Javascript : Classes abstraites et interfaces

8 novembre 2017 / Arkerone

Temps de lecture : 4 min

Nous allons aujourd'hui nous attaquer à la notion d'interface et de classe abstraite en Javascript. Ces deux notions fondamentales en programmation orientée objet n'existent pas en Javascript, nous allons donc voir comment implémenter ces deux notions tout au long de cet article.

Une Classe abstraite, c'est quoi au juste?

Une classe abstraite en programmation orientée objet se définit comme suit (merci Wikipédia) :

> Une classe abstraite est une classe dont l'implémentation n'est pas complète et qui n'est pas instanciable. Elle sert de base à d'autres classes dérivées (héritées).

Prenons un exemple pour illustrer ce concept. Imaginons que l'on ait besoin d'écrire des classes permettant de lire plusieurs types de fichiers de configuration. On pourrait par exemple avoir une classe qui s'occupe de lire un fichier XML et une autre un fichier JSON.

Ces deux classes partagent des fonctionnalités communes (vérification de l'existence et récupération d'une propriété par exemple), la seule chose qui diffère entre ces deux classes est la lecture du fichier.

Voyons comment écrire une classe abstraite fournissant ces fonctionnalités communes :

Fichier AbstractConfig.js

```
const fs = require('fs');

class AbstractConfig {

  constructor(path) {
    if (this.constructor === AbstractConfig) {
      throw new TypeError('Abstract class "AbstractConfig" cannot be instant
    }
    this.properties = {};
    if (!fs.existsSync(path)) {
      throw new Error(`File ${path} not found`);
    }
    this.parse(path);
  }

  get(name) {
    if (!this.has(name)) {
      throw new Error(`Property ${name} not found`);
    }
    return this.properties[name];
  }

  has(name) {
    return name in this.properties;
  }

  parse(path) {
    throw new Error('You must implement this function');
  }
}

module.exports = AbstractConfig;
```

La classe `AbstractConfig` respecte bien la définition d'une classe abstraite, celle-ci ne peut pas être instanciée :

```
if (this.constructor === AbstractConfig) {
  throw new TypeError('Abstract class "AbstractConfig" cannot be instantiated
}
```

La propriété `constructor` permet de renvoyer une référence à la classe de l'instance (cf constructor). On vérifie donc que la référence n'est pas la classe `AbstractConfig` , si c'est le cas, on lève une exception.

L'implémentation est également incomplète :

```
parse(path) {
  throw new Error('You must implement this function');
}
```

Il incombe donc aux classes dérivées d'implémenter cette fonction :

Fichier XmlConfig.js

```
const AbstractConfig = require('./AbstractConfig');

class XmlConfig extends AbstractConfig {

  constructor(path) {
    super(path);
  }

  parse(path) {
    /* Read the file */
  }
}

module.exports = XmlConfig;
```

La classe `XmlConfig` hérite de la classe abstraite `AbstractConfig` et implémente la fonction `parse` (j'ai omis volontairement le code, ce n'est pas ce qui est important ici).

Petit rappel, en Javascript l'héritage multiple n'existe pas, une classe ne peut hériter que d'une seule autre classe.

Comme nous l'avons vu, la création d'une classe abstraite est relativement simple, voyons maintenant les interfaces.

Et une interface c'est quoi ?

Une interface, en programmation orientée objet, peut-être définit comme suit :

> Une interface définit un ensemble de méthodes sans leurs implémentations

La différence entre une classe abstraite et une interface est qu'une classe abstraite peut avoir des méthodes concrètes qui implémentent des fonctionnalités générales ou communes, elle sert donc à factoriser du code. Une interface quant à elle définit uniquement les méthodes à implémenter, elle permet de définir un contrat : chaque classe implémentant l'interface sera tenue d'implémenter les méthodes de l'interface.

