JavaScript, les functions

Alvin Berthelot

Version 1.0.0

Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons.

Attribution - Partage dans les Mêmes Conditions 3.0 non transposé.

La licence, ses explications ainsi que les moyens de contribution et réappropriation sont détaillés à la fin.

Déclaration des fonctions

Définition d'une fonction

Rappel

Les fonctions en JavaScript permettent d'organiser et de regrouper des comportements au sein de notre code afin de les réutiliser.

```
function sayHello(name) {
  console.log('Hello ' + name);
};

var sayGoodbye = function(name) {
  console.log('Goodbye ' + name);
}

sayHello('Grumpy');

sayGoodbye('Grumpy');
```

Constitution d'une fonction

Une fonction se constitue de la sorte.

```
function sum(a, b, c) {
  return a + b + c;
}
```

- Un mot clef `function`.
- Un nom (facultatif), ici `sum`.
- Des arguments (facultatifs), ici `a`, `b` et `c`.
- Des instructions entre accolades (accolades obligatoires).
- Un mot clé `return` (facultatif) qui renvoie une valeur.

Le nom d'une fonction

Une fonction sans nom est appelée une fonction anonyme. Elle ne peut être invoquée que dans des conditions particulières.

Soit en étant associée à une variable.

```
var sum = function(a, b, c) {   return a + b + c;   };
console.log(sum(1, 2, 3)); // 6
```

Soit en étant définie comme un "callback" d'une autre fonction.

```
setTimeout(function() { console.log('Sorry I\'m late !'); }, 4000);
```

Soit via une IIFE : Immediately-Invoked Function Expression.

```
(function() {   console.log('Hello World immediately !'); })();
```

Les arguments d'une fonction

Une fonction peut être appelée avec un nombre d'arguments différent du nombre défini (aussi bien plus que moins).

Si il y a plus d'arguments fournis, ceux en trop sont ignorés.

```
console.log(sum(1, 2, 3, 4)); // 6
```

Si il y a moins d'arguments, ceux non définis sont `undefined`.

```
console.log(sum(1, 2)); // NaN
```

L'instruction d'interruption `return`

Lorsque l'éxecution arrive sur une instruction `return`, on sort immédiatement de la fonction.

Si une fonction ne définit pas l'instruction `return` alors elle retourne `undefined` par défaut.

```
function sayHello(name) {
  console.log('Hello ' + name);
};

console.log(sayHello('Grumpy')); // undefined
```

Les fonctions et le "scope"

La notion de "scope"

Par "scope", on définit la portée et la durée de vie des variables conditionnées par l'endroit de leur déclaration.

Contrairement à d'autres langages, JavaScript n'a pas de "scope" au niveau des blocs (c'est à dire entre {}) mais a **un scope au niveau des fonctions**.

```
var sum = function(a, b) {
  var baseSum = 4;
  return baseSum + a + b;
};
console.log(sum(1, 2)); // 7
console.log(baseSum); // ERROR

function substract(a, b) {
  if(true) {
    var baseSubstract = 100;
  }
  return baseSubstract - a - b;
};

console.log(substract(1, 2)); // 97
console.log(baseSubstract); // ERROR
```

Le "scope" global

JavaScript dispose d'un "scope" global, il s'agit d'un `object` disposant déjà d'un certain nombre de propriétés et de méthodes.

Dans un navigateur

```
console.log(location.href); // affiche l'URL du navigateur
console.log(setTimeout); // affiche la fonction setTimeout
```

Par défaut on est automatiquement positionné sur le "scope" global.

Dans un navigateur

```
var magicNumber = 42;

window.magicNumber++;

console.log(magicNumber); // 43
```

Limitation de l'utilisation du "scope" global

L'isolation de variables dans des "scopes" autres que global réduit le risque de collision et optimise l'usage de la mémoire.

C'est donc une très bonne pratique de ne pas gonfler le "scope" global.

