Programmation web
JavaScript - Langage 1 - Bases du langage

Vincent Padovani, PPS, IRIF

Cette introduction au langage JavaScript présuppose une connaissance des notions de type, de variable, d'expression et d'affectation, d'objet et de référence, ainsi que celle des structures de contrôle de Java ou C. Elle décrit les éléments les plus simples du langage, immédiatement utilisables dans les exercices du cours : les types de base, la syntaxe des déclarations de variables et de fonctions, les objets littéraux et les constructeurs.

La notion de variable en JavaScript est la même qu'en C ou Java. Les opérateurs arithmétiques et logiques ont la même syntaxe et (presque) la même sémantique. Les opérateurs `==` et `!=` existent mais obéissent à des règles assez confuses : les équivalents stricts de `==` et `!=` de C/Java s'écrivent `===` et `!==`, et il vaut mieux se servir de ces opérateurs que des précédents.

On retrouve les structures et instructions de contrôle habituelles, avec la même syntaxe et le même sens : `if`/`else`, `for`, `while`, `do`/`while`, `switch`, `break`, `continue`. On retrouve également dans le langage les notions d'objet et de référence. Le nom du langage et sa syntaxe peuvent créer l'illusion que la notion d'objet est essentiellement la même que celle de Java, mais il s'agit d'une notion différente – ce point sera détaillé au chapitre suivant.

1 Types et variables

1.1 Types de base.

Les principaux types de bases de JavaScript sont :

- `undefined`, de seule valeur `undefined`,
- `boolean`, de valeurs `true`, `false`,
- `number`, le type de toutes les valeurs numériques (entières ou non),
- `string`, le type des valeurs de chaînes de caractères.

Types `boolean` et `number`. Comme en Java, le type `boolean` est celui pris par toutes les expressions de comparaisons. Le type `number` contient deux valeurs singulières, `NaN`, qui est la valeur prise par une expression dont le type attendu est `number` mais dont le résultat est incalculable (*e.g.* `"abc" * 42` ou `parseInt("abc")`), et `Infinity`, le résultat par exemple d'une division par zéro.

Type `string`. Les chaînes sont des valeurs non mutables. La comparaison des chaînes se fait par valeur. L'opérateur de concaténation s'écrit `+`. Comme en java, un `+` est interprété comme cet opérateur si l'un au moins de ses arguments est une chaîne. Voici par exemple, écrites en commentaires, les valeurs renvoyées par la console JavaScript de Firefox à l'entrée des expressions suivantes :

```
"abc" + 42;                    // "abc42"
42 + 42;                       //  84
42 + 42 + "abc";               // "84abc"
```

1.2 Déclaration de variable et affectation.

Le mot-clef `let`. Le mot-clef `let` permet de déclarer une ou plusieurs nouvelles variables. Les variables ne sont pas typées au moment de leur déclaration, et peuvent contenir des valeurs de types distincts au cours du temps. Une variable non initalisée reçoit la valeur `undefined`.

```
let n, m;                       // n === undefined, m === undefined
n = 42;
n = 3.14;
n = "abc";
m = n;                          // m === "abc"
```

L'opérateur `typeof`. L'opérateur `typeof` renvoie le type du contenu courant d'une variable ou le type d'une expression sous la forme d'une chaîne de caractères.

```
let n;                         // typeof n === "undefined"
n = "abc";                     // typeof n === "string"
```

1.3 Tableaux

Tableaux littéraux. La syntaxe de tableaux "littéraux" (de contenu explicitement écrit dans le code) est un peu différente de celle de C ou Java, mais l'acces indexé a la même syntaxe. Rien n'oblige les éléments d'un tableau à être de même type.

```
let t = ["a", "b", 1.5];       // Array (3) ["a", "b", 1.5]
t[1] = 42;;                    // Array (3) ["a",  42, 1.5]
                               // t[3] === undefined
```

