POO en JS 1 - Procédural vs Objet, classes, instances

La Programmation Orientée Objet ou POO est un “paradigme” de programmation, né dans les années 60. Il a été peu à peu adopté par de nombreux langages, parmi lesquels C++, Java, Python, Ruby, PHP, etc. On utilise le terme de “langage objet” pour désigner un langage implémentant le modèle de Programmation Orientée Objet.

JavaScript permet de faire de la programmation orientée objet, mais repose sur des mécanismes différents de la plupart des autres langages, ce qui le classe un peu à part. La norme ES6 a cependant amené la possibilité d’écrire des “classes” d’une façon semblable aux autres langages objet.

Objectifs

- Programmation procédurale
- Programmation objet
- Instance de classe vs objet littéral

Etapes

Programmation procédurale

La programmation dite “procédurale” est un modèle de programmation

basée sur l'appel de procédures. Une procédure est un bloc de code effectuant une certaine tâche. Pour nommer une procédure en JavaScript, on utilise plutôt le synonyme *fonction*.

Les fonctions ont souvent besoin de paramètres pour accomplir leur tâche. Prenons l'exemple d'une application qui gère des compteurs. L'exemple complet se trouve sur le repo GitHub associé.

Chaque compteur est un objet ayant trois propriétés :

- son nom (`name`),
- la valeur courante du compteur (`value`),
- la valeur maximale qu'il peut atteindre (`max`)

On crée deux compteurs ayant cette forme :

```
const counter1 = {
  name: 'Counter #1', value: 0, max: 5
};

const counter2 = {
  name: 'Counter #2', value: 0, max: 3
};
```

Ainsi que deux fonctions ; la première permettant d'incrémenter la valeur d'*un* compteur, si celle-ci n'a pas déjà atteint son maximum.

```
function incrementCounter(counter) {
```

```
  if (counter.value < counter.max) {
    counter.value += 1;
    console.log(`Counter ${counter.name}'s value is now ${
counter.value}`);
  } else {
    console.error(`Reached max (${counter.max}) - Can't in
crement ${counter.name}`);
  }
}
```

La deuxième, permettant de vérifier si le compteur *peut* être incrémenté :

```
function canIncrementCounter(counter) {
  return counter.value < counter.max;
}
```

On crée ensuite deux boucles qui incrémentent les compteurs :

```
while(canIncrementCounter(counter1)) {
  incrementCounter(counter1);
}
while(canIncrementCounter(counter2)) {
  incrementCounter(counter2);
}
```

Dans ce code, les fonctions qui accèdent aux propriétés d'un objet "compteur" doivent recevoir un tel objet en paramètre.

Ressources

-

Programmation objet

Dans l'exemple précédent, les données sur lesquelles on agit (les objets "compteur") et le code qui agit dessus (les fonctions) sont bien séparées.

Dans le paradigme de Programmation Orientée Objet, on regroupe dans la même structure des données, et le code agissant sur ces données.

Le modèle objet vise à "modéliser" et à traduire en code des concepts du "monde réel". Il nous est ainsi plus facile de raisonner sur les données et sur les actions qui permettent de les modifier.

Par exemple, une voiture dans le monde réel possède certains attributs (une marque, un modèle, une année de construction, une couleur, un type de moteur), et il est possible d'effectuer différentes actions dessus (déverrouiller, verrouiller, mettre le contact, couper le contact, accélérer, freiner, changer de vitesse).

Pour "modéliser" une voiture en code, on va utiliser les concepts de *classe* et d'*objet*. Une classe permet de définir un "modèle" type, à

partir duquel un objet va être créé. La classe permet de définir à la fois des *attributs* (les données propres que chaque objet va posséder), et des *méthodes* (le code qui va agir sur ces données).

Exemple : le compteur revisité

Reprenons l'exemple du compteur, cette fois en version objet. L'exemple complet se trouve sur le repo GitHub associé.

