Classes et Programmation Orientée Objet

Concepts Fondamentaux

Pour les étudiants de 1ère année Supinfo

Au Programme

1. Introduction à la POO - Concepts et principes fondamentaux
2. Classes - Définition, constructeur, propriétés et méthodes
3. Héritage et Encapsulation - Réutilisabilité et contrôle d'accès
4. Polymorphisme et Concepts Avancés - Surcharge, composition, design patterns

Qu'est-ce que la POO ?

DÉFINITION

La **Programmation Orientée Objet (POO)** est une approche de programmation qui utilise des **objets** pour modéliser et organiser des données et des comportements logiciels. Ces objets sont des **instances de classes**, qui peuvent être considérées comme des plans ou des modèles.

AVANTAGES

- **Réutilisabilité du code** - Les classes peuvent être réutilisées et étendues
- **Gestion de la complexité** - Organise le code de manière structurée et logique
- **Maintenabilité** - Facilite la modification et la correction du code
- **Modularité** - Divise le programme en unités indépendantes et cohérentes

CONCEPTS CLÉS

- **Classe** - Un modèle ou un plan pour créer des objets
- **Objet/Instance** - Une réalisation concrète d'une classe

POO vs Programmation Procédurale

Programmation Procédurale

Approche traditionnelle centrée sur les **fonctions et procédures** qui manipulent les données.

- Données et fonctions séparées
- Flux d'exécution linéaire
- Difficile à maintenir pour les grands projets
- Réutilisabilité limitée

Programmation Orientée Objet

Approche moderne qui **regroupe données et comportements** dans des objets.

- Données et méthodes encapsulées
- Modularité et réutilisabilité
- Facilite la maintenance et l'évolution
- Meilleure organisation du code

> 💡 **En JavaScript**
>
> En JavaScript, vous avez utilisé la programmation procédurale jusqu'à présent (fonctions, variables globales). La POO vous permet d'organiser votre code de manière plus structurée et professionnelle, surtout pour les projets complexes.

Qu'est-ce qu'une Classe ?

DÉFINITION

Une **classe** est un **modèle ou un plan** à partir duquel des objets (instances) sont créés. En JavaScript, les classes sont définies avec le mot-clé **class**, suivi du nom de la classe.

CARACTÉRISTIQUES

- Une classe contient des **propriétés** (variables) et des **méthodes** (fonctions)
- Les classes sont des **plans réutilisables** pour créer plusieurs objets similaires
- Chaque objet créé à partir d'une classe est appelé une **instance**

SYNTAXE

```
class Personne {
constructor(nom, age) {
this.nom = nom;
this.age = age;
}

afficherInfo() {
console.log(`${this.nom}, ${this.age} ans`);
}
}
```

Constructeur et Instanciation

QU'EST-CE QU'UN CONSTRUCTEUR ?

Le **constructeur** est une méthode spéciale appelée automatiquement lors de la création d'une instance de classe. Il initialise les propriétés de l'objet avec des valeurs par défaut ou fournies en paramètres.

SYNTAXE DU CONSTRUCTEUR

```
class Personne {
constructor(nom, age) {
this.nom = nom;
this.age = age;
}
}
```

INSTANCIATION AVEC LE MOT-CLÉ NEW

Pour créer une instance d'une classe, on utilise le mot-clé **new** suivi du nom de la classe et des paramètres du constructeur. Cela crée un nouvel objet et appelle le constructeur.

```
const alice = new Personne("Alice", 25);
const bob = new Personne("Bob", 30);
console.log(alice.nom); // "Alice"
console.log(bob.age); // 30
```

PROPRIÉTÉS VS INSTANCES

Classe (Modèle)

```
class Personne {
constructor(nom) {
this.nom = nom;
}
}
```

Instances (Objets)

```
const p1 = new Personne("Alice");
const p2 = new Personne("Bob");
// p1 et p2 sont des instances différentes
```

Propriétés et Méthodes

Propriétés

Une **propriété** est une variable associée à une classe. Elle stocke des données ou l'état d'un objet. Les propriétés sont généralement définies dans le **constructeur**.

```
class Personne {
constructor(nom, age) {
this.nom = nom;
this.age = age;
}
}

// nom et age sont
// des propriétés
```

Méthodes

Une **méthode** est une fonction associée à une classe. Elle définit des comportements ou des actions que les instances de la classe peuvent effectuer.

```
class Personne {
afficherInfo() {
console.log(
`${this.nom}, ${this.age}`
);
}
}

// afficherInfo est
// une méthode
```

📌 **Résumé**

Les **propriétés** stockent les données (l'état de l'objet), tandis que les **méthodes** définissent les actions ou les comportements. Ensemble, elles forment une classe complète qui modélise un concept du monde réel.