Références vers les tableaux. Les tableaux en JavaScript sont des objets (en un sens à préciser). La déclaration ci-dessous crée un nouvel objet en mémoire, et `t` devient une référence vers cet objet. L'affectation de `t` à `u` copie la valeur de référence de `t` dans `u`, qui peut alors accèder au même objet :

```
let t = [0, 1, 2];
let u = t;                     // u === t
u[0] = 42;
console.log(t);                // 42, 1, 2
console.log(u);                // 42, 1, 2
u = [0, 1, 2];                 // t !== u;
```

Ajouts et suppressions d'éléments. Les tableaux sont extensibles et retrécissables. Ils peuvent contenir des cases vides. La longueur courante d'un tableau `t` est récupérable via l'expression `t.length`.

```
let t = [0, 1, 2];              // Array (3) [0, 1, 2]
t[t.length] = 42;               // Array (4) [0, 1, 2, 42]
t.length = 3;                   // Array (3) [0, 1, 2]
t.length = 10;                  // Array (4) [0, 1, 2, <7 emptyslots>]
                                // t[3] === undefined
delete(t[2]);                   // Array (4) [0, 1, <8 emptyslots>]
```

Parcours de tableaux. Une forme particulière de boucle, `for`/`of`, permet de parcourir la suite des éléments d'un tableau :

```
let sum = 0;
for (let i of [1, 3, 5]) { // pour chaque valeur i du tableau [1, 3, 5]
  sum += i;
}
sum;                       // => 9 === 1 + 3 + 5
```

2 Fonctions

2.1 Déclaration et appel.

Le mot-clef `function`. La syntaxe des déclarations de fonctions en JavaScript se fait sur le modèle suivant :

```
function plus(x, y) {
  return x + y;
}
```

On peut manipuler des fonctions anonymes, à la manière de OCaml :

```
(function (x, y) { return x + y; }) (3, 4);  // => 7
```

Les arguments attendus pour une fonction peuvent être de type référence vers fonction, ce qui permet d'écrire des fonctionnelles :

```
function appliquer(f, x) {
  return f(x);
}
console.log(appliquer(succ, 42)); // 43
```

De même, la valeur de retour d'une fonction peut être de type référence vers fonction :

```
function decalage(x) {
  return function (y) {
    return x + y;
  }
}
decalage(42)(3)            // => 45
```

On peut imbriquer les fonctions, c'est-à-dire déclarer des fonctions locales à une fonction.

Arguments manquants ou supplémentaires. Une fonction peut être appelée sur un nombre quelconque d'arguments. S'il y a plus d'arguments que de paramètres, les arguments supplémentaires sont simplement ignorés. S'il y a trop peu d'arguments, les paramètres sans arguments associés prendront la valeur `undefined`. On peut cependant spécifier pour un paramètre une valeur par défaut, qui sera celle prise par ce paramètre si son argument est absent :

```
function push(v, stack = []) {
  stack.push(v);    // ajouter v en tête du tableau stack, ou en tête
  return stack;     // du tableau vide si stack est absent.
}
push(42);       // => [42]
push(42, [10])  // => [42, 10];
```

Collecte des arguments restants Le dernier paramètre d'une fonction peut être spécifié comme devant collecter l'ensemble des arguments qui suivent les arguments donnant une valeur aux paramètres précédents, sous la forme d'un tableau éventuellement vide (mais jamais en `undefined`)– ce paramètre doit être précédé de trois points :

```
function push_on_all(a, ...stacks) {
  for (let t of stacks) {
    t.push(v);
  }
  return stacks;
}
push_on_all(42);                      // []
push_on_all(42, [0, 1], [2, 3, 4]);  // [[42, 0, 1], [42, 2, 3, 4]]
```

2.2 Règles de portée

1. La portée des éléments déclarés par `let` est une portée de *bloc*. Les éléments déclarés globalement sont vues par tous les éléments qui suivent leur déclaration. Les éléments déclarées dans un bloc (*e.g.* le corps d'une structure de contrôle, le corps d'une fonction)) ne sont visibles que dans ce bloc.
2. Le langage autorise dans un bloc la déclaration locale par `let` d'un élément de même nom qu'un élément déclaré dans le contexte de ce bloc, mais pas deux déclarations de mêmes noms à la même profondeur de bloc.

