



Javascript

Date de publication :

10 février 2017

Cet article est issu de : **Technologies de l'information | Technologies logicielles
Architectures des systèmes**

par **Christian QUEINNEC**

Mots-clés

Informatique | JavaScript |
Programmation | langage de
programmation

Résumé Javascript est un langage de programmation pour les applications Web. Quoique assez jeune, puisque né en 1995, l'essor du Web a conféré à Javascript une position dominante tant dans les navigateurs que maintenant côté serveurs. Cet article présente ce langage dans sa version standardisée de 2015.

Abstract Javascript is a programming language created for Web applications. Even if still young, the terrific development of the Web has been pushing Javascript as the dominant programming language for browsers and, now, for servers. This article presents Javascript in its most recent standard version (2015).

Keywords

Computer | JavaScript |
programming | programming
language

Pour toute question :

Service Relation clientèle
Techniques de l'Ingénieur
Immeuble Pleyad 1
39, boulevard Ornano
93288 Saint-Denis Cedex

Par mail :

infos.clients@teching.com

Par téléphone :

00 33 (0)1 53 35 20 20

Document téléchargé le : **18/10/2018**

Pour le compte : **7200043497 - imt atlantique brest et rennes // 192.108.116.193**

Javascript

Christian QUEINNEC

Professeur émérite de l'UPMC

1. Historique et emplois	H 3 120 - 2
2. Syntaxe	— 2
2.1 Instruction.....	— 2
2.2 Commentaires.....	— 2
2.3 Nombres	— 2
2.4 Chaînes de caractères.....	— 2
2.5 Identificateurs.....	— 3
2.6 Expressions rationnelles	— 3
2.7 Données composites	— 3
3. Valeurs	— 3
3.1 Nombres	— 3
3.2 Valeurs de vérité	— 4
3.3 Objets	— 4
3.4 Tableaux	— 5
3.5 Fonctions	— 5
4. Instructions.....	— 9
4.1 Opérateurs	— 9
4.2 Contrôle	— 9
5. Environnement global	— 9
5.1 Anciens modules.....	— 10
5.2 Modules d'ECMAScript 2015	— 10
6. Héritage et prototypes	— 11
6.1 Propriétés.....	— 11
6.2 Comportements	— 11
6.3 Constructeurs	— 12
6.4 Factorisation	— 12
6.5 Héritage.....	— 13
6.6 Hiérarchie.....	— 13
6.7 Syntaxes	— 14
6.8 Conclusions	— 14
7. Concurrence	— 14
7.1 Programmation par suites.....	— 15
7.2 Séquence par enchaînement de suites.....	— 15
7.3 Concurrence avec suite similaire	— 15
7.4 Promesses	— 16
7.5 Arbre de promesses.....	— 16
8. Générateur.....	— 17
8.1 Générateur linéaire	— 17
8.2 Générateur bouclant	— 17
8.3 Générateur et asynchronisme.....	— 17
8.4 Futur	— 18
9. Conclusions	— 18
Pour en savoir plus	Doc. H 3 120

Javascript est désormais le langage du Web tant du côté des serveurs que du côté des navigateurs qu'il s'exécute sur téléphone, tablette ou ordinateur. Cet article décrit synthétiquement le langage (dans sa version ECMAScript 2015) et ses principales spécificités afin de permettre aux lecteurs, ayant une certaine pratique de l'informatique, d'appréhender les caractéristiques majeures de ce langage de programmation. Tout ne sera bien évidemment pas dit (la norme fait 566 pages) mais l'essentiel le sera.

Les exemples figurant dans cet article n'ont pas une tonalité uniforme, ils varient les styles, emploient parfois des caractéristiques en avance ou des caractéristiques mineures non nécessairement détaillées dans le texte, mais qui illustrent l'éventail des possibilités qu'offre ECMAScript 2015.

1. Historique et emplois

Javascript est un langage de programmation défini en 10 jours et en 1995 par Brendan Eich au sein de la société Netscape. Javascript a été rapidement incorporé dans le navigateur que produisait cette société. Procurer un langage s'exécutant dans le navigateur s'est avéré une riche idée permettant de manipuler dynamiquement les entités propres au navigateur, à savoir textes et formulaires à la fois en contenu et en apparence.

Javascript a longtemps souffert d'implantations concurrentes et divergentes : au début étaient Javascript de Netscape, JScript de Microsoft et ActionScript de Macromedia. Un effort de standardisation, nommé ECMA-262, débuté en 1997, culmine aujourd'hui avec la norme actuelle, ECMAScript 2015 (aussi dite ES6 car c'est la sixième version) qui, adoptée en juin 2015, précède même les implantations actuellement déployées. C'est cette version qui sera présentée dans cet article.

Javascript est un langage héritant principalement de Scheme (un langage fonctionnel typé dynamiquement) et de Self (un langage où les objets sont des prototypes et non des instances de classes) avec des adjonctions provenant de Perl (tables associatives omniprésentes, conversions de type généreuses et implicites et expressions rationnelles de première classe). Javascript adopte une syntaxe à la C, il est muni de capacités réflexives lui permettant d'inspecter son propre mode de fonctionnement et procure une gestion automatique de la mémoire. Ce cocktail de traits l'a longtemps rendu lent car uniquement interprété. Deux décennies de travaux ont permis d'élaborer des techniques d'évaluation efficaces. Ces avancées ont fait naître les premières grandes applications tournant dans les navigateurs clients (GMail par exemple) que les navigateurs soient sur des ordinateurs, des tablettes ou des téléphones. Mais ces avancées ont aussi permis l'écriture de programmes côté serveur : Javascript (ou plus précisément Rhino) est présent dans les JVM (machine d'exécution de Java) depuis Java5 et, plus récemment, Node.js est un moteur complet de Javascript pour l'écriture de serveurs totalement rédigés en Javascript. Enfin, on trouve Javascript comme langage de script ou langage de requête dans des systèmes comme, par exemple, la base de données noSQL : MongoDB.

Tant côté client que serveur, Javascript est maintenant le langage de programmation du Web.

2. Syntaxe

Javascript a une syntaxe dans la lignée de C mais possède quelques singularités.

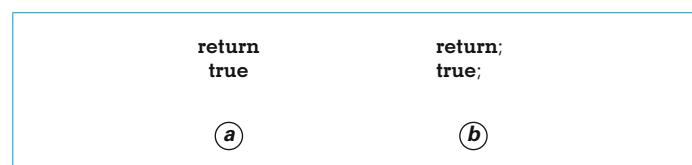


Figure 1 – Exemple de mauvaise interprétation liée à une écriture sur deux lignes (a) ou d'une écriture correcte (b)

2.1 Instruction

Contrairement à C, le point-virgule qui marque la fin d'une instruction est optionnel : une fin de ligne peut donc signifier une fin d'instruction. Toute instruction écrite sur plus d'une ligne (figure 1a) peut ainsi être mal comprise (figure 1a) et pourra être interprétée en marquant explicitement les fins d'instruction avec des points-virgules comme la figure 1b.

Cette licence peut s'avérer dangereuse et c'est pourquoi les guides de style recommandent de terminer systématiquement les instructions (sauf les définitions de fonctions) par des points-virgules.

2.2 Commentaires

Les commentaires à la C (entre /* et */) ou à la C++ (débutant par //) sont possibles.

2.3 Nombres

Les nombres s'écrivent comme usuellement. Les écritures binaires (préfixées par 0b), octales (préfixées par 0o), hexadécimales (préfixées par 0x) sont également permises.

2.4 Chaînes de caractères

Elles s'écrivent enserrées entre simples ou doubles guillemets ; elles doivent tenir sur une unique ligne logique. Techniquement, ce sont des séquences d'unités de code (*code units*) Unicode (UTF-16). Une ou deux unités de code permettent de représenter un caractère Unicode.

Tous les caractères Unicode sont possibles (possiblement représentés par \uHHHH où HHHH est le nombre hexadécimal identifiant le caractère Unicode). Comme en C, certains caractères spéciaux peuvent être obtenus en les préfixant d'une barre oblique inverse \ par exemple pour \n (fin de ligne) ou \\. Enfin une barre oblique en fin de ligne permet de décomposer une ligne logique en plusieurs lignes physiques.

```
'Première chaîne\n d\'aujourd\'hui'
"Première"
chaîne\n d'\n
aujourd'hui"
```

Figure 2 – Différentes écritures de chaîne

```
var moi = 'Christian';
'Bonjour$(' + moi) === 'Bonjour Christian' // est vrai
```

Figure 3 – Création d'une chaîne par patron

Les deux chaînes de la figure 2 sont égales quoique différemment écrites.

ECMAScript 2015 permet aussi de créer des chaînes par instanciation d'un patron (entre accents graves) où les parties à calculer sont signalées par un dollar et enserrées entre accolades (figure 3).

Les chaînes ont une propriété `length` représentant leur longueur en unités de code. Heureusement, presque tous les caractères usuels tiennent en une seule unité de code et non en deux. Ainsi, souvent, `length` est le nombre de caractères présents.

2.5 Identificateurs

Les noms en Javascript sont semblables à ceux de C sauf que \$ est permis (y compris en première position) ainsi que tous les caractères Unicode. Bien entendu, certains noms sont réservés, ce sont les mots-clés suivants :

```
var delete in with let
new typeof instanceof null void
if else true false
while break continue do for
switch case break default
try catch finally throw
function return yield this
export import const static
class extends super
debugger
```

Quelques autres sont aussi réservés pour des extensions futures. La liste des mots réservés évoluant de version en version, on s'abstiendra d'utiliser les mots-clés d'autres langages bien connus comme Java (par exemple, `implements`, `private`, etc.).

La figure 4 montre l'usage d'Unicode pour une variable, dans un commentaire et dans des chaînes de caractères.

2.6 Expressions rationnelles

Les expressions rationnelles (figure 5) (ou, en jargon, expressions régulières ou `regexp`) sont des valeurs de première classe : elles peuvent être manipulées telles quelles, stockées dans des

```
/^fo+.*r/.test('foobar')      // vaut vrai
'fooorbi'.match(/^fo+.*r/) // vaut vrai
```

Figure 5 – Exemple d'expressions rationnelles

```
{ pi: 3.14,
pi: 31.415/10,
"repI": /^3[.]14.*$/i
[ 3.14, "p"+"i", 10*pi, true ]
```

Figure 6 – Exemples d'écriture de tables associatives

variables, passées en argument de fonction, rendues en résultat de fonction. Comme en Perl, elles s'écrivent entre barres obliques.

2.7 Données composites

Deux syntaxes permettent de décrire des tables associatives (entre accolades) ou des tableaux (entre crochets) (figure 6).

Deux points particuliers sont illustrés dans cet exemple, il ne faut pas de virgule après le dernier terme et il est possible d'insérer des expressions calculées parmi les termes d'un tableau ou parmi les valeurs des clés d'une table associative.