On peut omettre le mot clé `var` pour déclarer une variable, mais cette particularité fait que celle-ci aura une portée globale même si elle est déclarée dans une fonction !

```
var sum = function(a, b) {
  baseSum = 4;
  return baseSum + a + b;
};
console.log(sum(1, 2)); // 7
console.log(baseSum); // 4
```

Imbrication de "scopes"

Une fonction en plus des paramètres qui lui sont passés, peut accéder au "scope" l'ayant déclaré.

```
var sayHref = function() {   console.log(location.href); };
sayHref(); // affiche l'URL du navigateur
```

Ce mécanisme est vrai quel que soit le niveau d'imbrication des "scopes" (c'est à dire des fonctions), on appelle cela une "closure";

```
var sayHelloHref = function() {
  var hello = 'Bonjour';
  var sayHref = function() {
    console.log(hello + ' ' + location.href);
  };
  return sayHref;
};
var welcomeBrowser = sayHelloHref();
welcomeBrowser(); // Bonjour + affiche l'URL du navigateur
```

Définition de "closure"

En JavaScript une "closure" ou fermeture (lexicale) est un objet spécial qui combine une fonction et l'environnement dans lequel elle a été créée.

Le mot "closure" pourrait être expliqué par le fait que l'on a enfermé le "scope" auquelle la fonction se réfère dans la fonction elle même au moment de sa déclaration.

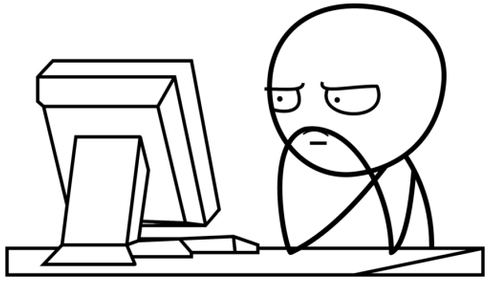
Pour des raisons de simplicité dans la suite de ces slides je parlerai plutôt de contexte de déclaration ; mais attention il ne s'agit absolument pas d'un terme que vous retrouverez dans la littérature à ce sujet.

Module pattern

Le fonctionnement des "closures" et des IIFE permettent l'exposition publique vs privée de propriétés d'un objet.

En voici l'exemple avec le module pattern.

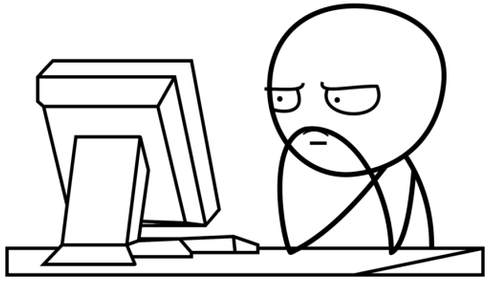
```
var namespace = (function() {
  var privVar = 'secret'; // propriétée privée
  var pubVar = 'not secret'; // propriétée publique
  var privMethod = function() { return privVar };
  return {
    pubVar: pubVar,
    pubMethod: privMethod
  };
})();

console.log(namespace.pubVar); // not secret
console.log(namespace.pubMethod()); // secret
console.log(namespace.privVar); // undefined
```



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "javascripting".

⇒ FUNCTIONS
⇒ FUNCTION ARGUMENTS
⇒ SCOPE



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Scope Chains & Closures".

⇒ Scopes
⇒ Scope Chains
⇒ Global Scope & Shadowing
⇒ Closures

Invocation des fonctions

Déclaration Vs Invocation

Pour bien comprendre les fonctions, il faut bien distinguer leur déclaration de leur invocation.

```
// Déclaration
function sayHello(name) {
  console.log('Hello ' + name);
};

// Invocation
sayHello('Grumpy');
```

Une fonction est forcément déclarée si on trouve le mot clé `function` (en ES5 en tout cas) et forcément invoquée si elle est suivie de parenthèses `()` mais sans le mot clé `function`.

Quizz

Voyons si vous devinerez ce qui s'affiche.

```
function hello() {
  console.log('Hello World !');
}; // QUIZZ 1 ?

var politeServer = {
  connect: hello // QUIZZ 2 ?
};

politeServer.connect; // QUIZZ 3 ?

function goodbye() {
  console.log('Hello World immediately !');
}(); // QUIZZ 4 ?
```

Les paramètres d'invocation

Lorsqu'une fonction est invoquée, en plus des paramètres déclarés (éventuellement passés), 2 autres paramètres sont accessibles : `arguments` et `this`.

`arguments` retournant un tableau avec tous les paramètres passés lors de l'invocation.

`this` retournant le contexte d'invocation de la fonction.

```
var sum = function(a, b, c) {
  console.log(arguments);
  console.log(this);
  return a + b + c;
};
```

What is `this` ?

