

Javascript

Programmation Orientée Objet

Objets Littéraires

- Sorte de tableau associatif
- Associe des clefs à des valeurs
- Une fonction est une valeur

Example

```
var person = {  
  name: 'Joe',  
  age: 43,  
  
  greet: function(){  
    console.log('Hello');  
  }  
}  
  
person.greet();
```


What is this

- fait référence au contexte d'exécution d'une méthode
- this est déterminé par qui invoque la méthode (la partie gauche de l'invocation)
- sa valeur est donc contextuelle


```
function showThis(){  
    console.log(this);  
}
```

//this est une référence de l'objet window
showThis();

```
var person = {  
    firstName: 'Noriko',  
    showThis: function(){  
        console.log(this);  
    }  
};
```

//this est une référence de l'objet Person
person.showThis();


```
var person = {  
  firstName: 'Noriko',  
  showThis: function(){  
    console.log(this);  
  }  
};
```

```
var showAgain = person.showThis;
```

```
//this est une référence de l'objet window  
showAgain();
```

```
var bt = document.getElementById('bt-1');  
bt.addEventListener('click', function () {  
    //this fait référence à la cible de l'écouteur  
    console.log(this);  
});  
  
person = {  
    firstName: 'Noriko',  
    greet: function(){  
        console.log('Hello' + this.firstName);  
    }  
};  
//this fait également référence à bt  
bt.addEventListener('click', person.greet);
```



```
var person = {  
    firstName: 'Noriko',  
    greet: function(){  
        console.log('Hello' + this.firstName);  
    }  
};  
  
var person2 = {firstName: "Wakuko"};  
  
person2.doGreet = person.greet;  
  
//This fait référence à person2  
person2.doGreet();
```


Méthode qui changent le
contexte (réinitialisent this)

```
var list = [5,8,3];  
list.forEach(function (item) {  
    console.log(item);  
    //this = window et pas array  
    console.log(this);  
});
```


Méthode qui changent le
contexte (réinitialisent this)

```
var db = {  
  title: 'Friends',  
  people: ['Scott', 'Wakuko', 'Seamus'],  
  print: function(){  
    this.people.forEach(function (friend) {  
      //this = window  
      console.log(this.title + ' : ' + friend);  
    })  
  }  
};  
  
db.print();
```


solution

```
var db = {  
  title: 'Friends',  
  people: ['Scott', 'Wakuko', 'Seamus'],  
  print: function(){  
    var that = this;  
    this.people.forEach(function (friend) {  
      //this = window  
      console.log(that.title + ' : ' + friend);  
    })  
  }  
};  
  
db.print();
```


autre solution

```
db = {  
  title: 'Friends',  
  people: ['Scott', 'Wakuko', 'Seamus'],  
  print: function(){  
    var callback = function (friend) {  
      console.log(this.title + ' : ' + friend);  
    };  
  
    //Détermination du contexte en deuxième argument  
    this.people.forEach(callback, this);  
  
    //Injection d'un contexte personnalisé  
    this.people.forEach(callback, {title: 'amis'});  
  }  
};
```


Injection du contexte

```
person = {  
  firstName: 'Noriko',  
  greet: function(){  
    console.log('Hello ' + this.firstName);  
  }  
};  
person2 = {firstName: 'Leila'};  
sayHello = person.greet;  
  
//Injection du contexte  
sayHello.call(person2);  
person.greet.call(person2);
```


Passage de paramètres

```
person = {  
  firstName: 'Noriko',  
  greet: function(greeting, title){  
    console.log(greeting + ' '  
      + title + ' ' + this.firstName);  
  }  
};  
  
person2 = {firstName: 'Leila'};  
sayHello = person.greet;  
  
//Injection du contexte  
//Les paramètres additionnels sont passés  
//à la méthode  
sayHello.call(person2, 'Hola', 'Miss');
```


function apply

```
person = {  
    firstName: 'Noriko',  
    greet: function(greeting, title){  
        console.log(greeting + ' '  
            + title + ' ' + this.firstName);  
    }  
};  
  
person2 = {firstName: 'Leila'};  
sayHello = person.greet;  
  
//Les arguments de la méthode sont passés  
//dans un tableau  
sayHello.apply(person2, ['Hola', 'Miss']);
```


