Chapitre 2 : Concepts avancés JavaScript

1. Structures de données imbriquées

Objets dans des Objets
Exemple d'un objet dans un autre objet:

```
var schools = {
    IT: {
        address: "North Ave NW, Atlanta, GA 30332",
        phoneNumber: "(404) 894-2000",
        dateEstablished: "October 13, 1885"
    }
};
var gtObject = schools.IT;
```

Pour accéder à l'objet interne, on peut utiliser la notation point (dot notation):
avec la variable gtObject on peut accéder à des attributs de l'objet:

```
gtObject.address; // returns "North Ave NW, Atlanta, GA 30332"
gtObject.phoneNumber; // returns "(404) 894-2000"
gtObject.dateEstablished; //returns "October 13, 1885"
//on peut utiliser directement schools
schools.IT.address;
schools.IT.phoneNumber;
schools.IT.dateEstablished;
```

1.1 Objets Complexes

Exemple:

```
var instructorData = {
    name: "Tim",
    favoriteHobbies: ["Sailing", "Hiking", "Coding"]
};
```

La méthode d'accès est comme suit:

```
instructorData.favoriteHobbies[0]; // returns "Sailing"
```

Exemple combinant des objets et des tableaux :

```
var instructorData = {
    name: "Elie",
```

```
    additionalData: {
        instructor: true,
        favoriteHobbies: ["Playing Chess", "Tennis", "Coding"],
        moreDetails: {
            basketballTeam: "NYJ",
            numberOfSiblings: 3,
            isYoungest: true,
            hometown: {
                city: "West Orange",
                state: "NJ",
            },
            citiesLivedIn: ["Seattle", "Providence", "New York"]
        }
    }
};

instructorData.name; // "Elie"
instructorData.additionalData.instructor; // true
instructorData.additionalData.favoriteHobbies[2]; // "Coding"
instructorData.additionalData.moreDetails.fbasketballTeam; // "NYJ"
instructorData.additionalData.moreDetails.hometown.state; // "NJ"
instructorData.additionalData.moreDetails.citiesLivedIn[1]; //"Providence"
```

L'utilisation des objets complexes est une pratique courante en JS.

1.2 Accès et modification des contenus des objets imbriqués

- L'utilisation de la notation point est pratique lorsqu'on connaît les noms des attributs.
- Si on ne connaît pas exactement les noms des attributs, on doit utiliser la notation entre crochets (bracket notation).

Dans l'exemple suivant, on voudrait écrire une fonction qui ajoute un attribut (ici nameOfLanguage) à l'objet imbriqué programmingLanguages.

```
var programmingLanguages = {
    java: {
        yearCreated: 1995,
        creator: "James Gosling"
    },
    javaScript: {
        yearCreated: 1995,
        creator: "Brendan Eich"
    }
}

function addProgrammingLanguage(nameOfLanguage, yearLanguageCreated, creatorOfLanguage){
    programmingLanguages[nameOfLanguage] = {
        yearCreated: yearLanguageCreated,
        creator: creatorOfLanguage
    }
}

addProgrammingLanguage("ruby", 1995, "Yukihiro Matsumoto");
```

(voir exemple en TD)

2. Fonctions d'ordre supérieur (higher order functions)

- Les fonctions en JS sont une partie fondamentale. Elles peuvent accepter tous les types de paramètres: (strings, numbers, booleans, arrays objects).
- On peut aussi passer des fonctions comme paramètres d'autres fonctions!
- Une fonction ayant parmi ses paramètres une autre fonction est appelée fonction d'ordre supérieur ("higher order function."), c'est spécifique à JavaScript. La fonction en paramètre est appelée callback function.

