582-21F-MA Programmation d'interface Web 1

cours 15

- Principes de la programmation orientée objet (*POO*)
- Classes *ES6*


JS

Objets littéraux (rappel)

- Un objet est une entité qui possède des propriétés et des méthodes.
- En *JavaScript*, on peut créer un objet littéral en définissant ses propriétés et méthodes à l'intérieur d'une paire d'accolades.
- Chaque propriété et méthode correspond à une paire clé - valeur. La clé est le nom de la propriété ou de la méthode accessible via la notation pointée.
- La valeur d'une propriété peut être une donnée (nombre, chaîne, booléen, etc) ou bien une fonction. Dans ce cas, la propriété est appelée une méthode.
- À l'intérieur d'une méthode, le mot-clé `this` représente l'objet courant sur lequel la méthode s'applique.

Exemple d’objet littéral (rappel)

```
let livre = {
    nombrePage: 300,
    pageActuelle: 1,
    auteur: 'John Doe',
    avancePage: function() {
        if (this.pageActuelle < this.nombrePage) {
            this.pageActuelle++;
        }
    },
    reculePage: function() {
        if (this.pageActuelle > 0) {
            this.pageActuelle--;
        }
    }
};
```

Les limites des objets littéraux

Les objets littéraux sont très utiles afin de représenter et de manipuler des données rapidement et efficacement. Cependant, leur structure ne permet pas facilement leur reproduction (on dira instanciation) de plusieurs copies. Dans l'exemple de l'objet `livre`, il faudrait définir plusieurs fois les mêmes méthodes et les mêmes propriétés pour chacun des livres dans le programme. Cette structure est donc limitée.

Programmation orientée objet (POO)

La programmation orientée objet est un paradigme de programmation qui a été défini au tournant des années 1960. On le distingue habituellement de la programmation procédurale, composée d'une longue liste d'instructions contenues dans diverses fonctions qui, vous l'aurez constaté, deviens rapidement difficile à organiser.

Le modèle de la programmation orientée objet consiste à faire interagir ensemble des entités de codes, appelés objet, qui regroupe des données (*data*), mais aussi des opérations (ou comportements) spécifiques à l'objet. Dans ce contexte, un objet représente une entité conceptuelle ou physique tel qu'un usager, un livre ou tout autre élément, selon le contexte du programme.

Paradigmes de la programmation orientée objet JS

Il existe deux principaux paradigmes de programmation orientée objet (*Object-oriented programming* (*OOP*)) : la première utilisant les *prototypes*, et la deuxième - depuis *ES6* - les classes (`class`).

Javascript utilise la programmation orientée objet par prototype, mais *ES6* introduit une syntaxe beaucoup plus claire qui aide grandement la cohérence, la structure et la modularité de vos scripts.

Notez toutefois que *JavaScript* est un langage de programmation orientée objet basé sur les prototypes. Cela-dit, que vous utilisiez la syntaxe des prototypes ou des classes *ES6*, ces deux paradigmes reposes sur les trois principes fondamentaux suivants.

Principe 1 : encapsulation

Contrairement aux langages procéduraux qui définissent des données (variables) et des procédures pour traiter ces données (fonction), l'encapsulation réfère au principe d'enfermer dans une même entité à la fois les données (propriétés) et les procédures (méthodes) de traitement. Ce qui permet de cacher (*scoper*) tous les rouages internes du traitement des données, mais aussi de laisser certaines données et procédures accessibles à l'extérieur de l'objet.

Les données et le traitement (aka logique) vont de pair. Cela est particulièrement vrai lorsqu'il s'agit d'applications plus volumineuses où il est très important que le traitement des données soit limité à des emplacements définis.

Bien fait, la *POO* produit la modularité par conception. Cela éloigne le code spaghetti redouté où tout est étroitement lié et difficilement maintenable.

Principe 2 : héritage

L'héritage permet de transmettre certaines caractéristiques entre les entités qui partagent des caractéristiques communes. De fait, il arrive qu'une classe partage plusieurs méthodes avec une classe plus générique. Pour minimiser, organiser et clarifier le code, il est possible de créer des sous-classes (*subclass*) qui hérite d'une classe dite mère.