Prenons l'exemple du précédent article, dans lequel nous avons une classe `Mailer` qui attend comme paramètre du constructeur un objet `logger` avec une méthode `log` :

```
class Mailer {

  constructor(settings, logger) {
    // ... Initialisation de la classe

    this.logger = logger;
  }
}

module.exports = Mailer;
```

Afin de garantir que notre objet `logger` possède bien une méthode `log`, celui-ci devra implémenter une interface qui définit cette méthode `log`.

Dans d'autres langages comme PHP par exemple, une interface se déclare comme suit :

Déclaration de l'interface LoggerInterface

```
interface LoggerInterface
{
  public function log($msg);
}
```

et s'implémente de cette manière :

Implémentation de l'interface LoggerInterface

```
class FileLogger implements LoggerInterface
{
    private $file;

    function __construct($file)
    {
        $this->file = $file;
    }

    function log($msg)
    {
        file_put_contents($this->file, $msg . "\n", FILE_APPEND);
    }
}
```

Mais comme je l'ai dit, malheureusement Javascript n'a pas de notion d'interface. Mais plusieurs solutions existent pour simuler cette notion, voyons l'une d'entre elles: le duck typing.

Le duck quoi?

Derrière ce nom étrange se cache un concept tout simple, le duck typing fait référence au test du canard qui dit :

> Si ça ressemble à un canard, si ça nage comme un canard et si ça cancane comme un canard, c'est qu'il s'agit sans doute d'un canard

L'idée derrière est que plutôt de se préoccuper du type d'un objet, on se préoccupe de son comportement (de ses méthodes).

Reprenons l'exemple précédent, notre classe `Mailer` doit attendre un objet qui possède une méthode `log` au final peu importe le type de cet objet du moment qu'il possède cette méthode :

```
class Mailer {

  constructor(settings, logger) {
    // ... Initialisation de la classe

    if (typeof logger['log'] !== 'function') {
      throw new TypeError("Le paramètre n'est pas un logger");
    }
    this.logger = logger;
  }
}

module.exports = Mailer;
```

En reprenant le test du canard, on peut dire "Si le paramètre `logger` possède une méthode `log` c'est qu'il s'agit sans doute d'un logger".

Cette technique permet donc de "simuler" la notion d'interface. On vérifie simplement l'existence des méthodes, mais notre code pose un souci : on doit à chaque fois vérifier dans le constructeur que le paramètre reçu possède bien les méthodes attendues, et ce pour chacune des classes qui utiliseront un logger. Ici, nous avons qu'une seule méthode (`log`), mais si l'on a plusieurs méthodes, dites bonjour à la duplication de code.

Encapsulons donc cette vérification. Dans un premier temps, créons une classe permettant de définir une interface :

Fichier Interface.js

```
class Interface {

  constructor(name, ...methods) {
    if (typeof name !== 'string' || name.length === 0) {
      throw new Error('The "name" argument must be a non-empty string')
    }
```

```
    if (!Array.isArray(methods) || methods.length === 0) {
      throw new Error('The "methods" argument must be a non-empty array of st
    }

    this.name = name
    this.methods = []
    for (let method of methods) {
      if (typeof method !== 'string') {
        throw new Error('The "methods" argument must contains only string')
      }
      this.methods.push(method)
    }
  }
}

module.exports = Interface;
```

Le constructeur prend en paramètre le nom de l'interface ainsi qu'un nombre indéfini de chaînes de caractères contenant les noms des méthodes de l'interface (Cf Rest parameters).