Dans l'exemple suivant, la fonction **sendMessage** accepte comme paramètres un String et une fonction. La fonction **sendMessage** va retourner comme résultat le résultat de la fonction qui lui est passé comme paramètre avec pour argument le message (string):

```
function sendMessage(message, fn){
   return fn(message);
}

sendMessage("Hello World", console.log); // Hello World
sendMessage("Hello World", alert); // Hello World is alerted
sendMessage("What is your name?", prompt); //value from prompt is returned
sendMessage("Do you like JavaScript?", confirm);//true or false is returned
```

Important:

- Il faut différencier entre une fonction référencée et une fonction invoquée (appelée)
- Dans sendMessage("Hello World", console.log);, console.log est une fonction **référencée** mais **pas encore invoquée**. Rien n'est écrit sur la console jusqu'à ce que la fonction soit invoquée.
- Une fonction passé en paramètre à une fonction d'ordre supérieur doit être référencée seulement par son nom (c'est pourquoi on ne voit pas les paramètres de cette fonction dans l'invocation)
- C'est à l'intérieur de la fonction d'ordre supérieur que la fonction référencée sera invoquée (appelée avec des paramètres entre ( )) .
- Dans l'exemple précédent, la ligne **return** fn(message); montre l'invocation de la fonction, on aura un affichage sur la console.
- La fonction console.log retourne toujours *undefined*, donc dans ce cas sendMessage("Hello World", console.log); retournera *undefined*.

2.1 Fonctions anonymes comme Paramètres

On peut passer une fonction anonyme comme paramètre.

```
sendMessage("Hello World", function(message){
   // message refers to the string "Hello World"
   console.log(message + " from a callback function!");
});  // Hello World from a callback function!
```

L'exemple précédent est équivalent à:

```
var myFunction = function(message){
   // message refers to the string "Hello World"
   console.log(message + " from a callback function!");
};

sendMessage("Hello World", myFunction);
```

En JS, il est fréquent de passer des fonctions anonymes comme paramètres.

2.2 Pourquoi les fonctions d'ordre supérieur?

Utilité : Réutilisation du code.
Dans l'exemple précédent, on devrait écrire beaucoup de code pour avoir le même résultat. Les fonctions d'ordre supérieur nous évitent d'écrire des fonctions séparées pour chaque cas, comme illustré ci-après:

```
function sendMessageWithConsoleLog(message){
   return console.log(message);
}

function sendMessageWithAlert(message){
   return alert(message);
}

function promptWithMessage(message){
   return prompt(message);
}

function confirmWithMessage(message){
   return confirm(message);
}

function sendMessageWithFromCallback(message){
   return console.log(message + " from a callback function!");
}
```

Au lieu d'écrire 5 fonctions différentes, il nous suffit d'en écrire une seule et de lui passer une autre fonction comme paramètre, cette fonction est appelée callback function.

Exemples de fonctions d'ordre supérieur :

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

function subtract(a,b){
   return a-b;
}

function math(a,b,callback){
   return callback(a,b);
```

```
}

math(1,4,add); // returns 5
math(5,5,subtract); // returns 0
```

Application

Écrire une fonction each qui accepte 2 paramètres: un tableau et une callback function. La fonction each doit boucler sur le tableau et appliquer la callback function sur chacun de ses éléments.

```
function each(){
   // ici le code
}
each([1,2,3,4], function(val){
   console.log(val);
});
// affichage obtenu

// 1
// 2
// 3
// 4

each([1,2,3,4], function(val){
   console.log(val*2);
});

// Affichage obtenu
// 2
// 4
// 6
// 8
```

Solution: la fonction each est:

```
function each(array, fn){
   for(var i=0; i< array.length; i++){
      fn(array[i]);
   }
}
```

Les functions setTimeout + setInterval, vues précédemment sont des fonctions d'ordre supérieur.

3. Itérateurs JavaScript

Ce sont des méthodes prédéfinies en JS pour le traitement des tableaux, utilisées pour optimiser le code (en taille et en facilité de lecture). Nous verrons les itérateurs : *forEach, map, filter, reduce.*

forEach

Son premier paramètre est une fonction en callback ayant au maximum 3 arguments : *value, index* et *array*. forEach retourne toujours *undefined*

Voici des exemples:

