

Programmation Orienté Objet

1WEBD – Javascript Web Development



Sommaire

1. Introduction.
2. Classes.
3. Héritage et Encapsulation.
4. Polymorphisme et Concepts Avancés.



1. Introduction

1. Introduction

POO ?

- 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

1. Introduction

POO et Javascript

- les classes sont introduites avec ES6
- permettent de regrouper des comportements connexes
- facilite la gestion de la complexité et augmente la réutilisabilité du code

1. Introduction

POO et Programmation Procédurale

- la programmation procédurale conçoit les programmes autour de procédure et fonctions (c'est ce qui est fait normalement en Javascript)
- la programmation orienté objet regroupe les données et les fonctions qui les manipulent

1. Introduction




2. Classes

2. Classes

Introduction

- une classe est un modèle à partir duquel des objets sont créés
- en JS, elles sont définies par le mot-clé **class**, suivi du nom de la classe

2. Classes



```
class Voiture {  
    constructor(marque, modele) {  
        this.marque = marque;  
        this.modele = modele;  
    }  
}
```

2. Classes

Instanciation

- quand on instance une classe, on appelle son constructeur avec le mot-clé **new**
- une instance est un objet (comme tout en JS)

2. Classes

Instanciation



```
let maVoiture = new Voiture("Toyota", "Corolla");
```


2. Classes

Propriétés

- les propriétés sont des variables associées à une classe
- elles peuvent être définie dans le constructeur

2. Classes

Propriétés



```
class Voiture {  
    constructor(marque, modele) {  
        this.marque = marque;  
        this.modele = modele;  
    }  
}
```

2. Classes

Méthodes

- les **méthodes** sont des fonctions associées à une classe
- elles définissent des comportements ou des actions que les instances de la classe peuvent effectuer

2. Classes

Méthodes

```
class Voiture {  
    constructor(marque, modele) {  
        this.marque = marque;  
        this.modele = modele;  
    }  
  
    afficherDetails() {  
        console.log(`Marque: ${this.marque}, Modèle: ${this.modele}`);  
    }  
}  
  
let maVoiture = new Voiture("Toyota", "Corolla");  
maVoiture.afficherDetails(); // Marque: Toyota, Modèle: Corolla
```


2. Classes



3. Héritage et Encapsulation

3. Héritage et Encapsulation

Héritage – 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

3. Héritage et Encapsulation

Héritage – Utilisation

- en JavaScript, l'héritage se fait avec le mot-clé `extends`
- la classe qui hérite des propriétés est appelée classe dérivée ou enfant, et la classe dont les propriétés sont héritées est appelée classe parent

3. Héritage et Encapsulation

Héritage – Utilisation

```
class Vehicule {
  constructor(marque) {
    this.marque = marque;
  }

  klaxonner() {
    console.log('Tut tut!');
  }
}

class Voiture extends Vehicule {
  constructor(marque, modele) {
    super(marque);
    this.modele = modele;
  }

  afficherDetails() {
    console.log(`Marque: ${this.marque}, Modèle: ${this.modele}`);
  }
}

let maVoiture = new Voiture("Toyota", "Corolla");
maVoiture.afficherDetails(); // Marque: Toyota, Modèle: Corolla
maVoiture.klaxonner(); // Tut tut!
```

3. Héritage et Encapsulation

Encapsulation – 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 méthodes 'getter' et 'setter'

3. Héritage et Encapsulation

Encapsulation – Utilisation

- les **getters** et **setters** sont des méthodes 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.