L'initialisation de l'argument `this` va dépendre de la façon dont la fonction est invoquée, cela se distingue en 4 cas :

- method : si la fonction est définie comme une propriété d'un objet, `this` fera référence à cet objet.
- function : si la fonction n'est pas une propriété d'objet, `this` fera référence au contexte global.
- constructor : l'invocation d'une fonction avec le préfixe `new` (constructeur, par convention nommée avec une majuscule) va induire un comportement particulier, un objet va être créé et `this` y fera référence.
- apply : les fonctions disposent d'une méthode `apply`, leur permettant d'être appelées en définissant `this` (premier argument de `apply`) tout en passant les paramètres (second argument, sous forme de tableau).

Quizz

Voyons si vous devinerez ce qui s'affiche.

```
var name = 'Wonderful program';

function hello() {
  console.log('Hello my name is ' + this.name);
};

var politeServer = {
  name: 'localhost',
  connect: hello
};

var Server = function() { this.name = 'aws' };
var awsServer = new Server();

console.log(hello()); // QUIZZ 1 ?

console.log(politeServer.connect()); // QUIZZ 2 ?

console.log(awsServer.hello()); // QUIZZ 3 ?

console.log(hello(awsServer)); // QUIZZ 4 ?

console.log(hello.apply(awsServer, ['cloud'])); // QUIZZ 5 ?
```

La programmation orientée objet

Héritage prototypal

Rappel

En JavaScript, un objet peut être "dérivé" d'un autre avec la propriété `__proto__`. Ce dernier est alors considéré comme son prototype.

```
// prototype
var animal = {
  "name": 'Grumpy',
  "age": 3,
  "male": true
};

var cat = {};
cat.__proto__ = animal;

console.log('My cat ' + cat.name + ' is ' + cat.age + ' years old.');
```

Un héritage prototypal plus conventionnel

Le fait d'utiliser la propriété `__proto__` n'est pas l'approche adéquate car cette technique n'est pas toujours bien supportée selon les navigateurs.

Pour arriver à un résultat équivalent on préferera utiliser `Object.create()` qui s'utilise de la manière suivante.

```
// prototype
var animal = { name: 'Grumpy', age: 3, male: true };
var cat = Object.create(animal);
console.log('My cat ' + cat.name + ' is ' + cat.age + ' years old.');
```

Comme auparavant on peut étendre ou surcharger certaines propriétés.

```
var dog = Object.create(animal, {
  name: {value: 'Droopy'}
});
console.log('My dog ' + dog.name + ' is ' + dog.age + ' years old.');
```

Factoriser l'héritage

On pourrait être tenté de créer une fonction pour faciliter cela :

```
function createAnimal(animalToInstantiate) {
  return Object.create(animal, animalToInstantiate);
};

var dog = createAnimal({
  name: {value: 'Droopy'}
});
```

Mais cela pose des problèmes :

- Ce n'est vraiment pas naturel à écrire.
- Même si on dispose de nouvelles propriétés, on dépend toujours directement de `Object.prototype` dans la chaîne de dérivation.

Les constructeurs

Pour créer une véritable hiérarchie par classes on va passer par la notion de constructeurs.

Un contructeur est une fonction en soit mais qui a va être invoquée avec le mot clé `new`, ce qui aura une importance particulière sur le `this` lors de l'invocation et sur le `return` qui renverra la référence de l'objet créé (si on ne définit pas un objet dans le `return`).

```
function Animal(name, age, male) {
  this.name = name;
  this.age = age;
  this.male = male;
};

var dog = new Animal('Droopy', 3, true);

console.log('My dog ' + dog.name + ' is ' + dog.age + ' years old.');
```

Les conventions des constructeurs

```
function Animal(name, age, male) {
  this.name = name;
  this.age = age;
  this.male = male;
};

var dog = new Animal('Droopy', 3, true);

console.log('My dog ' + dog.name + ' is ' + dog.age + ' years old.');
```

⚠ Ce n'est pas obligatoire mais la convention veut que l'on commence un constructeur par une majuscule.

⚠ En théorie vous ne devriez pas spécifier quoi que ce soit dans le return d'un constructeur.

