

Programmation Web Avancée

Cours 2

Objets

Portée des variables

Tableaux

Rappels MVC

kn@lri.fr

1 Introduction/ Généralité et rappels sur le Web/ Javascript : survol du langage



2 Objets/Portée des variables/Tableaux/Rappels MVC

2.1 Objets

2.2 Portée

2.3 Tableaux

2.4 MVC

Principes de la programmation orientée objet

Un général, un **langage orienté objet statiquement typé** propose une notion de **classe** et **d'objet**. Par exemple en Java :

```
class Point {  
    private int x;  
    private int y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void move(int i, int j) {  
        this.x += i;  
        this.y += j;  
    }  
    public void getX() { return this.x; }  
    public void getY() { return this.y; }  
}
```

Une **classe** définit un ensemble **d'objets** contenant un état interne (les attributs : x, y) ainsi que du code (les méthodes : move, ...) permettant de manipuler cet état.

Un objet est **l'instance** d'une classe.

Concepts objets

Les langages orientés objets exposent généralement plusieurs concepts :

- La notion de **constructeur** : une fonction ayant un statut spécial, qui est appelé pour **initialiser** l'état interne de l'objet à sa création.
- Une notion de **contrôle d'accès** aux attributs et méthodes.
- Un moyen de **référencer** l'objet dans lequel on se trouve (**this**).
- Plusieurs notions permettant de partager du code ou des données (**static**, héritage, ...)

Objets en Javascript

Rappel : Javascript ne fait pas de différences entre « **attributs** » et « **méthodes** ». Les « champs » d'un objet sont appelés « **propriétés** ». Elles peuvent contenir des valeurs scalaires ou des fonctions.

Rappel : l'affectation à une propriété en Javascript **ajoute** la propriété si elle était inexistante :

```
var p1 = { };           //Un objet vide
p1.x = 0;               //On ajoute un champ x initialisé à 0
p1.y = 0;               //On ajoute un champ y initialisé à 0

//Ajout de « méthodes » move, getX et getY
p1.move = function (i, j) { p1.x += i; p1.y += j; };

p1.getX = function () { return p1.x; };
p1.getY = function () { return p1.y; };
```

Quels sont les problèmes avec le code ci-dessus ?

1. Il faut copier-coller **tout** le code si on veut créer un autre point p2
2. Pour chaque objet p_i , on va allouer 3 fonctions **différentes** (qui font la **même** chose pour l'objet considéré.)

Première solution

On englobe le tout dans une **fonction** :

```
var mkPoint = function (x, y) {  
  var p = { };  
  p.x = x;  
  p.y = y;  
  p.move = function (i, j) { p.x += i; p.y += j; };  
  p.getX = function () { return p.x; };  
  p.getY = function () { return p.y; };  
  return p;  
};  
...  
var p1 = mkPoint(1,1);  
var p2 = mkPoint(2, 10);  
var p3 = mkPoint(3.14, -25e10);  
...
```

La fonction **mkPoint** fonctionne comme un **constructeur**. Cependant, les trois « méthodes » sont allouées **à chaque fois**.

Function, prototype et new

En Javascript, le type « Function » (des fonctions) a un statut particulier. Lorsque l'on appelle l'expression `new f(e1, ..., en)` :

1. Un nouvel objet `o` (vide) est créé.
2. Le champ `prototype` de `f` est copié dans le champ `prototype` de `o`.
3. `f(e1, ..., en)` est évalué et l'identifiant spécial `this` est associé à `o`
4. Si `f` renvoie un **objet**, alors cet objet est le résultat de `new f(e1, ..., en)`, sinon l'objet `o` est renvoyé.

L'expression `new e` où `e` n'est pas un appel de fonction provoque une **erreur**.

Comment créer des objets avec ça ?

