Programmation Web Client

Benjamin Canou, OCaml PRO

24 octobre 2016

Développement d'Applications Réticulaires

Plan de ce cours

Cette semaine

- Historique & caractéristiques
- Premiers pas et langage de base
- Structures de données et modèle objet

La semaine prochaine

- Modifications dynamiques de la page
- Gestion d'évènements

Petit historique

- Les débuts :
  - Introduit dans Netscape en 1995
  - Conçu comme un petit langage de script pour compléter Java
  - Inplantation partiellement compatible par Microsoft
- Évolution et adoption :
  - Longue histoire d'incompatibilités
  - Tentatives d'évolution / harminisation abandonnées
  - Normalisation sous le nom d'ECMAScript
  - Performances originellement très mauvaises, et très variables
- Aujourd'hui :
  - Bientôt ECMAScript 6 (noté ES6 dans ce cours)
  - Performances raisonables, encore variables
  - Encore des variations entre les bibliothèques
  - Sorti du navigateur : Node.JS

Caractéristiques principales

- Langage principalement impératif
- Traits fonctionnels (principalement pour fonctions de rappel)
- Modèle objet inhabituel, très flexible
- Typage dynamique très laxiste (conversions et valeurs implicites)
- Typage usuel pour les objets : typage canard (structurel dynamique)
- Portée lexicale un peu inhabituelle + portée globale

Comment programmer en JavaScript ?

- Dans un fichier HTML(5) :

```
1 : <html>
2 :   <head>
3 :     <meta charset="utf-8">
4 :     <script language="JavaScript">
5 :       // code JavaScript
6 :     </script>
7 :   </head>
8 :   <body></body>
9 : </html>
```

- Dans un fichier JS appelé par un fichier HTML :

```
1 : <script language="JavaScript" src="code.js"></script>
```

- Dans la console du navigateur
- Dans un interprète indépendant : `#!/usr/bin/env node`

- Bonjour tout le monde

```
1 : alert ("Bonjour") ;
```

- Bonjour madame

```
1 : var name = prompt ("Entrez votre nom") ;
2 : alert (`Bonjour " + name) ;
3 : 
```

Syntaxe de base

À la C / Java :

- Blocs : `{ ... }` (attention, pas des blocs lexicaux)
- Expressions, operateurs, etc.
- Affectations : `x = expr`
- Contrôle : `for (;;;){}`, `while (){}`, `if(){}`, etc.

Fonctions :

- Appel de fonction : `f (x, y)`
- Définition de fonction :

```
1 : function f(x, y) {
2 :   return (x + y) ;
3 : }
```

- Paramètres optionnels ES6 : `function f(x = 3, y = 4){ ... }`

Types de base :

- Nombres (pas de différence flottant / entier)
- Chaînes (non mutables), expressions rationnelles
- Expressions rationnelles
- Chaînes de format ES6
- Tableaux extensibles / creux
- Matrices numériques ES6
- Générateurs ES6
- Objets : tableaux associatifs + prototypes
- Null
- Undefined

Tableaux :

- Creation : `[]`, `var t = []`
- Initialisation : `[ 1, 2, 4, 8, 16 ]`
- Lecture : `t[index]` ($0 \leq$ index $<$ t.length)
- Affectation : `t[index] = expr`
- Itération (indices) : `for (index in tab) { ... }`
- Itération (valeurs) : `for (val of tab) { ... }` ES6
- Déconstruction : `var [x, y] = [1, 2]` ES6

Objets (tableaux associatifs) :

- Creation : `{}`, `var t = {}`
- Initialisation : `{ x : 12, y : 23 }`
- Lecture : `t.x`, `t["x"]`
- Affectation : `t.x = expr`, `t["x"] = expr`
- Suppression : `delete t.x`
- Itération (clés) : `for (key in obj) { ... }`
- Itération (valeurs) : `for (val of obj) { ... }` ES6
- Déconstruction : `var {x, y} = {x: 1, y: 2}` ES6

Méthodes :

- Appel : `o.m (x, y)` (cf. la suite du cours)
- Définition :

```
1 : function f (x, y) { return (x + y); } ;
2 : obj = {} ;
3 : obj.m = f ;
4 : obj.m (2, 3) ; // pour appeler
```

- Définition (alternative) :

```
1 : obj = { f: function (x, y) { return (x + y); } }
```

Portée globale dynamique :

- Variables définies partout dès leur première affectation
- Masquées par les variables locales
- Toujours accessibles par `window.variable`

Exemple :

```
1 : function f() { alert (xx); }
2 : f() ; // affiche undefined ou lève une erreur
3 : xx = 3;
4 : f() ; // affiche 3
```

Portée locale lexicale :

- Mot-clef `var` pour définir une locale
- Portée : la fonction courante
- Par défaut, les variables sont globales !

