

Javascript

Michel Gagnon

École polytechnique de Montréal



Types

- Javascript n'a que six types de données:
 - nombre (contient la valeur spéciale **NaN**)
 - **string**
 - booléen (**true** et **false**)
 - objet
 - **null**
 - **undefined**

Conversion de type

- Dans certains contextes, Javascript effectue des conversions automatiques
- Conversion vers un **string**:
 - Si l'objet a une méthode **toString()**, elle est exécutée
 - Si la méthode **valueOf()** existe, elle est exécutée
 - Sinon une exception est lancée
- Conversion vers une valeur numérique:
 - Si la méthode **valueOf()** existe, elle est exécutée
 - Si l'objet a une méthode **toString()**, elle est exécutée
 - Sinon une exception est lancée
- On peut convertir explicitement: `String(obj)`, `Number(obj)`

Conversion de type (suite)

- Opérateurs non-strict d'égalité (==), < et >:
 - Des conversions implicites sont effectuées
 - Pas de conversion quand on a deux objets: ils doivent référer au même objet
- Contexte booléen (**while(...)**, **if(...)**, etc):
 - **false** → false, **true** → true
 - **undefined** → false, **null** → false
 - **0** → false **NaN** → false
 - **""** → false
 - Objet → true
 - Autres cas → true

Fonctions

- Trois manières de créer une fonction:
 - Déclaration
 - Expression de fonction (fonction anonyme)
 - Par l'appel de **new Function**
- Une fonction retourne toujours une valeur (par défaut c'est la valeur **undefined**)
- Une variable déclarée avec le mot-clé **var** est une variable locale
- Les fonctions déclarées sont parsées avant l'exécution du script (on peut donc l'appeler avant sa définition)
- Une fonction est un objet qu'on peut manipuler comme tout autre objet

Expression de fonction

- Partout où on peut mettre une valeur (un objet par exemple), on peut mettre une expression de fonction

```
var increment = function(x) { return x + 1; };  
x = increment(3);
```

- *Attention:* dans ce cas, la fonction n'est pas créée avant l'exécution du script

Traitement d'un script

- Toutes les variables locales et les fonctions sont des propriétés d'un objet interne **LexicalEnvironment**
- L'environnement global pour le script est **window**
- Le traitement d'un script suit les étapes suivantes:
 1. Traitement des déclarations de fonction, qui sont ajoutées à **window**
 2. Traitement des variables déclarées avec **var**, qui sont elles aussi ajoutées à **window**, mais avec **undefined** comme valeur
 3. Le code est exécuté

Environnement lexical d'une fonction

Voici ce qui se passe quand une fonction est exécutée:

1. Son environnement lexical est créé
2. Son environnement lexical est peuplé par les variables paramètres, les variables locales (celles déclarées avec **var**) et les fonctions imbriquées déclarées
3. Le code est exécuté
4. À la fin de l'exécution l'environnement lexical est détruit (sauf en situation de fermeture, comme nous le verrons plus loin)

Portée

- Les blocs n'ont pas de portée
- Les variables utilisées dans une expression **for** existent encore à la sortie de la boucle

Fermetures

- Quand une variable n'est pas trouvée dans l'environnement lexical d'une fonction, on la cherche dans le premier environnement englobant
- À noter qu'une variable initialisée sans le mot-clé **var** sera toujours placée dans l'environnement lexical global, soit **window**
- Une fonction peut continuer d'exister une fois que l'exécution de sa fonction englobante est terminée (ce sera le cas si on retourne cette fonction)
- Dans ce cas, la fonction conserve un lien vers l'environnement lexical de la fonction englobante

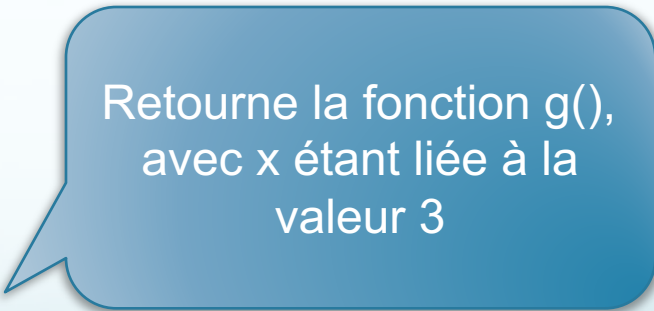
Exemple de fermeture

```
function f(x) {  
    function g(y) {  
        return x + y;  
    };  
    return g;  
}
```

```
var ajouterTrois = f(3);  
ajouterTrois(4);
```

