Angular

## 8. Typage en TypeScript

TypeScript est un sur-ensemble de JavaScript qui introduit le typage statique. Le typage statique signifie que les types des variables, des paramètres de fonction, et des valeurs de retour sont connus et vérifiés au moment de la compilation. Cela permet de détecter les erreurs potentielles plus tôt dans le cycle de développement et de rendre le code plus explicite et sûr.

- **Détection précoce des erreurs** : Le typage statique aide à identifier les erreurs de type avant même que le code ne soit exécuté, réduisant ainsi les bugs en production.
- **Clarté et lisibilité** : Le typage explicite améliore la lisibilité du code en rendant les intentions du développeur claires. Un développeur qui lit le code peut comprendre immédiatement quels types de données sont attendus, sans avoir à deviner ou à exécuter le code.
- **Auto-documentation** : Le typage en TypeScript sert de documentation intégrée, rendant souvent les commentaires superflus pour expliquer les types de données manipulés par les fonctions et les méthodes.
- **Sécurité** : En définissant explicitement les types, vous réduisez les risques de comportements imprévisibles causés par des valeurs inattendues.

### Mauvais Exemple (JavaScript pur) :

```
function calculateTotal(price, quantity) {  
  return price * quantity;  
}  
const total = calculateTotal(10, '5'); // total sera "105", ce qui est incorrect
```

# Présentation des bonnes pratiques JS/TS/Angular

Bon Exemple (TypeScript avec Typage) :

```
function calculateTotal(price: number, quantity: number): number {  
    return price * quantity;  
}  
  
const total = calculateTotal(10, 5); // total sera 50, ce qui est correct
```

## 9. Utilisation des Interfaces et des Types

TypeScript permet de définir des interfaces et des types pour structurer les objets et les fonctions. Les interfaces définissent la forme d'un objet, tandis que les types peuvent combiner plusieurs types existants ou être utilisés pour des unions de types.

**Pourquoi utiliser des interfaces et des types :**

- **Clarification des attentes** : Les interfaces et les types rendent explicite la structure des données manipulées, ce qui améliore la compréhension du code et réduit les erreurs.
- **Réutilisabilité** : En définissant des interfaces réutilisables, vous pouvez appliquer une structure cohérente à travers l'ensemble de l'application, améliorant ainsi la cohérence et la maintenabilité.
- **Encapsulation** : Les interfaces permettent de masquer les détails de l'implémentation tout en exposant seulement ce qui est nécessaire, ce qui est un principe clé de Clean Code.

# Présentation des bonnes pratiques JS/TS/Angular

## Mauvais Exemple :

```
function getUserInfo(user) {  
  console.log(user.name);  
  console.log(user.email);  
}
```

## Bon Exemple :

```
interface User {  
  name: string;  
  email: string;  
}  
  
function getUserInfo(user: User) {  
  console.log(user.name);  
  console.log(user.email);  
}
```

# Présentation des bonnes pratiques JS/TS/Angular

## 10. Modules TypeScript

Les modules TypeScript sont des fichiers qui encapsulent du code (classes, fonctions, constantes, etc.) et les exportent pour qu'ils puissent être utilisés dans d'autres parties de l'application. L'utilisation de modules favorise une organisation claire du code, améliore la réutilisabilité, et permet de maintenir un code propre et bien structuré.

- **Organisation** : Les modules permettent de diviser l'application en parties logiques et cohérentes, ce qui rend le code plus facile à gérer et à naviguer.
- **Encapsulation** : Les modules limitent la portée des variables et des fonctions à l'intérieur du module, réduisant ainsi les risques de conflits de nom et rendant le code plus sûr.
- **Réutilisabilité** : Le code encapsulé dans un module peut être facilement réutilisé dans d'autres modules ou applications, suivant le principe DRY (Don't Repeat Yourself) du Clean Code.



# Présentation des bonnes pratiques JS/TS/Angular

## 11. Respect des Principes SOLID avec TypeScript et Angular

Les principes SOLID sont des principes de conception orientés objet qui sont essentiels pour écrire un code propre et maintenable. TypeScript, avec son système de typage et ses modules, facilite l'application de ces principes dans Angular.

- **Single Responsibility Principle (SRP)** : Chaque module ou classe doit avoir une seule responsabilité.
- **Open/Closed Principle (OCP)** : Les modules doivent être ouverts à l'extension mais fermés à la modification.
- **Liskov Substitution Principle (LSP)** : Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe mère sans affecter le fonctionnement du programme.
- **Interface Segregation Principle (ISP)** : Les clients ne doivent pas être obligés de dépendre d'interfaces qu'ils n'utilisent pas.
- **Dependency Inversion Principle (DIP)** : Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.

## Sujet du TP

En utilisant l'application de Gestion des commandes. Adopter les Principes de Clean Code et SOLID

### Contexte :

Vous êtes chargé de refactorer une application de gestion des commandes pour un petit commerce. Le code de cette application a été écrit sans suivre les bonnes pratiques de Clean Code et SOLID. Votre tâche est de corriger ces problèmes en appliquant les principes suivants :

1. Nommage clair et significatif
2. Fonctions et méthodes courtes
3. Un seul niveau d'abstraction par fonction
4. Application du principe DRY (Don't Repeat Yourself)
5. Utilisation de l'égalité stricte (===)
6. Gestion appropriée des erreurs
7. Utilisation appropriée des commentaires
8. Utilisation du typage en TypeScript
9. Organisation modulaire avec TypeScript
10. Application des principes SOLID