3. La portée des éléments déclarés par `function` est une portée de *fonction*. Une fonction déclarée globalement est vue par tous les éléments qui suivent sa déclaration, même si celle-ci est imbriquée dans un ou plusieurs blocs. Une fonction déclarée dans une autre fonction n'est visible que dans le corps de celle-ci.
4. Contrairement aux éléments déclarés par `let`, une fonction peut-être redéfinie.

```
let x = 42;                 // x global                        (1)
{
  let x = 0;                // x local => 0                    (2)
  let y = 1;                // y local => 1                    (1)
}
x;                          // x global => 42                  (1)
// let x = 1;               // ERREUR : redeclaration de x.    (2)
y;                          // ERREUR : y indefini             (1)

function f() {
  let y = 1;                // y : vu seulement par f, g       (1)
  function g() {            // g : vu seulement par f          (3)
      let z = 2;            // z : vu seulement par g          (1)
      return x + y + z;
  }
  return g();
}

f();                        // => 45
g();                        // ERREUR : g indefinie            (3)

function f() {
  return 48;
}
f();                        // => 48                           (4)
```

3 Introduction aux objets

Un objet en JavaScript n'est rien de plus qu'une suite de valeurs nommées par des chaînes de caractères. Ces valeurs peuvent être de type quelconque, y compris des références vers d'autres objets, par exemple des fonctions.

Il y a deux manières, équivalentes mais syntaxiquement différentes, de construire un objet en JavaScript : soit en décrivant littéralement son contenu ; soit en déléguant son initialisation à une fonction appelée *constructeur*, par analogie avec Java. La notion de constructeur est introduite à la section suivante, mais sera développée au chapitre suivant.

3.1 Objets littéraux.

La déclaration suivante crée à la volée un nouvel objet en mémoire, et `monRectangle` devient une référence vers celui-ci. Les éléments `largeur`, `hauteur` et `aire` sont des chaînes de caractères et pourraient être encadrés par des guillements doubles, mais ils sont facultatifs et l'usage est de ne pas les écrire.

```
let monRectangle = {
  largeur: 200,
  hauteur: 100,
  aire: function () {                    // ou simplement : aire() { ... }
    return this.largeur * this.hauteur;
  }
}
```

Chaque couple *nom*:*valeur* est appelé une *propriété* de l'objet `monRectangle`. Les noms des propriétés permettent d'accéder à leur valeur, avec deux syntaxes possibles. La seconde nécéssite de rétablir les guillemets omis autour du nom de la propriété.

```
monRectangle.hauteur = 42;
monRectangle["hauteur"] = 42;
```

Toujours par analogie avec Java, une propriété dont la valeur est fonctionnelle est appelée une *méthode*. Une instruction de la forme `monRectangle.aire()` est appelée une *invocation* de méthode. Comme en Java, le mot-clef `this` dans le corps de cette méthode est une référence vers l'objet lui-même, *i.e.* vers `monRectangle`.

3.2 Ajout et suppression de propriétés.

L'ensemble des propriétés d'un objet, littéral ou non, peut varier au cours du temps.

Si l'on tente de donner une valeur à une propriété dont le nom n'est pas celui de l'une des propriétés de l'objet, le couple formé par ce nom et cette valeur sera simplement ajouté en propriété supplémentaire. L'opérateur `in` permet de déterminer si un objet possède ou non une propriété à partir de son nom :

```
let ceCours {
  titre: "Programmation Web",
  cursus: "L3 Info"
}
```

```
// (suite de l'exemple precedent)
// messages de la console en mode interactif :
"horaire" in ceCours;                // false
ceCours.horaire;                     // undefined

ceCours.horaire = "13h30";
// messages :
"horaire" in ceCours;                // true
ceCours.horaire;                     // "13h30"
```

Une propriété peut aussi être supprimée :

```
delete(ceCours.titre);
// message :
"titre" in ceCours;                 // false
```

