Design Patterns :
Développement Web

Julien Provillard

LES OBJETS EN JAVASCRIPT

Les bases

- Un objet est une valeur qui en agglomère d'autres. Ce n'est pas un type primitif.
  - Un objet est composé d'une liste de propriétés.
  - Une propriété est un couple clé/valeur.
  - La clé d'une propriété (son nom) est une chaîne de caractères.
  - Les valeurs sont arbitraires.
  - Intuitivement, on peut voir un objet comme un dictionnaire / une table associative.

Les bases

- On peut créer un objet "vide" avec la syntaxe :

```
let student = new Object();
```

- On peut créer un objet littéral de la manière suivante :

```
let professor = {
  firstName: "Julien",
  lastName: "Provillard",
  id: getNewID(),
  birthDate: {
    day: 4,
    month: 0,
    year: 1985,
  }, // virgule optionnelle ici
};
```

Les bases

- On peut accéder à la valeur d'une propriété.

```
let name = professor.firstName; // "Julien"
name = professor.nickname; // undefined, la propriété n'existe pas
```

- On peut modifier une propriété (ou en ajouter une).

```
professor.id = 123456789;
professor.nickname = "Error 404";
name = professor.nickname; // "Error 404"
```

- On peut supprimer une propriété.

```
delete professor.nickname;
name = professor.nickname; // undefined
```

Les bases

- Les noms de propriétés sont des chaînes de caractères.

```
let obj = { nom: val } ⇔ let obj = { "nom": val }
```

- Si le nom contient un espace ou commence par un chiffre, la deuxième syntaxe est nécessaire.

```
let obj = { "nom composé": val }
let c = { "2x3+1": 7 }
```

- On peut avoir des propriétés « numériques ».

```
let obj = { 0: "zero", 1: "un" } ⇔ let obj = { "0": "zero", "1": "un" }
```

- Ce qui peut entraîner des pièges.

```
let obj = { 0x1F: 31 } ⇔ let obj = { "31": 31 }
```

Les bases

- Pour accéder à une propriété, on peut utiliser les crochets.
  - L'argument entre crochets est converti en chaîne comme nom de la propriété.
  - C'est la seule manière d'interagir avec les propriétés qui commencent par un chiffre ou qui comportent plusieurs mots.
  - On peut accéder à des propriétés de manière dynamique.

```
delete professor["birthDate"].year;

let propName = prompt("property wanted?");
let name = professor[propName]; // Julien si propName == "firstName"
```

Les bases

- On peut aussi utiliser les crochets pour créer dynamiquement des propriétés à la créations d'un objet.

```
let create = function(name, index, value) {
  return { [name + index]: value }
}
let ex = create("exemple", 0, 42); // { exemple0: 42 }
```

- On préfère généralement une utilisation plus classique en deux temps.

```
let create = function(name, index, value) {
  let res = {}; // autre façon de créer un objet vide
  res[name + index] = value;
  return res;
}
let ex = create("exemple", 0, 42); // { exemple0: 42 }
```

Existence d'une propriété

- On peut tester si une propriété est définie en comparant sa valeur à `undefined`.

```
let user = {};
alert(user.notDefined === undefined); // true
alert(user["notDefined"] === undefined); // true
```

- L'opérateur `in` permet de simplifier ce test.

```
alert("notDefined" in user); // false
let key = "notDefined";
alert(key in user); // false
```

Raccourcis

- Un motif courant est la création d'un objet via des variables qui portent le même nom que les propriétés.

```
let firstName = "Julien";
let lastName = "Provillard";
let user = { firstName: firstName, lastName: lastName, id: 42 };
```

- Javascript permet d'ignorer la répétition dans ce cas.

```
let firstName = "Julien";
let lastName = "Provillard";
let user = { firstName, lastName, id: 42 };
```

