JavaScript

Michel Gagnon et Nikolay Radoev

École Polytechnique de Montréal

Historique

- Crée en 1995 pour le navigateur Netscape Navigator
- Standarisé sous le nom ECMAScript (ES) en 1996
  - Souvent les versions de JS sont référencés par ESX (ex : ES5).
- Entre 1998 et 2005, JScript, le langage de Microsoft domine le Web
  - La création de Mozilla en 2004 et le développement de techniques comme AJAX et des librairies comme jQuery (2006) amènent une renaissance du JS
- En 2008, Chrome est rendu disponible avec un engin très compétitif (V8)
  - ES5 (2009) est la première mise à jour depuis 1999
  - NodeJS (2009) permet d'utiliser JS hors d'un navigateur grace à V8
- Entre 2009 et 2015, beaucoup de travail permet à JS de devenir plus mature et populaire.
  - Les lacunes et les particularités de JS dans les années 2000 lui donnent une réputation mixte auprès la communauté de développeurs logiciels
  - ECMAScript6 (renommé à ES2015) en 2015 améliore grandement JS et est reconnu comme le « JavaScript moderne »

ECMAScript et JavaScript

- Le langage ECMAScript n'existe pas réellement, JavaScript est une implémentation de la spécification ECMA-262
- ES3 – (1999-2009) : quelques amélioration mineures, mais reste stagnant pendant très longtemps
- ES4 – Abandonné : projet ambitieux d'amélioration de JS, mais échoue par manque de collaboration.
- ES5 – 2009 : amène des nouvelles fonctionnalités ainsi que du support pour le format JSON.
  - **Beaucoup** du code JS trouvé sur StackOverflow date de cette époque
- ES2015 (originalement ES6) : le « JavaScript moderne ». Amène la syntaxe de « class », les mots clés « let » et « const », etc.
  - Chaque année suivante amène des améliorations itératives (ES2016, ES2017, etc)
- ES.NEXT : nom dynamique pour le prochain ensemble de modifications à JS

Types

- JavaScript n'a que huit types de données
  - Les types de JS sont « dynamiques » (on peut changer le type d'une variable pendant l'exécution)

- 4 Types principaux:
  - number
    - Chiffre à virgule flottante : $[-(2^{53}-1), (2^{53}-1)]$
    - Contient la valeur spéciale **NaN (Not a Number)**
  - string
    - chaîne de caractères (pas de type *character*)
    - Peut s'écrire avec des guillemets doubles, simples ou avec *backtick (`)*
  - boolean
    - **True** ou **False**
  - object
    - Représente des entités plus complexes
    - Utilise le concept de *Prototype* pour créer des comportements plus complexes
    - En JS, une fonction est un object

Types

- JavaScript n'a que huit types de données
- Undefined et Null:
  - Undefined
    - Représente une valeur non assignée
    - Valeur par défaut d'une variable déclarée sans assignation
    - Traité comme « false » dans les opérateurs logiques
  - Null :
    - Représente une valeur vide, mais existante (différent du null en C++ ou C#)
    - typeof null donne 'object' pour des raisons historiques
  - En général, évitez d'utiliser Null et optez pour Undefined

Types

- JavaScript n'a que huit types de données
- 2 Types « niches »:
  - BigInt
    - Représente un ***number*** de longueur arbitraire
    - Ajout récent à JS. Pas encore supporté par tous
    - Permet de dépasser les limites de *number* et ne pas à avoir à utiliser des librairies comme JSBI
  - Symbol :
    - Représente un identifiant unique qui ne peut pas être dupliqué
    - Utile comme attribut d'un objet comme identifiant unique

Conversion de type

- Dans certains contextes, JS 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
    - Ex : (0 == false) est **true** puisque 0 est converti en **boolean**
  - Pas de conversion quand on a deux objets: ils doivent référer au même objet
- Opérateur strict d'égalité (===, !==) :
  - Aucune conversion implicite effectué
  - Ex : (0 === false) est **false** puisqu'on a 2 types différents
- Contexte booléen (**while(...)**, **if(...)**, etc):
  - **false** → false, **true** → true
  - **undefined** → false, **null** → false
  - **0** → false **NaN** → false
  - "" → false
  - Objet → true
  - Autres cas → true
  - Les valeurs qui retournent **false** sont appelées ***falsy***