3. Valeurs

Javascript est dynamiquement typé : toute valeur produite par un calcul a un type qui peut être booléen, nombre, chaîne de caractères, fonction, objet. En revanche, les variables n'ont pas de type et peuvent contenir successivement des valeurs de types différents (même si ce n'est pas nécessairement recommandable). Les valeurs de Javascript sont dites des **citoyens de première classe** car il n'y a pas de restriction sur leur emploi : elles peuvent être stockées dans des variables, des tableaux, des objets, être passées en argument de fonction ou renvoyées en résultat de fonction.

On distingue les types primitifs comme booléen, nombre, chaîne de caractères ou symboles, des objets qui sont fondamentalement des tables associatives. Existent aussi des natures d'objets prédéfinies (des classes), par exemple, Date, RegExp ou Error. Existent également deux singulaires `null` (pour représenter l'absence de valeur) et `undefined` (la « valeur » initiale d'une variable ou d'une propriété d'un objet non encore initialisée) qui ont pour type, respectivement, `object` (*sic*) et `undefined`.

3.1 Nombres

Du point de vue de l'implantation, il n'y a, en Javascript, que des flottants sur 64 bits. Les entiers (entre -2^{53} et $+2^{53}$) ne sont que des flottants sans partie décimale. Si le type des entiers et des flottants est l'unique `number`, on peut toutefois distinguer les entiers avec le prédictat `Number.isInteger`.

Conformément à la norme IEEE 754, quelques valeurs spéciales existent : +0, -0, Infinity, -Infinity et NaN (pour *Not a Number*). Ces valeurs peuvent être distinguées grâce aux prédictats spécialisés : `isFinite` et `isNaN` (figure 7).

NaN est souvent utilisé par Javascript lorsque le résultat d'une opération censée produire un résultat numérique a des opérandes incorrects, ainsi `[] * "foo"` donne NaN.

```
var π = +31.415926535e-1; // Valeur approximative de \u03C0 (pi)
'La valeur de "\u03C0" (π) est ${π}';
```

Figure 4 – Exemple de caractères Unicode dans une chaîne et une variable

```

-1/0 === -Infinity; // est vrai
isFinite(1/0);      // est faux
isNaN(0/0);         // est vrai
isNaN(1/0);         // est faux
NaN === NaN || Nan == NaN; // est faux
Object.is(NaN, NaN)           // est vrai
-0 == +0 && -0 === +0 && ! Object.is(+0, -0) // est vrai

```

Figure 7 – Utilisation de prédictats spécialisés pour nombres

3.2 Valeurs de vérité

C'est un des points délicats de Javascript. Aux valeurs **true** et **false** de type **boolean** s'ajoutent de nombreux autres cas dus à des conversions automatiques de type. Ainsi distingue-t-on deux opérateurs de comparaison **==** (égalité) et **===** (identité) et, pour faire bonne mesure, un prédictat supplémentaire de comparaison **Object.is**. Pour les types primitifs (booléen, nombre, chaîne de caractères, symboles), égalité et identité donnent les mêmes résultats. Ainsi, les chaînes de caractères sont comparées caractère par caractère. Et, comme vu ci-dessus, on fera attention à la comparaison de nombres.

Le comparateur **Object.is** n'est vrai que si ses deux opérandes sont une seule et même valeur (et **Nan** est **Nan**). L'identité a un comportement semblable sauf que **+0** et **-0** sont considérés comme identiques alors que **Nan** n'est jamais identique à **Nan**. L'identité implique l'égalité, mais le contraire n'est pas toujours vrai (figure 8).

Les conversions de type automatiques introduisent des phénomènes étranges. Les valeurs **null** et **undefined** mais aussi **0** et la chaîne vide **" "** sont toutes considérées comme fausses. Les chaînes ne contenant que des chiffres peuvent être converties en nombres, les valeurs peuvent être converties en des chaînes, les booléens peuvent être dégradés en nombres, etc. (figure 9).

Bref, le problème de ces conversions est que tout ou presque a un sens, mais pas nécessairement celui auquel on s'attend !

3.3 Objets

Les objets de Javascript peuvent être vus, dans un premier temps, comme des tables associatives associant clés et valeurs. C'est une vision inspirée des langages de programmation Perl et Self. Si les valeurs sont de type quelconque, les clés (ou **propriétés** dans le vocabulaire de Javascript) doivent être des chaînes de

```

var a = [1, "2"];
var b = [1, "2"];
a === a && a == a; // est vrai
a === b || a == b;   // est faux
a[1] === b[1];      // est vrai

```

Figure 8 – Comparaisons : identité et égalité

```

/* étranges mais tous vrais! */
"1" == true
0 == false
(9 - "1") === 8
(9 + "1") === "91"
isNaN({} - '2')

```

Figure 9 – Résultats inattendus de conversion automatique

```

var ti = {
  nom: "Techniques de l'ingénieur",
  article: 'Javascript'
};
ti.article === "Javascript"; // est vrai
ti['art' + 'icle'] === 'Javascript'; // est vrai
'nom' in ti; // est vrai
delete ti.article;
'title' in ti; // est maintenant faux
ti.article == undefined; // est vrai
ti.article = {
  surnom: 'JS',
  'mon age': undefined
};
for (let key in ti) {
  key === 'article' || key === "nom" // est vrai
}
for (const key in ti.article) {
  key === 'surnom' || key === 'mon age' // est vrai
}

```

Figure 10 – Description des objets

caractères, des symboles ou bien des valeurs convertibles en des chaîne de caractères (nombres ou objets munis d'une méthode **toString**).

On peut accéder aux propriétés des objets par la notation pointée usuelle aux langages à objets. On peut aussi calculer la propriété, savoir si une propriété est présente et énumérer les propriétés d'un objet. Un objet (figure 10) est dynamique : on peut lui ajouter des propriétés, mais aussi en supprimer. La valeur associée à une propriété absente est **undefined**.

3.3.1 Propriétés

Les propriétés d'un objet peuvent être modulées par des attributs. Afin de protéger certaines propriétés, il est possible d'interdire leur modification ou de les déclarer non énumérables. Fixer les attributs des propriétés peut s'effectuer syntaxiquement ou dynamiquement.

On peut encapsuler l'accès aux propriétés dans des fonctions et ainsi donner l'illusion d'un champ lu ou écrit. C'est ce qu'illustre le code de la figure 11 où la variable enclose **myage** ne peut être accédée que par la propriété **age**.

```

var personl = (function (myage) {
  return {
    get fullname () {
      return `${personl.firstname} ${personl.lastname}`;
    },
    get age () {
      return myage;
    },
    set age (n) {
      myage = n;
    }
  };
})(13);
personl.age === 13 // est vrai
personl.age = 18;
personl.age === 18 // est vrai

```

Figure 11 – Encapsulation de l'accès aux propriétés dans des fonctions

```

var o = { a: 1 };
Object.defineProperty(o, 'b', {
  configurable: false,
  enumerable: false,
  writable: true,
  value: 22
});
o.b === 22 // est vrai
o.b = 3;
'b' in o; // est vrai
for (let key in o) {
  key === 'a' || key === 'b' // est vrai
}
delete o.b; // erreur

```

Figure 12 – Définition des attributs des propriétés avec Object.defineProperty

Dans l'exemple de la figure 11, on voit la définition de deux lecteurs (pour *getter*) et un écrivain (pour *setter*). La propriété *fullName* n'ayant pas d'écrivain associé est donc non modifiable. Notez que la lecture ou l'écriture des propriétés déclenche l'invocation des fonctions associées mais respecte la syntaxe de lecture ou d'écriture des propriétés.

On peut aussi programmatiquement définir ou ajuster finement les attributs des propriétés avec *Object.defineProperty* (figure 12). Parmi ces attributs, on trouve :

- *configurable* est vrai si l'on peut encore ajuster les attributs de la propriété, voire la supprimer ;
- *writable* est vrai si l'on peut modifier la valeur que stocke la propriété ;

- *enumerable* indique si la propriété figure parmi celles listées par un *for (key in ...)* ... ou obtenues par la fonction *Object.keys*. Cet attribut ne modifie pas le comportement de l'opérateur *in* (qui permet de tester si une chaîne de caractères nomme l'une des propriétés d'un objet), ni le comportement de la fonction *Object.getOwnPropertyNames* qui retourne le tableau des noms des propriétés d'un objet.

Les attributs ont des valeurs par défaut lorsqu'ils ne sont pas spécifiés. On peut les inspecter avec la fonction *Object.getOwnPropertyDescriptor*.

3.3.2 Caractéristiques

Au-delà des attributs sur les propriétés, les objets eux-mêmes peuvent avoir des caractéristiques globales ajustables. Ainsi *Object.preventExtensions* prévient un objet d'acquérir de nouvelles propriétés (il n'empêche toutefois pas de perdre des propriétés). Cette caractéristique peut être testée par *Object.isExtensible* et, une fois posée, ne peut être défaite. Quelques autres caractéristiques existent décelées par les prédictats *Object.isSealed* ou *Object.isFrozen*.

Les objets bénéficient d'une notion d'héritage qui sera approfondie au § 6. Les objets possèdent généralement les méthodes que procurent la classe *Object* et notamment la méthode *toString*.

3.4 Tableaux

Les tableaux (figure 13) sont des structures de données indexées par des nombres. Un certain nombre de méthodes sont communes aux tableaux comme d'ajouter ou de retirer des objets, en tête ou en queue, afin d'implanter piles ou files. Les index d'un tableau débutent à zéro mais peuvent ne pas être contigus. Dans ce cas, la « taille » d'un tableau est le plus petit entier supérieur à

```

var t = [];
t.push('a'); // est vrai
t[0] === 'a'; // est vrai
t[2/2] === undefined; // est vrai
t.length === 1; // est vrai
t[1e2] = 3.14;
t.length === 101; // est vrai
t[100] === 3.14 // est vrai
t["100"] === 3.14 // est vrai
t["le2"] === undefined // est vrai
t.pop() === 3.14; // est vrai
t.length === 100; // est vrai
t['length'] === 100; // est vrai
t[2] = 2;
t.length = 2;
t[2] === undefined; // est vrai

```

Figure 13 – Exemple de tableau

tous les index présents dans le tableau. La taille est accessible, en lecture, via la propriété *length*. En écriture, elle permet de réduire la taille du tableau en supprimant tous les éléments d'index supérieurs ou égaux à cette taille.

De fait, les tableaux sont implantés par des objets. Un tableau est un objet doté d'une propriété spécifique : *length*. Les index des tableaux sont convertis en des chaînes de caractères. Du coup, les tableaux peuvent également détenir des propriétés mais ce n'est pas nécessairement raisonnable même si Javascript se l'autorise comme le montre le résultat de la méthode *match* sur une expression rationnelle.

En revanche, un objet n'est pas un tableau même s'il ne contient que des propriétés numériques. Un tableau est de type *object* mais de classe *Array* et on peut les reconnaître avec le prédictat spécialisé *Array.isArray*.

3.5 Fonctions

Une fonction (figure 14) est définie par un ensemble ordonné de variables et un corps, c'est-à-dire une série d'instructions. Le mot clé *return* permet d'imposer le résultat de la fonction.