Exemple :

```
1 : function f() {
2 :   x = 3 ; // référence la globale x
3 :   var y = 2 ; // crée une nouvelle locale y
4 : }
5 : f() ;
6 : // la globale x est affectée
```

Fonctions anonymes :

```
1 : var f = function (x) { return x; }
2 : (function (x, y) { return (x + y); }) ()
```

Fermetures :

```
1 : function f (x) {
2 :   var y = ... ;
3 :   return (function () {
4 :     return (x + y) ;
5 :     // x et y de la fonction parente
6 :     // sont enregistrées dans la fermeture
7 :   }) ;
8 : }
```

Attention ! les variables de l'environnement ne sont pas figées.

```
1 : function what () {
2 :   var fs = [] ;
3 :   for (var i = 0; i < 10; i++) {
4 :     fs[i] = function () { alert (i) ; }
5 :   }
6 :   fs[4]() ;
7 : }
```

Quel est le résultat ?

Variables spéciales :

- `this` : dans le code d'une méthoode
- `arguments` : dans le code d'une fonction
  - tableau des arguments,
  - utile au cas où la fonction est appelée avec trop de paramètres,
  - sert à implanter des arguments optionnels.

Lambda expressions ES6 :

```
1 : x => x
2 : (x, y) => (x + y)
3 : (x, y) => { return (x + y); }
```

Un troisième forme de portée ES6 :

```
1 : { let x = (...) ; /* portée de x */ }
2 : for (let i = ...) { /* portée de i */ }
```

Restriction plus classique au bloc local.

Définition de générateur ES6 :

```
1 : function range (min, max) {
2 :   for (let i = min ; i <= max ; i++) { yield i ; }
3 : }
```

Utilisation ES6 :

```
1 : for (i of range (4, 8)) { console.log (i)) ; }
2 : let t = [ ...range (4, 8) ] ;
```

Modèle d'héritage par prototypes :

- Chaque objet a un champ caché `__prototype__` ,
- ce champ est lui-même un objet,
- si un champ `o.m` n'est pas trouvé, `o.__prototype__.m` est recherché,
- et récursivement,
- jusqu'à la fin de la chaine de prototypes : `Object` .

Le prototype d'un objet :

- Est fixé à sa création,
- dépend de son constructeur (cf. suite),
- peut être modifié dynamiquement.

Définition de constructeur personnalisé :

- Mot-clef `function` pour définir une fonction de construction,
- mot-clef `new` devant l'appel de fonction,
- crée un nouvel objet (`this` dans le corps du constructeur),
- le prototype des futurs objets est accessible via le champ `prototype` du cosntructeur.

Exemple :

```
1 : function Vec(u, v) {
2 :   this.u = u ;
3 :   this.v = v ;
4 : }
5 : Vec.prototype.norm = function () {
6 :   return Math.sqrt (this.u * this.u + this.v * this.v) ;
7 : }
8 :
9 : v = new Vec (2, 3) ;
10 : alert (v.norm()) ;
```

Chaînage (héritage) de prototypes :

```
1 : function ColVec(u, v, c) {
2 :   Vec.call (this, u, v) ;
3 :   this.c = c;
4 : }
5 : ColVec.prototype = new Vec();
6 :
7 : v = new Vec (2, 3) ;
8 : alert (v.norm()) ;
```

(version ancestrale, exercice : récrire avec `Object.create`)

Sucre syntaxique pour simuler les classes ES6 :

```
1 : class Vec {
2 :   constructor (u, v) {
3 :     this.u = u ;
4 :     this.v = v ;
5 :   }
6 :   norm () {
7 :     return Math.sqrt (this.u * this.u + this.v * this.v) ;
8 :   }
9 : }
10 : class ColVec extends Vec {
11 :   constructor (u, v, c) {
12 :     super (u, v);
13 :     this.c = c ;
14 :   }
15 : }
```

Destruction : les regexps

```
1 : var [ _, lhs, rhs] =
2 :   /^\(([0-9]+), ([0-9]+)\)$/
3 :   .exec("(3, 4)")
```

Construction : les templates ES6

```
1 : var age = 3, nom = 'bob' ;
2 : alert (`mon nom est ${nom}, j'ai ${age} ans !`)
3 : ␣␣
```

Programmation Web Client

Questions ?

Développement d'Applications Réticulaires