3. Héritage et Encapsulation

Encapsulation - Utilisation

```
class Vehicule {
    constructor(marque, portes) {
        this._marque = marque
        this._portes = portes
    }

    get marque() {
        return this._marque
    }

    set portes(nbPortes) {

        if (nbPortes > 0) {
            this._portes = nbPortes
        } else {
            console.log("il faut au moins une porte")
        }
    }
}
```


3. Héritage et Encapsulation



4. Polymorphisme et Concepts Avancés

4. Polymorphisme et Concepts Avancés

Définition Polymorphisme

- le polymorphisme est la capacité d'un objet à prendre plusieurs formes
- en POO, cela se traduit souvent par la capacité d'une classe enfant à surcharger une méthode de sa classe parent, offrant ainsi un comportement spécifique

4. Polymorphisme et Concepts Avancés

Utilisation Polymorphisme

- en JavaScript, le polymorphisme se manifeste lorsque les classes enfant surchargent les méthodes de la classe parent
- cela permet aux objets de différentes classes de répondre différemment à la même méthode

4. Polymorphisme et Concepts Avancés

Utilisation Polymorphisme

```
class Animal {  
    parler() {  
        console.log("Son générique d'animal");  
    }  
}  
  
class Chien extends Animal {  
    parler() {  
        console.log("Woof!");  
    }  
}  
  
class Chat extends Animal {  
    parler() {  
        console.log("Miaou!");  
    }  
}  
  
let monChien = new Chien();  
let monChat = new Chat();  
  
monChien.parler(); // Woof!  
monChat.parler(); // Miaou!
```

4. Polymorphisme et Concepts Avancés

Composition vs Héritage

- la **composition** est une alternative à l'héritage où une classe est construite à partir de l'utilisation d'autres classes, plutôt que de l'héritage de la classe parent
- cela offre une plus grande flexibilité et évite certains problèmes courants de l'héritage, comme [le problème du diamant](#)

4. Polymorphisme et Concepts Avancés

Composition vs Héritage

```
class Vehicule {
    constructor(marque) {
        this.marque = marque;
    }

    demarrer() {
        console.log("Le véhicule démarre.");
    }
}

class Voiture extends Vehicule {
    constructor(marque, nombreDePortes) {
        super(marque);
        this.nombreDePortes = nombreDePortes;
    }
}

class Moto extends Vehicule {
    constructor(marque, aUnSidecar) {
        super(marque);
        this.aUnSidecar = aUnSidecar;
    }
}
```

```
class Moteur {
    demarrer() {
        console.log("Le moteur démarre.");
    }
}

class Radio {
    allumer() {
        console.log("La radio s'allume.");
    }
}

class Voiture {
    constructor(marque) {
        this.marque = marque;
        this.moteur = new Moteur();
        this.radio = new Radio();
    }

    demarrerVoiture() {
        this.moteur.demarrer();
    }

    allumerRadio() {
        this.radio.allumer();
    }
}
```

4. Polymorphisme et Concepts Avancés

Méthodes et Propriétés Statiques

- les **méthodes statiques** sont des fonctions associées à une classe plutôt qu'à ses instances
- elles sont souvent utilisées pour des fonctionnalités qui ne nécessitent pas les données d'un objet particulier

4. Polymorphisme et Concepts Avancés

Méthodes et Propriétés Statiques

```
class Calculatrice {  
    static addition(a, b) {  
        return a + b;  
    }  
}  
  
let resultat = Calculatrice.addition(5, 7); // 12, pas besoin d'instancier Calculatrice
```

4. Polymorphisme et Concepts Avancés

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
- elle peut par exemple envoyer une erreur (avec le mot-clé **throw**)

4. Polymorphisme et Concepts Avancés

Classes Abstraites

```
class Vehicule {
  constructor() {
    if (this.constructor === Vehicule) {
      throw new Error("Classe abstraite 'Vehicule' ne peut pas être instanciée directement.");
    }
  }

  demarrer() {
    throw new Error("Méthode 'démarre' doit être implémentée");
  }
}

class Voiture extends Vehicule {
  demarrer() {
    console.log("La voiture démarre.");
  }
}

// let vehicule = new Vehicule(); // Erreur
let maVoiture = new Voiture(); // Correct
maVoiture.demarrer(); // La voiture démarre.
```

4. Polymorphisme et Concepts Avancés

Design Patterns – Intro

- Les design patterns sont des structures génériques prévues pour résoudre des problèmes courants en programmation
- Ensemble de schémas