Déstructurer un objet

- On peut récupérer la valeur des propriétés en déstructurant un objet.

```
let user = { firstName: "Julien", lastName: "Provillard", id: 42 };

let { firstName, lastName } = user; // déstructuration partielle
⇔
let firstName = user.firstName;
let lastName = user.lastName;
```

- Cela fonctionne aussi pour les arguments des fonctions.

```
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
let name = getFullName(user); // Julien Provillard
```

Déstructurer un objet

- On peut combiner les deux techniques pour des manipulations élégantes des objets.

```
let user = { firstName: "Julien", lastName: "Provillard", id: 42 };

function cloneUser({ firstName, lastName }, newID) {
  return { firstName, lastName, id: newID };
}
let clone = cloneUser(user, 421);
```

Déstructurer un objet

- On peut donner des alias pour les propriétés et/ou leur assigner des valeurs par défaut.

Propriétés à déstructurer
Variables à initialiser
Valeur par défaut

```
let rectangle = { width: 20, height: 10 };
let { width: w, height: h, anchor: a = { x: 0, y: 0 } } = rectangle;
let message = `Rectangle of size ${w * h} in (${a.x}, ${a.y}).`;
// "Rectangle of size 200 in (0, 0)"
```

- On peut assigner des variables par déstructuration. En ce cas, toutes les variables utilisées doivent être déclarées.

```
let w, h;
({ width: w, height: h } = rectangle);
// parenthèses obligatoires pour l'analyse syntaxique correcte
```

Itération sur les propriétés d'un objet

- On itère sur les propriétés d'un objet à l'aide de la boucle `for in`.

```
let obj = { b: "foo", a: "bar", 1: null, 0: 42 };
for (let key in obj) {
  alert(`${key}: ${obj[key]}`);
}
// affiche "0: 42", "1: null", "b: foo", "a: bar"
```

- Pourquoi cet ordre ?

Copie de propriétés

- Pour assigner toutes les propriétés d'un objet à un autre, on utilise Object.assign.

```
let target = { firstName: "Julien", lastName: "Provillard", version: 1 };
let analytics = { color: "Purple", version: 2 };
Object.assign(target, analytics); // renvoie également target
// target devient { firstName: "Julien", lastName: "Provillard",
                    version: 2, color: "Purple" }
```

- On peut chaîner les assignements

```
Object.assign(obj1, obj2, obj3, ..., objN)
⇔
Object.assign(Object.assign(obj1, obj2), obj3, ..., objN)
```

Références

- Les objets sont des références.
  - Comparaison par identité mémoire

```
let obj = {};
obj == {}; // false
obj == obj; // true
```

  - Pas de copie implicite

```
let obj1 = {};
let obj2 = obj1;
obj2.x = 10;
(obj => obj.y = 20)(obj2);
alert(obj1.x + obj1.y); // 30
```

Méthodes

- Une méthode est simplement une fonction stockée dans une propriété.

```
let professor = {
  firstName: "Julien",
  lastName: "Provillard",
  sayHi: function () { alert("Hi!"); },
};
professor.sayHi(); // appel de méthode
```

- C'est un cas tellement classique qu'il existe une syntaxe simplifiée.

```
let professor = {
  firstName: "Julien",
  lastName: "Provillard",
  sayHi() { alert("Hi!"); },
};
```

Méthodes

- Lors d'un appel de méthode, on peut utiliser le mot clé `this` pour référencer l'objet appelant.

```
let professor = {
  firstName: "Julien",
  lastName: "Provillard",
  sayHi() {
    alert(`${this.firstName} says hi!`);
  },
};
professor.sayHi(); // Julien says hi!
```

Méthodes

- Le mot clé `this` n'a de sens que dans un appel de méthode. Il réfère donc au dernier objet appelant… ou à l'objet global. (???)

```
let f = professor.sayHi;
f(); // appel normal, this n'est plus lié à professor => undefined says hi!
```

- On peut se servir de ce phénomène de manière inverse.

```
let getName() {
  return `${this.firstName} ${this.lastName}`; // ici this n'est pas lié
}
professor.getName = getName; // <=> Object.assign(professor, { getName })
let name = professor.getName(); // Julien Provillard
```

Méthodes

- Attention, les fonctions fléchées et les fonctions classiques ont des comportements différents par rapport à `this`.

```
let professor = { firstName: "Julien", lastName: "Provillard",
  buildSayHi(title) {
    return function () { alert(`I am ${title} ${this.lastName}.`); }},
};

let sayHi = professor.buildSayHi("professor");
sayHi(); // I am professor undefined.
```

- La référence à `this` a été perdue.

