Introduction au langage Javascript

Nicolas Le Sommer
Nicolas.Le-Sommer@univ-ubs.fr
Université Bretagne Sud, IUT de Vannes, Département Informatique

Le langage Javascript : un langage normé et plutôt « libre »

- Le langage Javascript est un langage normé
  - respectant le standard/norme ECMAScript
    - Dernière norme : ECMAScript Edition 11 (juin 2020)
  - orienté objet reposant sur des prototypes plutôt que sur des classes
    - La notion de classe a été introduite dans ECMAScript 6 (2015)
  - interprété, faiblement typé et typé dynamiquement
    - Les variables et constantes ne sont pas explicitement typées
    - Les paramètres et les valeurs de retour des fonctions ne sont pas explicitement typés
- Le langage Javascript est un langage plutôt « libre » qui permet de programmer suivant plusieurs paradigmes de programmation :
  - programmation fonctionnelle,
  - programmation impérative,
  - programmation orientée objet

Environnement d'exécution pour Javascript

- Un code Javascript peut être exécuté
  - dans un navigateur Web
  - sur un système d'exploitation

Navigateur/Environnement	Moteur d'exécution Javascript	Organisation
Chrome/Chromium Node.js	V8 Javascript Engine	Google
Firefox	SpiderMonkey	Mozilla
Opera	V8 Javascript Engine	Google
Edge	Chakra	Microsoft
WebKit/Safari	JavaScriptCore	Apple
IoT.js	JerryScript	Samsung
JRE/JDK ≥ vers. 8	Nashorn	OpenJDK/Oracle
JRE/JDK	Rhino	Mozilla

Langages reposant sur le langage Javascript

- Langages reposant sur Javascript et « transpilés » vers Javascript
  - TypeScript
    - Langage libre et open source développé par Microsoft
    - Langage permettant notamment
      - un typage statique (optionnel) des variables et des fonctions,
      - un typage générique,
      - la création de classes et d'interfaces,
      - l'import de modules
    - https://www.typescriptlang.org
  - CoffeeScript
    - Langage ajoutant « du sucre syntaxique » pour améliorer la brièveté et la lisibilité du JavaScript
    - https://coffeescript.org/
- « Transpilation » : processus de traduction d'un langage vers un autre

Langages à objets à classes

- Une classe définie par son code source est statique
- Une classe représente une définition abstraite de l'objet
  - Elle décrit la structure interne des données
  - Elle définit les méthodes qui s'appliqueront aux objets de même type
  - Elle propose des méthodes de création des objets
- Un objet est une instance d'une classe
- L'héritage se situe au niveau des classes

Langages à objets à prototypes

- Un prototype défini par son code source est mutable
- Un prototype est un objet à part entière qui sert de « modèle » de définition de la structure interne et des messages
  - Il a donc une existence physique en mémoire
  - Il peut être modifié et être appelé
  - Il est obligatoirement nommé
- Un des intérêts majeurs des prototypes est l'héritage dynamique
  - Tout objet peut changer de parent à l'exécution, n'importe quand
  - Les caractéristiques héritées par un objet peuvent varier dans le temps

Langages à objets à prototypes

- Les autres objets de mêmes types sont créés par « clonage »
  - Un objet hérite des propriétés (attributs et messages) de son prototype
  - Chaque ajout ou modification d'une propriété d’un objet prototype influence l'objet prototype lui même et l'ensemble de ses clones
  - Chaque ajout ou modification d'une propriété sur un objet cloné n'affecte pas les autres clones du prototype

Exemple de modification de la structure d'un prototype en Javascript

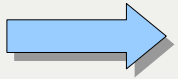
```
function proto(instance_name){
    this.name = instance_name;
    this.f1 = function(){
       console.log(`${this.name}.f1()`);
    }
}
var inst1 = new proto("inst1");
inst1.f1();
console.log('--------------');
proto.prototype.f2 = function(){
    console.log(`${this.name}.f2()`);
}
inst1.f2();
console.log('--------------');
var inst2 = new proto("inst2");
inst2.f1();
inst2.f2()
console.log('--------------');
proto.prototype.f2 = function(){
    console.log(`${this.name}.f2_1()`);
}
inst1.f2();
inst2.f2();
console.log('--------------');
inst1.f2 = function(){
    console.log(`${this.name}.f2_2()`);
}
inst1.f2();
inst2.f2();
```