On commence par définir une classe `Counter`.

```
class Counter {
  constructor(name, max) {
    this.value = 0;
    this.max = max;
    this.name = name;
  }

  getValue() {
    return this.value;
  }

  canIncrement() {
    return this.value < this.max;
  }

  increment() {
    if (this.value < this.max) {
```

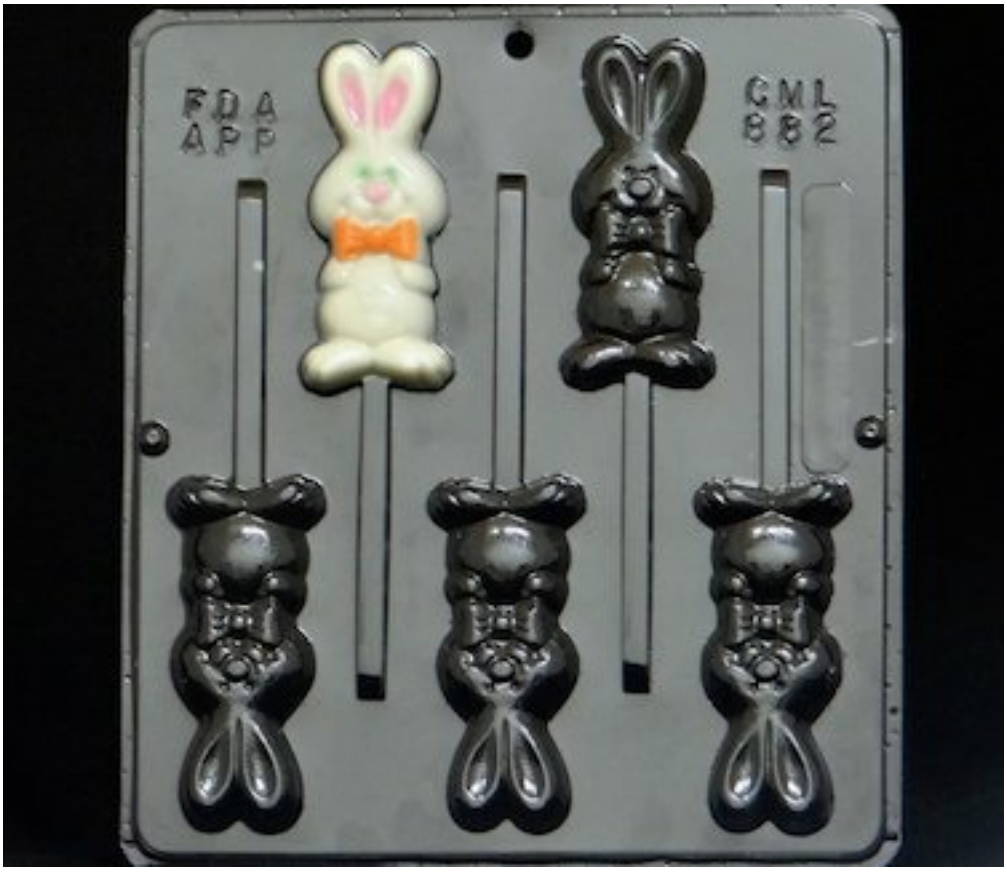
```
      this.value += 1;
      console.log(`Counter ${this.name}'s value is now ${t
his.value}`);
    } else {
      console.error(`Reached max (${this.max}) - Can't inc
rement ${this.name}`);
    }
  }
}
```