Déclarations de variables

- JS a 3 manières de déclarer des variables. Avant ES2015, seulement « var » existait.
- **let**:
  - Déclaration d'une variable avec une portée de block (block statement : `{}`)
  - Ne peut pas être utilisée avant sa déclaration
- **const** :
  - Déclaration d'une variable qui ne peut être assignée qu'une seule fois
  - La variable n'est pas immutable (ex : un tableau const peut avoir ses éléments modifiées, mais pas sa valeur propre)
- **var**:
  - Ancienne manière de déclarer des variables.
  - Portée de fonction ou globale (utilisable hors de son bloc)
  - Peut être utilisée avant sa déclaration grâce au *hoisting* : si rencontrée, une déclaration **var** est levée (*hoisted*) au début de son contexte lexique

Déclarations de variables - Exemple

```
console.log(hoist); // Output: undefined

var hoist = 'The variable has been hoisted.';
```

Ce qui arrive vraiment :

```
var hoist;

console.log(hoist); // Output: undefined

hoist = 'The variable has been hoisted.';
```

Déclarations de variables - Exemple

```
function hoist() {

  a = 20;

  var b = 100;

}


hoist();


console.log(a); // 20

console.log(b); // Uncaught ReferenceError: b is not defined
```

Déclarations de variables - Exemple

```
console.log(hoist); // Output: ReferenceError:
hoist is not defined

let hoist = 'The variable was not hoisted.';
```

Ce qu'il faut faire:

```
let hoist;

console.log(hoist); // Output: undefined

hoist = 'The variable was not hoisted.';
```

Fonctions

- Trois manières de créer une fonction:
  - Déclaration
  - Expression de fonction (fonction anonyme)
  - Par l'appel de **new Function** (très rare)
- Une fonction retourne toujours une valeur (par défaut c'est la valeur **undefined**)
- Les fonctions déclarées sont *hoisted* avant l'exécution du script (on peut donc l'appeler avant sa définition)
- Les expressions de fonction doivent être déclarées dans une variable avant d'être utilisées
- 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

```
let increment = function(x) { return x + 1; };
x = increment(3);
// x = 4 après appel
```

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

Fonctions lambda

- On peut utiliser la notation flèche (*arrow*) pour des expressions de fonctions

```
let increment = (x)=>{ return x + 1; };
x = increment(3);
// x = 4 après appel
```

- S'il y a une seule instruction à exécuter, les { } sont optionnelles ainsi que le return

```
let increment = (x)=> x + 1;
x = increment(3);
// increment = x => x+1   est valide aussi
```

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**
  - Dans le cas de NodeJS, cet objet est nommé **global**
- 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 (n'est pas le cas pour **let** et **const**)
  3. Le code est exécuté

Environnement lexical d'une fonction

Voici se 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, let** et **const**) 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 encapsulent la portée des variables déclarées avec **let** et **const**
  - Ex : { let a = 12;} console.log(a); // Uncaught ReferenceError: a is not defined
- Les variables déclarées avec **var** existent en dehors de la portée de bloc. Leur portée est au niveau des fonctions ou globalement.
- Les fonctions imbriquées ont accès aux variables dans les couches supérieures
  - Ex :

```
function sayHiBye(firstName, lastName) {
    function getFullName() { return firstName + " " + lastName;  }
    alert( "Hello, " + getFullName() );
    alert( "Bye, " + getFullName() );
}
```

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;
}

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

Exemple de fermeture

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

let ajouterTrois = f(3); // g(y)
ajouterTrois(4);// retourne g(4) = 7
```

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)); // 5
console.log(ajouterQuatre(4)); // 6
```

Autre exemple de fermeture

```
for (let x = 0; x < 2; x++){
    let f = function() {
        return function(y){return x + y;};
    }

    if (x == 0) {ajouterTrois = f(); };
    if (x == 1) {ajouterQuatre = f(); };
}

console.log(ajouterTrois(3)); // 3
console.log(ajouterQuatre(4)); // 5
```

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());
```

Objets

- Un object se comporte comme un dictionnaire dont la clé est le *string* du nom d'un attribut et la valeur est l'attribut en soi:

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

console.log(michel["age"]); // 29
michel["incrementerAge"](4) // michel.age = 33 maintenant
michel["id"] = 10 // michel a un attribut id de valeur 10
```