```
var arr = [4,3,2,1];
arr.forEach(function(val,index,arr){
   console.log(val);
});

// 4
// 3
// 2
// 1

arr.forEach(function(val,index,arr){
   console.log(index);
});

// 0
// 1
// 2
// 3
var doubledValues = arr.forEach(function(val,index,arr){
   return val*2;
});

doubledValues; // undefined
```

Les 3 arguments ne sont pas nécessaires :

```
var arr = [4,3,2,1];
// donne les memes resultats que précedemment
arr.forEach(function(val){
   console.log(val);
});
```

map

Contrairement à forEach, map retourne un tableau des valeurs retournées dans le callback. La fonction en callback est la même que pour forEach: *value, index* et *array* (dans cet ordre).

```
var arr = [1,2,3,4];
arr.map(function(val, index, array){
   return val * 2;
}); // [2,4,6,8]

var tripledValues = arr.map(function(val,index,arr){
   return val*3;
});
```

```
tripledValues; // [3,6,9,12]

// une autre facon d'utiliser map pour doubler les valeurs du tableau
function doubleArray(arr){
   // return the result of arr.map
   return arr.map(function(val){
      // return a new array with each value doubled
      return val *2;
   });
}

doubleArray([2,4]); // [4,8]
```

filter

filter retourne un tableau comme map, mais dans la fonction callback, on doit retourner une expression logique qui donne true ou false (pour filtrer les éléments du tableau).

Exemple :

```
var arr = [1,2,3,4];
var valuesGreaterThanTwo = arr.filter(function(val){
   return val > 2;
});

valuesGreaterThanTwo // [3,4]
```

reduce

Le but de reduce est de réduire un tableau à une valeur unique de n'importe quel type (string, boolean, object, array etc). Les argument de la fonction callback sont 4: les 3 premiers sont identiques à ceux de map ou forEach, ils ne sont pas tous obligatoires, le 4ème appelé start, previous, est une accumulation des résultats sur les différentes itérations.

Exemples : (snip 8)

```
var arr = [2,4,6,8];
arr.reduce(function(acc,next){
   return acc + next;
},5);

/*
premiere iteration, acc = 5 et next = 2; le callback retourne 5 + 2 = 7.
Seconde iteration, acc = 7 et next = 4; le callback retourne 7 + 4 = 11.
troisième iteration, acc = 11 et next = 6;le callback retourne 11 + 6 = 17

Dernière iteration, acc = 17 et next = 8;le callback retourne 17 + 8 = 25.
reduce returne 25 (somme de tous les elements du tableau, plus 5)
*/

var arr = [2,4,6,8];
arr.reduce(function(acc,next){
   return acc + next;
});
```

4. Le mot-clé 'this'

this est un mot-clé en JavaScript qui indique un objet. Sur la console Chrome, si on tape this, le résultat est l'objet window. Mais en général, la valeur indiquée par le mot-clé this dépend de l'endroit où il est utilisé dans le code.

Les 4 modes d'utilisations possibles du mot-clé "this"

i)default binding/contexte global,
ii) implicit binding,
iii) explicit binding,
iv) new binding

1) Default Binding/ Contexte Global

Dans le contexte global, this fait référence à l'objet window.