Héritage : Étendre les Classes

DÉFINITION

L'**héritage** permet à une classe de **hériter des propriétés et méthodes** d'une autre classe. C'est un moyen de créer une nouvelle classe en tant que version améliorée ou spécialisée d'une autre classe, favorisant la **réutilisabilité du code**.

SYNTAXE : LE MOT-CLÉ EXTENDS

En JavaScript, l'héritage se fait avec le mot-clé **extends**. La classe qui hérite est appelée **classe enfant** ou **classe dérivée**, et la classe dont les propriétés sont héritées est appelée **classe parent** ou **classe de base**.

```
class Animal {
constructor(nom) {
this.nom = nom;
}
parler() {
console.log(this.nom + " fait du bruit");
}
}

class Chien extends Animal {
parler() {
console.log(this.nom + " aboie");
}
}

const monChien = new Chien("Rex");
monChien.parler(); // "Rex aboie"
```

AVANTAGES

- **Réutilisabilité** - Évite la duplication de code
- **Hiérarchie** - Organise les classes en une structure logique
- **Spécialisation** - Crée des versions spécialisées de classes existantes

Encapsulation : Getters et Setters

DÉFINITION

L'**encapsulation** est un principe de la POO qui consiste à **restreindre l'accès direct** aux composants d'un objet. En JavaScript, cela se fait généralement via des **getters** et **setters** pour contrôler comment les propriétés sont accédées et modifiées.

GETTERS ET SETTERS

Les **getters** et **setters** sont des méthodes spéciales utilisées pour accéder et modifier les propriétés d'un objet de manière contrôlée. Ils permettent d'ajouter une logique de validation ou de traitement lors de l'accès ou de la modification des propriétés.

Getter (Lecture)

```
get age() {
return this._age;
}

// Utilisation:
console.log(personne.age);
```

Setter (Écriture)

```
set age(valeur) {
if (valeur > 0) {
this._age = valeur;
}
}

// Utilisation:
personne.age = 30;
```

EXEMPLE COMPLET

```
class Personne {
constructor(nom, age) { this.nom = nom; this._age = age; }
get age() { return this._age; }
set age(v) { if (v > 0 && v < 150) this._age = v; else console.log("Âge invalide"); }
}
const alice = new Personne("Alice", 25); console.log(alice.age); alice.age = 30; alice.age = -5;
```

Polymorphisme : Surcharger les Méthodes

DÉFINITION

Le **polymorphisme** est la capacité d'un objet à prendre plusieurs formes. En POO, cela se traduit par la capacité d'une **classe enfant à surcharger une méthode** de sa classe parent, offrant ainsi un comportement spécifique tout en conservant l'interface commune.

AVANTAGES

- **Flexibilité** - Les objets répondent différemment à la même méthode
- **Code générique** - Traiter différents objets de la même manière
- **Extensibilité** - Ajouter facilement de nouvelles classes sans modifier le code existant

SURCHARGE DE MÉTHODES

Quand une classe enfant définit une méthode avec le même nom qu'une méthode de la classe parent, elle **surcharge** cette méthode. Cela permet à chaque classe d'avoir son propre implémentation du même comportement.

```
class Animal {
parler() {
console.log("Bruit générique");
}
}
class Chien extends Animal {
parler() {
console.log("Woof! Woof!");
}
}
class Chat extends Animal {
parler() {
console.log("Meow!");
}
}
const chien = new Chien();
const chat = new Chat();
chien.parler(); // "Woof! Woof!"
chat.parler(); // "Meow!"
```

Composition vs Héritage

Héritage

Une classe enfant **hérite** de toutes les propriétés et méthodes de la classe parent. Relation "est-un".

- Relation hiérarchique
- Couplage fort
- Peut causer le problème du diamant
- Réutilisabilité limitée

```
class Vehicule {
avancer() { }
}

class Voiture extends Vehicule {
// Hérite de avancer()
}
```

Méthodes et Propriétés Statiques

DÉFINITION

Les **méthodes et propriétés statiques** sont associées à la **classe elle-même** plutôt qu'aux instances. Elles sont définies avec le mot-clé **static** et sont accessibles directement sur la classe sans avoir besoin de créer une instance.