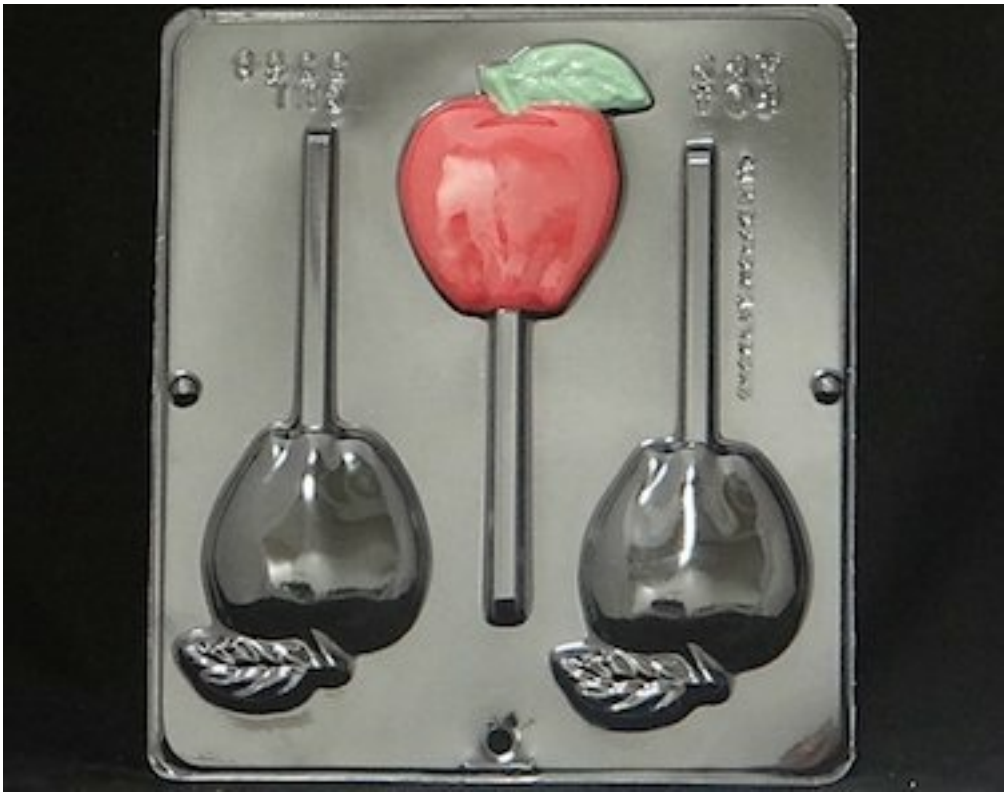
Il y a déjà beaucoup à dire sur cet exemple, et il nous faut détailler le sens de :

- `class`
- `constructor`
- `this`
- *attributs* et *méthodes*

Classe et mot-clé `class`

La *classe* est le “modèle” qui permet de définir une certaine “forme” qu’aura chaque objet. On peut la comparer à un “moule” qui permet de donner une forme à un objet.





Deux moules différents donneront des objets de forme différente, et de même, deux classes différentes donneront des objets de forme

différente. Dans le cas des classes, la “forme”, ce sont les attributs que pourront avoir les objets : `value`, `max` et `name` pour un compteur ; `brand`, `model`, `year`, `color`, `engine` pour une voiture.

Chaque lapin sortant du moule ci-dessus a la même “forme”, mais chaque lapin peut ensuite avoir des caractéristiques propres : tel lapin peut être fait de chocolat blanc, tel autre de chocolat noir, tel autre de chocolat au lait. De même, chaque objet créé à partir d’une même classe aura des valeurs qui lui sont propres pour chaque attribut.

Constructeur et mot-clé `constructor`

Le *constructeur* (`constructor`) est une fonction appelée lors de la création de d’un objet, et qui permet d’initialiser ses attributs. Pour créer un compteur, on va toujours utiliser le même constructeur, qui va initialiser les mêmes attributs (et donner ainsi la même “forme” à notre nouvel objet). Par contre, les valeurs qu’on passe en paramètre au constructeur sont utilisées pour donner au nouvel objet ses caractéristiques propres. Ainsi, deux compteurs distincts peuvent avoir un nom et une valeur maximale distincts.