Méthodes

- Attention, les fonctions fléchées et les fonctions classiques ont des comportements différents par rapport à `this`.

```
let professor = { firstName: "Julien", lastName: "Provillard",
  buildSayHi(title) {
    return function () { alert(`I am ${title} ${this.lastName}.`); }},
};

let sayHi = professor.buildSayHi("professor");
let professor2 = { firstName: "Horatiu", lastName: "Cirstea", sayHi };
professor2.sayHi(); // I am professor Cirstea.
```

- La référence à `this` s'est adaptée au contexte.

Méthodes

- Attention, les fonctions fléchées et les fonctions classiques ont des comportements différents par rapport à `this`.

```
let professor = { firstName: "Julien", lastName: "Provillard",
  buildSayHi(title) {
    return () => alert(`I am ${title} ${this.lastName}.`); },
};

let sayHi = professor.buildSayHi("professor"); // capture ici
sayHi(); // I am professor Provillard.
```

- La référence à `this` a été maintenue. La fonction fléchée capture la valeur de `this` qui n'est plus affecté par les changements de contexte.

Méthodes

- Attention, les fonctions fléchées et les fonctions classiques ont des comportements différents par rapport à `this`.

```
let professor = { firstName: "Julien", lastName: "Provillard",
  buildSayHi(title) {
    return () => alert(`I am ${title} ${this.lastName}.`); },
};

let sayHi = professor.buildSayHi("professor"); // capture ici
let professor2 = { firstName: "Horatiu", lastName: "Cirstea", sayHi };
professor2.sayHi(); // I am professor Provillard.
```

- La référence à `this` a été à nouveau maintenue.

Méthodes

- Peut-on faire de même avec les fonctions classiques ?

```
let professor = { firstName: "Julien", lastName: "Provillard",
  buildSayHi(title) {
    return function () { alert(`I am ${title} ${this.lastName}.`); }},
};

let sayHi = professor.buildSayHi("professor").bind(professor);
let professor2 = { firstName: "Horatiu", lastName: "Cirstea", sayHi };
professor2.sayHi(); // I am professor Provillard.
```

- L'appel à bind permet de modifier le contexte de la fonction en fixant la valeur de `this`.

Méthodes

- Il est possible de modifier le context d'appel d'une fonction à l'aide de la méthode `call`.
- L'appel `f.call(context, arg1, ..., argN)` exécute la fonction `f` avec les arguments `arg1, ..., argN` en liant `this` à `context`.
- Avec cette syntaxe, on peut emprunter des méthodes à d'autres objets.

```
let professor = {
  firstName: "Julien",
  lastName: "Provillard",
  sayHi() { alert(`${this.firstName} says hi!`); },
};
let professor2 = { firstName: "Horatiu", lastName: "Cirstea" };
professor.sayHi.call(professor2); // Horatiu says hi!
// <=> professor.sayHi.bind(professor2)()
```

Méthodes

- De manière générale, écrire

```
let g = f.bind(context, arg1, ..., argN)
```

revient à definir la fonction g par

```
let g = function(argN+1, ..., argM) {
   f.call(context, arg1, ..., argN, argN+1, ..., argM)
};
```

- Habituellement, on appelle `bind` avec un seul argument.
- Avec plus d'arguments, on peut simuler des appels de fonction partiels.