Parcours des noms de propriétés. Une boucle `for`/`in` (à ne pas confondre avec `for`/`of`, *c.f.* la section sur les tableaux ci-dessus) permet de parcourir l'ensemble des noms de propriétés d'un objet, sur le modèle suivant :

```
for (let prop in ceCours) {
  console.log(prop + " " + ceCours[prop]);
}
// cursus "L3Info"
// horaire "13h30"
```

3.3 L'objet global, `let` et `var`

Toutes les fonctions déclarées à l'aide du mot-clef `function` sont en fait des propriétés d'un *objet global*, ajoutées à cet objet au fil des déclarations. Les fonctions ne sont donc rien de plus que des méthodes de cet objet global.

```
this;                           // => Window about:blank
function plus(x, y) { return x + y; }
plus(40, 2)                     // => 42
this.plus(40, 2);               // => 42
```

En revanche, les variables déclarées à l'aide du mot-clef `let` ne sont pas ajoutées à cet objet global :

```
let n = 42;
n;                              // => 42
this.n;                         // => undefined
let plus = function (x, y) { return x + y; }
plus(40, 2)                     // => 42
this.plus;                      // => undefined
this.plus(40, 2);               // => TypeError: this.plus is not a function
```

Un mot-clef du langage, `var` permet de déclarer une variable en forçant cet ajout, mais les règles de portée des éléments ou déclarés ou redeclarés en `var` sont suffisamment confuses pour qu'il soit préférable de ne pas s'en servir – sauf, de manière implicite, dans les déclarations de fonctions.

4 Les constructeurs

La notion de classe n'existe pas dans JavaScript. La possibilité de définir des "classes" à la Java existe en apparence dans les versions récentes du langage, mais il ne s'agit que d'une simulation syntaxique. On retrouve bien cependant dans le langage une notion de *constructeur*, une fonction chargée de l'initialisation d'objets au moment de leur création.

4.1 Appels de constructeurs

La création d'un objet et son initialisation par un constructeur s'effectue à l'aide de l'opérateur `new` suivi d'un appel du constructeur. Par convention, les noms de constructeurs commencent par une majuscule.

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

let p1 = new Personne("John", "Doe") // {prenom: "John", nom: "Doe"}
let p2 = new Personne("Jane", "Doe") // {prenom: "Jane", nom: "Doe"}
// etc.
```

Le `new` crée d'abord un nouvel objet en mémoire. L'appel du constructeur est ensuite exécuté, `this` étant dans le corps de celui-ci une référence vers l'objet créé.

4.2 Constructeurs implicites.

Presque tous les objets, y compris les objets littéraux, sont initialisés par un constructeur. La déclaration d'un objet littéral, par exemple, est un simple raccourci d'écriture pour `new` suivi de l'appel du constructeur prédéfini `Object`, suivi de l'ajout des propriétés littéralement écrites :

```
let p = new Object();          // let indiv = {
p.prenom = "John";             //   prenom: "John".
p.nom = "Doe";                 //   nom: "Doe"
                               // };
```

De même, les fonctions déclarées sont des objets initialisés par le constructeur prédéfini `Function`, les tableaux le sont par le constructeur `Array`, etc.

À noter qu'il existe aussi des constructeurs `String`, `Number`, `Boolean`. Lorsque l'on tente d'invoquer une méthode sur une chaîne, un nombre ou un booléen, un nouvel objet encapsulant cette valeur est crée à la volée en mémoire et initialisé par le constructeur correspondant : c'est sur cet objet que la méthode sera invoquée.

4.3 Appels de plusieurs constructeurs.

Un constructeur peut déléguer une partie de l'initialisation d'un objet à un ou plusieurs autres constructeurs. Toute méthode (donc toute fonction, qui est aussi une méthode de l'objet global) est munie d'une méthode `call` invocable sur un objet `o` sur le modèle suivant :