Autrement dit, en *POO*, il est possible de créer de nouvelles classes à partir d'autres classes, ces classes enfants peuvent hériter des propriétés et méthodes de leur classe parent. Il est ainsi possible d'avoir des attributs partagés à l'ensemble des classes plutôt que de les dupliquer. Au besoin, il est tout à fait possible d'ajouter des propriétés et méthodes sur chaque classe enfant afin de spécialiser ses comportements.

Par exemple, un objet `Administrateur` et un objet `Client` hérite des propriétés et méthodes communes de l'objet parent `Usager`, mais chacun d'eux définissent des caractéristiques et des fonctionnalités spécifiques. Un objet `MembrePrivilege` hérite des propriétés et méthodes jumelées des objets parents `Client` et `Usager`.

Principe 3 : polymorphisme

Le polymorphisme décrit la capacité d'un objet qui hérite des méthodes de ses parents de modifier cette dernière ou bien d'appeler la méthode du parent. Un objet parent pourra définir une méthode générale et en faire hériter ses enfants, mais ceux-ci pourront redéfinir pour eux-mêmes cette méthode.

Par exemple, si un objet de `Bateau` et un objet `Voiture` hérite de la méthode `seDeplacer()` de l'objet `Véhicule`, il faut que `Bateau` et `Voiture` puissent redéfinir cette méthode afin de l'adapter à leur caractéristique.

Pour reprendre l'exemple d'héritage entre l'objet `Usager` et l'objet `Administrateur`, nous pourrions y redéfinir la méthode commune `seConnecter()` hérité de `Usager` pour y ajouter des éléments de vérification sécuritaire supplémentaire.

Classe ES6

La plupart des langages orientés objet (*C++*, *Java*, *C#*, etc) sont basés sur l'utilisation de classes. Une classe est une abstraction qui représente une idée ou un concept manipulé par le programme. Elle facilite la création d'objets ayant le même modèle et partageant ainsi leurs données et comportements.

Bien qu'auparavant le concept de `class` n'existait pas en *JavaScript* (la *POO* en *JS* utilisait les prototypes vus précédemment), *ECMAScript 2015* (*ES6*) introduit le mot-clé `class` afin d'écrire des classes réutilisables à l'aide d'une syntaxe plus simple, claire et propre, qui ressemble davantage aux classes en *C++* ou en *Java*.

Création d'une classe

Reprenons notre exemple **`livre`**, mais cette fois comme **`class`** *JavaScript* :

```
class Livre {
    constructor(nombrePage, pageActuelle, auteur) {
        this._nombrePage = nombrePage;
        this._pageActuelle = pageActuelle;
        this._auteur = auteur;
    }

    avancePage = () => {
        if (this._pageActuelle < this._nombrePage) this._pageActuelle++;
    }

    reculePage = () => {
        if (this._pageActuelle > 0) this._pageActuelle--;
    }
};
```

Syntaxe

Une classe est créée avec le mot-clé `class` suivi du nom de la classe - qui commence par convention avec une majuscule - puis l'ouverture et fermeture d'accolades `{}`.

La méthode spéciale nommée `constructor()` peut être ajoutée à la définition de la classe. Cette méthode est exécutée lors de la création d'un objet et permet de définir les données propres à celui-ci, sous forme de propriétés. Comme les prototypes, les propriétés ne sont pas séparées par des virgules mais bien des points-virgules.

Toutes les méthodes que vous souhaitez associer à la classe *ES6* seront définies à l'intérieur de celle-ci, après le constructeur.

À l'extérieur du constructeur, une classe *ES6* ne contient que des définitions de méthodes (rappelons que `constructor()` est une méthode).

Comme pour les objets littéraux, le mot-clé `this` représente à l'intérieur d'une méthode l'objet sur lequel la méthode a été appelée. Autrement dit, `this` représente la classe elle-même. Par exemple `this._nombrePage` peut se lire : le nombre de page de l'instance de ce `Livre`.

`constructor()`

Le `constructor` est une méthode qui est utilisée par l'*API JavaScript* pour créer et initialiser un objet et ses propriétés lorsqu'on utilise le mot-clé `class`. Notez qu'il ne peut y avoir qu'une seule méthode utilisant le mot-clé `constructor` au sein d'une classe, ce serait d'ailleurs autrement bizarre. Une exception *SyntaxError* sera levée dans la console du navigateur si la classe contient plusieurs méthodes `constructor`.

La méthode `constructor` n'est pas obligatoire, si elle n'est pas définie, un constructeur sans propriété sera initialisé par défaut.