Les fonctions sont anonymes : elles n'ont pas de nom aussi les stocke-t-on souvent dans des variables ou des tables. Les fonctions sont des citoyens de première classe, c'est-à-dire des valeurs manipulables comme toutes les autres valeurs de Javascript. C'est ce qui rapproche Javascript des langages fonctionnels comme Scheme ou ML. Les récentes évolutions stylistiques de Javascript comme celles induites par les bibliothèques JQuery ou lodash ainsi que la récente introduction des promesses montrent l'influence grandissante des techniques fonctionnelles en Javascript.

```

const min2 = function (a, b) {
  // rendre le plus petit de a et b
  if (a < b) {
    return a;
  } else {
    return b;
  }
};

```

Figure 14 – Exemple de fonction

Une graphie usuelle permet de raccourcir la précédente définition : le mot-clé **function** est encore utilisé (figure 15).

Javascript est, encore une fois, assez laxiste quant à l'invocation des fonctions. Bien que la fonction min2 soit binaire (ou ait une arité de 2 ce que l'on peut d'ailleurs obtenir avec min2.length) on peut l'invoquer avec un nombre quelconque d'arguments. Les arguments manquants sont considérés comme undefined.

Afin qu'une fonction puisse recevoir un nombre quelconque d'arguments, il existe un mot-clé **arguments** permettant l'accès à la séquence de tous les arguments (figure 16). La valeur d'arguments est presqu'un tableau et s'accède similairement. Dans l'exemple de la figure 16, on utilise la boucle **for of** qui permet d'itérer sur tout objet itérable.

Existe aussi (depuis ECMAScript 2015) la possibilité de définir une fonction rassemblant ses arguments d'appel en un tableau (figure 17a).

Inversement, il existe aussi la possibilité d'éclater un itérable en une séquence de valeurs (figure 17b)

```
function min2(a, b) {
    // rendre le plus petit de a et b
    if (a < b) {
        return a;
    } else {
        return b;
    }
}
```

Figure 15 – Autre syntaxe pour la définition d'une fonction

```
function sum() {
    // rendre la somme de tous les arguments
    var result = 0;
    //for (var i=0; i<arguments.length; i++) {
    //    result += arguments[i];
    //}
    for (let arg of arguments) {
        result += arg;
    }
    return result;
}
0 === sum()
10 === sum(1, 2, 3, 4)
```

Figure 16 – Écriture d'une fonction avec le mot-clé **arguments**

```
function prefixAll(mot, ...mots) {
    return mots.map(m => mot + m);
}
prefixAll("J'aime pas ", 'les schtroumpfs-bouchons',
          'les tire-bouschtroumpfs');
(a)
let quelquesMots = [ 'les schtroumpfs-bouchons',
                      'les tire-bouschtroumpfs' ];
prefixAll("J'aime pas ", ...quelquesMots);
(b)
```

Figure 17 – Écriture d'une fonction avec arguments en tableau (a) et d'un itérable en une séquence de valeurs (b)

```
var plus1 = (x) => x + 1;
var plus2 = x => x + 2;
var plus = (x, y) => { return x + y; };
```

Figure 18 – Graphie ECMAScript 2015 avec une flèche =>

```
function f(x = 0, y = 1) {
    return x + y;
}
f(1, 2) === 3 // est vrai
f(1) === 2 // est vrai
f() === 1 // est vrai
```

Figure 19 – Initialisation explicite de variables manquantes

Une nouvelle graphie a été introduite par ECMAScript 2015 à l'aide de la flèche => (figure 18). À gauche de la flèche sont les variables (les parenthèses peuvent être omises s'il n'y a qu'une variable). À droite de la flèche se trouve le corps de la fonction. Si le corps est une expression, le mot-clé **return** est implicite. Les fonctions définies avec la flèche n'ont pas de **this** propre (cf. § 6.2), elles utilisent le **this** environnant.

Une dernière caractéristique des fonctions est de permettre d'initialiser explicitement des variables manquantes (figure 19).

3.5.1 Portée

Définir une fonction est possible à l'intérieur d'une fonction (figure 20).

Les variables d'une fonction ont pour portée le corps de cette fonction : elles ne peuvent être utilisées que dans le corps de la fonction. La **portée** d'une variable (ou plus généralement d'une définition) est donc la zone de texte où l'on peut utiliser cette variable (respectivement, cette définition). Inversement l'**environnement** présent en un point de programme est l'ensemble des variables (ou définitions) dont la portée contient ce point : ce sont donc les variables ou définitions utilisables en ce point.

Les variables des fonctions internes masquent les variables externes de mêmes noms. Ainsi, dans l'exemple de la figure 20, les variables de min2 masquent-elles les variables de même nom de min4.

Dans l'exemple de la figure 21, la fonction interne walk, qui parcourt un arbre pour rechercher s'il contient des étiquettes supérieures à une étiquette donnée, profite de la portée de la variable externe tag et évite ainsi de l'avoir comme variable supplémentaire explicite de la fonction walk.

```
function min4(a, b, c, d) {
    // rendre le plus petit de a, b, c et d
    function min2(a, b) {
        if (a < b) {
            return a;
        } else {
            return b;
        }
    }
    return min2(min2(a, b), min2(c, d));
}
```

Figure 20 – Définition d'une fonction dans une fonction

```

var tree = {
    tag: 'a',
    left: {
        tag: 'b',
        left: {
            tag: 'c',
        },
        right: {
            tag: 'd'
        }
    },
    right: {
        tag: 'e',
        right: {
            tag: 'f'
        }
    }
}

function hasGreaterTag(tag, tree) {
    // prédictat vérifiant que l'arbre tree possède des
    // étiquettes supérieures à tag.
    const walk = function (tree) {
        if (tree) {
            if (tree.tag && tree.tag > tag) {
                return true;
            } else {
                return walk(tree.left) || walk(tree.right);
            }
        } else {
            return false;
        }
    }
    return walk(tree);
}

```

Figure 21 – Utilisation des variables externes et internes

La directive `var` permet de déclarer une variable locale à la fonction qui contient cette directive. La portée des variables n'est donc pas liée aux blocs (instructions enserrées entre accolades) comme elle serait en C ou en Java. Ainsi, dans l'exemple précédent, la portée de la variable `walk` est l'ensemble du corps de la fonction `hasGreaterTag`.

À l'extérieur d'une fonction, la directive `var` permet de déclarer une variable globale ou plutôt moins locale (cf. § 5 pour la notion de variable globale).

Il suffit donc d'une directive `var` pour qu'une variable soit considérée comme locale même si cette directive apparaît après le premier emploi de cette variable ! C'est pourquoi les guides de style recommandent de placer les directives `var` définissant des variables locales ou globales avant les emplois de ces mêmes variables (figure 22).

Pour éviter l'abomination précédente, il est préférable d'utiliser le mode strict de Javascript (obtenu en plaçant 'use strict' ; en première position dans le corps d'une fonction) qui permet de sanctionner certaines de ces malhabiletés et notamment de les transformer en des erreurs explicitement signalées.

Une autre solution est d'utiliser la directive `let` (ou `const`) qui réduit la portée des variables mentionnées au bloc englobant (`const` spécifie aussi que la variable ne peut pas être modifiée). Là encore, les guides de style recommandent de placer ces directives avant les emplois des variables concernées. On trouvera des exemples plus bas.

```

var g = 1;
function abominable1(a) { // NE PAS IMITER!
    g = 2; // Non! g n'est pas la variable globale!
    if (a) {
        var g = 3;
    }
    return g; // rend 3 si a est vrai sinon 2
}
g === 1 // toujours vrai

```

Figure 22 – Utilisation des directions `var`

3.5.2 Fermeture

Comme pour tous les langages fonctionnels, lorsqu'une fonction est créée, elle capture son environnement de définition. C'est une **fermeture** puisqu'elle enferme son environnement de définition. Les références aux variables que son corps contient et qui ne sont pas ses propres variables (on parle alors de variables libres) sont liées comme elles l'étaient lors de la définition de la fonction. Cette caractéristique ne survient que parce que des fonctions peuvent apparaître à l'intérieur d'autres fonctions ce qui n'est, par exemple, pas possible en C.

Voici un exemple (figure 23) où `jaimepas` est une fermeture, c'est-à-dire, dans ce cas précis, une fonction unaire qui attend une chaîne de caractères (avec sa variable propre `s2`) et qui la préfixe avec la chaîne "J'aime pas" (capturée dans `s1`). Cette nouvelle fonction `jaimepas` peut être utilisée autant que de besoin. On remarquera que `jaimepas` est le résultat d'un calcul, d'où l'emploi de la directive `var` pour la définir, la directive `function` ne pouvant être utilisée.

Ce mécanisme permet de préserver le secret et l'inaltérabilité des captures effectuées : il n'est pas possible d'extraire `s1` de la fermeture `jaimepas`. Ce mécanisme est notamment utilisé pour supporter la notion de module : un ensemble de ressources dont on ne laisse connaître que celles composant l'interface. La figure 24 présente une façon de définir un « module du pauvre » exportant une fonction `f` définie à l'aide des données `data` et des fonctions utilitaires `helper1` et `helper2`, toutes cachées de la vue des utilisateurs de `f` puisque toutes locales à une fonction anonyme immédiatement invoquée.

```

function prefix(s1) {
    return function (s2) {
        return `${s1} ${s2}`;
    };
}
var jaimepas = prefix("J'aime pas");
jaimepas("les schtroumpfs-bouchons");
jaimepas("les tire-bouschtroumpfs");

```

Figure 23 – Exemple de fermeture

```

var f = (function () {
    var data = ...;
    function helper1 () {...}
    var helper2 = function () {...}
    return function (...) {...};
})();

```

Figure 24 – Exemple de « module du pauvre »

```

function makeCounter (start) {
  var counter = start;
  return {
    get: function () {
      return ++counter;
    },
    reset: function () {
      return (counter = start);
    }
  };
}

var counter1 = makeCounter(10);
counter1.get(); // 11
var counter2 = makeCounter(100);
counter1.get(); // 12
counter2.get(); // 101
counter1.reset(); // 10
counter1.get(); // 11

```

Figure 25 – Exemple de lecture et modification d'une variable capturée

```

function makeClosures (end) {
  var closures = [];
  for (var i=1; i<end; i++) {
    closures[i] = function () {
      return i;
    };
  }
  return closures;
}

var closures = makeClosures(10);
closures[0]() === 10 // est vrai!
closures[3]() === 10 // est vrai!

```

Figure 26 – Erreur classique dans la capture de liaisons

Cet idiotisme (function () {...})() est très utilisé pour éviter, par exemple, des conflits de noms lorsqu'une page Web charge de nombreuses bibliothèques. Les greffons (pour *plugins*) de JQuery utilisent beaucoup cette technique.

Les exemples précédents pourraient laisser penser qu'une fermeture capture les valeurs des variables libres au moment de sa création. Il n'en est rien, une fermeture capture les liaisons qu'avaient les variables libres au moment de sa création (figure 25).

Lorsque la fonction makeCounter est invoquée, deux nouvelles liaisons sont créées, la variable start est liée au premier argument de l'invocation, la variable locale counter est liée à cette même valeur. C'est cette liaison qui est capturée et partagée par les deux fonctions get (qui la lit) et reset (qui l'écrit).