Définition du prototype

Ajout d'une propriété au prototype

Modification d'une propriété du prototype

Modification d'une propriété d'un objet cloné



```
inst1.f1()
--------------
inst1.f2()
--------------
inst2.f1()
inst2.f2()
--------------
inst1.f2_1()
inst2.f2_1()
--------------
inst1.f2_2()
inst2.f2_1()
```

Déclaration et portée des variables

- JavaScript est sensible à la casse de caractères
  - ***maVariable*** est une variable différente de ***mavariable***.
- Les variables peuvent être déclarées via les mots-clés ***var*** et ***let***
  - Les variables déclarées avec le mot-clé ***let*** sont des variables dont la portée est celle du bloc courant (ECMAScript 6)
  - Les variables déclarées avec le mot-clé ***var*** peuvent être globales ou locales à une fonction
    - Les variables globales sont en réalité des propriétés de l'objet ***global***
    - Dans les pages web, l'objet global est l'objet ***window***

```
if (true) {
  var x = 5;
}
console.log(x); // affiche 5
```

Variable accessible dans la fonction ou dans le programme

```
if (true) {
  let x = 5;
}
console.log(x); // ReferenceError: x is not defined
```

Évaluation des variables

- Il est possible d'utiliser ***undefined*** pour déterminer si une variable possède une valeur.

```
var a;
if (a === undefined){ … }
```

- La valeur *undefined* peut
  - prendre la valeur ***false*** dans un contexte booléen

```
var monTableau = new Array();
if (!monTableau[0]){..}
```

  - être convertie en **NaN** (*Not a Number*) dans un contexte numérique

```
var a;
console.log(a + 2) ; // affichera NaN
```

- La valeur ***null*** sera considérée comme valant 0 dans un contexte numérique

```
var n = null;
console.log(n * 32); // Le log affichera 0
```

Les constantes

- Les constantes sont déclarées avec le mot-clé ***const***
  - Introduit dans ECMAScript 6

```
const PI = 3.141593 ;
console.log(PI > 3.0) ;
```

- Les règles de portée des constantes sont les mêmes que celles des variables

Remontée des variables (*hoisting*)

- Il est possible en Javascript de faire référence, sans recevoir d'exception, à une variable qui est déclarée plus tard
- Ce concept est appelé « remontée » (hoisting en Anglais)
- En revanche, les variables qui n'ont pas encore été initialisées renverront la valeur ***undefined***

```
console.log(x === undefined); // donne "true"
var x = 3;
```

Les types primitifs (1/3)

- ***boolean*** : littéraux ***true*** et ***false***
- ***null*** : valeur nulle
- ***undefined*** : pour les valeurs non définies
- ***number*** : un seul type pour les nombres
  - défini sur 64 bits à précision double selon le format IEEE-754
    - valeurs entre $-(2^{53}-1)$ et $2^{53}-1$
  - Littéraux de nombres entiers (binaire et octale dans ECMAScript 6)

```
0,117 et -345 (notation décimale, base 10)
0o767 === 503  (notation octale, base 8)
0x1123, 0x00111 et -0xF1A7 (notation hexadécimale, base 16)
0b111110111 === 503 (notation binaire)
```

  - Littéraux de nombres décimaux

```
// [chiffres][.chiffres][(E|e)[(+|-)]chiffres]

-3.1E12
```

Les types primitifs (2/3)

- ***string*** : un type pour les chaînes de caractères
- Un littéral de chaîne de caractères se compose de zéro ou plusieurs caractères encadrés par des guillemets droits doubles (") ou des guillemets droits simples (').
- À partir d'ECMAScript 6, on peut également utiliser des littéraux sous forme de gabarits (*templates*) en utilisant le caractère accent grave (`) comme séparateur.

```
var nom = "Alan", prenom= 'Turing';
console.log(`Bonjour ${nom} ${prenom}, comment allez-vous ?`) ;
```

Les types primitifs (3/3)