Constructeurs

- Les constructeurs sont des fonctions destinées à initialiser de nouveaux objets.
  - On les appelle à l'aide du mot clé `new`.
  - Leur nom commence conventionnellement par une majuscule.
  - Elles initialisent un nouvel objet référencé par `this` qu'elles renvoient automatiquement (`return` non nécessaire).

```
function File(fileName) {
  this.fileName = fileName;
  this.editable = true;
}
let file = new File("toto.txt"); // { fileName: "toto.txt", editable: true }
```

Symboles

- Cas d'usage:
  - Une équipe de développement propose une nouvelle bibliothèque basée sur des objets d'une certaine forme.
  - Des utilisateurs exploitent cette bibliothèque et ajoute une propriété `randomName` pour leurs besoins propres.
  - L'équipe de développement veut ajouter une nouvelle fonctionnalité et a besoin d'ajouter une nouvelle propriété `randomName` pour cela… Conflit de noms !
- Solutions possibles:
  - Mapping des objets vers les nouvelles données (pas toujours évident/efficace).
  - Encapsulation dans un décorateur (pas idéal, surtout en JS).
  - Propriétés cachées (comment?)

Symboles

- Les symboles sont un type primitif destiné à servir de clé à une propriété.
- On peut créer un symbole à l'aide de la fonction `Symbol`.

```
let s1 = Symbol();
let s2 = Symbol("description") // argument utilisé dans toString par exemple
```

- La fonction renvoie toujours des symboles distincts.

```
Symbol("foo") == Symbol("foo") // false
```

- Pour utiliser un symbole comme clé de propriété, il faut passer par la notation entre crochets.

Symboles: exemple

```
let data = { }; // données visibles par l'utilisateur

let nsa = { // entité extérieure quelconque
  key: Symbol("nsaSecret"),
  read(data) { return data[this.key]; },
  write(data, comment) { data[this.key] = comment; }
}

nsa.write(data, "has donuts");
```

- Si l'utilisateur ne connaît pas la clé `nsa.key`, il n'a aucun moyen d'accéder à la propriété associée… à moins d'utiliser des méthodes très spécifiques (*e.g.* Object.getOwnPropertySymbols).

Symboles

- Les propriétés symboliques n'apparaissent pas dans les boucles `for in`.
- Elles sont bien copiées par `Object.assign`.
- Javascript prédéfinit plusieurs symboles pour son usage interne :
  - `Symbol.iterator`
  - `Symbol.toPrimitive`
  - …
- L'utilisateur a accès à une table d'association globale entre chaînes de caractères et symboles.
  - `Symbol.for("name") // création si nécessaire`
  - `Symbol.keyFor(symb)`

Objets enveloppes

- Les types primitifs représentent une unique valeur.
- Les objets peuvent contenir plusieurs valeurs dont des méthodes.
- Mais pourquoi alors peut-on écrire ?

```
(3).toString()
"hello".toUpperCase()
```

- Un objet enveloppe est créé à partir des constructeurs `String`, `Number`, `Boolean`, `Symbol` et `BigInt`.
- La méthode est appelée sur l'objet enveloppe puis celui-ci est détruit.
- Ces constructeurs contiennent aussi de nombreuses méthodes utilitaires.

LES TABLEAUX

Tableaux

- Les objets sont des tables associatives. Pour des structures ordonnées, on préfère des tableaux… qui sont des objets particuliers.
- On crée un tableau à l'aide :
  - du constructeur associé, `let array = new Array();`
  - de manière littérale, `array = [null, 21, false, "zero", {}];`
- Implicitement les valeurs du tableau sont liées à des clés numériques croissantes.

```
array[3] = array[1] * 2;
```

- Les indices autorisés s'étendent de `0` à `array.length - 1`.