```
var thing = this;
thing; // window

function outer() {
   return this;
}

outer(); // window
```

Pour les autres modes, la valeur de this a une autre valeur quand il est utilisé à l'intérieur d'une fonction. Sa valeur est attribuée en fonction du lieu et de la façon dont la fonction est appelée.

2) Implicit/Object

Dans un objet, this indique l'objet parent le plus proche.

Exemple snp 4:

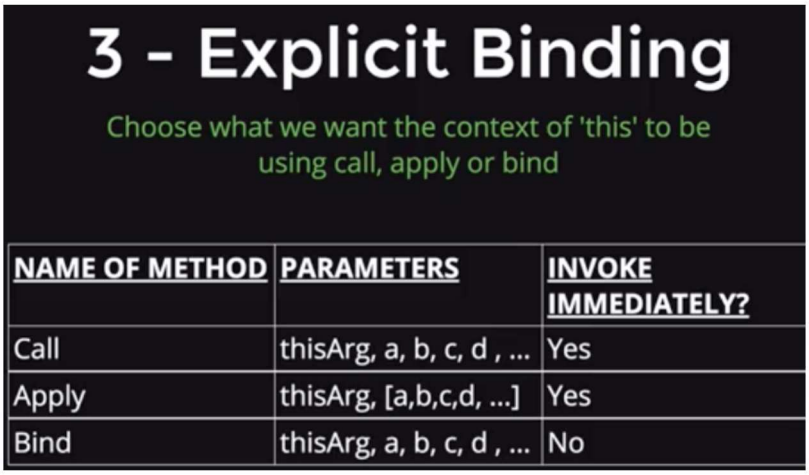
```
var person = {
        firstName: "Omar",
        sayHi : function (){
                return "Hi " + this.firstName
        },
        determineContext: function() {
                return this===person;
        },
        dog: {
        sayHello : function (){
                return "Hi " + this.firstName
        },
        determineContext: function() {
                return this===person;
        },
        }
}
```

```
person.sayHi(); // Hi Omar
person.determineContext(); // true

person.dog.sayHello(); // Hi undefined
person.dog.determineContext(); // false
```

3) Explicit binding

Dans ce cas, c'est l'utilisateur qui indique à quoi correspond this, en utilisant 3 méthodes : *call, apply ou bind*.



NAME OF METHOD	PARAMETERS	INVOKE IMMEDIATELY?
Call	thisArg, a, b, c, d , ...	Yes
Apply	thisArg, [a,b,c,d, ...]	Yes
Bind	thisArg, a, b, c, d , ...	No

- La méthode call:

Exemple sans la méthode call

```
var Omar = {
        firstName: "Omar",
        sayHi : function (){
                return "Hi " + this.firstName
        }
}

var ali = {
        firstName: "Ali",
        // duplication du code précédent
        sayHi : function (){
                return "Hi " + this.firstName
        }
}

Omar.sayHi(); // Hi Omar
ali.sayHi(); // Hi Ali
```

Même exemple avec la méthode call (évite la répétition du code)

```
var Omar = {
        firstName: "Omar",
        sayHi : function (){
                return "Hi " + this.firstName
        }
}

var ali = {
        firstName: "Ali",
}

Omar.sayHi(); // Hi Omar
Omar.sayHi.call(ali); // Hi Ali
```

- La méthode Apply :

Exemple sans apply (snippet 6)

```
var Omar = {
        firstName: "Omar",
        sayHi : function (){
                return "Hi " + this.firstName
        },
        addNumbers : function (a,b,c,d) {
                return this.firstName + "just calculated " + (a+b+c+d);
        }
}

var ali = {
        firstName: "Ali",
}

Omar.sayHi(); // Hi Omar
Omar.sayHi.call(ali); // Hi Ali
Omar.addNumbers(1,2,3,4); //Omar just calculated 10

//pour faire la meme chose avec Ali, on peut utiliser call
Omar.addNumbers.call(ali, 1,2,3,4);  //Ali just calculated 10

//on utilise apply

Omar.addNumbers.apply(ali,[1,2,3,4]); //Ali just calculated 10
```

- La méthode bind

Les paramètres fonctionnent comme pour call mais bind retourne une fonction dans laquelle le contexte de « this » est déjà précisé.

Exemple (snippet 7)

```
var Omar = {
        firstName: "Omar",
```

```
		sayHi : function (){
			return "Hi " + this.firstName
		},
		addNumbers : function (a,b,c,d) {
			return this.firstName + "just calculated " + (a+b+c+d);
		}
}

var ali = {
		firstName: "Ali",
}

var aliCalc = Omar.addNumbers.bind(ali, 1,2,3,4); // function(){…}
aliCalc(); //Ali just calculated 10


//avec bind, on n'a pas besoin de spécifier  tous les paramètres à l'avance

var aliCalc2 = Omar.addNumbers.bind(ali, 1,2); // function(){…}
aliCalc(3,4); //Ali just calculated 10
```

- Avec setTimeout, 'this' représente l'objet global window.

Snippet 8.