Voici comment on crée des objets “compteur” :

```
const counter1 = new Counter('Counter #1', 5);
const counter2 = new Counter('Counter #2', 3);
```

Pour créer un nouvel objet à partir de la classe, on appelle le nom de la classe, précédé du mot-clé `new`, et suivi des paramètres attendus par le

constructeur de `Counter`. L'appel du constructeur est implicite : on n'utilise pas le mot-clé `constructor` lors de la création de l'objet, mais celui-ci est appelé du fait qu'on ait utilisé le mot-clé `new`. L'objet ainsi créé est appelé une *instance* de la classe.

Mot-clé `this`

Le mot-clé `this` permet à un objet - une instance de classe - de se référencer lui-même. C'est à dire qu'un objet compteur, voiture, etc., peut accéder à *ses* propres attributs et *ses* propres méthodes via ce mot-clé.

Par exemple, le constructeur de `Counter` permet de donner des valeurs initiales pour les attributs `value`, `max` et `name` de *ce* nouveau compteur.

Attributs et méthodes

Ainsi qu'il a déjà été dit, les *attributs* sont les données propres à chaque objet, c'est à dire à chaque *instance* d'une classe.

Les *méthodes* sont le code qui permet de récupérer des informations d'un objet (comme `getValue` pour récupérer la valeur, ou `canIncrement` pour savoir si le compteur peut être incrémenté), ou de modifier les attributs de l'objet (`increment` qui modifie l'attribut `value`). Une méthode ressemble à une fonction, à ceci pres qu'en JavaScript (norme ES6), on n'utilise pas le mot-clé `function` pour déclarer une méthode dans une classe.

Voici comment on peut ré-écrire le code de l'exemple procédural, pour incrémenter en boucle chaque compteur :

```
while(counter1.canIncrement()) {
  counter1.increment();
}
while(counter2.canIncrement()) {
  counter2.increment();
}
```

La différence principale avec le code procédural est qu'on appelle directement une méthode *sur* l'objet lui-même.

Ressources

-

Instance de classe vs objet littéral

L'utilisation de classes est le moyen exclusif de créer des objets, dans la plupart des langages supportant la POO.

Mais ce n'est pas le cas en JavaScript, car celui-ci dispose d'un moyen plus direct pour créer des objets, sans créer de classes. C'est la syntaxe littérale, dont voici un exemple :

```
const counter = {
  value: 0
```

```
};
counter.getValue = function() {
  return this.value;
}
counter.increment = function() {
  this.value += 1;
}
counter.increment();
counter.increment();
console.log(counter.getValue());
```

Le modèle POO “classique” basé sur des classes n’est pas utilisé aussi systématiquement en JavaScript que dans d’autres langages, du fait de cette possibilité immédiate de création d’objets.

Mais il faut savoir que derrière cette syntaxe littérale, les objets créés sont des instances d’une classe générique `Object`. Ce sont bien des objets à part entière, mais ils ne sont pas “contraints” dans leur forme par une classe et un constructeur qui spécifient les attributs et méthodes dont un objet doit disposer.

Ressources

-

Challenge

Ecrire une classe représentant une personne

Tu vas devoir écrire une classe `Person`, permettant de modéliser une personne. Chaque instance de la classe `Person` aura deux attributs, `name` et `age`.

- Le constructeur de `Person` devra initialiser les attributs `name` et `age` à partir des paramètres qui lui sont passés.
- La classe `Person` disposera également de deux méthodes, `tellMyName` et `tellMyAge`, qui afficheront respectivement : `I am <name>` et `I am <age> years old` (remplacer `<name>` et `<age>` respectivement par les valeurs des attributs `name` et `age`).

Enfin, pour tester ta classe, tu devras instancier deux personnes ayant pour noms et âges respectifs “John” et 40, “Mary” et 35, et appeler les méthodes `tellMyName` et `tellMyAge` sur chacune de ces instances de `Person`.

Critères de validation

- Poster ton code sur un Gist. Les wilders qui valident le code sont encouragés à vérifier son bon fonctionnement avec Node.js.