Ajoutons maintenant la fonction permettant de vérifier qu'un objet implémente bien une interface :

Fichier Interface.js

```
class Interface {

  constructor(name, ...methods) {
    if (typeof name !== 'string' || name.length === 0) {
      throw new Error('The "name" argument must be a non-empty string')
    }
    if (!Array.isArray(methods) || methods.length === 0) {
      throw new Error('The "methods" argument must be a non-empty array of st
    }

    this.name = name
    this.methods = []
    for (let method of methods) {
      if (typeof method !== 'string') {
        throw new Error('The "methods" argument must contains only string')
      }
      this.methods.push(method)
    }
  }

  static checkImplements(object, ...interfaces) {
    if (!Array.isArray(interfaces) || interfaces.length === 0) {
      throw new Error('The "interfaces" argument must be a non-empty array of
    }

    for (let itf of interfaces) {
      if (itf.constructor !== Interface) {
```

```
        throw new Error('The "interfaces" argument must contains instances of
      }

      const missingMethods = [];
      for (let method of itf.methods) {
        if (!object[method] || typeof object[method] !== 'function') {
          missingMethods.push(method);
        }
      }

      if (missingMethods.length > 0) {
        throw new Error(`The object doesn't implement the ${itf.name} interfa
      }
    }
  }
}

module.exports = Interface;
```

La méthode `checkImplements` prend en paramètre l'objet à vérifier ainsi qu'un nombre indéfini d'instances de la classe `Interface`. Elle vérifie simplement que l'objet possède bien les méthodes des interfaces qui sont passées en paramètres.

Reprenons donc notre exemple précédent et créons une interface de logger :

Fichier LoggerInterface.js

```
const Interface = require('./Interface');

module.exports = new Interface('Logger', 'log');
```

Vérifions maintenant dans notre classe Mailer que l'objet passait en paramètre du constructeur implémente bien notre interface `Logger` :

Fichier Mailer.js

```
const Interface = require('./Interface');
const LoggerInterface = require('./LoggerInterface')

class Mailer {

  constructor(settings, logger) {
    // ... Initialisation de la classe

    Interface.checkImplements(logger, LoggerInterface);
    this.logger = logger;
  }
}

module.exports = Mailer;
```

Si nous passons en paramètre un objet qui ne contient pas une méthode log, une exception sera levée avec le message suivant :

```
The object doesn't implement the Logger interface. Methods not found : log
```

Il est également possible de vérifier qu'une classe implémente bien une interface lors de son instanciation :

```
const fs = require('fs');
const path = require('path');
const Interface = require('./Interface');
const LoggerInterface = require('./LoggerInterface')

class FileLogger {

  constructor(settings) {
    if (!settings.path || !settings.filename) {
      throw new Error('The settings "path" and "filename" are required');
    }
    this.stream = fs.createWriteStream(path.join(settings.path, settings.file

    Interface.checkImplements(this, LoggerInterface);
  }

  log(msg) {
    this.stream.write(`[${new Date().toISOString()}] ${msg}\n`);
  }

  close() {
    this.stream.end();
  }
}

module.exports = FileLogger;
```

Notre exemple est tout simple, notre interface `Logger` possède qu'une seule méthode `log`, mais il possible de définir plusieurs méthodes comme suit :

Fichier LoggerInterface.js

```
const Interface = require('./Interface');

module.exports = new Interface('Logger', 'debug', 'info', 'warning', 'error',
```

C'était simple non?

Les classes abstraites et interfaces ont chacune une fonction bien distincte : l'une sert à factoriser du code, tandis que l'autre à définir des contrats. Bien que le langage Javascript ne possède pas ces deux notions, nous avons vu qu'il était assez simple de les implémenter.

Les solutions ont été écrites pour Node.js, mais pourront très simplement être utilisées pour du Javascript côté front par exemple.

J'ai brièvement rappelé que l'héritage multiple n'existait pas en Javascript, nous verrons dans le prochain article une solution qui permet entre autres de pallier à ce problème (je vous laisse deviner sur quoi il portera 😉).

Publié dans Architecture logicielle, Javascript / Étiqueté avec javascript, node.js, poo

‹ Node.js : l'inversion de contrôle (IOC) et l'injection de dépendances (DI)

Javascript : l'héritage multiple ›

Une réflexion au sujet de « Javascript : Classes abstraites et interfaces »

Ping : Javascript : l'héritage multiple - Code Heroes

Laisser un commentaire

Commentaire

Nom *