```
Personne.call(o, "John", "Doe");
```

Le constructeur `Personne` sera appelé en lui passant les arguments qui suivent `o`, le mot-clef `this` étant alors dans le corps de celui-ci une référence vers `o`. Ceci permet par exemple de définir un nouveau constructeur complétant l'initialisation de nouveaux objets partiellement initialisés par `Personne` par l'ajout d'une adresse :

```
function Fiche(nom, prenom, adresse) {
  Personne.call(this, nom, prenom);
  this.adresse = adresse;
}
let f = new Fiche("John", "Doe", "Arkham");
// {prenom: "John", nom: "Doe", adresse: "Arkham"}
```

4.4 Appels sans `new`.

L'oubli du `new` dans un appel de constructeur donne une instruction sytaxiquement correcte, mais dont les effets de bord peuvent être indésirables. À défaut de la présence de `new`, dans le corps de `Personne`, `this` est une référence vers l'objet global :

```
// ???
let p = Personne("John", "Doe")
// messages :
p;                              // undefined (pas de return dans Personne)
this;                           // Window : about blank
prenom; this.prenom;            // "John" "John"
nom;    this.nom;               // "Doe"  "Doe"
```

4.5 Accesseurs

En Java, on attend d'un "accesseur" (un "getter" ou un "setter") qu'il permette d'accéder en lecture ou en écriture à un champ, en particulier si ce champ est privé.

En JavaScript, un accesseur est bien une méthode, mais cette méthode n'est invocable que lorsque l'on utilise son nom comme s'il s'agissait d'une propriété ordinaire [1].

1. À mon avis, ceci n'apporte grand chose ni à la lisibilité du code, ni à sa compréhension - il me semblerait plus raisonnable que ce qui est syntaxiquement identique à une simple lecture de valeur de propriété ne puisse avoir aucun effet de bord, et que ce qui est sytaxiquement identique à écriture de valeur de propriété ne puisse avoir aucun autre effet de bord que celui attendu – ce que ne garantit en rien, dans chaque cas, une invocation de méthode.

Dans l'exemple suivant deux accesseurs, un getter et un setter de noms `radian` permettent d'accéder à une propriété représentant une mesure d'angle en degrés, syntaxiquement *comme si* celle-ci était une propriété de l'objet `radian` exprimant la même mesure en radians :

```
let o = {
  degree : 0.,
  // getter de nom radian
  get radian ()  {
    return (this.degree * 2. * Math.PI) / 360. ;
  },
  // setter de nom radian
  set radian (x) {
    this.degree = (x * 360.) / (2. * Math.PI);
  }
};

o.degree = 90;
o.radian;               // invocation implicite du getter
                        // => 1.5707963267948966

o.radian = Math.PI;     // invocation implicite du setter
o.degree;               // => 180;
```

L'exemple suivant montre cependant les limites du choix syntaxique de l'invocation des accesseurs. Il pourrait sembler naturel d'écrire, à la manière de Java :