Mot-clé this

Comme pour les objets littéraux et prototypes, les mot-clé `this` représente l'instance de la classe elle-même. Ainsi vous pourrez, à l'intérieur de la classe, avoir directement accès à toutes les valeurs des propriétés définies dans le constructeur lors de l'instanciation. Vous pourrez également appeler directement toutes les méthodes développées dans la classe.

Autrement dit, le mot-clé `this` assure que les valeurs des propriétés et les comportements des méthodes sont autonomes et contextualisés à chaque instance de la classe.

Par exemple, à l'intérieur de la classe `Livre`, faire référence à `this._nombrePage` équivaut à manipuler la valeur de la propriété `_nombrePage` de cette instance (objet) de la classe `Livre`.

_propriete

Vous aurez remarqué que j'utilise un *underscore* pour déclarer mes propriétés. Ce n'est pas une obligation ni même une convention, mais c'est une syntaxe qui permet de clairement différencier les propriétés des méthodes et des *getters / setters* (à venir la session prochaine) à l'intérieur d'une classe.

Fonctions fléchées (arrow function)

Depuis *ES6*, il est possible de déclarer les fonctions à l'aide de fonctions fléchées (*arrow function*). La syntaxe se rapproche de celle-vu précédemment et va comme suit :

```
nomDeLaFonction = (params) => {
    // Comportement(s) de la fonction
}
```

Au-delà du gain de quelques caractères, le grand avantage est la différence de valeur du mot-clé `this`. Sans entrer dans les détails, dans la fonction fléchée `this` représente toujours l'objet qui a défini la fonction, donc ici l'instance de l'objet.

Comme pour les *back tick*, notez bien que la fonction fléchée a été introduite avec *ES6*, si le devis demande de supporter *IE11*, utilisez la déclaration à l'aide du mot-clé `function`.

L’instance d’une classe est un objet

L’opération de création d’un objet à partir d’une classe s’appelle instanciation. Dans le contexte de classe *JavaScript*, chaque instance d’une classe est un objet. Ainsi vous assurez l’autonomie de chaque objet et la modularité de votre script, qu’il y ait une seule ou 8000 instance(s).

Dans l’exemple ci-dessous, deux objets `Livre` sont créés par l’intanciation de la classe `Livre` à l’aide de l’instruction `new`. Bien qu’elles partagent les mêmes propriétés et méthodes héritées de la classe `Livre`, il est fondamental de comprendre que ces deux instances (ou objets) sont complètement indépendantes l’une de l’autre.

```
const livre1 = new Livre(300, 1, 'John Doe'),
      livre2 = new Livre(400, 1, 'Jane Doe');

livre1.avancePage();
livre1.avancePage();
console.log(livre1._pageActuelle);
livre2.avancePage();
console.log(livre2._pageActuelle);
```

Instancier les classes initiales d'une page

Pour éviter le code spaghetti, une bonne pratique est d'instancier les classes *JavaScript* nécessaires à une page depuis un fichier *JS* autre que la classe elle-même, par exemple un fichier `script.js` ou même une autre classe. Aussi - contrairement à l'exemple précédent - il ne faut pas écrire de code à l'extérieur des accolades d'une classe. Ainsi le comportement de l'objet instancié sera propre, indépendant et modulaire.

Pour instancier les classes initiales d'une page, il faut d'abord déclarer les fichiers *JS* des classes avant le fichier *JS* qui les appellera dans le `<head>`, autrement le mot-clé `new` fera référence à une classe que *JavaScript* ne pourra reconnaître. Exemple :

```
<head>
    ...
    <script src="./chemin-relatif/Class.js"></script>
    <script src="./chemin-relatif/appel-de-la-class.js"></script>
    ...
</head>
```

Récupérer les propriétés et méthodes d'un objet

De l'exemple précédent, vous aurez remarqué qu'il est tout à fait possible de récupérer les propriétés et appeler les méthodes d'un objet (instance de classe) avec la notation pointée à partir du fichier où vous avez fait l'instanciation de la classe ...évidemment du moment où vous placez l'objet instancié à l'intérieur d'une variable à laquelle vous pouvez faire référence.

Cela-dit, un objet instancié n'a pas besoin d'être placé dans une variable pour exister. Il arrive souvent - surtout dans le contexte de manipulations de *DOM* - qu'un objet n'ait pas à retourner de valeur(s). Du coup l'instance appelée n'a qu'à être lancée pour exécuter son comportement attendu.