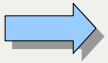
- ***symbol*** : type introduit dans ECMAScript 6 pour représenter des données immuables et uniques
  - peut être utilisé pour ajouter des propriétés à un objet existant sans risquer d'interférer avec les propriétés propres de l'objet, et sans visibilité involontaire

```
var key = Symbol("description");

function MyClass(privateData) {
   this[key] = privateData;
}

MyClass.prototype = {
    someFunc: function() {
       return "data: " + this[key];
    }
};

var c = new MyClass("private data")
console.log(key);
console.log(c["key"]);
console.log(c.someFunc());
```



```
undefined
undefined
data: private data
```

Littéraux d’objets, de tableaux, et d'expressions rationnelles

- Littéraux d'objets

```
var obj = {key1 : 12, key2 : 'Hello'} ;
```

- Littéraux de tableaux

```
var tab = [12, 'Hello'] ;
```

- Littéraux d’expression rationnelles (motif encadré par deux barres obliques)

```
var re = /ab+c/;
```

Les objets natifs

- Objets modélisant les types primitifs
  - Boolean, String, Number, Object, Function, Symbol
- Structures de données
  - Array, Map, Set, WeakMap, WeakSet
- Autres classes
  - Math, Date, Reflect, Proxy, RegEx, JSON, Error, Promise

Transtypage

- Javascript est un langage faiblement typé et typé dynamiquement

```
x = "La réponse est " + 42; // "La réponse est 42"
x = "37" - 7; // 30
X = "37" + 7; // "377"
```

- Les fonctions *parseInt()* et *parseFloat()* permettent de convertir une représentation textuelle d'un nombre en nombre en entier ou flottant

Les structures conditionnelles

- Structures conditionnelles habituelles *if* et *switch*

```
// conditionnelle if
if (condition_1) {
  instruction_1;
} else if (condition_2) {
  instruction_2;
} else {
  dernière_instruction;
}

// conditionnelle switch
switch (expression) {
  case label_1:
    instructions_1
    [break;]
  case label_2:
    instructions_2
    [break;]
  ...
  default:
    instructions_par_defaut
    [break;]
}
```

Les structures itératives

- Les instructions *for*
  - *for*

```
for (var i = 0; i < tab.length; i++) {
  tab[i] = i ;
}
```

  - *for… in* permet d'itérer sur l'ensemble des propriétés énumérables d'un objet

```
for (var i in obj) {
   result += nomObj + "." + i + " = " + obj[i] + "\n";
 }
```

  - *for… of* pour le parcours d'objets itérables (Map, Set, Array) (introduit dans ECMAScript 6)

```
let arr = [3, 5, 7];
arr.toto = "coucou";
for (let i in arr) {
  console.log(i); // affiche "0", "1", "2", "toto" dans la console
}
for (let i of arr) {
  console.log(i); // affiche "3", "5", "7" dans la console
}
```

Les structures itératives

- Boucles *do...while* et *while*

```
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

```
while (i < 5) {
  i += 1;
  console.log(i);
}
```

- Instructions *label*, *break et continue*
  - *label* : associer une étiquette à un bloc d'instructions
  - *break :* sortir d'une conditionnelle switch ou d'une boucle
    - *break*
    - *break label* : sortir et aller au bloc d'instructions identifié par l'étiquette
  - *continue* : permet de reprendre une boucle while, do-while, for, ou une instruction label.
    - *continue*
    - *continue label*

Parcours de tableaux

- 3 façons de parcourir des tableaux : *for*, *for...in* et *forEach*

```
var tab = ['elem1', 'elem2', 'elem3'];
for(let i = 0; i < tab.length; i++){
    console.log(tab[i]);
}

for(let i in tab){
    console.log(tab[i]);
}

tab.forEach(function(v){
    console.log(v);
});
```

Gestion des exceptions

- Génération d'une exception avec l'instruction ***throw***
  - ***throw*** *expression* : n'importe quel type d'expression (chaîne de caractères, objet).

```
throw "Erreur2"; // type String
throw 32;        // type Number
throw true;      // type Boolean
throw {toString: function() {return "je suis un objet !";}};
```

- Capture des exceptions dans un bloc ***try... catch()***
- Le bloc ***finally*** contient les instructions à exécuter après les blocs ***try*** et ***catch***, mais avant l'instruction suivant le ***try...catch***. Le bloc ***finally*** est exécuté dans tous les cas.

