Développement Web

Julien Provillard

PROTOTYPES ET HÉRITAGE

Prototypes

- Tout objet javascript a un prototype qui est soit un autre objet soit `null`.
- Si on recherche une propriété dans un objet et qu'elle ne s'y trouve pas, la recherche continue récursivement dans le prototype jusqu'à la trouver ou atteindre le prototype `null`.
- Par défaut, les objets ont pour prototype `Object.prototype`.
- Les tableaux et les fonctions ont leurs propres prototypes.
- Les objets créés par l'instruction `new Constructor(...)` ont `Constructor.prototype` comme prototype.

Prototypes

- Le prototype d'un objet est stocké dans une propriété cachée nommée `__proto__`.
- Son usage est déprécié et n'est officiellement supporté que dans le navigateur.
- Dans les faits, on peut l'utiliser partout.
- Le Javascript moderne préfère utiliser :
  - l'accesseur `Object.getPrototypeOf(obj)`,
  - le mutateur `Object.setPrototypeOf(obj, proto)`.

Prototypes : exemples

```
const obj1 = {
  f() { alert("f de obj1"); },
  g() { alert("g de obj1"); },
  h() { alert("h de obj1"); },
};
const obj2 = {
  __proto__: obj1,
  f() { alert("f de obj2"); },
  g() { alert("g de obj2"); },
};
const obj3 = {
  __proto__: obj2,
  f() { alert("f de obj3"); },
};

obj3.f(); // f de obj3
obj3.g(); // g de obj2
obj3.h(); // h de obj1
```

Modifications

- L'écriture ou la suppression de propriété n'affecte que l'objet courant.

```
const obj1 = { name: "obj1" };
const obj2 = { __proto__: obj1 };

alert(obj2.name); // obj1
obj2.name = "obj2";
alert(obj2.name); // obj2
alert(obj1.name); // obj1
delete obj2.name;
alert(obj2.name); // obj1
delete obj2.name;
alert(obj2.name); // obj1
```

Prototypes et `this`

- Le mot clé `this` désigne toujours l'objet appelant même si la méthode appartient à un prototype.

```
const obj1 = {
  name: "obj1",
  setName(name) { this.name = name; },
};

const obj2 = { __proto__: obj1, name: "obj2" };

alert(obj2.name); // "obj2"
obj2.setName("foo");
alert(obj2.name); // "foo"
alert(obj1.name); // "obj1"
```

Itération et héritage

- Les boucles `for in` itèrent sur les propriétés héritées. Les autres façons d'itérer ignorent les prototypes (`Object.keys`, `Object.entries`, …).

```
const obj1 = { a: "a", b: "b" };
const obj2 = { __proto__: obj1, b: "newB", c: "c" };
for (let key in obj2) { alert(`${key}: ${obj2[key]}`); }
// b: newB, c: c, a: a
for (let [key of Object.keys(obj2)) { alert(`${key}: ${obj2[key]}`); }
// b: newB, c: c
```

- On peut savoir si une propriété est propre à un objet par :

```
obj2.hasOwnProperty("a"); // false
```

- On peut obtenir les propriétés propres d'un objet par :

```
Object.getOwnPropertyNames(obj2); // ["b", "c"]
```

Prototypes et itération

- Pourquoi la propriété `toString` n'apparaît pas dans les itérations ?

```
alert({}.toString); // [Function: toString]
for (let key in {}) { alert(key); } // rien
```

- Les propriétés ont des descripteurs.

```
alert(Object.getOwnPropertyDescriptor({a: 0}, "a"));
// { value: 0, writable: true, enumerable: true, configurable: true }
alert(Object.getOwnPropertyDescriptor({}.__proto__, "toString"));
/* { value: [Function: toString],
     writable: true,
     enumerable: false,
     configurable: true } */
```

Descripteurs

- On peut modifier les descripteurs.

```
const obj = {};
Object.defineProperty(obj, "a", { value: 0, configurable: true });
// writable et enumerable implicitement à faux
alert(obj.a); // 0
for (let key in obj) { alert(key); } // rien
obj.a = 1; // erreur
```

- `writable` : autorisation en écriture,
- `enumerable` : apparition dans les énumérations,
- `configurable` : si faux, non supprimable et descripteurs fixes.