Instancier des classes à partir de noeud DOM

Souvent un comportement est lié à la présence d'un noeud *DOM*, par exemple un script de traitement de formulaire (validation et envoie). Toutefois, pour ne pas alourdir le nombre de traitements *JS* sur une page, on souhaite instancier le(s) script(s) lié(s) à ce noeud seulement lorsque celui-ci est présent dans le *DOM*.

Pour ce faire, comme toujours lorsqu'il s'agit de manipuler du *DOM*, il faudra d'abord récupérer tous les noeuds éléments souhaités puis boucler à travers ceux-ci afin d'instancier la classe *ES6* où sont situés leurs comportements. Rappelons encore ici qu'une boucle est également une structure conditionnelle.

Pour assurer la modularité du code et bien circonscrire la portée de la classe, on passe en paramètre son propre noeud *DOM* pour encapsuler clairement son cadre d'action(s). Exemple :

this._el

Afin d'assurer l'autonomie et la modularité d'un objet associé à un comportement d'un module du *DOM*, on voudra passer la référence à lui-même pour limiter sa portée. De cette façon, le point d'entrée du sélecteur d'un élément *DOM* à l'intérieur d'un module ne sera plus `document` mais bien `this._el`, ainsi on assure l'autonomie, la précision et la modularité des comportements de chaque objet instancié. Exemple :

```
class Form {
    constructor(el) {
        this._el = el;
        //console.log(this._el);

        this._elInputs = this._el.querySelectorAll('input');
        //console.log(this._elInputs);

    }
};
```

Méthode init

Contrairement à la méthode `constructor`, `init` n'est pas une méthode de l'*API JavaScript*. Je vous invite toutefois à utiliser cette syntaxe sémantique pour assurer l'organisation de votre code.

On déclare la méthode `init` immédiatement après le constructeur. La méthode `init` sert à initialiser les comportements de l'objet - souvent les gestionnaires d'événements globaux - comme par exemple l'événement clic sur le bouton soumettre d'un formulaire.

Il est important de notez que, pour être automatiquement exécutée, vous aurez besoin de lancer la méthode `init()` dans le constructeur.

Structure d’une classe ES6

Un objectif en programmation est la clarté. Ainsi, placer les méthodes comportementales d’une classe *ES6* à la suite du constructeur et de la méthode `init()` rend possible une structure simple et logique qui aide énormément la lisibilité et la cohérence de vos scripts.

Pour résumé, la structure du script d’une classe *ES6* suit ce fil de développement :

- déclaration des propriétés globales de la classe dans le constructeur ;
- initialisation des comportements et/ou événements globaux dans la méthode `init` ;
- définition des méthodes comportementales de la classe.

Instancier dynamiquement des classes ES6

Lors de l'ajout dynamique de nouveaux contenus, il se peut que ceux-ci doivent avoir leurs propres comportements dynamiques. Rappelons qu'en *JavaScript* vanille, il n'est pas possible d'ajouter un gestionnaire d'événements sur un/des élément(s) inexistant(s), *JavaScript* soulève alors une erreur qui stoppe l'exécution du script.

Pour pallier à ce problème, vous devrez faire l'instantiation de(s) classe(s) voulue(s) après l'injection du *DOM*. Vous pourrez alors récupérer ce nouveau *DOM* avec les techniques de manipulation *DOM* déjà vu afin d'instancier la classe - avec lui-même en paramètre - pour initialiser ses comportements.

En résumé

- Les objets basés sur des objets du monde réel constituent la pièce maîtresse de toute application basée sur la *POO*.
- Les classes sont les modèles utilisés pour instancier des objets.
- Les objets ont des méthodes qui agissent sur les données qu'ils contiennent.
- L'encapsulation protège les données des accès incontrôlés.
- La *POO* est plus détaillée mais plus facile à lire que les autres paradigmes de codage.
- Comme la programmation orientée objet avec `class` est intervenue plus tard dans le développement de *JavaScript*, vous pouvez rencontrer un code plus ancien qui utilise des techniques de programmation prototypes ou fonctionnelles.

Références

class
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/class

Classes
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes

JavaScript ES6: Classes
https://medium.com/@luke_smaki/javascript-es6-classes-8a34b0a6720a

JavaScript Class
https://www.javascripttutorial.net/es6/javascript-class/

constructor
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes/constructor