```
try{
    écrireFichier(données); // une erreur peut se produire
}
catch(e){
    gérerException(e); // On gère le cas où on a une exception

} finally{
    fermetFichier(); // On n'oublie jamais de fermer le flux.
}
```

Gestion des exceptions

```
// On crée le constructeur pour cet objet
function ExceptionUtilisateur(message){
    this.message = message;
    this.name = "ExceptionUtilisateur";
}

// On surcharge la méthode toString pour afficher
// un message plus explicite (par exemple dans la console)

ExceptionUtilisateur.prototype.toString = function(){
    return this.name + ': "'+ this.message +'"';
}

// On crée une instance pour ce type d'objet)(
// et on renvoie une exception avec cette instance
function test(){
    throw new ExceptionUtilisateur("La valeur fournie est trop élevée !");
}

try{
    test();
}catch(e){
    console.log(e.toString());
}
```

Les fonctions

- En JavaScript, les fonctions sont des objets de première classe, au même titre que les chaînes de caractères, les nombres, etc.
- Cela veut dire qu'une fonction peut :
  - être exprimée comme une valeur anonyme littérale
  - être affectée à des variables ou des structures de données
  - avoir une identité propre
  - être comparable pour l'égalité ou l'identité avec d'autres entités
  - être passée comme paramètre à une procédure ou à une fonction
  - être retournée par une procédure ou une fonction
  - être construite lors de l'exécution
- Suivant les langages, les fonctions ne sont pas toujours des objets de première classe : en C, par exemple, ce n'est pas le cas. Mais quand c'est le cas, cela permet des constructions intéressantes : les fonctions d'ordre supérieur.
  - Une fonction d'ordre supérieur est une fonction qui accepte au moins une autre fonction en paramètre, et/ou qui retourne une fonction en résultat.

Fonctions nommées et fonctions anonymes

- Les fonctions sont déclarées à l'aide du mot-clé ***function***
- Les fonctions peuvent retourner un résultat via l'instruction ***return***
- Les types primitifs sont passés par valeurs, les autres (objets, tableaux, fonctions) par références
- Une fonction peut être anonyme ou être nommée
- Une fonction peut être passée en argument d'une autre fonction

```
function myFunction(f,a){
    f(a);
}
var func1 = function(x){
    console.log(`parameter of func1 is: ${x}`);
}
myFunction(func1,3);

myFunction(function(x){
    console.log('parameter of the anonymous function is: '+x);
},3);
```

```
// fonction nommée
function square(x){return x*x;};
console.log(square(3));

// function anonyme
var square2 = function(x){return x*x;};
console.log(square2(3));
```

Remontée des fonctions (*hoisting*)

- La portée d'une fonction est
  - la fonction dans laquelle elle est déclarée
  - ou le programme entier si elle est déclarée au niveau le plus haut.
- En Javascript, une expression de fonction (i.e. une fonction anonyme) n'est pas « remontée » à la différence d'une fonction nommée
  - Une fonction anonyme ne peut pas être invoquée avant sa déclaration

```
f1("boo");
f2("boo");

var f2=function(x) {console.log("f2:"+x)};
function f1(x){console.log("f1:"+x)};
```

```
f1:boo
/tmp/test.js:2
f2("boo");
^
TypeError: f2 is not a function
    at Object.<anonymous> (/tmp/test.js:2:1)
```

```
f1("boo");
          var f2=function(x) {console.log("f2:"+x)};
function f1(x){console.log("f1:"+x)};
f2("boo");
```

```
f1:boo
f2:boo
```

Fonctions flêchées (*Arrow Functions*)

- Une expression de fonction fléchée permet d'avoir une syntaxe plus courte que les expressions de fonctions
- Une fonction fléchée ne possède pas ses propres valeurs ***this***, ***arguments***, ***super*** ou **new.*target***
  - Elles utilisent les valeurs de leur contexte
- Syntaxe :

```
([param] [, param]) => { instructions }

(param1, param2, …, paramN) => expression
// équivalent à (param1, param2, …, paramN) => { return expression; }