Comprendre la capture des liaisons permet d'expliquer l'erreur classique de la figure 26.

La variable i est locale à la fonction makeClosures. Ainsi, toutes les fonctions contenues dans le tableau closures capturent la même liaison ! Et la valeur finale que possède la variable i est 10. Afin que chaque fermeture retourne ce qu'il faut, on peut par exemple s'arranger pour créer de nouvelles variables que l'on pourra capturer (figure 27) ou plus simplement, en ECMAScript

```

function makeClosures (end) {
  var closures = [];
  for (var i=0; i<end; i++) {
    closures[i] = (function (ii) {
      return function () {
        return ii;
      };
    })(i);
  }
  return closures;
}

var closures = makeClosures(10);
closures[0]() === 10 // est vrai!
closures[3]() === 10 // est vrai!

```

Figure 27 – Définition correcte de capture de liaisons

```

function makeClosures (end) {
  var closures = [];
  for (let i=0; i<end; i++) {
    closures[i] = function () {
      return i;
    };
  }
  return closures;
}

var closures = makeClosures(10);
closures[0]() === 0 // est vrai!
closures[3]() === 3 // est vrai!

```

Figure 28 – Définition correcte de capture de liaisons avec ECMAScript 2015

2015, avec un let qui limite la portée de la variable ii au corps de la boucle for (figure 28).

3.5.3 Fonction et objet

Les fonctions sont aussi des objets ! Ce sont des objets invocables de la classe **Function** et dotés des méthodes de cette classe. La méthode length rend le nombre de variables de la fonction : son arité naturelle donc.

Dans l'exemple de la figure 29, des valeurs (fantaisistes) par défaut sont stockées dans la propriété mydefault de la fonction.

```

function foo(x, y) {
  x = x || foo.mydefault.x;
  y = y || foo.mydefault.y;
  return x * y;
}

foo.mydefault = {
  x: 1,
  y: 0,
  nickname: 'multiplication'
}

```

Figure 29 – Exemple de stockage de valeurs fantaisistes

4. Instructions

Les instructions dont dispose Javascript sont assez classiques. Les instructions comportent les opérations et les structures de contrôle.

4.1 Opérateurs

De manière générale, on retrouve en Javascript, les opérateurs classiques de C ou Java. En voici quelques-uns :

<code>+ - * / % ++ --</code>	<code>// arithmétique</code>
<code>< <= == != === != > = >=</code>	<code>// comparaison</code>
<code>! && </code>	<code>// booléens</code>
<code>~ <<>> >> & </code>	<code>// chaînes de bits</code>

Attention à l'addition qui, comme en Java, lorsqu'un de ses opérandes n'est pas un nombre mais convertible en chaîne de caractères, concatène plutôt qu'ajoute. Les opérateurs sur chaînes de bits utilisent des entiers (de 32 bits) pour représenter ces chaînes de bits.

À ces opérateurs s'ajoutent :

<code>delete</code>	<code>in</code>	<code>typeof</code>	<code>void</code>	<code>new</code>	<code>instanceof</code>
---------------------	-----------------	---------------------	-------------------	------------------	-------------------------

`delete` supprime une propriété d'un objet. `in` teste si une propriété existe dans un objet. `typeof` renvoie le type d'une valeur. Les types possibles sont `number`, `string`, `symbol`, `boolean`, `object`, `function` et `undefined`. `void` agit comme une fonction qui prend une valeur quelconque et rend `undefined`.

Un objet créé par `new` a une classe que l'on peut vérifier avec `instanceof` (§ 6).

4.2 Contrôle

Les structures de contrôle de Javascript sont celles de tous les langages modernes de haut niveau. Elles ont une syntaxe héritée de C et de Java. Le code de la figure 30 adopte la variante syntaxique utilisant systématiquement des blocs enserrés entre accolades.

Toutes ces instructions peuvent être préfixées par un label (un identificateur suivi de deux points). Ce label est surtout utile pour les boucles imbriquées car les instructions `break` et `continue` peuvent prendre un label pour spécifier de quelle boucle l'on sort ou quelle boucle on reprend.

L'instruction `for in` permet d'itérer sur les propriétés énumérables d'un objet tandis que `for of` itère sur les valeurs d'un itérable. Sont itérables les tableaux, les chaînes de caractères, les ensembles, les générateurs et, plus généralement, les objets pourvus d'une méthode d'itération.

La signalisation des exceptions s'effectue avec `throw`. Une expression peut donc s'achever soit en rendant une valeur, soit en signalant une exception. Les exceptions signalées lors du calcul d'une instruction `try` peuvent être rattrapées avec la clause `catch` associée. L'exception signalée est alors liée à la variable de la clause `catch` et les instructions associées sont exécutées. À la fin, que le corps de `try` rende une valeur ou signale une exception traitée par `catch`, les instructions de la clause `finally` sont exécutées.

```

if( condition ) {
    alors
} else if( condition2 ) {
    alors2
} else {
    sinon
}

switch ( valeur ) {
    case cas1: {
        alors1
    }
    case cas2: {
        alors2
    }
    default: {
        sinon
    }
}

while ( condition ) {
    corps
}

do {
    corps
} while ( condition );

for ( initialisation ; condition ; incrementation ) {
    corps
}

for ( variable in objet ) {
    corps
}

for ( variable of objet ) {
    corps
}

try {
    corps
    catch (variable) {
        instructions
    } finally {
        instructions
    }
}

```

Figure 30 – Structure de contrôle Javascript

Il est bien sûr possible, tant dans `catch` que dans `finally`, de signaler une exception avec `throw`.

Il existe une instruction `with` décrite, inefficace et donc à éviter.

5. Environnement global

Jusqu'ici, ont été exposés de Javascript les mots-clés, quelques fonctions essentielles et les notions de portée et d'environnement. Quel est l'environnement global et que contient-il ? La réponse dépend de la mise en œuvre particulière de Javascript.

Dans un navigateur, Javascript s'exécute dans un contexte restreint pour des raisons de sécurité. En effet, autoriser Javascript dans son navigateur, c'est permettre à du code écrit l'on ne sait où de s'exécuter dans son navigateur et donc sur sa machine. On ne cite plus les attaques où ces codes pouvaient dérober des

```
var d = '2016jan01';
d === window.d           // est vrai
window.d === window[d]   // est vrai
window.window === window // est vrai
```

Figure 31 – Définition d'une variable globale

fichiers sensibles, envoyer du spam, inspecter le réseau interne d'une société ou espionner les autres codes Javascript afin de trouver des numéros de cartes bancaires ou des mots de passe. Les navigateurs ont fait beaucoup d'efforts pour limiter ces risques et, en conséquence, procurent des bibliothèques restreintes en fonctionnalités : il n'est ainsi pas possible qu'un programme Javascript s'exécutant dans un navigateur puisse lire, sans restriction, le disque dur de l'utilisateur.

Dans cette mise en œuvre, chaque fenêtre du navigateur procure un environnement global au code Javascript qu'il héberge. Cet environnement est accessible, sous la forme d'une table associative, valeur de la variable `window`. Ainsi, chaque variable globale est aussi vue comme une propriété de l'environnement global (figure 31).

Dans l'environnement global d'un navigateur est aussi exposé le contenu de la page web : le **DOM** (pour *Document Object Model*) sous la forme d'un arbre réifié que l'on peut non seulement inspecter mais aussi modifier en provoquant ainsi des changements d'affichage. Certains fonctionnalités du navigateur sont aussi rendues disponibles comme créer de nouvelles fenêtres, modifier leur aspect, etc.

Mais Javascript est aussi un langage de programmation comme C ou Java et, à ce titre, procure l'ensemble des bibliothèques usuelles permettant de lire ou d'écrire des fichiers, d'interagir avec des serveurs, des bases de données, etc. Node.js est une telle mise en œuvre dont le succès a permis d'écrire des serveurs en Javascript. L'environnement global se nomme `global` en Node.js.

ECMAScript 2015 a défini un nouveau système de modules dont les relations sont réglées avec de nouveaux mots-clés : `import` et `export`. Modules et fichiers sont liés, le nom d'un module étant dérivé du nom du fichier qui le contient. Répertoires et URL sont mis à profit pour la recherche de modules à la discréption de l'évaluateur utilisé. Cette nouvelle sorte de module n'est pas encore répandue, aussi allons-nous exposer ce que sont les modules anciens et ce que seront les modules d'ECMAScript 2015 lorsque disponibles.

5.1 Anciens modules

Plusieurs définitions de modules co-existent : AMD (*Asynchronous Module Definition*), CommonJS et UMD (*Universal Module Definition*) qui les unifie. Plutôt que de les passer tous en revue, car ils sont en voie d'obsolescence, nous allons nous intéresser aux seuls modules de Node.js inspirés de CommonJS. Ces modules peuvent être chargés dynamiquement. Ils définissent leurs dépendances, c'est-à-dire les modules qu'ils vont requérir (avec la fonction `require`) ainsi que ce qu'ils vont exporter sous la forme, bien sûr, de table associative.

La figure 32a présente un module minimalist (et fantaisiste). On y trouve l'importation du module `util` que procure Node.js au sein de la variable `util`. La ligne suivante (ré-)exporte une valeur sous la forme de la propriété `dump` de la table associative des exports. On exporte également la valeur de la variable `pi` ainsi qu'une fonction modifiant (*sic*) la valeur de `pi`. Dans ce module, `util` et `pi` sont des variables globales pour ce module. On peut alors lancer Node.js avec les instructions de la figure 32b.

```
// Module testmodule1.js
var util = require('util');
exports.dump = util.inspect;
var pi = exports.pi = 3.14;
exports.setpi = function (v) {
    return (pi = v);
};
```

(a)

```
var tm = require('./testmodule1.js');
tm.pi === 3.14 // est vrai
tm.util === undefined // util non exporté
tm.setpi(3.1415926535);
tm.pi === 3.14 // est vrai
```

(b)

Figure 32 – Importation de module `util` (a) et utilisation avec Node.js

Dans ce fragment de programme, on charge le fichier `testmodule1.js` dans la variable toujours globale `tm`. Après avoir vérifié la valeur de la propriété `tm.pi`, on vérifie que la variable `util` du module `testmodule1` n'a pas été exportée.

Dans les deux dernières lignes, enfin, on utilise la fonction exportée `setpi`. Cette fonction est une fermeture ayant capturé la variable `pi` du module `testmodule1`. C'est cette variable qui est donc modifiée mais cela ne change pas la valeur de la propriété `pi` de la table associative `exports`. De fait, le module `testmodule1` est traité comme s'il était enclos dans une construction du genre `(function (module, exports) {...})()` ce qui assure la protection des variables locales à cette fonction anonyme bien que ces variables soient globales vues du module. Dans cette fonction anonyme, `module` est l'objet représentant le module (avec des propriétés comme son nom, ses dépendances, ses exportations, etc.) et `exports` est la table associative qui sera remplie avec les exportations du module. Ces deux objets sont alloués par la fonction `require` et, au début, `module.exports === exports`.