Function, prototype et new (suite)

```
var Point = function (x, y) {  
    this.x = x;  
    this.y = y;  
};  
...  
var p1 = new Point(1,1);  
var p2 = new Point(2, 10);  
var p3 = new Point(3.14, -25e10);  
...
```

1. Un nouvel objet **p₁** (vide) est créé.
2. Le champ **prototype** de **Point** est copié dans le champ **prototype** de **p₁**.
3. **Point(e₁, ..., e_n)** est évalué et l'identifiant spécial **this** est associé à **p₁**
4. Si **Point** renvoie un **objet**, alors cet objet est le résultat de **new Point(e₁, ..., e_n)**, sinon l'objet **p₁** est renvoyé.

prototype et les propriétés propres

La **résolution de propriété** en Javascript suit l'algorithme ci-dessous. Pour rechercher la propriété p sur un objet o :

```
soit  $x \leftarrow o$ ;  
répéter:  
  si  $x.p$  est défini alors renvoyer  $x.p$ ;  
  si  $x.prototype$  est défini, différent de null et est un objet,  
  alors  $x \leftarrow x.prototype$ 
```

Une propriété p d'un objet o peut donc être :

- Soit rattachée à o lui-même (on dit que p est une **propriété propre** de o , *own property*)
- Soit rattachée à l'objet o_1 se trouvant dans le champ `prototype` de o (s'il existe)
- Soit rattachée à l'objet o_2 se trouvant dans le champ `prototype` de o_1 (s'il existe)

■ ...

Function, prototype et new (fin)

```
var Point = function (x, y) {  
    this.x = x;  
    this.y = y;  
};  
Point.prototype.move = function (i, j) {  
    this.x+= i;  
    this.y+= j;  
};  
Point.prototype.getX = function () {  
    return this.x;  
};  
Point.prototype.getY = function () {  
    return this.y;  
};  
...  
var p1 = new Point(1,1);  
p1.move(2, 10);  
...
```

Lors de l'appel à `move` l'objet `p1` ne possède pas directement de propriété `move`. La propriété `move` est donc cherchée (et trouvée) dans son champ `prototype`.

Parallèle avec les langages compilés

Le fonctionnement par **prototype** est identique à la manière dont les langages OO statiquement typés (Java, C++, C#) sont **compilés**.

En Java, chaque objet contient un pointeur (caché) vers un **descripteur de classe** (une structure contenant les adresses de toutes les méthodes de la classe + un pointeur vers le descripteur de la classe parente) \equiv **prototype**.


Qu'offre Javascript en plus ?

- Redéfinition **locale** de propriétés (**monkey patching**)
- Définition manuelle du prototype pour « hériter » d'un type existant

Monkey patching

Technique qui consiste à redéfinir une méthode sur un **objet spécifique** (impossible à faire en Java).

```
var p1 = new Point(1, 1);  
var p2 = new Point(1, 1);  
  
p2.move = function () { this.x = 0; this.y = 0;};  
p1.move(10, 10);    //appelle Point.prototype.move  
p2.move(10, 10);    //appelle la méthode move définie ci-dessus  
  
var x1 = p1.getX(); //x1 contient 11  
var x2 = p2.getX(); //x2 contient 0
```

 : c'est une technique dangereuse, car elle donne un comportement **non-uniforme** à des objets du même « type ». On l'utilisera à des fins de débuggages, jamais pour spécialiser durablement le type d'un objet (et encore moins d'un objet système tel que Math).

Différence entre propriété propre et prototype

On peut savoir à tout moment si un objet **o** a une propriété **p** propre en utilisant la méthode **.hasOwnProperty(...)**

```
var p = new Point(1, 2);  
p.hasOwnProperty('x');      // renvoie true  
p.hasOwnProperty('move');   // renvoie false
```

« Héritage »

L'algorithme de résolution de propriété peut être utilisé pour simuler l'héritage.

```
var ColoredPoint = function (x, y, c) {  
    Point.call(this, x, y);    //appel du constructeur parent  
  
    this.color = c || "black";    //si c est convertible en false (en particulier  
                                //undefined), on initialise à black  
  
};  
  
ColoredPoint.prototype = Object.create(Point.prototype);  
//Object.create crée un nouvel objet dont le champ prototype est  
//une copie de celui passé en argument.  
  
ColoredPoint.prototype.getColor = function () { return this.color; };  
  
var p = new ColoredPoint(1, 2, "red");  
p.move(10, 10);    //move se trouve dans ColoredPoint.prototype.prototype !  
p.getColor();    //getColor se trouve dans ColoredPoint.prototype !
```