// Parenthèses non nécessaires quand il n'y a qu'un seul argument
param => expression

// Une fonction sans paramètre peut s'écrire avec un couple de parenthèses
() => { instructions }

// Gestion des paramètres du reste et paramètres par défaut
(param1, param2, ...reste) => { instructions }
(param1 = valeurDefaut1, param2, …, paramN = valeurDefautN) => {
instructions }
```

```
var simple = a => a>15?15:a;
simple(16); // 15
simple(10); // 10

var max = (a, b) => {
  return a > b ? a: b;
}
```

Fonctions et portée des variables

- Les variables définies dans une fonction ont comme portée la fonction
- Une fonction peut accéder aux variables définies dans une fonction parente

```
function f1(){
    let x = 2;
    function f2(){
        let y = 3;
        console.log(x+y); // affichera 5
    }
    f2();
}
f1();
```

Fonctions et récursivité

- Récursivité
  - Une fonction peut faire référence à elle-même et s'appeler elle-même.
  - 3 solutions possibles:
    - Le nom de la fonction
    - *arguments.callee*
    - Une variable de la portée qui fait référence à la fonction

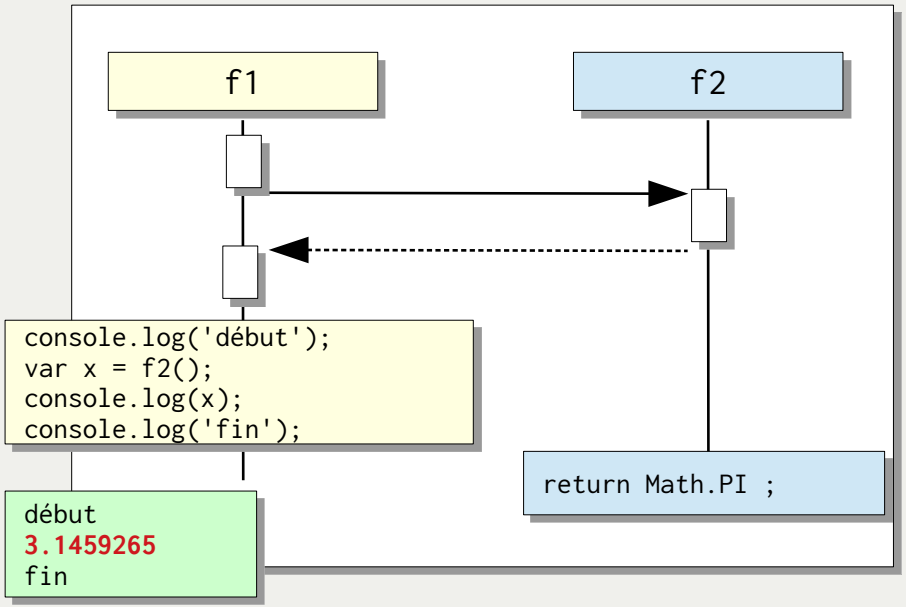
```
var r = function recursive(x){
    if(x < 3){
        console.log(x);
        recursive(x+1);
    }
}
recursive(0);
```