La notion de module introduite côté serveur a son équivalent côté client. Les modules permettent de discipliner les portées des variables et de régler finement les partages de valeurs. En résumé, si les fermetures capturent des liaisons, ces anciens modules ne font qu'exporter des valeurs.

5.2 Modules d'ECMAScript 2015

Les anciens modules sont par trop dynamiques, ils règlent mal le problème des modules cycliques. CommonJS charge les modules de façon synchrone tandis qu'AMD les charge de façon asynchrone ce qui paraît plus approprié à la récupération de fichiers possiblement fort distants. ECMAScript 2015 propose une synthèse de CommonJS et AMD avec de nouveaux mots-clés `import` et `export`. Les modules peuvent être chargés de façon synchrone ou asynchrone, ce sont les liaisons qui sont exportées et non plus les valeurs, on peut renommer les liaisons à l'export ou à l'import, on peut les réexporter, etc.

La figure 33a présente une solution pour écrire le module précédent. Ici figurent plusieurs manières d'importer le module `util` et de nommer la fonction `inspect`.

Et la figure 33b illustre une importation de ce module en opérant une sélection ou en renommant une importation.

ECMAScript 2015 ne spécifie pas comment les modules sont chargés (de façon synchrone ou non). En revanche, les nouveaux mots-clés permettent d'organiser statiquement les partages des

```
// Module testmodule2.js
//import u from "util";
import * as util from "util";
//import {inspect} from "util";
//import {inspect as inspector} from "util";

export let dump = util.inspect;
// u.inspect === util.inspect === inspect === inspector
export let pi = 3.14;
export let setpi = function (v) {
    return (pi = v);
};
```

(a)

```
import {pi, setpi} from "./testmodule2.js";
import {setpi as setPI} from "./testmodule2.js";
import * as tm from "./testmodule2.js";

pi === 3.14 && tm.pi === pi // est vrai
setPI(3.1415926535);
pi === 3.1415926535 // est vrai
```

(b)

Figure 33 – Réécriture du module de la figure 31 avec ECMAScript 2015 (a) et importation avec Node.js (b)

liaisons et les noms qui permettent d'y accéder. Pour utiliser ces nouvelles notations, il faut les compiler d'ECMAScript 2015 en vieux Javascript avec, par exemple, babel.js.

6. Héritage et prototypes

Javascript est un langage à prototypes, il procure donc des objets mais, nativement, pas de classe. La différence entre langages à prototypes et langages à classes tient au mécanisme de création des objets. Dans les langages à classes, tous les objets d'une même classe ont la même structure (les mêmes champs) et les mêmes comportements (méthodes). Dans les langages à prototypes, les objets sont créés par clonage. Un clone acquiert initialement toutes les propriétés et tous les comportements de l'objet souche (en Javascript, on parle de **prototype**). Il peut évoluer indépendamment de son prototype et acquérir (ou perdre) dynamiquement de nouveaux comportements ou propriétés.

Nous allons présenter dans un premier temps les prototypes, concept sur lequel ECMAScript 2015 a bâti un système de classes associé à de nouvelles syntaxes pour leur déclaration.

Tout objet (y compris table, fonction, expression rationnelle, etc.) possède un prototype que l'on peut obtenir via `Object.getPrototypeOf` et modifier via `Object.setPrototypeOf`.

6.1 Propriétés

Lire une propriété consiste à explorer l'objet vu comme une table associative. Si la propriété n'y est pas, alors l'exploration continue avec le prototype puis avec le prototype du prototype et ainsi de suite tant que le prototype n'est pas `null`. On peut donc dire qu'un objet hérite de son prototype.

En revanche, modifier une propriété s'effectue dans l'objet. Une propriété modifiée masque donc la propriété qui aurait éventuellement été héritée (figure 34).

```
var o1 = { a: 1 };
var o2 = {};
Object.setPrototypeOf(o2, o1);
o2.a === 1 // est vrai
o2.a = 2;
o2.a === 2 // est vrai
o1.a === 1 // est toujours vrai
```

Figure 34 – Modification d'une propriété

Le chaînage des prototypes fait que l'on distingue les propriétés propres à un objet de celles qui sont héritées via ses prototypes. La fonction `Object.hasOwnProperty` permet de vérifier si un objet `a`, en propre, une certaine propriété. La fonction `Object.getOwnPropertyNames` permet de lister toutes les propriétés (énumérables ou pas) contenues en propre, par un objet alors que `for (var name in object) ...` liste toutes les propriétés énumérables que l'objet possède en propre ou pas. Enfin, `Object.keys` ne liste que les propriétés énumérables que l'objet possède seulement en propre.

6.2 Comportements

Une propriété dont la valeur est une fonction peut être invoquée comme une méthode et alors accéder à l'objet sur lequel elle est appliquée grâce au mot-clé `this`. La syntaxe (figure 35a) pour invoquer une méthode est la syntaxe usuelle des langages à objets : la notation pointée où l'objet figure en tant qu'expression avant le point et où le nom de la méthode à invoquer figure après le point.

La fonction `getName` est destinée à être utilisée comme une méthode car elle contient une référence à `this`. L'expression `getName()` où l'on invoque la fonction `getName` sans indiquer explicitement quelle sera la valeur de `this` a toutes les chances de ne pas faire ce que l'on attend d'elle. De fait, toute fonction définie par `function` introduit un nouveau `this` dont la valeur est définie par la forme de l'appel (fonction ou méthode). En revanche, une fonction définie par la flèche `=>` n'introduit pas de nouveau `this` et peut donc utiliser le `this` environnant.

```
var getName = function (prefix) {
    return `${prefix} ${this.name}`;
};
var o1 = {
    name: 'moi',
    getName: getName
};
o1.getName('hello ') === 'hello moi' // est vrai
var o2 = {};
Object.setPrototypeOf(o2, o1);
o2['get' + Name]('hello ') === 'hello moi' // est vrai
```

(a)

```
var o3 = { name: 'lui' };
getName.call(o3, 'bonjour ') === 'bonjour lui' // est vrai
getName.apply(o3, ['bonsoir ']) === 'bonsoir lui' // est vrai
```

(b)

Figure 35 – Syntaxe d'une propriété (a) et utilisation de la méthode particulière call

```

function Point(x, y) {
    this.x = x;
    this.y = y;
    this.d = function () {
        return this.x + this.y;
    }
}
var p1 = new Point(1, 2);
p1.x === 1 // est vrai
p1.y += 2;
p1.y === 4 // est vrai
p1.d() === 5 // est vrai
p1.constructor === Point // est vrai
p1 instanceof Point // est vrai

```

Figure 36 – Mécanisme de constructeur

Attention, `this` est un mot-clé, ce n'est pas une variable ! En conséquence, il ne pourrait pas être capturé par une fermeture. Une méthode des fonctions existe, nommée `call`, qui permet d'invoquer une fonction tout en spécifiant la valeur du `this` à prendre en compte (figure 35b).

Est aussi illustrée dans l'exemple de la figure 35, une variante de `call`, nommée `apply`, qui permet de calculer l'ensemble des arguments (réunis dans un tableau) avec lesquels invoquer la fonction.

6.3 Constructeurs

Le mot-clé `typeof` est peu discriminant, le mécanisme de constructeur permet de discriminer plus finement les objets. Un **constructeur** est une fonction (dont le nom débute traditionnellement par une majuscule) destinée à être utilisée avec le mot-clé `new` (figure 36).

L'exemple de la figure 36 illustre la création d'un point, la lecture et l'écriture de ses propriétés `x` et `y` ainsi que l'usage d'une méthode `d` calculant la distance de Manhattan du point par rapport à l'origine.

Notez que, comme en Java, le nom de « constructeur » est usurpé car c'est le mot-clé `new` qui alloue un `Point`, point qui est ensuite initialisé par la fonction `Point`. Cette fonction est mémorisée par la propriété spéciale `constructor`.

Les deux dernières lignes démontrent deux nouvelles caractéristiques. D'une part, une nouvelle propriété, `constructor`, permet de découvrir le constructeur de l'objet.

Le mécanisme de constructeur donne l'illusion de classes : on parlera donc de la classe `Point`. Les instances de la classe sont créées par `new`, l'appartenance à une classe se vérifie avec `instanceof`, méthodes et champs sont disponibles grâce aux propriétés. Il ne manque plus que la notion d'héritage.

6.4 Factorisation

Dans l'exemple de la figure 36, chaque instance de la classe `Point` possédait sa propre méthode `d` puisqu'à chaque construction était recréée une nouvelle fonction anonyme stockée dans la propriété `d` : on pourrait même parler de « **méthode d'instance** ». C'est un gâchis de place qu'un usage raisonnable du mécanisme de prototype permettrait d'éviter.

La construction est un mécanisme plus complexe qu'habituellement perçu. Lorsque la construction d'un premier point est exigée, un objet est alloué et stocké dans la propriété `prototype` de la fonction `Point` (vu comme un objet invoquable). Cet objet devien-

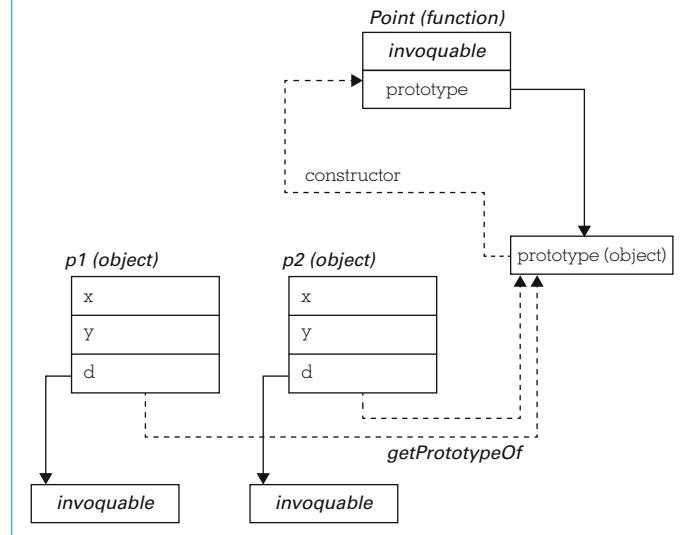


Figure 37 – Crédit de deux instances de Point, leur constructeur et leur prototype.

```

Point.prototype.tg = function () {
    return this.y / this.x;
};
p1.tg() === 4 // est vrai

```

Figure 38 – Définition d'une méthode de classe

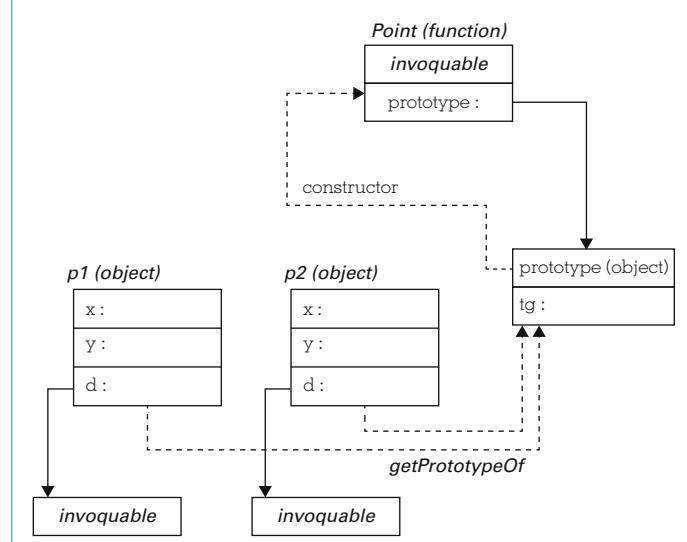


Figure 39 – Méthode tg de classe

dra le prototype commun à tous les points qui seront créés comme indiqué dans la figure 37.