1 Introduction/ Généralité et rappels sur le Web/ Javascript : survol du langage



2 Objets/Portée des variables/Tableaux/Rappels MVC

2.1 Objets ✓

2.2 Portée

2.3 Tableaux

2.4 MVC

Objet Global et variables globales

La norme Javascript (ECMA-262) définit un **Objet Global** initialisé avant le début du programme.

Les variables globales en Javascript ne sont que des **propriétés propres** de cet objet.

Dans les navigateurs Web, cet objet global représente l'«onglet courant». Il possède une propriété **window** qui pointe sur lui même.

```
//On suppose que l'on est dans un fichier test.js inclus directement  
//dans la page
```

```
var foo = 123;      // variable « globale »  
bar = 456;          // assigne la propriété bar à l'objet global  
this.baz = 789;     // idem mais avec un this explicite
```

```
window.foo;        // vaut 123  
window.bar;        // vaut 456  
window.baz;        // vaut 789
```


Variables locales

En Javascript, la seule construction pouvant introduire une nouvelle portée (*scope*) est `function (...) { }`.

Une variable déclarée (au moyen de `var`) dans une fonction est **locale** à cette fonction.



les « blocs » ne créent pas de nouvelle portée ! :

```
//On suppose que l'on est dans un fichier test.js inclus directement
//dans la page
for(var i = 0; i < 10; i++) {
    var j = 2 * i;
    ...
}
i;           //bien défini après la boucle, vaut 9
j;           //bien défini après la boucle, vaut 18

var f = function () {
    var k = 20;
    ...
};
k;           //vaut undefined
```

Shadowing

Une variable peut masquer une variable de même nom se trouvant dans une portée englobante:

```
//On suppose que l'on est dans un fichier test.js inclus directement
//dans la page
var x = 123;
console.log(x);                // affiche 123

function f () {
    var x = 456;
    function g () {
        var x = 789;
        console.log(x);       // affiche 789 quand g est appelée
    };
    g ();
    console.log(x);            // affiche 456 quand f est appelée
};
f ();
console.log(x);                // affiche 123
```

Hoisting des déclarations

Les déclarations de variables **locale** sont déplacées (*hoisted*) en début de portée.
Ainsi :

```
function f () {  
  console.log('Hello !');  
  var x = 23;  
  ...  
};
```

est équivalent à :

```
function f () {  
  var x;  
  console.log('Hello !');  
  x = 23;  
  ...  
};
```

Identifiant this

L'identifiant `this` est similaire à celui de Java **mais** est **tout le temps défini**, avec des règles précises :

`o.f(e1, ..., en)` : `this` est initialisé à `o`

`new f(e1, ..., en)` : `this` est initialisé à **un objet fraîchement créé**.

`f.call(o, e1, ..., en)` : `this` est initialisé à `o`.

`f(e1, ..., en)` :  `this` est initialisé à **l'objet global** (`window`).

```
ColoredPoint.prototype.getColor = function () {  
    console.log(this.color);           //accès à la couleur  
    var g = function () {  
        console.log(this.color);       //undefined car équivalent à  
                                        //window.color  
    };  
    g();  
    return this.color;  
};
```

Encapsulation

Javascript ne disposant ni de **modules**, ni de **namespace**, ni de **classes**, ni de **packages**, il est possible rapidement « poluer » l'objet global :

```
//Définition de Point, move, getX, ...
var Point = function (x, y) {
    ...
};
...
var strAux (x, y) {          //on définit une fonction auxiliaire
    return "(" + x + ", " + y + ")";
};

Point.prototype.toString = function () {
    return strAux(x, y);
};

window.strAux (...);          //est défini
strAux(...);                  //est défini
```

Problème : une fonction auxiliaire qui n'est utile qu'à la classe Point est visible globalement.