Une application

```
var list = [89,3,9087,345];  
  
//Retourne le valeur max du tableau  
console.log(  
  Math.max.apply(null, list);  
);
```

utiliser un tableau pour passer des arguments à une fonction

Fixer le contexte

```
var log = console.log.bind(console);  
log("Hi");
```

```
person = {  
  firstName: 'Noriko',  
  greet: function(){  
    console.log('Hello' + this.firstName);  
  }  
};  
  
bt.addEventListener('click',  
  person.greet.bind(person)  
);
```


currying

déterminer certains arguments
d'une fonction

```
function add(n1, n2){  
  return n1 + n2;  
}  
//Le premier argument est fixé à 5  
var add5 = add.bind(null, 5);  
  
//Retourne 15  
console.log(add5(10));
```


Utilité

- Simplifier les appels à une fonction en préfixant certains arguments
- Emprunter des méthodes à un objet pour les utiliser dans un autre contexte
- Fixer le contexte d'un écouteur d'événement

Fonction
constructeur

Un constructeur

```
function Person(firstName, age){  
    this.firstName = firstName;  
    this.age = age;  
}  
  
var alice = new Person('Alice', 24);  
var pierre = new Person('Pierre', 21);
```


Méthode

```
function Person(firstName, age){  
  this.firstName = firstName;  
  this.age = age;  
  
  this.greet = function(){  
    console.log('Hello ' + this.firstName);  
  }  
}  
  
var alice = new Person('Alice', 24);  
  
alice.greet();
```


Visibilité des variables

```
//id n'est pas exporté, il est privé

function Person(firstName, age, id){
  this.firstName = firstName;
  this.age = age;

  this.greet = function(){
    //id est disponible au sein de l'objet
    console.log(id + ' ' + this.firstName);
  };
}

var alice = new Person('Alice', 24, '0012');

//id n'est pas disponible
console.log(alice.id)
```


Visibilité des méthodes

```
function Person(firstName, age){  
    this.firstName = firstName;  
    this.age = age;  
  
    var that = this;  
  
    //Méthode publique  
    this.greet = function(){  
        console.log(getGreetings());  
    };  
  
    //Méthode privée  
    function getGreetings(){  
        return 'Hello ' + that.firstName;  
    }  
}
```


Visibilité des méthodes

```
function Person(firstName, age){
  this.firstName = firstName;
  this.age = age;

  var that = this;

  //Méthode publique
  this.greet = function(){
    console.log(getGreetings());
  };

  //Méthode privée
  var getGreetings = function(){
    return 'Hello ' + that.firstName;
  }
}
```


Alternative

```
function PersonFactory(firstName) {  
  
    var sayHello = function(){  
        console.log('Hello ' + firstName);  
    };  
  
    //Retourne un objet littéral  
    return {  
        greet: sayHello,  
        setFirstName: function (value) {  
            firstName = value;  
        }  
    }  
}
```


Les propriétés

Objectifs

Créer ou modifier une propriété d'un objet en définissant ses attributs

Les attributs

propriété	description
configurable	Indique si Les attributs de la propriétés peuvent être modifiés et si la propriété peut être modifiée. False par défaut
enumerable	Indique si la propriété est listée lors du parcours des propriétés de l'objet. False par défaut
writable	Indique si la valeur de la propriété peut être modifiée avec un opérateur d'affectation. False par défaut
value	La valeur initiale de la propriété. Undefined par défaut
get	Une fonction utilisée pour accéder à la valeur de la propriété
set	Une fonction utilisée pour modifier la valeur de la propriété

Exemple

```
var person = {};
```

```
Object.defineProperty(person, 'name',  
  {  
    value: 'Inconnu',  
    writable: true  
  }  
);
```