```
/* ERREUR !!!
let o = {
  value : 42,
  get value ()  {
    return this.value; // Stack overflow !!!
  },
  set value (x) {
    this.value = x;    // Stack overflow !!!
  }
};
*/
```

Dans cet exemple, le nom de propriété `value` est immédiatement écrasé par celui du getter de même nom. Conformément à la règle d'appel implicite des accesseurs, chaque évaluation de `this.value` dans le getter provoquera un nouvel appel de ce getter, jusqu'à l'explosion de la pile d'appel. De même, la présence de `this.value` dans le setter à gauche d'une affectation provoquera un nouvel appel du setter jusqu'à explosion de la pile.

Le paradoxe des accesseurs en JavaScript est donc qu'ils ne doivent accéder, en lecture pour les getters, en écriture pour les setters, qu'à des propriétés de noms différents du leur. Dans l'exemple, placer la propriété `value` après les accesseurs ne ferait que masquer leurs noms par le nom de cette propriété, ce qui rendrait ces accesseurs inaccessibles.

5 Compléments : les fonctions en flèche

Nous avons vu dans les sections précédentes que les fonctions en JavaScript se divisent en trois catégories :

- les méthodes, c'est-à-dire les fonctions propriétés d'objets,
- les constructeurs, permettant avec l'opérateur `new` d'initialiser un objet,
- toutes les autres fonctions globales ou locales,

Les interprétations suivantes du mot-clef `this` s'appliquent à toutes les fonctions déclarées à l'aide du mot-clef `function` :

1. Au plus haut niveau du corps d'une méthode, `this` est un référence vers l'objet sur lequel on invoque cette méthode.
2. Au plus haut niveau du corps d'un constructeur invoqué avec l'opérateur `new`, `this` est une référence vers l'objet créé.
3. Au plus haut niveau du corps des autres fonctions, `this` est une référence vers l'objet global.

La règle (3) est sans exception, et il s'agit clairement d'une erreur de conception du langage. S'il avait été mieux conçu, `this` aurait dû avoir, comme en Java, un sens uniforme dans tout le corps d'une méthode ou d'un constructeur invoqué avec `new`, à moins qu'un contexte interne n'écrase sa valeur. Ce n'est pas le cas.

Dans l'exemple ci-dessous, la méthode `m` obéit à la règle (1), tandis que la fonction `f` interne à `m` obéit à la règle (3) : `this` est donc dans cette fonction une référence vers l'objet global, et non vers `o`.

```
let global = this;
let o = {
  m : function () {
    // this === o
    function f() {
      // this === global
    }
  }
```

Avant l'introduction des fonctions en flèche, cette faute de conception interdisait l'usage de fonctions auxiliaires internes à une méthode sans passer par un artifice d'écriture : la mise en mémoire de `this` au début du corps de `m` dans une autre variable, traditionnellement appelée `that`, et permettant aux fonctions internes d'accéder de manière indirecte à l'objet courant.

```
let global = this;
let o = {
  m : function () {
    let that = this;  // this === o, that === o
    function f() {
       // this === global, that === o
    }
  }
```

L'introduction des fonctions en flèche permet à présent d'éviter cette forme écriture. Une fonction en flèche se définit sur le modèle suivant :

```
let plus = (x, y) => { return x + 1; };
```

Tant que ce mot-clef `this` n'apparaît pas dans une fonction, cependant, le choix de se servir de `function` ou de `=>` est neutre. la flèche offre même des possiblités d'ecritures très concises que n'autorise pas la notation usuelle. Si l'argument d'une fonction en `=>` est unique, les parenthèses autour de son paramètres sont facultatives. De plus, si le corps d'une fonction est réduit à une unique instruction en `return` *expr*. Il peut être entièrement remplacé par *expr* sans accolades. La syntaxe des fonctions ainsi déclarées ressemble beaucoup à celle d'OCaml :

```
let succ  = x => x + 1;
let composition = f => g => x => f (g (x));
```

La règle pour les fonctions en flèche est celle-ci :

4. Dans une fonction en flèche et à l'extérieur de toute fonction interne, la valeur de `this` est la même que celle de `this` dans son contexte le plus proche.

Autrement dit, si une fonction en flèche est globale, `this` désigne dans celle-ci l'objet global. Mais si elle est interne au plus haut niveau du corps d'une autre fonction, `this` aura la même valeur pour la fonction en flèche que celle de cette fonction parente :

```
let o = {
  m : function () {
    // this === o
    let f = () => {
      // this === o
      let g = () => {
        // this === o
      }
    }
  }
```

Une conséquence de ce mécanisme est que si une fonction en flèche est utilisée comme une méthode, `this` n'y désignera pas l'objet courant mais l'objet global. Une fonction en flèche ne peut pas être utilisée comme constructeur – toute tentative de coupler un appel de cette fonction à un `new` déclenchera une erreur d'exécution. On peut cependant se servir de telles fonctions par exemple si l'objet dont elles sont les propriétés est vu comme un simple espace de noms – à la manière des méthodes statiques de Java :

```
const utils = {
  plus  : (x, y) => x + y,
  moins : (x, y) => x - y,
  fois  : (x, y) => x * y
};
let n = utils.plus(42, 3);
```