Pour ajouter une « **méthode de classe** » c'est-à-dire une méthode disponible pour toutes les instances d'une classe, il suffit de l'ajouter au prototype commun à toutes ces instances (figures 38 et 39).

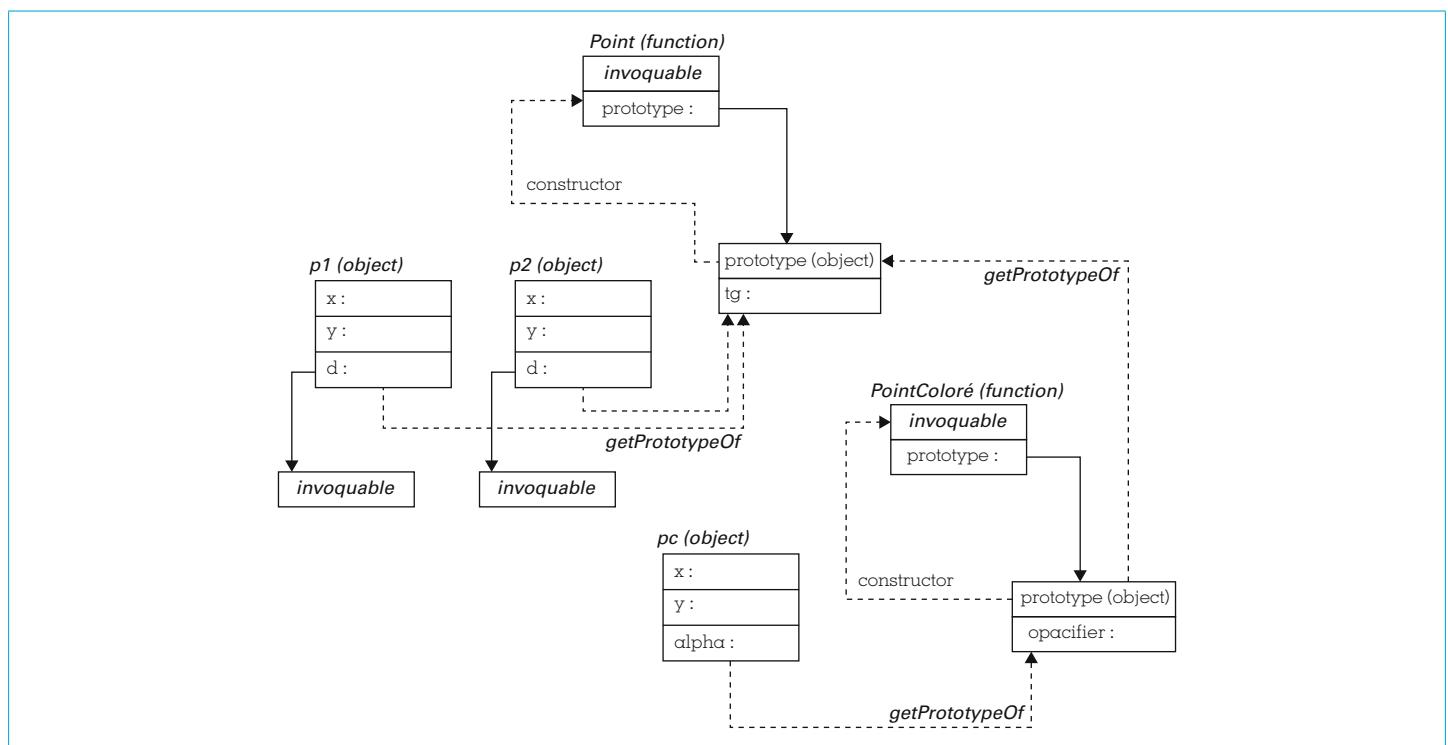


Figure 40 – Sous-classe des points colorés

6.5 Héritage

L'héritage de classe peut maintenant être réalisé au moyen d'un chaînage de prototypes. Supposons définir une sous-classe de Point que l'on nommera PointColoré. Une instance de PointColoré aura une opacité définie par une propriété alpha et une méthode de classe opacifier qui augmente la propriété alpha (figure 40 et 41a).

On remarquera qu'un point coloré n'a pas accès à la propriété d des points car d est une méthode d'instance et non de classe. On remarquera aussi que, dans l'exemple précédent, l'initialisation d'un point coloré n'utilise pas le constructeur des points : il n'y a pas d'équivalent à super dans cette simulation de classes. On peut toutefois écrire et obtenir le même effet avec le code de la figure 41b (à condition de ne pas avoir modifié les valeurs des propriétés constructor).

La graphie Point.prototype.constructor.call (**this**, x, y) (en commentaire dans le code de la figure 41b) est plus directe mais elle mentionne la super-classe Point. On pourra lui préférer les deux lignes suivantes (utilisant la variable locale ogt qui retrouve le constructeur de la super-classe sans présumer de son nom).

6.6 Hiérarchie

Les classes prédéfinies sont hiérarchiquement organisées. Tout en haut se trouve Object fournissant de nombreuses méthodes comme toString et de nombreuses fonctions comme Object.getPrototypeOf. Sous Object se trouvent la plupart des classes prédéfinies comme Array, RegExp, Date, etc. Se trouvent également les classes utilitaires introduites par ECMAScript 2015 : Set, Map, Symbol, Proxy, etc.

```

a
function PointColoré (x, y, alpha) {
  this.x = x;
  this.y = y;
  this.alpha = alpha;
}
Object.setPrototypeOf(PointColoré.prototype, Point.prototype);
PointColoré.prototype.opacifier = function () {
  this.alpha = Math.min(1, this.alpha * 1.1);
  return this;
}
var pc = new PointColoré(11, 22, 0.3);
pc instanceof PointColoré // est vrai
pc instanceof Point // est vrai
pc.tg() === 2 // est vrai
pc.d === undefined // est vrai

b
var PointColoré = function (x, y, alpha) {
  // Point.prototype.constructor.call(this, x, y);
  const ogt = Object.getPrototypeOf(this);
  Object.getPrototypeOf(ogt).constructor.call(this, x, y);
  this.alpha = alpha;
}
Object.setPrototypeOf(PointColoré.prototype, Point.prototype);
var pc2 = new PointColoré(3, 6, 0.4);
pc2 instanceof PointColoré // est vrai
pc2 instanceof Point // est vrai
pc2.tg() === 2 // est vrai
pc2.d() === 9 // est vrai

```

Figure 41 – Deux méthodes de définition de la sous-classe PointColoré

```

class Point {
  constructor (x, y) {
    this.x = x;
    this.y = y;
  }
  tg () {
    return this.y/this.x;
  }
}
class PointColoré extends Point {
  constructor (x, y, alpha) {
    super(x, y);
    this.alpha = alpha;
  }
  opacifier () {
    this.alpha = Math.min(1, this.alpha * 1.1);
    return this;
  }
  tg () {
    return 0 + super.tg();
  }
}

```

Figure 42 – Autre écriture de l'exemple des points colorés avec ECMAScript 2015

6.7 Syntaxes

ECMAScript 2015 introduit de nouveaux mots-clés et de nouvelles fonctionnalités permettant aux tenants des langages à objets de conserver leurs habitudes en Javascript. Ainsi l'exemple des points colorés (figure 41) pourrait s'écrire, à la Java, comme présenté à la figure 42.

La méthode `tg` de `PointColoré` est là juste pour montrer l'usage de `super` dans une méthode. D'un autre côté, la méthode d'instance `d` a été omise.

Il est aussi possible de placer des fonctions dans des classes avec le mot-clé `static`. Une caractéristique beaucoup plus intéressante est que la définition d'une classe est une expression tout comme l'est ce qui suit le mot-clé `extends` : on peut donc créer dynamiquement de nouvelles classes étendant des classes obtenues par calcul (figure 43).

Dans l'exemple de la figure 43, la fonction `AddAlphaChannel` prend une classe pour fabriquer une sous-classe dotée d'une méthode supplémentaire `opacifier`. La classe `PointColoré` est elle-même une nouvelle sous-classe de la précédente.

6.8 Conclusions

ECMAScript 2015 apporte de nouveaux mots-clés et de nouvelles fonctionnalités pour la définition de classes le rapprochant ainsi de Java. Hélas, à ce jour, aucun navigateur n'implante totalement ECMAScript 2015 et, au vu de la lenteur à laquelle les internautes mettent à jour leur navigateur, il faudra des années avant que ne tournent partout des programmes écrits en ECMAScript 2015. Il est donc nécessaire de savoir encore programmer avec des prototypes ou d'utiliser un compilateur d'ECMAScript 2015 vers du Javascript ancien comme Babel.js au prix d'une complexification de la chaîne de développement.

```

function AddAlphaChannel (klass) {
  return class extends klass {
    opacifier () {
      this.alpha = Math.min(1, this.alpha * 1.1);
      return this;
    }
  }
}
const Point = class Point {
  constructor (x, y) {
    this.x = x;
    this.y = y;
  }
  tg () {
    return this.y/this.x;
  }
}
class PointColoré extends AddAlphaChannel(Point) {
  constructor (x, y, alpha) {
    super(x, y);
    this.alpha = alpha;
  }
}
var pc = new PointColoré(1, 2, 0.5);
pc.x === 1 // est vrai
pc.tg() === 2 // est vrai
pc.opacifier().alpha === 0.55 // est vrai

```

Figure 43 – Création dynamique de nouvelles classes

7. Concurrence

Javascript est mono-tâche non préemptif. Ces mots signifient que l'évaluateur ne fait qu'une seule chose à la fois et qu'il ne fait que cette chose jusqu'à son achèvement. Deux grandes conséquences :

- chaque exécution peut se considérer comme en section critique puisque rien ne peut l'empêcher d'être menée à terme ;
- une boucle infinie ne se terminera donc jamais.

D'un autre côté, la bibliothèque d'exécution primitive de Javascript (par exemple, celle que fournissent les navigateurs) sait mener en parallèle des actions comme aller chercher des pages, des images, des scripts et les traiter de façon appropriée lorsque leur contenu a été obtenu. La communication entre ces deux mondes s'effectue par le biais d'événements et de suites (en jargon, *callback*). Le terme *callback* peut aussi être traduit par « fonction de renvoi » mais c'est moins léger que « suite » qui sera adopté ici. La littérature anglophone utilise souvent le terme de « continuation » qui, sémantiquement, n'a pas cette acception. À tout instant, la continuation représente l'ensemble des calculs restant à mener jusqu'à l'achèvement complet du programme. Une suite n'est qu'une fonction qui sera invoquée lorsque les conditions de son lancement seront réunies.