Relation constructeur / prototype

En réalité chaque objet dispose d'un constructeur avec la propriété `constructor` qui retourne une fonction.

```
var whatever = {};
console.log(whatever.constructor); // function Object() { ... }
```

Chaque fonction dispose d'une propriété `prototype` qui retourne un objet.

```
var sum = function(a, b, c) {   return a + b + c;   };
console.log(sum.prototype); // Object {}
```

Cela est possible car **une fonction en JavaScript est aussi un objet !**.

Hiérarchie plus fine

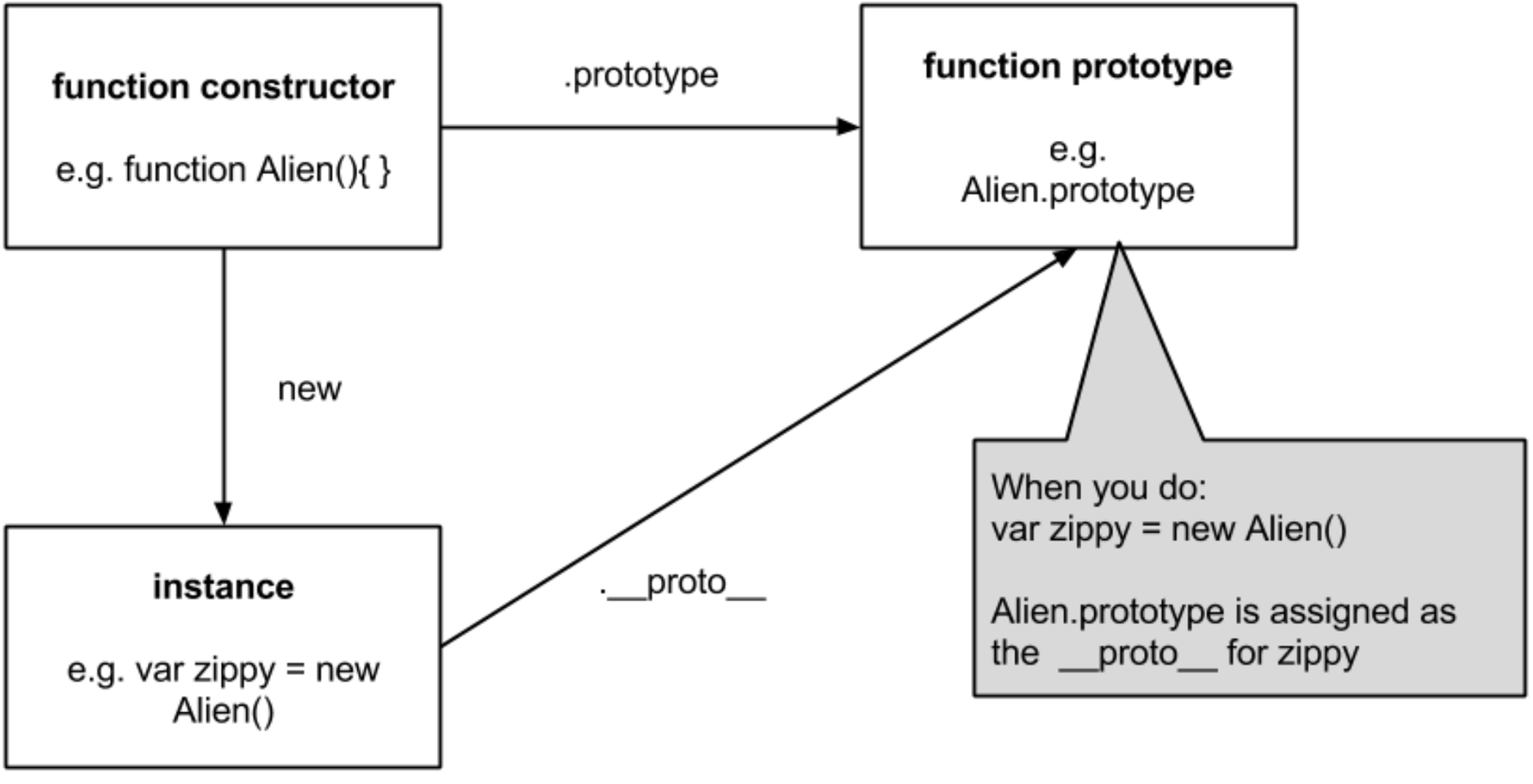
Avec l'approche des constructeurs la hiérarchie entre les objets est beaucoup plus fine car ils ne dépendent plus directement de `Object.prototype` mais du prototype qui est retourné par le constructeur.

```
function Animal(name, age, male) {
  this.name = name;
  this.age = age;
  this.male = male;
};
var dog = new Animal('Droopy', 3, true);
dog.toString(); // [object Object]
Animal.prototype.toString = function() {
  console.log('I\'m a animal');
};
dog.toString(); // I'm a animal
```

`Object.prototype` est le prototype des objets par défaut en JavaScript, il a une incidence sur tous les objets puisqu'ils dérivent tous de lui (y compris Animal).