Objets

- Une fonction est tout simplemet un object:

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

let michel = new Michel(); //
console.log(michel.age); // 29
michel.incrementerAge(4); // age = 33
console.log(michel.estVieux());
```

L'objet **this**

- **this** est dynamique en JS: il est identifié lors de *l'exécution* d'une fonction (sauf pour les *arrow functions*)
- Quand il est appelé dans une 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
- Une *arrow function* utilise le contexte de **création** et ne créent pas un nouveau contexte pour **this**
- L'objet **this** peut être fourni explicitement en paramètre (toute fonction a une méthode **call()** (et **apply()**) qui prend en argument l'objet **this** et une liste qui contient tous les paramètres de la fonction)

L'objet **this** - Exemples

```
console.log(this) // retourne Window {…}

function printThis(){ console.log(this)};
printThis() // retourne Windows {…}



let obj1 = {
   a: 12,
   f: function(){ console.log(this)}
}
obj1.f() // donne {a:12,f:ƒ}
obj1.f.call({a:'allo'}) // donne {a:'allo'}
```

L'objet **this** - Exemples

```
function MyObj(){
   return {
       a: 123,
       f: function(){ console.log(this)}
   }
}
let obj2 = new MyObj()
obj2.f(); // donne {a:123, f:ƒ}
```

Ceci permet de se rapprocher des classes des langages orienté-objet comme C++ sans utiliser la syntaxe *class* de JS

L'objet **this** - Exemples

```
// Cas special pour les fonctions lambda
let obj = {
  i: 10,
  b: () => console.log(this.i),
  c: function() { console.log(this.i); }
}
obj.b() // retourne undefined
obj.c() // retourne 10

const obj3 = {i:20}
obj.c.call(obj3) // retourne 20
obj.b.call(obj3) // retourne toujours undefined
```

Prototype

- JS est un langage avec des objets basé sur les prototypes et non des classes comme C++
- Il n'y a pas de définition d'objets, seulement des instances d'objets
- On peut étendre le comportement d'un objet en utilisant son instance comme prototype pour un autre objet
- Si un objet ne contient pas l'attribut ou la méthode recherchée, on remonte la chaîne de prototypes
- La chaîne se termine toujours par un pointeur vers **null**

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  (alterantive non standard)
paul = Object.create(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
```

Classes (ES2015)

- La syntaxe de classes a été rajouté dans ES2015
- Une classe en JS est du *sugar synatax* autour des fonctions et des prototypes avec quelques différences mineures
- Une classe contient des attributs et des méthodes, comme un objet normal.
- Le mot clé **this** fait référence à la classe dans laquelle on se trouve, mais reste dynamique (peut être problématique)
- On peut faire de l'héritage de classes (semblable aux prototypes)

Classes (ES2015) - Exemple

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }

}

// Utilisation
let user = new User("Michel");
user.sayHi();
```

Classes (ES2015) - Exemple

```
// Notre classe avec les prototypes
// Notre constructeur par défaut
function User(name) {
  this.name = name;
}

User.prototype.sayHi = function() {
  alert(this.name);
};

// Utilisation
let user = new User("John");
user.sayHi();
```

Classes (ES2015) – Syntaxe get/set

```
class User {

  constructor(name) {
   this.name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (value.length < 4) {
      alert("Name is too short.");
      return;
    }
    this._name = value;
  }

}

let user = new User("John");
alert(user.name); // John

user = new User(""); // Name is too short.
```

Classes – this dynamique

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click() {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); //
  undefined
```

Classes – this dynamique (solution #1)

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click() {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(()=> button.click(), 1000); //
  hello
```

Classes – this dynamique (solution #2)

```
class Button {
  constructor(value) {
    this.value = value;
  }

  click = ()=> {
    alert(this.value);
  }
}

let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

Classes – héritage

```
class Person {
  constructor(name){this.name = name;}
}

class Student extends Person {
  constructor(name, id){
      // faut toujours appeler le constructeur
  du parent avant d'utilizer this
      super(name);
      this.id = id;
  }
}
const Michel = new Student("Michel",1234);
Michel.id = 5678; // on peut changer la valeur
```