Utilisation de fonctions pour encapsuler

Comme les fonctions introduisent une nouvelle portée, on peut s'en servir pour encapsuler le code

```
//Définition de Point, move, getX, ...
var Point = function (x, y) {
    ...
};
...
Point.prototype = (function () {
    //On est maintenant dans une fonction, strAux ne pourra pas
    //s'échapper dans le contexte global

    var strAux (x, y) {
        return "(" + x + ", " + y + ")";
    };
    //On renvoie un objet faisant office de prototype :
    return {
        move : function (i, j) { ... },
        getX : function () { return this.x; },
        getY : function () { return this.y; },
        toString : function () { return strAux(x,y); }
    };
})(); //On applique immédiatement la fonction !
```

```
strAux( ).
```

```
//undefined
```

1 Introduction/ Généralité et rappels sur le Web/ Javascript : survol du langage



2 Objets/Portée des variables/Tableaux/Rappels MVC

2.1 Objets ✓

2.2 Portée ✓

2.3 Tableaux

2.4 MVC

Array

Les tableaux (classe **Array**) font partie de la bibliothèque standard Javascript. On peut créer un tableau vide avec **[]**.

```
var tab = [];  
tab[35];           //undefined  
tab[35] = "Hello"; //initialise la case 35 à "Hello"  
tab[0];           //toujours undefined;  
tab.length;       //36 ! indice le plus grand ayant été  
                  //initialisé + 1
```


Array

- `new Array(n)` : Initialise un tableau de taille n (indiqué de 0 à $n-1$) où toutes les cases valent `undefined`
- `.length` : renvoie la longueur du tableau
- `.toString()` : applique `.toString()` à chaque élément et renvoie la concaténation
- `.push(e)` : ajoute un élément en fin de tableau
- `.pop()` : retire et renvoie le dernier élément du tableau. **undefined** si le tableau est vide
- `.shift()` : retire et renvoie le premier élément du tableau. **undefined** si le tableau est vide
- `.unshift(e)` : ajoute un élément au début du tableau
- `.splice(i, n, e1, ..., ek)` : à partir de l'indice i , efface les éléments i à $i+n-1$ et insère les éléments e_1, \dots, e_k
- `.forEach(f)` : Applique la fonction f à tous les éléments du tableau qui ne valent pas **undefined**. f reçoit trois arguments (v, i, t) :
 - v : la valeur courante de la case visitée
 - i : l'indice courant (à partir de 0)
 - t : le tableau en entier

1 Introduction/ Généralité et rappels sur le Web/ Javascript : survol du langage



2 Objets/Portée des variables/Tableaux/Rappels MVC

2.1 Objets ✓

2.2 Portée ✓

2.3 Tableaux ✓

2.4 MVC

Qu'est-ce que le modèle MVC ?

C'est un *design pattern* qui permet de modéliser des applications « interactives » :

- L'application possède un état interne
- Un « utilisateur » (ça peut être un programme externe) interagit avec le programme pour modifier l'état interne
- L'application affiche à l'utilisateur le résultat de son opération

Ces trois aspects sont représentés par trois composants :

- Le **Modèle** (représentation de l'état interne)
- La **Vue** (affichage du modèle)
- Le **Contrôleur** (modification du modèle)

En quoi est-ce adapté aux applications Web ?

Une application Web typique :

- Présente au client un formulaire permettant de passer des paramètres (C)
- Effectue des opérations sur une base de donnée (M) à partir des paramètres
- Affiche une page Web montrant le résultat de l'opération (V)

Avantages du Modèle MVC ?

La **séparation** permet d'obtenir :

Maintenance simplifiée : Le code d'une action est centralisé à un seul endroit

Séparation des privilèges : Pas besoin que la vue ai un accès à la base de donnée
par exemple

Test simplifié : Les composants peuvent être testés indépendamment