Tableaux

- Pour itérer sur un tableau, on a les boucles `for` classiques.

```
for (let i = 0; i < array.length; i++) {
  process(array[i]);
}
```

- Si on n'a pas besoin de l'indice, on peut utiliser la boucle `for of` adaptée pour les objets itérables (dont les tableaux)

```
for (let value of array) {
  process(value);
}
```

- On peut techniquement utiliser une boucle `for in` mais c'est déconseillé (plus lent, attrape les propriétés non numériques).

Méthodes importantes

- Pour ajouter ou extraire des éléments, on utilise
  - `pop` / `push`: pour la fin d'un tableau O(1)
  - `shift` / `unshift`: pour le début du tableau O(length)
- Les méthodes `push` et `unshift` prennent un nombre quelconque d'arguments.

```
let array = [3];
array.push(4, 5);
array.unshift(0, 1, 2); // array == [0, 1, 2, 3, 4, 5]
```

Méthodes importantes

- La méthode `splice` permet de supprimer et/ou d'insérer des éléments de/dans un tableau.
- L'appel `array.splice(index, [nb, [e1, e2, …, eN]])` supprime `nb` éléments de `array` à partir de l'indice `index` et insère les éléments `e1`, `e2`, …, `eN` à la place.

```
let array = [0, 1, 2, 3, 4];
array.splice(2, 2); // array devient [0, 1, 4], suppression pure
array.splice(-2, 1, 5, 6); // array devient [0, 5, 6, 4], indice négatif OK
array.splice(2, 0, 7, 8); // array devient [0, 5, 7, 8, 6, 4], insertion pure
```

Méthodes importantes

- La méthode `slice` permet de dupliquer une tranche d'un tableau.
- L'appel `array.slice([from, [to]])` renvoie un tableau qui contient les éléments de array entre les indices `from` (inclus) et `to` (exclus). Par défaut, `from = 0` et `to = array.length`.

```
let array = [0, 1, 2, 3, 4];
array.slice(); // renvoie [0, 1, 2, 3, 4]
array.slice(3); // renvoie [3, 4]
array.slice(1, 3); // renvoie [1, 2]
array.slice(1, -1); // renvoie [1, 2, 3]
array.slice(-2, -1); // renvoie [3]
```

Méthodes importantes

- La méthode `concat` permet de dupliquer une tranche d'un tableau.
- L'appel `array.concat(arg1, arg2, …, argN)` renvoie un tableau qui contient les éléments de array concaténés avec les arguments.
  - Si un argument est un tableau, ses éléments sont ajoutés.
  - Sinon, la valeur est ajoutée telle quelle.

```
let array = [0, 1, 2];
let array2 = array.concat(3, 4, [5, 6], [7, [8], 9]);
// renvoie [0, 1, 2, 3, 4, 5, 6, 7, [8], 9]
```

Méthodes importantes

- Les tableaux possèdent des méthodes de recherche classiques :
  - `indexOf(item, pos)` et `lastIndexOf(item, pos)`

  Renvoie l'indice de `item` dans le tableau à partir de `pos`, recherche par indices croissants ou décroissants.
  - `includes(item)`

  Prédicat d'appartenance.
- On peut inverser les indices d'un tableau avec la méthode `reverse`.

Méthodes importantes

- On peut trier un tableau sur place avec la méthode `sort`.
- Par défaut, l'ordre utilisé est l'ordre lexicographique sur la représentation des éléments du tableau.
- On peut préciser l'ordre à utiliser en paramètre de `sort`.

```
let array = [10, 0, 5, 1];
array.sort(); // array devient [0, 1, 10, 5] ???
array.sort((a, b) => a - b); // array devient [0, 1, 5, 10]
array.sort((a, b) => { // comprendre le tri par défaut
  let strA = String(a);
  let strB = String(b);
  return strA < strB ? -1 : strA > strB ? 1 : 0;
});
```

Méthodes importantes

- L'appel `str.split(delim)` renvoie dans un tableau les éléments de `str` séparés par `delim`.

```
"note1, note2, note3".split(", "); // renvoie ["note1", "note2", "note3"]
```