```
var r = function recursive(x){
    if(x < 3){
        console.log(x);
        r(x+1);
    }
}
recursive(0);
```

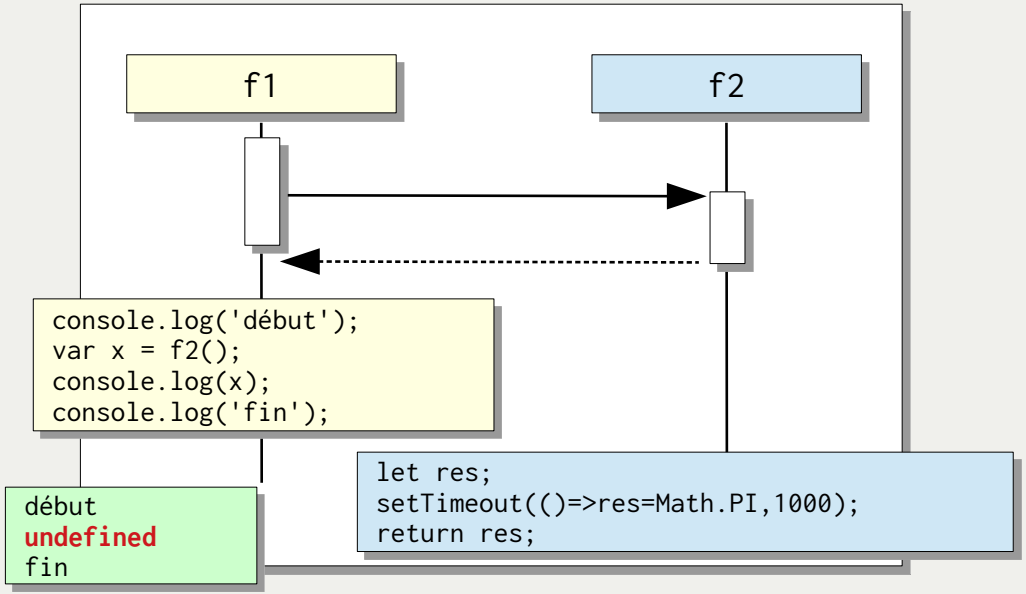
```
var r = function recursive(x){
    if(x < 3){
        console.log(x);
        arguments.callee(x+1);
    }
}
recursive(0);
```

Fonctions et asynchronisme

- Un appel à une fonction non bloquante peut provoquer un asynchronisme dans l'exécution du programme
  - Les instructions ne sont pas exécutées de manière séquentielle



Exécution séquentielle et synchrone



Exécution asynchrone

Appel asynchrone des fonctions

- Au lieu de rester bloquer en attente du résultat retourné par une fonction, on peut passer en paramètre une fonction (*callback*) qui sera chargée de traiter le résultat retourné.
- Exemple sans *callback*

```
function f1(){
    console.log('debut');
    var x = f2();
    console.log(x);
    console.log('fin');
}
function f2(){
    return Math.PI
}
f1();
```

- Exemple avec *callback*

```
function f1(){
       console.log('debut');
       f2(console.log);
       console.log('fin');
}
function f2(callback){
    setTimeout(()=>callback(Math.PI),10000);
}
f1() ;
```

Promesses (1/2)

- L'objet ***Promise*** est utilisé pour réaliser des traitements de façon asynchrone.
- Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais.
- Une promesse est créée en passant en paramètre une fonction « d'éxécution » qui réalise le travail asynchrone et qui prend 2 fonctions comme arguments : ***resolve*** et ***reject***

```
new Promise( function(resolve, reject) { ... } );
```

- La fonction d'exécution démarre le travail asynchrone puis, une fois le travail terminé, appelle
  - la fonction ***resolve*** (si tout s'est bien passé)
  - ou la fonction ***reject*** (lorsqu'il y a eu un problème) pour définir l'état final de la promesse.

Promesses (2/2)

- Une promesse peut être dans l'état
  - ***pending*** (en attente) : état initial, la promesse n'est ni remplie, ni rompue ;
  - ***fulfilled*** (tenue) : l'opération a réussi ;
  - ***rejected*** (rompue) : l'opération a échoué ;
  - ***settled*** (acquittée) : la promesse est tenue ou rompue mais elle n'est plus en attente.
- Les promesses peuvent être « chaînées » via les méthodes ***then*** (succès de la promesse) et ***catch*** (erreur de la promesse)

```
function resolveAfter2Seconds() {
  return new Promise((resolve,reject) => {
    setTimeout(() => {resolve('resolved');}, 2000);
  });
}
async function asyncCall() {
  console.log('calling');
  var prom = resolveAfter2Seconds();
  prom.then(val=>{console.log(val);}); // expected output: "resolved"
}
asyncCall();
```

Fonctions asynchrones

- La déclaration ***async function*** définit une fonction asynchrone qui renvoie un objet ***AsyncFunction***
- Une fonction asynchrone s'exécute de façon asynchrone grâce à la boucle d'évènement en utilisant une promesse (***Promise***) comme valeur de retour.
- Une fonction asynchrone peut contenir une expression ***await*** qui interrompt l'exécution de la fonction asynchrone et attend la résolution de la promesse. La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