SYNTAXE

```
class Compteur {
static count = 0; // Propriété statique

constructor(nom) {
this.nom = nom;
Compteur.count++; // Accès à la propriété statique
}

static afficherTotal() { // Méthode statique
console.log(`Total: ${Compteur.count}`);
}
}

const c1 = new Compteur("A");
const c2 = new Compteur("B");
Compteur.afficherTotal(); // "Total: 2"
```

PROPRIÉTÉS D'INSTANCE VS STATIQUES

Propriétés d'Instance

```
class Personne {
constructor(nom) {
this.nom = nom;
}
}

const p1 = new Personne("Alice");
const p2 = new Personne("Bob");

// Chaque instance
// a sa propre
// propriété nom
```

Propriétés Statiques

```
class Personne {
static espece = "Homo sapiens";
}

// Partagée par
// toutes les instances
console.log(
Personne.espece
);
// "Homo sapiens"
```

CAS D'USAGE

- **Compteurs** - Compter le nombre d'instances créées
- **Constantes** - Stocker des valeurs partagées par toutes les instances
- **Fonctions utilitaires** - Méthodes qui n'ont pas besoin d'accès aux données d'instance

Classes Abstraites et Design Patterns

CLASSES ABSTRAITES

En JavaScript, il n'y a pas de support direct pour les classes abstraites, mais vous pouvez **simuler ce concept**. Une classe abstraite est une classe qui n'est pas destinée à être instanciée directement, mais plutôt à être une classe de base pour d'autres classes.

```
class Animal {
constructor() {
if (new.target === Animal) {
throw new Error("Animal est abstraite");
}
}
parler() {
throw new Error("parler() doit être implémentée");
}
}

const animal = new Animal(); // Erreur!
```

DESIGN PATTERN : FACTORY

Le **Factory Pattern** est un design pattern de création qui permet de créer des objets sans spécifier leurs classes exactes. Utile pour avoir une interface commune avec des implémentations spécifiques.

Classe Factory

```
class AnimalFactory {
static create(type) {
if (type === "chien") {
return new Chien();
} else if
(type === "chat") {
return new Chat();
}
}
}
```

Utilisation

```
const chien = AnimalFactory
.create("chien");
const chat = AnimalFactory
.create("chat");

chien.parler();
chat.parler();
```

AVANTAGES DU FACTORY PATTERN

• **Abstraction** - Masque la complexité de la création d'objets • **Flexibilité** - Facile d'ajouter de nouveaux types • **Centralisation** - La logique de création est au même endroit

Bonnes Pratiques et Résumé

RÉSUMÉ DES 5 CONCEPTS CLÉS

1. **Classes** - Modèles pour créer des objets avec propriétés et méthodes
2. **Héritage** - Réutiliser et étendre des classes existantes
3. **Encapsulation** - Contrôler l'accès aux propriétés avec getters/setters
4. **Polymorphisme** - Surcharger les méthodes pour des comportements spécifiques
5. **Composition** - Combiner des classes pour plus de flexibilité

BONNES PRATIQUES

- Utilisez des **noms clairs et descriptifs** pour vos classes et méthodes
- Chaque classe doit avoir **une seule responsabilité** (Single Responsibility Principle)
- Préférez la **composition à l'héritage** quand c'est possible
- Utilisez l'**encapsulation** pour protéger les données sensibles

> 💡 **Quand Utiliser la POO ?**
>
> Utilisez la POO pour les projets **complexes et de grande taille** où vous avez besoin d'organiser votre code en structures logiques. Pour les scripts simples, la programmation procédurale peut suffire.

Questions ?

N'hésitez pas à poser vos questions sur les Classes, l'Héritage, l'Encapsulation, le Polymorphisme ou tout autre concept de la Programmation Orientée Objet en JavaScript.

RESSOURCES POUR APPROFONDIR

MDN Web Docs - Documentation complète sur les classes JavaScript

JavaScript.info - Tutoriels interactifs sur la POO en JavaScript

Eloquent JavaScript - Livre en ligne gratuit sur JavaScript avancé

Design Patterns - Apprenez les patterns de conception en JavaScript

Merci pour votre attention ! Continuez à pratiquer et à explorer la POO en JavaScript.