`Animal.prototype` a une incidence sur tous les objets qui dérivent de lui.

Tout en une image

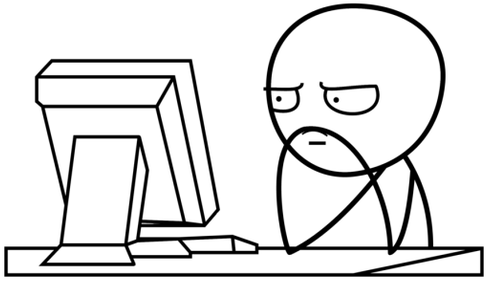


Héritage au final en ES5

```
function Animal(name, age, male) {
  this.name = name;
  this.age = age;
  this.male = male;
};

function Cat(name, age, male, color) {
  Animal.apply(this, [name, age, male]);
  this.color = color;
};
Cat.prototype = Object.create(Animal.prototype, {
  constructor: {value: Cat}
});
Cat.prototype.action = function() {
  return 'meaows';
};

var cat = new Cat('Grumpy', 3, true, 'grey');

console.log('My ' + cat.color + ' cat ' + cat.name + ' ' + cat.action());
```

Bilan

- La portée des variables se fait via le "scope" d'une fonction.
- La déclaration d'une fonction avec ce que cela implique sur les arguments et le contexte de déclaration (closure).
- L'invocation d'une fonction avec ce que cela implique sur les arguments, `this` et `return`.
- Les types de valeur objet et fonction sont en réalité fortement liés.
- La programmation orientée objet c'est compliqué en ES5 … vive la syntaxe ES6.



Exercices

On va s'appuyer sur la plateforme NodeSchool.io, pour cela vous devez installer le TP "Planet Proto".

⇒ Object Create
⇒ Dot New
⇒ Constructor functions
⇒ Implicit this
⇒ Function prototype

Licence

Pour toutes questions, réclamations ou remarques, merci d'envoyer un message à alvin.berthelot@webyousoon.com.

Explications licence CC BY-SA 3.0

Cette licence permet aux autres de remixer, arranger, et adapter votre œuvre, même à des fins commerciales, tant qu'on vous accorde le mérite en citant votre nom et qu'on diffuse les nouvelles créations selon des conditions identiques.

Cette licence est souvent comparée aux licences de logiciels libres, "open source" ou "copyleft".

Toutes les nouvelles œuvres basées sur les vôtres auront la même licence, et toute œuvre dérivée pourra être utilisée même à des fins commerciales.

C'est la licence utilisée par Wikipédia ; elle est recommandée pour des œuvres qui pourraient bénéficier de l'incorporation de contenu depuis Wikipédia et d'autres projets sous licence similaire.

Contribution et réappropriation

Ce fichier PDF est généré avec Asciidoctor à partir d'un dépôt Git se trouvant sous GitHub.

https://github.com/alvinberthelot/slides-js

Cela signifie que vous n'avez pas besoin de vous battre avec un fichier binaire (le PDF) pour **contribuer**, **vous réapproprier le contenu** ou **modifier le thème** de présentation.

Contribution

Vous voulez **contribuer au contenu** car :

- Il y a une erreur (ça arrive à tout le monde), de typographie, de compréhension, ou tout autre chose.
- Vous souhaitez apporter une précision.

Il vous suffit de contribuer au projet via Git par le moyen d'une "pull request" sur le dépôt Git.

Réappropriation

N'oubliez pas les conditions de la licence.

Vous voulez vous **réapproprier le contenu** car :

- Vous souhaitez donner un style différent.
- Vous souhaitez enlever/ajouter/modifier des sections dans votre contexte.

Il vous suffit de "forker" le dépôt Git et d'y apporter vos propres modifications, puis de générer par vous même le nouveau PDF.