```
function resolveAfter2Seconds() {
  return new Promise((resolve,reject) => {
    setTimeout(() => {resolve('resolved');}, 2000);
  });
}
async function asyncCall() {
  console.log('calling');
  var result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
}
asyncCall();
```

Les fermetures

- Une fonction peut être imbriquée dans une autre fonction
- La fonction imbriquée forme une fermeture (*closure* en anglais): elle peut utiliser les arguments et les variables de la fonction parente. En revanche, la fonction parente ne peut pas utiliser les arguments et les variables de la fonction fille.
  - La portée de la fonction fille (celle qui est imbriquée) n'est pas contenue dans la portée de la fonction parente
  - La fonction fille bénéficie bien des informations de la fonction parente grâce à sa portée
  - La fonction imbriquée ne peut être utilisée qu'à partir des instructions de la fonction parente

Le mot-clé *this*

- Le mot-clé ***this*** peut se comporter différemment selon le contexte utilisé, et selon qu'on utilise le mode ***strict*** ou pas
- Dans le contexte global
  - Fait référence à l'objet global (en mode **strict** ou non)
  - Dans un navigateur Web, le contexte global est l'objet ***window***
  - Avec Node.js c'est l'environnement d'exécution qui est l'objet global
- Dans le contexte d'une fonction
  - La valeur de ***this*** dépend de l'utilisation du mode ***strict***
    - En mode strict, la valeur de ***this*** est conservée entre le moment de sa définition et l'entrée dans le contexte d'exécution. S'il n'est pas défini, il vaut ***undefined***.

```
function f1(){
    console.log(this);
}
f1(); // retour l'objet global
```

```
function f1(){
      "use strict";
    console.log(this);
}
f1(); // retour undefined
```

Le mot-clé *this*

- Dans une méthode d'un objet
  - ***this*** correspondra à l'objet possédant la méthode appelée

```
var o ={
    val:12,
    getVal: function(){return this.val;}
};
console.log(o.getVal()); // retourne 12
```

- Dans les getters/setters
  - ***this*** correspondra à l'objet associé au getter/setter

```
var o ={
    val:12,
    get getVal(){return this.val;},
    set setVal(x){this.val = x;}
};

var obj = Object.create(o);
console.log(obj.getVal); // retourne 12
obj.setVal=3;
console.log(obj.getVal); // retourne 3
```

Le mot-clé *this*

- Dans un constructeur
  - ***this*** correspondra au nouvel objet en train d'être construit
- Dans la chaîne de prototype
  - Si une méthode se situe sur la chaîne de prototype, **this** fera référence à l'objet appelant

```
var o ={
    f: function(){return this.a + this.b;}
};
var obj = Object.create(o);
obj.a = 1;
obj.b = 2;
console.log(obj.f()); // retourne 3
```

Programmation orientée objet

- Les classes ont été introduites dans ECMAScript 6
  - « Sucre syntaxique » par rapport à l'héritage du modèle à prototype
  - Fournit uniquement une syntaxe plus claire pour utiliser un modèle classique et gérer l'héritage.
- Les classes contiennent
  - un constructeur
  - des méthodes (statiques)
  - des attributs (statiques)

```
// anonyme
var maClass = class{
    constructor(a){
       this.attribut = a;
    }
};
```

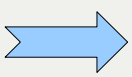
- Les classes peuvent hériter de classes parentes
  - Les attributs ou les méthodes des classes parentes sont accédés via le mot clé « ***super*** »
- Utilisation du mode **strict**

```
// nommée
class MaClasse{
    constructor(a){
       this.attribut = a;
    }
};
```

Programmation orienté objet : exemple

```
'use strict';
class MaClasse {
    constructor(a){
       this.attribut = a;
    }
    methode(){
       console.log('MaClasse.attribut = '+this.attribut);
    }
}

class MaClasseFille extends MaClass{
   constructor(){
       super('Fille');
    }
    methode(){
       super.methode();
       console.log('MaClasseFille.methode()');
    }
}
var obj1 = new MaClasse('Parente');
var obj2 = new MaClasseFille();
obj1.methode();
obj2.methode();
```



```
MaClasse.attribut = Parente
MaClasse.attribut = Fille
MaClasseFille.methode()
```

Les collections

- Collections indicées : objet *Array*