Exemple de fermeture

```
function f(x) {  
    function g(y) {  
        return x + y;  
    };  
    return g;  
}
```



Retourne la fonction g(),
avec x étant liée à la
valeur 3

```
var ajouterTrois = f(3);  
ajouterTrois(4);
```

Autre exemple de fermeture

```
for (x = 0; x < 2; x++){  
  var f = function() {  
    return function(y){return x + y;};  
  }  
  
  if (x == 0) {ajouterTrois = f(); };  
  if (x == 1) {ajouterQuatre = f(); };  
}  
  
console.log(ajouterTrois(3));  
console.log(ajouterQuatre(4));
```

Autre exemple de fermeture

```
for (x = 0; x < 2; x++){  
    var f = function() {  
        return function(y){return x + y;};  
    }  
  
    if (x == 0) {ajouterTrois = f(); };  
    if (x == 1) {ajouterQuatre = f(); };  
}  
  
console.log(ajouterTrois(3));  
console.log(ajouterQuatre(4));
```

La variable x fait partie du même environnement lexical pour les 2 fonctions. La valeur de x est 2, soit celle fixée à la sortie de la boucle.

Objets

- La manière la plus simple de créer un objet en Javascript est de spécifier une liste d'attributs:

```
michel = {  
    nom: "Michel",  
    age: 29  
};
```

```
console.log(michel.nom + " " + michel.age);
```

Objets

- On peut aussi avoir une méthode comme attribut:

```
michel = {  
  nom: "Michel",  
  age: 29,  
  incrementerAge: function(inc) { this.age += inc; },  
  estVieux: function() { return (this.age > 30); }  
};
```

```
michel.incrementerAge(4);  
console.log(michel.estVieux());
```


L'objet **this**

- **this** est dynamique en Javascript: il est identifié lors de *l'exécution* d'une fonction
- Quand il est déclaré comme méthode d'un objet, il représente cet objet
- S'il n'y a pas d'objet, **this** sera alors l'objet **window**
- Lorsqu'on exécute une fonction avec **new**, **this** sera le nouvel objet créé
- L'objet **this** peut être fourni explicitement en paramètre (toute fonction a une méthode **apply()** qui prend en argument l'objet **this** et une liste qui contient tous les paramètres de la fonction)

Prototype

- Supposons maintenant qu'on veut représenter Paul, qui a 46 ans
- Il partage les même deux méthodes, ainsi que les deux attributs de Michel (avec des valeurs différentes)
- On pourra créer Paul en utilisant Michel comme prototype
- Il héritera alors de tous les attributs de Michel
- On pourra changer la valeur de certains attributs, qui seront alors locaux à l'objet qui représente Paul

Prototype

```
// On crée l'objet vide
paul = { };

// Paul héritera des attributs de Michel
paul.__proto__ = michel;

// On redéfinit les attributs locaux de Paul
paul.nom = "Paul";
paul.age = 46;
console.log(paul.nom + " " + paul.age); // Paul 46

// Si l'objet ne possède pas d'attribut local, on
// le cherche dans son prototype
console.log(paul.estVieux()) // true
delete paul.nom
console.log(paul.nom + " " + paul.age); // Michel 46
```

Prototype

- On aimerait bien avoir une manière plus pratique de définir des "classes"
- Pour y arriver, il faut faire appel à des caractéristiques spéciales des fonctions:
 - Toute fonction f possède un attribut spécial `prototype`, qu'on peut faire pointer sur n'importe quel objet
 - Appelons p cet objet
 - L'exécution de l'opération `new f ()` créera un objet qui se verra automatiquement attribuer comme prototype l'objet p
 - Ce type de fonction est appelé **constructeur** et il est d'usage d'utiliser un nom commençant par une majuscule

Prototype

```
function Personne(nom, age) {  
    this.nom = nom;  
    this.age = age;  
}
```

```
Personne.prototype = {  
    ajusterAge: function(inc) { this.age += inc; },  
    estVieux: function() { return (this.age > 30); }  
};
```

```
michel = new Personne("Michel",29);  
paul = new Personne("Paul",46);  
console.log(paul.nom + " " + paul.age); // Paul 46  
console.log(paul.estVieux()) // true  
console.log(michel.nom + " " + michel.age); // Michel 29  
console.log(michel.estVieux()) // false
```