De façon générale, la bibliothèque d'exécution primitive propose, dès qu'une fonctionnalité prend un temps indéterminé pour s'achever, de lui procurer une interface asynchrone lui permettant de signaler, par des événements, les différents états auxquels elle aboutit. Ainsi, aller chercher le contenu d'une page connue par son URL peut signaler, au bout d'un certain temps, un parmi deux événements possibles : soit le bon achèvement de la requête, soit son échec (réseau inaccessible, mauvaise URL, page absente, mauvais format, etc.). Aux événements qui seront signalés, on peut associer des fonctions qui seront déclenchées lorsque surviendra l'événement, fonctions qui représentent donc la « suite » de ce qu'il faut faire.

```

fetch(url, function (data, error) {
  if ( data ) {
    // traiter la donnée
  } else {
    // traiter l'échec
  }
});

①

let urls = [url1, url2];
fetch(urls[0], function (data1, error1) {
  if ( data1 ) {
    fetch(urls[1], function (data2, error2) {
      if ( data2 ) {
        console.log(data1 + data2);
      } else {
        console.log('échec');
      }
    });
  } else {
    console.log('échec');
  }
});

```

②

```

let results = urls.map(item => undefined);
let failure = false;
urls.forEach(function (url, index) {
  fetch(url, function (data, error) {
    if ( failure ) {
      return;
    } else if ( data ) {
      results[index] = data;
      if ( results.every(data => !!data) ) {
        console.log(results.reduce((s1, s2) => s1 + s2, ""));
      }
    } else {
      failure = error;
      console.log('échec');
    }
  });
});

```

③

Figure 44 – Définition d'une fonction fetch (a) et mise en œuvre de façon « propre » séquentiellement (b) et en parallèle (c)

Le moteur d'exécution de Javascript est un ordonnanceur qui maintient à jour la liste des suites pouvant être exécutées et la liste des suites encore en attente d'un événement. Lorsqu'un événement survient, que l'ordonnanceur soit actif ou non, les suites en attente de cet événement migrent dans la liste des suites prêtes à être exécutées. Lorsque l'ordonnanceur ne fait rien, il choisit une suite parmi celles qui sont prêtes et l'exécute jusqu'à sa fin (tout en restant à l'écoute des événements pouvant survenir afin de tenir à jour la liste des suites prêtes à être exécutées).

7.1 Programmation par suites

La programmation par emploi de suites est particulière et nous allons l'illustrer avec un exemple que nous programmerons de différentes manières illustrant différents styles de programmation. Le but sera d'afficher la concaténation des contenus de deux URL. Nous supposerons disposer d'une fonction fetch (figure 44a).

La fonction fetch prend une URL, va chercher le contenu de la page référencée par cette URL et invoque la suite avec ce contenu en premier argument. S'il y a une anomalie dans la requête (réseau indisponible, serveur en panne, page manquante, etc.), alors la suite sera invoquée avec undefined en premier argument et une erreur en second argument.

La fonction fetch peut être aisément programmée avec JQuery.get côté navigateur ou, côté serveur, avec http.get. Nous avons donné à fetch l'allure d'une fonction asynchrone classique avec une suite binaire qui prend deux arguments et se comporte comme une suite de succès ou d'échecs suivant la nature des arguments.

La mise en œuvre de cette fonction et de sa suite demande à être bien comprise en termes d'environnement, de contexte et, notamment, de rattrapage d'erreur.

7.2 Séquence par enchaînement de suites

La figure 44b présente une première méthode qui consiste à aller séquentiellement chercher le premier contenu (phase 1 de figure 45) puis le second contenu (phase 2) et enfin de les concaténer et de les afficher (phase 3). La moindre erreur interrompt le calcul. On appréciera l'intérêt des fermetures imbriquées qui permet d'utiliser la variable data1 dans la seconde suite.

7.3 Concurrence avec suite similaire

Pourquoi séquentialiser ce que l'on pourrait paralléliser ? Cette seconde version (figure 44c) utilise le côté asynchrone de fetch pour aller chercher en parallèle les contenus à concaténer. Lorsque tous les contenus sont acquis, on peut les concaténer.

Dans cette seconde version, deux variables globales sont concernées : un vecteur pour stocker les contenus cherchés (initialisés à undefined) et un booléen signalant l'échec d'au moins un des appels à fetch. Les recherches des contenus sont lancées en parallèle et chaque suite doit regarder dans results si tous les résultats sont présents et si oui, les concaténer.

On remarquera dans le chronogramme potentiel de la figure 46 que le second appel à fetch est plus rapide que le premier et que la suite correspondant au premier appel doit attendre la fin de l'invocation de la suite du second appel pour s'exécuter. On remarquera également et contrairement à la première version que la taille du code ne dépend plus du nombre d'URL à aller chercher.

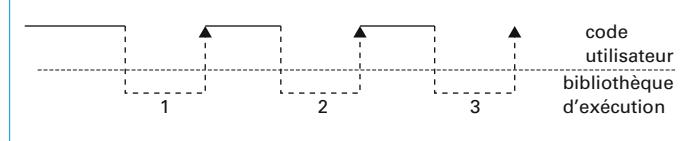


Figure 45 – version séquentielle

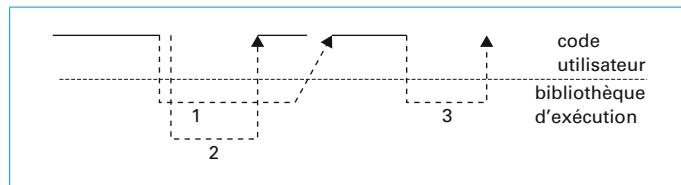


Figure 46 – version parallèle avec état global explicite

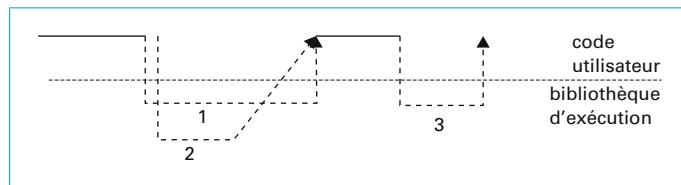


Figure 47 – version parallèle avec promesses

7.4 Promesses

La troisième version présentée pour ce problème récurrent utilise des promesses : un nouveau mécanisme introduit dans ECMAScript 2015, bien supérieur aux suites. Inspirée des techniques fonctionnelles des années 1980, une **promesse** est un objet qui démarre un calcul et auquel on peut attacher des conséquences : des fonctions qui seront déclenchées suivant que la promesse a été tenue ou non (figure 47). Les résultats de ces fonctions sont eux-mêmes des promesses ce qui permet de constituer un arbre de conséquences.

Le code de la figure 48 commence par créer un vecteur de promesses. Chacune de ces promesses est créée à partir d'un appel à fetch avec une suite qui rend le résultat conforme au comportement des promesses à savoir : invoquer la fonction resolve avec un résultat correct ou invoquer la fonction reject avec une représentation de l'erreur.

La classe Promise vient avec ses méthodes et notamment all qui prend un vecteur de promesses pour créer une nouvelle promesse qui sera en échec dès qu'une des promesses initiales est en échec. La conséquence du succès de toutes les promesses initiales est de fournir le vecteur des résultats. Les conséquences heureuses sont associées à une promesse avec la méthode then, les conséquences malheureuses le sont avec catch.

Cette version n'utilise plus aucune variable globale et ne demande plus à l'utilisateur de gérer la terminaison des différentes requêtes menées en parallèle puisque Promise.all s'en charge. Si d'ailleurs la fonction fetch avait été écrite sous forme d'une promesse plutôt que sous la forme d'une fonction avec suite, le code aurait été encore plus court !

7.5 Arbre de promesses

La dernière version ne rend pas justice aux promesses qui améliorent le rattrapage d'erreur. Pour illustrer ce point, supposons maintenant que l'on considère qu'une URL en erreur doit être assi-

```
function fetchPromise(url) {
  return new Promise(function(resolve, reject) {
    fetch(url, function(data, error) {
      if (data) {
        return resolve(data);
      } else {
        return reject(error);
      }
    });
  });
}

var promises = urls.map(fetchPromise);
Promise.all(promises)
.then(function(results) {
  console.log(results.reduce((s1, s2) => s1 + s2, ""));
})
.catch(function(error) {
  console.log('échec');
});
```

(a)

```
var promises = urls.map(url => {
  return fetchPromise(url).catch(error => "");
});
Promise.all(promises)
.then(function(results) {
  return results.reduce((s1, s2) => s1 + s2, "");
}).then(function(result) {
  console.log(result);
})
.catch(function(error) {
  console.log('échec');
});
```

(b)

Figure 48 – Création d'une promesse avec utilisation des méthodes then et catch (a) et en séparant calcul et affichage

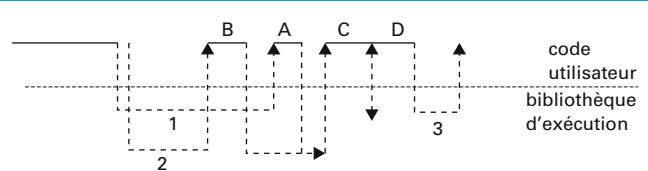


Figure 49 – Version parallèle avec arbre de promesses et leurs conséquences

milée à la chaîne vide. Séparons également calcul et affichage qui avaient été mélangés dans les versions précédentes (figure 48b).

Le chronogramme de la figure 49 montre un déroulement possible. Chaque promesse du vecteur promises a une suite d'erreur lui faisant renvoyer la chaîne vide en cas d'anomalie. Les suites devant renvoyer des promesses, on remarquera qu'une valeur peut être considérée comme une promesse déjà résolue qui activera donc sa suite de succès (ce sont les instants B et A de la figure 49 correspondant aux requêtes 1 et 2). Lorsque toutes les promesses du vecteur promises sont résolues prend place (C) la suite concaténant toutes les valeurs obtenues. Cette suite rend une valeur qui déclenche donc sa suite de succès qui enregistre que cette valeur doit être affichée (D). Plus aucune tâche ne restant à exécuter, la bibliothèque d'exécution peut mettre à jour l'affichage (3) et ainsi rendre visible le résultat final.

On remarquera dans ce dernier exemple, le tissage des suites d'échec et de succès sur les promesses permettant un code quasiment sans fermeture et d'imbrication réduite. Les promesses sont appelées à un bel avenir en ECMAScript 2015.

8. Générateur

Parmi les avancées d'ECMAScript 2015, on trouve les générateurs : des sortes de fonctions avec un état de contrôle persistant, pouvant être reprises plusieurs fois et rendre plusieurs résultats. Un générateur est défini avec un nouveau mot-clé : **function*** et il utilise le mot-clé **yield** pour interagir avec son appellant.

8.1 Générateur linéaire

Un premier exemple illustrant les pauses qui s'opèrent dans le générateur et les différentes reprises est présenté à la figure 50a.