```
var x = new Array('12','a');
var y = ['12','a']; // équivalent à la définition précédente
console.log(y.length); // taille du tableau y
x.forEach(function(a){console.log(a);}); // parcours du tableau
console.log(x.indexOf('a')); // indice de la première occurrence de a
x[2]='3'; // ajoute un élément à x
y.push('3'); // équivalent à l'instruction précédente
y.pop(); // supprime le dernier élément
y.shift(); // supprime le premier élément
```

- Collections avec clés : objets *Map*, *Set*, *WeakMap* et *WeakSet*

```
var map = new Map();
map.set('key1','value1');
map.set('key2','value2');
map.size; // 2
map.get('key3'); // undefined
map.has('key1'); // true;
map.delete('key1');

for(var [key, value] of map){
 console.log(`${key} = ${value}`);
}
```

```
var monEnsemble = new Set();
monEnsemble.add(1);
monEnsemble.add("du texte");
monEnsemble.add("ensemble");

monEnsemble.has(1); // true
monEnsemble.delete("ensemble");
monEnsemble.size; // 2

for(let item of monEnsemble){
    console.log(item);
}
```

Modules, import et export

- ECMAScript 6 (2015) introduit la notion de modules, d'import et d'export
- Les modules permettent d'organiser et de structurer le code
- Les modules sont supportés
  - dans les dernières versions des navigateurs
  - et dans Node.js
- Exemple d'utilisation des modules dans les navigateurs

```
<!DOCTYPE html>
<html lang="fr-FR">
<head>
  <meta charset="utf-8">
  <script type="module" src="./bundle.js"></script>
</head>
<body>
</body>
</html>
```

Export

```
// SYNTAXE
export { nom1, nom2, …, nomN };
export { variable1 as nom1, variable2 as nom2, …, nomN };
export let nom1, nom2, …, nomN; // fonctionne également avec var, const
export let nom1 = …, nom2 = …, …, nomN; // également avec var, const
export function nomFonction() { … };
export class nomClasse { … };

export default expression;
export default function (…) { … } // également avec class, function*
export default function nom1(…) { … } // également avec class, function*
export { nom1 as default, … };

export * from …;
export { nom1, nom2, …, nomN } from …;
export { import1 as nom1, import2 as nom2, …, nomN } from …;
export { default } from …;
```

Export nommés

Export par défaut

Redirection entre modules

```
// EXEMPLES
export const PI=3.14;
function square (x){return x*x} ;
export {const,square} ; // maFonction a été préalablement déclarée

let value = 12 ;
export default value;
export default function square (x){return x*x} ;

export * from 'autre_module.js' ;
export { default } from 'autre_module.js' ;
```

Import

```
// SYNTAXE
import exportParDefaut from "nom-module";
import * as nom from "nom-module";
import { export } from "nom-module";
import { export as alias } from "nom-module";
import { export1 , export2 } from "nom-module";
import { export1 , export2 as alias2 , [...] } from "nom-module";
import exportParDefaut, { export [ , [...] ] } from "nom-module";
import exportParDefaut, * as nom from "nom-module";
import "nom-module";
```

- `exportParDefaut` : Nom qui fera référence à l'export par défaut du module.
- `nom-module` :Le module depuis lequel importer.
- `alias` :Noms qui feront référence aux imports nommés.
- `nom` :Nom de l'objet module qui sera utilisé comme un genre d'espace de noms lors de références aux imports.
- `export` : Nom des exports à importer.

```
// EXEMPLES

// Ajoute tous les exports du module à la portée courante
import * as monModule from '/modules/mon-module.js';

// Ajoute les exports truc1 et truc2 du module à la portée courante
import {truc1, truc2} from '/modules/mon-module.js';

// Renome l'export lors de l'import
import {nomDExportDeModuleVraimentVraimentLong as nomCourt}
  from '/modules/mon-module.js'
```

Bibliographie

- Documentation du projet Mozilla
  - https://developer.mozilla.org/fr/docs/Web/JavaScript/
- Documentation Node.js
  - https://nodejs.org/documentation/
- Norme ECMAScript
  - http://www.ecma-international.org/