Objets vides

- Il est parfois préférable d'avoir un objet réellement vide (sans propriété même non énumérable).
- On peut définir un tel objet par `{ __proto__: null }` ou plus proprement par `Object.create(null)`.
- L'appel `Object.create(proto)` créé un objet vide avec proto comme prototype.

```
alert("toString" in {}); // true
alert("toString" in Object.create(null)); // false
```

Retour sur les constructeurs

- Toutes les fonctions ont une propriété `prototype`. Par défaut, pour une fonction `F`, elle vaut : `{ constructor: F }`.
- Le prototype de tous les objets créés à partir d'un constructeur `F` est `F.prototype`.

```
alert(new F().__proto__ === F.prototype); // true
```

- Pour éviter la duplication des méthodes dans chaque objet, on les place de préférence dans le prototype.
- On évite d'assigner `F.prototype`. On préfère lui ajouter des propriétés.
  - Évite de perdre des méthodes,
  - Permet aux interpréteurs de maintenir des optimisations.

Constructeurs et méthodes

```
function List() { this.head = null; }

List.prototype.isEmpty = function () { return this.head == null; };

List.prototype.get = function (index) {
  let node = this.head;
  while (node && index > 0) {
    node = node.next;
    index--;
  }
  return node?.value; // ⇔ node != undefined ? node.value : undefined
};

List.prototype.add = function (value) {
 this.head = { value, next: this.head };
};
```

Chaîne d'héritage

- Pour réaliser l'héritage entre deux constructeurs `Pere` et `Fils`, il faut :
  - Que le constructeur `Pere` soit appelé en partageant la valeur `this` du constructeur `Fils`,
  - Que le prototype de `Fils.prototype` soit `Pere.prototype`.

- Pour accéder aux méthodes de `Pere`, notamment en cas de redéfinition dans `Fils`, il faut passer explicitement par `Pere.prototype` .

Chaîne d'héritage : exemple

```
function LoggedList(logger) { // LoggedList hérite de List
  List.call(this);            // logger doit être de la forme
  this.logger = logger;       // { log: (string value) => void }
}

Object.setPrototypeOf(LoggedList.prototype, List.prototype);

LoggedList.prototype.add = function (value) {
  this.logger.log(`adding ${value}`);
  List.prototype.add.call(this, value);
};

const list = new LoggedList(console);
[3, 2, 1, 0].forEach((i) => list.add(i));
```

CLASSES

Enfin !

Classes

- La définition de constructeurs et de chaînes de prototypage peut-être grandement simplifiée par l'utilisation de classes.
- La syntaxe de base est :

```
class Class {
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  ...
  methodN() { ... }
}
```

Exemple

```
class List {
  constructor() { this.head = null; }
  isEmpty() { return this.head == null; }
  get(index) {
    let node = this.head;
    while (node && index > 0) {
      node = node.next;
      index--;
    }
    return node?.value;
  }
  add(value) {
    this.head = { value, next: this.head };
  }
}
```

Différences avec les constructeurs

- Une classe est bien une fonction qui s'utilise comme un constructeur.

```
alert(typeof List); // function
const list = new List();
```

- Le code à l'intérieur d'une classe est toujours en mode strict.
- Par défaut, les méthodes d'une classe sont non-énumérables.
- Le constructeur d'une classe a une propriété spéciale qui indique qu'il est issu d'une classe. Il **doit** être appelé avec `new`.

```
function LoggedList(logger) { // LoggedList hérite de List
  List.call(this); // ne fonctionne plus car List est désormais une classe.
  this.logger = logger;
}
```

Fonctionnalités classiques

```
class Test {
  static #count = 0; // privé et statique (récent)
  #value = 0; // privé (récent)
  constructor(value) {
    if (value) { this.value = value; } // appelle le mutateur
    alert(`${++Test.#count} object(s) created`);
  }

  get value() { return this.#value; } // accesseur

  set value(value) {// mutateur
    alert(value);
    this.#value = value;
  }
}
```

Héritage

- On peut utiliser les mots-clés `extends` et `super` de la même manière que dans le langage Java.

```
class LoggedList extends List {
  constructor(logger) {
    super(); // obligatoirement en premier
    this.logger = logger;
  }
  add(value) {
    this.logger.log(`adding ${value}`);
    super.add(value);
  }
}
```

