L'encapsulation en JavaScript

L'encapsulation est un principe de la programmation orientée objet qui consiste à regrouper les données et les méthodes qui agissent sur ces données dans un même objet. Cela permet de protéger les données de l'objet en les rendant privées et de les manipuler uniquement à travers les méthodes de l'objet.

Dans plusieurs langages de programmation orientée objet, comme Java ou C#, il est possible de définir des attributs privés qui ne peuvent être accédés directement depuis l'extérieur de l'objet.

En JavaScript, c'est plus complexe.

Propriétés publiques et privées

Une propriété publique est une propriété qui peut être accédée et modifiée directement depuis l'extérieur de l'objet.

Ex:

```
let personne = new Personne("Jean", 30);
personne.nom = "Paul"; // Propriété publique
personne.age = 25; // Propriété publique
personne.direBonjour(); //Méthode publique
```

Une propriété privée est une propriété qui ne peut être accédée et modifiée que depuis l'intérieur de l'objet.

Il existe depuis peu, une nouvelle fonctionnalité en JavaScript appelée **Private Class Fields** qui permet de déclarer des propriétés privées et des méthodes dans une classe.

On préfixe simplement le nom de la propriété par un # pour la rendre privée. Elle sera visible dans la console, mais impossible à modifier. À noter, qu'il faut aussi les déclarer avant le constructeur.

Ex:

```
class Personne {
    #nom;
    #age;

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

    #formaterNom(nom) {
        return nom.toUpperCase();
```

```
    }

    direBonjour() {
        console.log(`Bonjour, je m'appelle ${this.#nom} et j'ai ${this.#age} ans.`);
    }
}
```

Pourquoi l'encapsulation ?

L'encapsulation permet de protéger les données de l'objet en les rendant privées. Cela permet de contrôler l'accès aux données et de garantir leur intégrité.

Il faut prendre pour acquis que les développeurs qui utiliseront votre code pourrait faire n'importe quoi avec les données de l'objet. L'encapsulation permet de limiter les risques d'erreurs et de bugs en contrôlant l'accès aux données.

Une bonne pratique est toujours de déclarer les propriétés d'un objet comme privées et de fournir des méthodes publiques pour accéder et modifier ces propriétés. Cela permet de garantir que les données de l'objet sont toujours valides et cohérentes.

Accesseurs et mutateurs (Getters et Setters)

Les accesseurs et les mutateurs sont des méthodes spéciales qui permettent d'accéder et de modifier les propriétés privées d'un objet.

Les accesseurs sont des méthodes qui permettent de lire la valeur d'une propriété privée. Ils sont généralement appelés des getters.

Les mutateurs sont des méthodes qui permettent de modifier la valeur d'une propriété privée. Ils sont généralement appelés des setters.

En JavaScript, on utilise les mots-clés `get` et `set` pour définir des accesseurs et des mutateurs.

Ex:

```
class Personne {
    #nom;
    #age;

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

    get nom() {
        return this.#nom;
    }
```

```
    set nom(nom) {
        this.#nom = nom;
    }

    get age() {
        return this.#age;
    }

    set age(age) {
        this.#age = age;
    }

    #formaterNom(nom) {
        return nom.toUpperCase();
    }

    direBonjour() {
        console.log(`Bonjour, je m'appelle ${this.#nom} et j'ai ${this.#age} ans.`);
    }
}
```

Pourquoi ne pas utiliser des propriétés publiques alors?

Parce que cela permet de contrôler l'accès aux données et de garantir leur intégrité. Cela permet de limiter les risques d'erreurs et de bugs en contrôlant l'accès aux données.

Ex: on pourrait créer une propriété en lecture seule en ne définissant que le getter.

On pourrait aussi formatter la valeur avant de la retourner ou de la modifier dans le setter.

Utiliser les mutateurs et accesseurs

Pour accéder à une propriété privée, on utilise les accesseurs (getters) et pour modifier une propriété privée, on utilise les mutateurs (setters) sans les parenthèses. Le navigateur prend pour acquis que c'est une propriété et non une méthode.

```
let personne = new Personne("Jean", 30);
console.log(personne.nom); // Jean
console.log(personne.age); // 30

personne.nom = "Paul";
personne.age = 25;

console.log(personne.nom); // Paul
console.log(personne.age); // 25
```

Retrocompatibilité - Les closures pour les propriétés privées

En JavaScript, avant les classes et les private class fields, il était possible de créer des propriétés privées en utilisant le concept de fermeture (closure).

Cela permet de créer des variables privées qui ne sont pas accessibles depuis l'extérieur de la fonction grâce à la portée des variables en JavaScript.

En gros, il s'agit d'une fonction dans une autre fonction qui capture les variables de la fonction externe. Cela permet de créer des variables privées qui ne sont pas accessibles depuis l'extérieur de la fonction à cause de la portée des variables en JavaScript.

Voici un exemple de closure pour une fonction constructeur `Personne` avec des attributs privés en utilisant les closures :

```
function Personne(nom, age) {
    // Attributs privés

    let _nom = nom;
    let _age = age;

    // Méthodes publiques
    // Grâce au retour et à la portée des variables, sont accessibles uniquement
dans la méthode genererPersonne
    return function genererPersonne() {
        return `<p>Nom : ${_nom}, Age : ${_age}</p>`;
    };
}

let personne = Personne("Jean", 30);
console.log(personne); // <p>Nom : Jean, Age : 30</p>
```

Le concept est important à comprendre, mais il est plus simple d'utiliser les private class fields pour déclarer des propriétés privées dans une classe.