

5. Orienté Objet

Jean-Luc Falcone

Août 2019

Orienté-Objet en JavaScript

- Le langage JavaScript est orienté-objet, mais **sans classes** !

Orienté-Objet en JavaScript

- Le langage JavaScript est orienté-objet, mais **sans classes** !
- Il existe des mots-clé `this` et `new` mais la **sémantique est différente** de celle de Java, C++, Scala, C#, etc.

Orienté-Objet en JavaScript

- Le langage JavaScript est orienté-objet, mais **sans classes** !
- Il existe des mots-clé `this` et `new` mais la **sémantique est différente** de celle de Java, C++, Scala, C#, etc.
- La version ES6 introduit une nouvelle syntaxe plus habituelle avec des "classes". **Elle est cependant déconseillée.**

Le mot clé `this`

En Java, `this` représente l'instance à partir de laquelle la méthode est définie.

En JS, `this` représente l'objet dans lequel la méthode est appelée.

Le mot clé this (1)

```
let counter = {  
  value: 0,  
  incr: function() {  
    this.value += 1;  
  }  
};  
  
> counter  
{ value: 0, incr: [Function] }  
> counter.incr(); counter.incr(); counter.incr();  
> counter  
{ value: 3, incr: [Function] }
```

Le mot clé this (2)

```
function f() {  
  this.x += 1;  
}  
  
let a = { x: 2, incr: f };  
let b = { x: "hello", incr: f };  
  
> a.incr();  
> b.incr();  
> a  
{ x: 3, incr: [Function: f] }  
> b  
{ x: 'hello1', incr: [Function: f] }
```

Mot clé new

Le mot clé `new` permet d'interpréter une fonction comme un constructeur:

```
function Counter( x ) {  
  this.value = x;  
  this.incr = function() {  
    this.value += 1;  
  }  
}  
  
> let c = new Counter(2);  
> c  
Counter { value: 2, incr: [Function] }  
> c.incr();  
> c  
Counter { value: 3, incr: [Function] }
```


Mot clé new (2)

Que se passe-t-il si on oublie le new ?

```
function Counter( x ) {  
  this.value = x;  
  this.incr = function() {  
    this.value += 1;  
  }  
}
```

```
> let c = Counter(2);
```

Mot clé new (3)

Si l'on oublie le new on va polluer l'espace global !

```
> let c = Counter(2);
```

```
> c
```

```
undefined
```

```
> value
```

```
2
```

```
> incr()
```

```
undefined
```

```
> value
```

```
3
```

Prototype: fonctions

Chaque objet a un prototype qui contient des méthodes par défaut:

```
function Counter(x) {  
  this.value = x;  
}  
Counter.prototype.incr = function() {  
  this.value += 1;  
}  
> let c = new Counter(0);  
> c  
Counter { value: 0 }  
> c.incr()  
undefined  
> c  
Counter { value: 1 }
```

Prototype: propriétés

Cela fonctionne également avec les propriétés. Lorsqu'un objet la modifie, elle est clonée dans l'objet.

```
function Counter() { }  
Counter.prototype.value = 0;  
Counter.prototype.incr =  
  function() {  
    this.value += 1;  
  }
```

Prototype: propriétés

Cela fonctionne également avec les propriétés. Lorsqu'un objet la modifie, elle est clonée dans l'objet.

```
function Counter() { }  
Counter.prototype.value = 0;  
Counter.prototype.incr =  
  function() {  
    this.value += 1;  
  }
```

```
> let c1 = new Counter();  
> let c2 = new Counter();  
> c1.incr();  
> c1  
Counter { value: 1 }  
> c2  
Counter {}  
> c2.incr()  
> c2  
Counter { value: 1 }
```

Prototype: juste un objet

Le prototype est juste un objet qui contient les propriétés et méthode assignées.

```
> Counter.prototype
```

```
Counter { value: 0, incr: [Function] }
```

Prototype: héritage

On peut utiliser le chaînage des prototypes pour hériter d'un autre constructeur.

```
function Counter() {}  
Counter.prototype.incr = function() { this.value += 1; }  
Counter.prototype.value = 0;
```

```
function CounterDeluxe() { }  
CounterDeluxe.prototype = new Counter();  
CounterDeluxe.prototype.reset =  
    function() { this.value = 0 };
```

Prototype: héritage (démonstration)

```
> let c = new Counter();  
> c.incr(); c.incr();  
> c  
Counter { value: 2 }  
> let cd = new CounterDeluxe();  
> cd.value  
0  
> cd.incr();  
> cd  
Counter { value: 1 }
```


Méthode alternative

On peut se passer facilement de `new` et de `this` en utilisant une autre approche:

```
function Counter( x ) {  
  let value = x;  
  return {  
    incr: function() { value += 1; },  
    get: function() { return value; },  
    reset: function() { value = 0; }  
  }  
}
```

Méthode alternative (démonstration)

```
> let c = Counter(2);  
> c  
{ incr: [Function], get: [Function], reset: [Function] }  
> c.get()  
2  
> c.incr()  
> c.get()  
3
```

Méthode alternative: Encapsulation

On peut également obtenir des méthodes **privées** avec cette approche:

```
function Counter( x ) {  
  let value = x;  
  function add( n ) {  
    value += n;  
  }  
  return {  
    incr: function() { add(1); },  
    decr: function() { add(-1); },  
    get: function() { return value; },  
    reset: function() { value = 0; }  
  }  
}
```

Chaînage de méthodes

```
function Counter( x ) {  
  let value = x;  
  function add( n ) {  
    value += n;  
  }  
  let c = {  
    incr: function() { add(1); return c; },  
    decr: function() { add(-1); return c; },  
    get: function() { return value; },  
    reset: function() { value = 0; return c; }  
  }  
  return c;  
}
```

Méthode alternative: Chaînage (démonstration)

```
> let c = Counter(2);  
> c.reset().incr().incr().incr().get();  
3
```

Méthode alternative: Héritage

On peut utiliser le chaînage des prototypes pour hériter d'un autre constructeur.

```
function Counter() {  
  let value = 0;  
  return {  
    incr: function() { value += 1; },  
    get: function() { return value; },  
    set: function(i) { value = i; }  
  };  
}  
  
function CounterDeluxe() {  
  let c = Counter();  
  c.reset = function() { c.set(0); };  
  c.incrN = function( n ) { c.set( c.get() + n ); }  
  return c;  
}
```

Méthode alternative: Héritage (démonstration)

```
> let cd = CounterDeluxe();  
> cd.incr();  
> cd.incrN(100);  
> cd.get();  
101
```