```
var Omar = {
		firstName: "Omar",
		sayHi : function (){
			seTimeout(function(){
			console.log("Hi " + this.firstName)
		},1000)
		}
}


Omar.sayHi(); // Hi undefined —after 1000 ms


// use bind to set the correct context of this
var Omar = {
		firstName: "Omar",
		sayHi : function (){
			seTimeout(function(){
			console.log("Hi " + this.firstName)
		}.bind(this),1000)
		}
}

Omar.sayHi();
```

4) Le mot-clé new

Utilisé comme pour le constructeur d'une classe avec les langages orientés objet.

/* la partie qui suit est remplacée par la notion de classe en ES6 dans la partie 2 du chapitre 2 */

5. JavaScript et l'Orientée Objet

JavaScript n'est pas un langage OO à l'origine. On lui a ajouté des caractéristiques qui le rendent semblable à un langage OO (comme Java ou C++), à savoir : l'existence des classes, le constructeur d'une classe pour créer les objets instances et l'héritage.

5.1 le mot-clé *new*

new permet de créer des instances de classes d'objets basées sur un constructeur qui est une fonction particulière en JS. L'utilisation de new et du constructeur permet de réduire le code et évite la répétition de ce dernier.
Dans l'exemple suivant, il y a beaucoup de répétition de code :

```
var car1 = {
   make: "Honda",
   model: "Accord",
   year: 2002
}
var car2 = {
   make: "Mazda",
   model: "6",
   year: 2008
}
var car3 = {
   make: "BMW",
   model: "7 Series",
   year: 2012
}
var car4 = {
   make: "Tesla",
   model: "Model X",
   year: 2016
}
```

Solution à la duplication du code :

5.2 Fonction constructeur

C'est une fonction JS ordinaire, elle doit commencer par convention par une lettre majuscule (bonne pratique) pour indiquer que c'est une fonction constructeur.

Snippet 9

```
function Car(make, model, year){
   this.make = make;
   this.model = model;
   this.year = year;
}
```

Pour construire des objets instances, on utilise new

```
var probe = new Car('Ford', 'Probe', 1993);
var cmax = new Car('Ford', 'C-Max', 2014);

probe.make;  // Returns "Ford"
cmax.year;
// Returns 2014
```

Rôle du mot-clé ***new*** :

- Création d'un objet vide qui est référencé par « *this* » dans la fonction constructeur
- Ajout automatique de 'return this' à la fin de la fonction constructeur (on n'a pas besoin de retourner explicitement une valeur).

5.3 Classes

En JavaScript le concept de classe n'existait pas, il a été "imité" pour ressembler aux classes en OO (existe dans les versions ultérieures de JavaScript), en utilisant les fonctions constructeur et invoqués par new.

L'héritage n'est pas non plus explicite mais il est assuré par la notion de *prototype* (pour les attributs et les méthodes statiques) pour emprunter les propriétés d'un constructeur et les utiliser dans un autre. On peut créer un héritage par utilisation de call.
Exemple :

```
function Vehicle(make,model,year){
   this.make = make;
   this.model = model;
   this.year = year;
}

function Motorcycle(make,model,year,motorcycleType){
   Vehicle.call(this,make,model,year)
   this.motorcycleType = motorcycleType;
}

var moto = new Motorcycle("Kawasaki", "Ninja 500", 2006, "Sports")
```