- L'appel `array.join(delim)` transforme les éléments de `array` en chaînes de caractères et les concatènes, séparés par `delim`.

```
[3, 2, 1, "ignition"].join("-") // renvoie "3-2-1-ignition"
```

Méthodes importantes

- Il est possible de manipuler les tableaux à l'ordre supérieur.
  - La méthode `forEach` applique une fonction à tous les éléments d'un tableau.
  - La méthode `map` applique une fonction à tous les éléments d'un tableau et collecte les résultats dans un nouveau tableau.
  - La méthode `filter` collecte les éléments du tableau qui vérifient un prédicat.
  - Les méthodes `reduce` et `reduceRight` permettent d'accumuler les éléments d'un tableau à l'aide d'une fonction d'accumulation et d'une valeur initiale. Le tableau est parcouru de gauche à droite pour `reduce` et de droite à gauche pour `reduceRight`.

Exemple

```
const coeff = [0.3, 0.3, 0.4];

let dataCSV = `id;note1;note2;note3
alpha;7;12;11
beta;8;5;2
gamma;14;8;15`;

let lines = dataCSV.split("\n").map((line) => line.split(";"));
let headers = lines[0];
let data = lines.slice(1);

headers.push("moyenne");
data.forEach(addMean);
data.sort((s1, s2) => s2.at(-1) - s1.at(-1));
let newCSV = [headers].concat(data).map((line) => line.join(";")).join("\n");
alert(newCSV);

function addMean(line) {
  let mean = line.slice(1).map((val, i) => val * coeff[i]).reduce((a, b) => a + b, 0);
  line.push(mean);
}
```

Déstructuration

- On peut déstructurer un tableau :

```
let [v1, v2] = [1, 2, 3, 4]; // v1 = 1, v2 = 2
```

- On peut ignorer certains indices :

```
let [v1, , v3] = [1, 2, 3, 4]; // v1 = 1, v3 = 3
```

- Comme pour un objet quelconque, il est possible de donner des valeurs par défaut et/ou d'assigner des variables déclarées.

```
let [v1 = 1, v2 = 2] = [10]; // v1 = 10, v2 = 2
[v1, v2] = [v2, v1]; // v1 = 2, v2 = 10
```

Paramètre de reste

- L'opérateur de reste (...) permet de collecter dans un tableau toutes les valeurs restantes d'une décomposition partielle.

```
let array = [10, 20, 30, 40];
let [v1, v2, ...v] = array; // v = [30, 40]
```

- Il fonctionne aussi sur les objets.

```
let rectangle = { width: 20, height: 10, border: "black", color: "blue" };
let { width, height, ...options } = rectangle;
// options = { border: "black", color: "blue" }
```

- On l'utilise beaucoup dans les fonctions.

```
function sum(...args) { return args.reduce((a, b) => a + b, 0); }
let res = sum(1, 2, 3, 4); // 10
```

Opérateur de décomposition

- L'opérateur de décomposition (`...`) permet de « déplier » un tableau.
- Écrire `...[v1, v2, v3]` revient à écrire `v1, v2, v3`.

```
let array = [1, 2, 3, 4];
alert(sum(array)); // 01,2,3,4 ???
alert(sum(...array)); // 10
let array2 = [0, ...array, 5]); // [0, 1, 2, 3, 4, 5]
```

Des objets aux tableaux

- L'appel `Object.keys(obj)` renvoie un tableau des clés des propriétés de `obj`.
- L'appel `Object.values(obj)` renvoie un tableau des valeurs des propriétés de `obj`.
- L'appel `Object.entries(obj)` renvoie un tableau des propriétés de `obj`. Chaque propriété est alors un tableau `[key, value]`.

```
let rectangle = { width: 20, height: 10 };
let keys = Object.keys(rectangle); // ["width", "height"]
let values = Object.values(rectangle); // [20, 10]
let entries = Object.entries(rectangle); // [["width", 20], ["height", 10]]
```