Particularités

- Les mots-clés `class` et `extends` fonctionnent dans des contextes dynamiques.

```
function makeClass(title) {
  return class {
    constructor(name) { this.name = `${title} ${name}`; }
  };
}

class Professor extends makeClass("Professor") {
  // constructeur par défaut <=> constructor(...args) { super(...args); }
  display() { alert(this.name); }
}

new Professor("Julien").display(); // Professor Julien
```

Particularités

- La liaison du mot clé `super` est fixe.

```
class A { f() { alert("Je suis un A"); } }
class B extends A {
  f() {
    super.f();
    alert("Mais surtout un B");
  }
}

const obj = { // ressemble à un B avec une classe mère différente
  __proto__: { f() { alert("Je suis un imposteur"); } },
  f: B.prototype.f,
};
obj.f(); // super.f() appelle bien A.prototype.f() et non obj.__proto__.f()
```

GESTIONS DES ERREURS

Expression `try-catch`

- La gestion des erreurs se fait en plaçant le code pouvant générer un problème dans un bloc `try` et de prévoir une gestion particulière dans un bloc `catch`.

```
try {
  // code pouvant générer une erreur
} catch (err) { // objet contenant des informations sur l'erreur
  // code gérant l'erreur
}
```

- Comme en Java, on peut ajouter un bloc `finally` qui s'exécutera dans tous les cas après l'un ou l'autre des blocs `try-catch`.

Erreurs

- Une erreur peut-être lancée par l'expression `throw err` où `err` peut-être n'importe quelle valeur.

```
try { throw 0; } catch (err) { alert("got it"); } // ne pas faire ça
```

- Une erreur est usuellement un objet avec (au moins) les propriétés :
  - `name` pour le nom de l'erreur,
  - `message` pour la description de l'erreur.
- Il est préférable qu'une erreur soit une instance de la class `Error`. Vos classes d'erreurs personnelles devraient en hériter (directement ou indirectement).

Exemple

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name; // nom du constructeur initial
  }
}
class NoDataError extends ValidationError {
  constructor(message = "No data to process") { super(message); }
}
class MissingDataError extends ValidationError {
  constructor(property) {
    super(`The property {${property}} is missing.`);
    this.property = property;
  }
}
```

Exemple

```
function getData() {
  let r = Math.random();
  if (r < 0.1) { return null; } // pas de données
  if (r < 0.3) { return { name: "foo" }; } // donnée incomplète
  return { name: "foo", id: 42 };
}

function validateData(properties, data) {
  if (!data) { throw new NoDataError(); }
  properties.forEach((name) => {
    if (!data[name]) { throw new MissingDataError(name); }
  });
}
```

Exemple

```
function processData(process) {
  const data = getData();
  try {
    validateData(["name", "id"], data);
    process(data);
  } catch (err) {
    if (err instanceof NoDataError) {
      alert("Erreur 403");
    } else if (err instanceof MissingDataError) {
      alert(`Donnée corrompue, champ ${err.property} manquant`);
    } else if (err instanceof ValidationError) {
      alert("Échec de la validation");
    } else { throw err; } // erreur inconnue, on la relance
  }
}
```

MODULES

Présentations

- Pour un projet conséquent, vos scripts seront répartis dans plusieurs fichiers. Chaque fichier est un module qui pourra importer et exporter des fonctionnalités.
- Une classe devrait occuper un fichier à elle seule.
- Deux systèmes de gestion des modules existent et se côtoient en ce moment : CommonJS et ECMAScript module.
- Les nouvelles ressources devraient être des modules ES.
- Les modules ES sont compatibles avec les modules CommonJS.
- Attention, par défaut Node utilise des modules CommonJS.

Modules ECMAScript

- Dans une page Web, il faut ajouter l'attribut `type="module"` dans la balise `script`. On a alors accès à tous les modules locaux en spécifiant des chemins relatifs ou absolus.
- **Il faut nécessairement passer par un serveur pour la page Web**. L'extension *Live Server* de VSCode est utile pour cela.

- Dans une application Node, il faut ajouter `"type": "module"` dans le fichier `package.json`. On a accès à la fois aux modules locaux et à l'écosystème de modules de `npm`.
- En ajoutant cette association, le système de modules passe à ES pour le projet.

Exemple