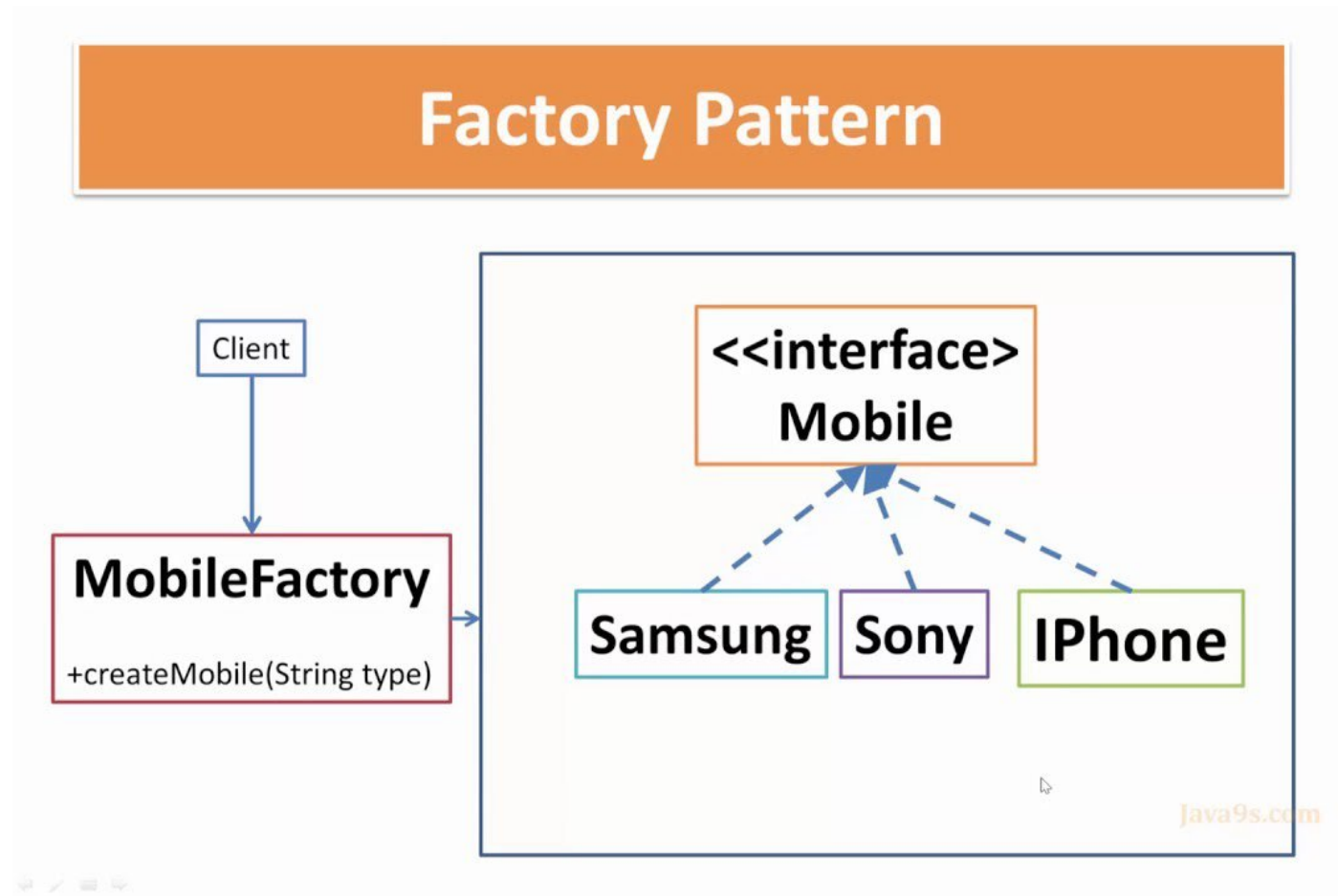
4. Polymorphisme et Concepts Avancés

Design Patterns – Factory

- Design Pattern de création
- Utile pour avoir une interface commune et des implémentation spécifique
- Peut se baser sur l'héritage et le polymorphisme
- Permet de ne pas exposer de la complexité logicielle

4. Polymorphisme et Concepts Avancés

Design Patterns – Factory



4. Polymorphisme et Concepts Avancés