C'est le premier appel à `next` qui démarre l'exécution du générateur qui s'arrête au premier `yield` et renvoie comme résultat un objet dont la propriété `value` est la valeur de l'expression qui suit `yield`. Lors du second appel à `next`, l'exécution du générateur reprend là où elle s'était arrêtée et s'interrompt au second `yield`. Le troisième appel à `next` prend un argument qui devient la valeur de l'expression `yield 2` et se trouve donc stockée dans la variable `trois`. Invoquer une quatrième fois `next` conduit à exécuter `return` ce qui interrompt le générateur ce que l'on peut vérifier à l'aide de la propriété `done`.

8.2 Générateur bouclant

On le voit, le générateur possède un état interne qui lui permet d'arrêter puis de reprendre son exécution. Un générateur peut ne pas être linéaire et placer des `yield` au sein de boucle infinie. La figure 50b présente un exemple qui calcule la suite de Fibonacci (une suite définie par $f_0 = f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$).

Dans l'exemple de la figure 50b, le premier appel `g.next()` exécute le corps du générateur jusqu'au premier `yield`. Le second appel `g.next(1)` demande le premier nombre de Fibonacci non encore calculé tandis que `g.next(3)` demande le troisième nombre de Fibonacci non encore calculé.

En encapsulant ce générateur dans une fonction, on peut donner l'illusion d'une fonction qui rend, tour à tour, les nombres de Fibonacci.

Les générateurs étendent les itérateurs et itérables (et le mot clé `yield*` permet d'adapter ces derniers en ces premiers).

8.3 Générateur et asynchronisme

Pouvoir pauser un générateur peut être combiné avec des promesses pour redonner une apparence linéaire et synchrone à des appels asynchrones. Reprenons l'exemple en section 7.5 (figure 48b) où l'on devait concaténer les contenus de deux pages Web tout en assimilant à la chaîne vide une page à laquelle on ne peut pas accéder. Voici ce code, écrit de façon linéaire, où l'on peut admirer que les appels à `fetch` Promise qui ne fonctionnent pas sont rattrapés par des `try catch` (figure 51a).

La clé de ce code tient à la fonction `process` qui prend en argument un générateur, le lance et attend de chaque `yield` qu'il ramène une promesse. Cette promesse est alors évaluée avec des

```
function* oneTwoThree (log) {
  log.push('bientôt 1');
  yield 1;
  log.push('bientôt 2');
  var trois = yield 2;
  log.push('enfin ${trois}');
  return yield trois;
}

var log = [];
var g = oneTwoThree(log);
log.length === 0           // est vrai
g.next().value === 1         // est vrai
log.length === 1             // est vrai
log[0] = 'bientôt 1'         // est vrai
g.next().value === 2         // est vrai
g.next(3).value === 3        // est vrai
g.next().done                // est vrai
```

(a)

```
function* fibgen () {
  var previous = 1;
  var current = 1;
  while ( true ) {
    var n = yield current || 0;
    while ( n-- > 0 ) {
      var next = previous + current;
      previous = current;
      current = next;
    }
  };
  var g = fibgen();
  let fib1 = g.next().value;
  fib1 === 1           // est vrai
  let fib2 = g.next(1).value;
  fib2 === 2           // est vrai
  let fib5 = g.next(3).value;
  fib5 === 8           // est vrai
  lg.next(1).done       // est vrai
```

(b)

```
const fib = (function (g) {
  return function () {
    return g.next(1).value;
  };
})(fibgen());
fib() === 1           // est vrai
fib() === 2           // est vrai
fib() === 3           // est vrai
```

(c)

Figure 50 – Utilisation d'un générateur : (a) illustration des pauses possibles (b) génération bouclant pour le calcul de la suite de Fibonacci (c) encapsulation du générateur dans une fonction

suites appropriées afin de reprendre l'exécution du générateur avec une valeur (avec `next`) ou en signalant une exception avec `throw`. Cette méthode permet de signaler une exception à l'endroit même du `yield` en pause ; le générateur reprend donc son exécution jusqu'à la prochaine pause ou terminaison. La figure 51b présente donc cette fonction `process`.

```

process(function* () {
  let a, b;
  try {
    a = yield fetchPromise('http://a.fr');
  } catch (exc) {
    a = '';
  }
  try {
    b = yield fetchPromise('http://b.fr');
  } catch (exc) {
    b = '';
  }
  let result = a + b;
  console.log(result);
});

```

(a)

```

function process(generator) {
  var g = generator();
  function throwError(error) {
    step(g.throw(error));
  }
  function resumeNext(v) {
    step(g.next(v));
  }
  function step(result) {
    while (!result.done) {
      // On suppose que result.value est une promesse:
      var promise = result.value;
      return promise.then(resumeNext).catch(throwError);
    }
    step(g.next());
  }
}

```

(b)

Figure 51 – Concaténation de deux pages Web (a) écriture linéaire du code (b) écriture de la fonction process

8.4 Futur

Les prochaines versions de Javascript devraient encore améliorer syntaxiquement l'usage du précédent asynchronisme avec deux nouveaux mots-clés, `async` et `await`. On pourrait ainsi encore réécrire notre exemple récurrent comme dans la figure 52.

```

async function main () {
  let a = "", b = "";
  try {
    a = await fetchPromise('http://a.fr');
  } catch (exc) {}
  try {
    b = await fetchPromise('http://b.fr');
  } catch (exc) {}
  return a + b;
};

var result = main();
console.log(result);

```

Figure 52 – Usage des mots-clés `async` et `await`

Les deux dernières versions avec générateur explicite ou annotation `async` procurent un code particulièrement lisible car séquentiel. L'asynchronisme est relégué dans les `yield` ou `await`, le rattrapage d'erreur est clairement délimité par des `try-catch`. Mais cette lisibilité vient au détriment du parallélisme que l'on pouvait obtenir comme en section 7.2 ou 7.3. Le parallélisme peut se retrouver mais requiert de manipuler des promesses et leurs méthodes spécifiques telles que `Promise.all`.

9. Conclusions

Cet article n'a fait que donner les grandes lignes d'ECMAScript 2015, il a omis la description de caractéristiques réflexives comme la possibilité d'invoquer le compilateur Javascript à l'exécution avec la fonction `eval` ou le constructeur `Function`. Il n'a décrit que le langage et s'est abstenu de décrire la manipulation du DOM et les innombrables modules écrits en Javascript. De même, il est resté muet sur les outils : environnement intégré de développement (Éclipse, Atom, etc.), outils de tests (Jasmine, Mocha, etc.), utilitaires (grunt, gulp, etc.), gestionnaire de modules (bower, npm, etc.), ainsi que sur les bonnes pratiques. En revanche, nous avons insisté sur les faits saillants de Javascript et ses particularismes lexicaux ou sémantiques.

Les navigateurs sont les modernes conteneurs d'applications. Ils procurent les fonctionnalités d'affichage via HTML et CSS, ils permettent les interactions avec des serveurs, ils n'utilisent qu'un unique langage de programmation, Javascript, facilitant ainsi le déploiement et la portabilité des applications. Des ordinateurs aux téléphones, Javascript tourne partout et est là pour rester.

Javascript

Christian QUEINNEC
Professeur émérite de l'UPMC

Sources bibliographiques

- [1] SELF UNGAR (D.) et SMITH (R.B.). – *Self: The Power of Simplicity*. OOPSLA '87 Conference Proceedings, pp. 227-241, Orlando, FL, October 1987

Normes

ECMA-262	2016	ECMAScript® 2016 language specification	ISO/IEC 10646 : 2014	2014	Information technology Universal Coded Character Set (UCS)
IEEE Std 1178-1990	1990	IEEE Standard for the Scheme Programming Language	IEEE 754	2008	Standard for Binary Floating – Point Arithmetic

Sites Internet

Node https://nodejs.org/	lodash https://github.com/lodash/lodash
Babel.js http://babeljs.io/	Rhino https://github.com.mozilla/rhino
MongoDB https://www.mongodb.org/	ES6 compatibility https://kangax.github.io/compat-table/es6/
jQuery https://jquery.com/	

GAGNEZ DU TEMPS ET SÉCURISEZ VOS PROJETS EN UTILISANT UNE SOURCE ACTUALISÉE ET FIABLE

Techniques de l'Ingénieur propose la plus importante collection documentaire technique et scientifique en français !

Grâce à vos droits d'accès, retrouvez l'ensemble des **articles et fiches pratiques de votre offre, leurs compléments et mises à jour,** et bénéficiez des **services inclus.**



- + de 350 000 utilisateurs
- + de 10 000 articles de référence
- + de 80 offres
- 15 domaines d'expertise

- | | |
|---|---|
| <input type="radio"/> Automatique - Robotique | <input type="radio"/> Innovation |
| <input type="radio"/> Biomédical - Pharma | <input type="radio"/> Matériaux |
| <input type="radio"/> Construction et travaux publics | <input type="radio"/> Mécanique |
| <input type="radio"/> Électronique - Photonique | <input type="radio"/> Mesures - Analyses |
| <input type="radio"/> Énergies | <input type="radio"/> Procédés chimie - Bio - Agro |
| <input type="radio"/> Environnement - Sécurité | <input type="radio"/> Sciences fondamentales |
| <input type="radio"/> Génie industriel | <input type="radio"/> Technologies de l'information |
| <input type="radio"/> Ingénierie des transports | |

Pour des offres toujours plus adaptées à votre métier,
découvrez les offres dédiées à votre secteur d'activité

Depuis plus de 70 ans, Techniques de l'Ingénieur est la source d'informations de référence des bureaux d'études, de la R&D et de l'innovation.

www.techniques-ingenieur.fr

CONTACT : Tél. : + 33 (0)1 53 35 20 20 - Fax : +33 (0)1 53 26 79 18 - E-mail : infos.clients@teching.com

LES AVANTAGES ET SERVICES compris dans les offres Techniques de l'Ingénieur

ACCÈS



Accès illimité aux articles en HTML

Enrichis et mis à jour pendant toute la durée de la souscription



Téléchargement des articles au format PDF

Pour un usage en toute liberté



Consultation sur tous les supports numériques

Des contenus optimisés pour ordinateurs, tablettes et mobiles

SERVICES ET OUTILS PRATIQUES



Questions aux experts*

Les meilleurs experts techniques et scientifiques vous répondent



Articles Découverte

La possibilité de consulter des articles en dehors de votre offre



Dictionnaire technique multilingue

45 000 termes en français, anglais, espagnol et allemand



Archives

Technologies anciennes et versions antérieures des articles



Impression à la demande

Commandez les éditions papier de vos ressources documentaires



Alertes actualisations

Recevez par email toutes les nouveautés de vos ressources documentaires

*Questions aux experts est un service réservé aux entreprises, non proposé dans les offres écoles, universités ou pour tout autre organisme de formation.

ILS NOUS FONT CONFIANCE



www.techniques-ingénieur.fr

CONTACT : Tél. : + 33 (0)1 53 35 20 20 - Fax : +33 (0)1 53 26 79 18 - E-mail : infos.clients@teching.com