```
// fichier random.js
export function randInt(from, to) {
  return Math.floor(Math.random() * (to - from) + from);
}
export function randomItem(t) {
  return t[randInt(0, t.length)];
}
```

```
<!-- fichier random.html -->
<script type="module">
  import { randomItem } from "./random.js";
  const t = ["foo", "bar", "baz"];
  alert(randomItem(t));
</script>
```

Les deux fichiers doivent être servis par le même serveur !

requête `GET` sur `"./random.js"`

Spécificité des modules

- Chaque module dispose de son propre environnement (noms indépendants entre modules).
- Dans la portée globale d'un module, `this` est `undefined`.
- Ils sont automatiquement en mode strict.
- Les modules se chargent en parallèle du code html et s'exécutent une fois celui-ci entièrement chargé. L'ordre relatif des scripts est maintenu.
- On peut exécuter un module dès que possible en ajoutant l'attribut `async` dans la balise `script`.

Exporter des définitions

- Pour exporter une définition (variable, fonction, classe), il suffit d'ajouter le mot clé `export` avant la définition.
- On peut également spécifier ce que l'on souhaite exporter du module en écrivant `export { nom1, ..., nomN };`
- Les deux possibilités ne sont pas exclusives tant qu'un même nom n'est exporté qu'une seule fois.

```
export function randInt(from, to) {
  return Math.floor(Math.random() * (to - from) + from);
}
function randomItem(t) {
  return t[randInt(0, t.length)];
}
export { randomItem };
```

Importer des définitions

- Il faut lister les fonctionnalités voulues et indiquer leur module d'origine.

```
import { randomItem } from "./random.js"; // randomItem disponible
```

- L'extension du fichier est obligatoire. L'import peut être partiel.

- On peut importer l'ensemble des exports d'un fichier et les stocker dans un objet.

```
import * as random from "./random.js"; // random.randomItem disponible
```

Renommage

- On peut renommer les définitions à l'export comme à l'import.

```
// fichier random.js
const { floor, random } = Math;
function randInt(from, to) { return floor(random() * (to - from) + from); }
function randomItem(t) { return t[randInt(0, t.length)]; }
export { randInt, randomItem as randItem };

<!-- fichier random.html -->
<script type="module">
  import { randItem as pickOne } from "./random.js";
  alert(pickOne(["foo", "bar", "baz"]));
</script>
```

Export par défaut

- Un module peut avoir un (au maximum) export par défaut.
- La définition est précédée de `export default`.
- Il suffit alors d'importer le module vers un nom arbitraire.

```
// fichier example.js
export default class Example { ... }

// fichier useExample.js
import Example from "./example.js";

// fichier useExampleAgain.js
// à l'import, le nom n'a pas besoin de correspondre à l'export
import SomeClass from "./example.js";
```

Export par défaut

- Comment combiner export par défaut et exports classiques ?

```
// fichier random.js
const { floor, random } = Math;
export default function randInt(from, to) {
  return floor(random() * (to - from) + from);
}
export function randomItem(t) { return t[randInt(0, t.length)]; }

// solution 1, deux imports
import randInt from "./random.js";
import { randomItem } from "./random.js";
```

Export par défaut

- Comment combiner export par défaut et exports classiques ?

```
// fichier random.js
const { floor, random } = Math;
export default function randInt(from, to) {
  return floor(random() * (to - from) + from);
}
export function randomItem(t) { return t[randInt(0, t.length)]; }

// solution 2, renommage de l'export par défaut
import { default as randInt, randomItem } from "./random.js";
```

Modules CommonJS

- Depuis un module ES, on peut importer des modules CommonJS.
- Les modules CommonJS exportent toujours une valeur nommée `exports`.
- Depuis un module ES, on peut récupérer cet valeur en tant qu'export par défaut.
- De manière générale, il faut se référer à la documentation pour exploiter correctement un module.

Récapitulatif

- Vous avez étudié :
  - les types primitifs,
  - les structures de contrôle,
  - les objets et les tableaux,
  - l'aspect orienté objet du langage,
  - la gestion d'erreur,
  - l'organisation du code.
- Vous en savez assez pour développer en Javascript.
- Ce qui vient ensuite va consister en des points particuliers du langage et de son usage (interaction avec la page web, architecture client/serveur).