Example

```
function Person(name, firstName){
    this.name = name;
    this.firstName = firstName;
}

Object.defineProperty(person, 'fullName',
    {
        get: function(){
            return this.firstName + ' ' + this.name;
        }
    }
);
```


Tester L'existence d'une propriété

```
//Avec un opérateur  
if('name' in amadeus){  
    console.log("amadeus has a name");  
}
```

```
//Avec une fonction  
if (amadeus.hasOwnProperty('name')){  
    console.log("amadeus has a name");  
}
```


Prototype

Définition

- Chaque objet JavaScript possède un lien vers un objet parent, c'est son prototype
- Un objet hérite de méthodes et propriétés de son prototype
- L'héritage s'effectue en cascade jusqu'à atteindre l'objet Object dont le prototype est null

Prototypes par défaut

- Le prototype d'un objet littéral est Object.
- Le prototype d'un objet construit avec le mot clef new est la fonction constructeur de l'objet

Extension du prototype

```
function Person(name, firstName) {  
    this.name = name;  
    this.firstName = firstName;  
}
```

```
//Ajout d'une méthode au prototype  
Person.prototype.getFullName = function () {  
    return this.firstName + ' ' + this.name;  
};
```

```
var joe = new Person('Smith', 'Joe');  
console.log(joe.getFullName());
```


Extension du prototype

```
//Ajout d'une propriété  
Person.prototype.kind = "friends";  
  
var jane = new Person('Doe', 'Jane');  
  
//La propriété est partagée  
//par toutes les instances  
console.log(jane.kind);  
console.log(joe.kind);
```


Utilité

- Les méthodes définies dans une fonction constructeur sont dupliquées pour chaque instance
- Les méthodes définies dans un prototype sont partagées

Héritage

Points clefs

- Chaîne de prototypes
- `Object.create(parent)`


```
function Animal() {}

function Mamal() {};
Mamal.prototype = Object.create(Animal.prototype);

function Cat () {};
Cat.prototype = Object.create(Mamal.prototype);


Animal.prototype.eat = function () {
    console.log("I eat");
};
Mamal.prototype.milk = function () {
    console.log("I produce milk");
};
Cat.prototype.fight = function () {
    console.log("I fight with my claws");
};
```



```
var sam = new Cat();
```

```
sam.fight();
```

```
sam.milk();
```

```
sam.eat();
```


Factory

```
function catFactory(name) {  
    var cat = Object.create(Cat.prototype);  
    cat.name = name;  
    return cat;  
}
```

```
function catFactory(name) {  
    var cat = Object.create(new Cat());  
    cat.name = name;  
    return cat;  
}
```


Helper

```
function Animal() {}  
function Mammal() {};  
function Cat () {};
```

```
Function.prototype.extends = function(parent){  
    this.prototype = Object.create(parent.prototype);  
    this.prototype.constructor = this;  
};
```

```
Mammal.extends(Animal);  
Cat.extends(Mammal);
```


Constructeur parent

```
function Person(name) {  
    this.name = name;  
}  
function Employee(name, salary) {  
    Person.call(this, name);  
    this.salary = salary;  
}  
  
var joe = new Employee('Joe', 5000);  
  
console.log(joe);
```


Injection de méthodes

```
Function.prototype.extends = function(parent){
    this.prototype = Object.create(parent.prototype);
    this.prototype.constructor = this;
};

function Vehicle(){};
function Plane(){};
Plane.extends(Vehicle);

function Animal(){};
function Bird(){};
Plane.extends(Animal);

function addFlight(obj){
    obj.fly = function () {
        console.log("I fly");
    };

    obj.land = function () {
        console.log("I land");
    };
}
```


Injection de méthodes

```
addFlight(Plane.prototype);  
addFlight(Bird.prototype);
```

```
var donald = new Bird();  
donald.fly();
```

```
var rafale = new Plane();  
rafale.fly();
```