📚 Cours : JavaScript Avancé - Programmation Orientée Objet et Design Patterns

🎯 Objectif du cours :

Ce cours vous initie aux concepts avancés de la **programmation orientée objet (POO)** en JavaScript et aux **design patterns**. Vous apprendrez comment structurer vos applications de manière modulaire et réutilisable grâce à des concepts comme les classes, l'héritage, l'encapsulation, ainsi que des patterns classiques utilisés dans le développement logiciel pour résoudre des problèmes récurrents de manière élégante et efficace.

---

1. 🚀 Révision des Concepts de Base de la Programmation Orientée Objet (POO)

La **programmation orientée objet** est un paradigme de programmation basé sur la notion d'objets, qui sont des instances de **classes**. En JavaScript, bien que l'on utilise des **fonctions constructrices** pour créer des objets, le langage a évolué pour offrir des **classes** et une gestion de l'héritage plus claire.

a. Classes et Objets en JavaScript

Les **classes** en JavaScript sont introduites dans ECMAScript 6 (ES6). Elles servent de gabarits pour créer des objets, avec des propriétés et des méthodes spécifiques.

Exemple de définition d'une **classe** :

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

    sePresenter() {
        console.log(`Je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
    }
}

let personne1 = new Personne('Jean', 30);
personne1.sePresenter(); // Affiche "Je m'appelle Jean et j'ai 30 ans."
```

Ici, **`constructor()`** est une méthode spéciale qui est appelée lors de la création de chaque nouvel objet à partir de la classe.

---

2. ⚙ Encapsulation et Abstraction

L'encapsulation en POO fait référence à l'idée de cacher l'état interne de l'objet et de ne permettre l'accès à cet état qu'à travers des méthodes spécifiques. Cela permet de contrôler l'accès aux propriétés de l'objet et de protéger l'intégrité de ces propriétés.

a. Encapsulation avec des méthodes privées

JavaScript permet de déclarer des **propriétés privées** dans les classes, ce qui ajoute une couche d'abstraction et protège l'état interne de l'objet.

Exemple avec des propriétés privées (introduites dans ES2022) :

```
class Personne {
    #nom; // Propriété privée
    #age; // Propriété privée

    constructor(nom, age) {
        this.#nom = nom;
        this.#age = age;
    }

    getNom() {
        return this.#nom;
    }

    setNom(nouveauNom) {
        this.#nom = nouveauNom;
    }

    sePresenter() {
        console.log(`Je m'appelle ${this.#nom} et j'ai ${this.#age} ans.`);
    }
}

let personne2 = new Personne('Alice', 28);
console.log(personne2.getNom()); // Alice
personne2.setNom('Bob');
console.log(personne2.getNom()); // Bob
```

Ici, **`#nom`** et **`#age`** sont des **propriétés privées** et ne peuvent être accédées directement de l'extérieur de la classe. Elles sont encapsulées à l'intérieur de l'objet.

---

3. 🔄 Héritage et Polymorphisme

L'**héritage** est un principe clé de la POO, permettant à une **classe enfant** de hériter des propriétés et des méthodes d'une **classe parente**. Le **polymorphisme** permet de redéfinir des méthodes dans les classes enfants, offrant une plus grande flexibilité.

a. Héritage en JavaScript

En JavaScript, l'héritage est facilité grâce aux **classes** et au mot-clé **`extends`**.

Exemple d'héritage :

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

    parler() {
        console.log(`${this.nom} fait un bruit`);
    }
}

class Chien extends Animal {
    constructor(nom, race) {
        super(nom); // Appel du constructeur de la classe parente
        this.race = race;
    }

    parler() {
        console.log(`${this.nom} aboie`);
    }
}

let monChien = new Chien('Rex', 'Labrador');
monChien.parler(); // Affiche "Rex aboie"
```

Ici, **`Chien`** hérite de la classe **`Animal`** et redéfinit la méthode **`parler()`** pour l'adapter au comportement spécifique du chien.

b. Polymorphisme

Le polymorphisme permet à différentes classes d'avoir des comportements similaires mais différents. Par exemple, deux objets peuvent avoir une méthode avec le même nom mais qui se comporte différemment selon l'objet.

---

4. 📐 Design Patterns

Les **design patterns** sont des solutions éprouvées à des problèmes récurrents rencontrés lors de la conception de logiciels. En JavaScript, certains design patterns courants incluent :

a. Le Singleton Pattern

Le **Singleton** est un pattern qui garantit qu'une classe n'a qu'une seule instance, et fournit un point d'accès global à cette instance.

Exemple du pattern Singleton :

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            this.value = 42;
            Singleton.instance = this;
        }
        return Singleton.instance;
    }

    getValue() {
        return this.value;
    }
}

const instance1 = new Singleton();
const instance2 = new Singleton();

console.log(instance1 === instance2); // true (Les deux sont la même instance)
```

Ici, le **Singleton** garantit qu'il n'y ait qu'une seule instance de la classe, ce qui peut être utile lorsque vous voulez centraliser la gestion de l'état dans une application.

b. Le Factory Pattern

Le **Factory Pattern** est un pattern de création d'objets, qui permet de créer des objets sans spécifier la classe exacte de l'objet qui sera créé.

Exemple du pattern Factory :

```
class Animal {
    parler() {
        console.log("L'animal fait un bruit");
    }
}

class Chien extends Animal {
    parler() {
        console.log("Le chien aboie");
    }
}

class Chat extends Animal {
    parler() {
        console.log("Le chat miaule");
    }
}

class AnimalFactory {
    static creerAnimal(type) {
        if (type === 'Chien') {
            return new Chien();
        } else if (type === 'Chat') {
```

```
            return new Chat();
        }
        return new Animal();
    }
}

let chien = AnimalFactory.creerAnimal('Chien');
chien.parler(); // Affiche "Le chien aboie"

let chat = AnimalFactory.creerAnimal('Chat');
chat.parler(); // Affiche "Le chat miaule"
```

Ici, le **Factory Pattern** permet de créer des objets sans connaître exactement la classe à instancier, ce qui rend le code plus flexible et extensible.

---

5. 🏭 Autres Design Patterns Importants

a. Le Observer Pattern

Le **Observer Pattern** permet de définir une relation de type "un à plusieurs" entre objets, où plusieurs observateurs sont notifiés lorsqu'un objet change d'état.

b. Le Module Pattern

Le **Module Pattern** est utilisé pour encapsuler des parties de code dans des espaces de noms distincts, permettant ainsi une meilleure organisation et isolation du code.

---

6. 🔄 Résumé

Ce cours vous a permis d'explorer les concepts avancés de la **programmation orientée objet** en JavaScript, y compris l'héritage, l'encapsulation et le polymorphisme. Vous avez également appris à utiliser certains **design patterns** populaires, comme le **Singleton**, le **Factory Pattern**, et plus encore.

La maîtrise de ces concepts vous permettra de créer des applications JavaScript plus modulaires, réutilisables et évolutives.