Labo 4B : prototypes et constructeurs

Objectifs

- Manipuler des objets et des prototypes pour bien comprendre leur fonctionnement
- Créer et utiliser une fonction constructrice via la console
- Manipuler les prototypes prédéfinis
- Critiquer et corriger du code orienté objet en Javascript

Exercice 1 : paragraphes et prototypes

Le but de cet exercice est de manipuler l'héritage prototypal sur un exemple assez simple afin de bien en comprendre le fonctionnement. Les objets utilisés correspondent à des paragraphes qui peuvent être ajoutés au document HTML.

Étape 1 : une page web comme cadre de travail

Créez tout d'abord le document suivant, qui servira de base à cet exercice. Examinez son contenu et comparez-le avec ce que le navigateur affiche quand vous l'ouvrez.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <style>
      #cadre {
        background-color: #EFFFEF;
        border: 1px solid blue;
        padding: 4px;
      }
    </style>
  </head>
  <body>
    <div id="cadre">
      <p>Le cadre commence ici.</p>
    </div>
  </body>
</html>
```

Ouvrez la page sous Firefox et ouvrez la console. Tout l'exercice qui suit se déroule dans la console. Vous allez construire des objets qui correspondent à de nouveaux paragraphes qui pourront être affichés dans le cadre `#cadre`. Pour rappel, pour ajouter du texte à l'intérieur du cadre, vous pouvez vous inspirer du code suivant.

```
let cadreHTML = document.getElementById("cadre");
cadreHTML.innerHTML += "<p>Nouveau paragraphe</p>";
```

Cela peut également se faire en une seule commande :

```
document.getElementById("cadre").innerHTML +=
  "<p>Nouveau paragraphe</p>";
```

mais cela implique que Firefox devra rechercher l'élément identifié par `#cadre` à chaque ajout. (Dans le cadre de cet exercice, ce n'est pas dramatique mais, dans le cadre d'un script plus long, il peut être intéressant de stocker l'élément-cible dans une variable…)

Étape 2 : deux premiers paragraphes

Définissez les deux objets suivants appelés `p1` et `p2` et censés représenter des paragraphes à afficher.

Objet	Propriétés	
	Nom	**Valeur**
p1	`texte`	"Quelle prétention de prétendre que l'informatique est récente : Adam et Ève avaient déjà un Apple !"
	`auteur`	"Anonyme"
	`couleur`	"purple"
	`gras`	true
p2	`texte`	"Cookie : anciennement petit gâteau sucré qu'on acceptait avec plaisir. Aujourd'hui : petit fichier informatique drôlement salé qu'il faut refuser avec véhémence."
	`auteur`	"Luc Fayard"
	`couleur`	"darkgreen"
	`gras`	false

Dans la console, affichez les valeurs de `p1` et de `p2` puis, en cliquant sur Object, observez leur contenu dans la partie droite de la console. Notez entre autres la valeur de la propriété `__proto__`.

« Object » est le prototype par défaut, de même que la classe « Object » est automatiquement classe-mère de toutes les classes créées en Java. Si vous cliquez sur le triangle à gauche de `__proto__ : Object`, vous pouvez avoir un aperçu de toutes les propriétés que l'objet `Object` possède et dont les objets nouvellement créés héritent.

Parmi celles-ci se trouve la méthode `.toString()` qui, comme en Java, est appelée automatiquement lorsqu'il faut convertir un objet en chaînes de caractères. Vous pouvez examiner son effet et tester son fonctionnement en entrant une après l'autre les lignes suivantes. (Comme en Java, la définition par défaut de `toString` n'est pas très utile.)

```
p1.toString();
"Hello " + p2;   "hello [object Object]"
```

Étape 3 : un premier prototype

Définissez un troisième objet appelé `protoPara` en tant qu'objet vide. Affichez sa valeur dans la console et cliquez sur Objet pour vérifier qu'il est bien « vide » (enfin, si on omet la propriété par défaut `__proto__` et tout ce dont il hérite).

Ajoutez-lui la méthode suivante.

```
protoPara.toString = function () {
  return this.texte + " (" + this.auteur + ")";
```

```
};
```

Utilisez ensuite `setPrototypeOf` pour faire en sorte que `protoPara` devienne le prototype de `p1` et de `p2`. `Object.setPrototypeOf(p1,protoPara);`

Entrez à nouveau les lignes suivantes une par une et assurez-vous de comprendre pourquoi on obtient ces nouveaux résultats.

```
p1.toString();
"Hello " + p2;
```

Étape 4 : un prototype plus fourni

Ajoutez à `protoPara` une nouvelle méthode appelée `affiche()` et qui va ajouter au cadre `#cadre` un paragraphe…

- dont le contenu sera `this.texte` suivi d'un retour à la ligne, de trois tirets et de `this.auteur` entre des balises `<em>`.
- dont le texte sera écrit dans la couleur `this.couleur`,
- et dont le texte sera gras ou non gras en fonction de la valeur de `this.gras`.

Exécutez ensuite cette méthode sur les objets `p1` et `p2`. Voici le résultat que vous devriez obtenir.

Quelle prétention de prétendre que l'informatique est récente : Adam et Ève avaient déjà un Apple !
---anonyme

Cookie : anciennement petit gâteau sucré qu'on acceptait avec plaisir. Aujourd'hui : petit fichier informatique drôlement salé qu'il faut refuser avec véhémence.
---Luc Fayard

Étape 5 : deux autres paragraphes

Créez maintenant un objet `p3` en utilisant `Object.create` de manière à ce qu'il possède, dès sa création, `protoPara` comme prototype. Ajoutez-lui les propriétés suivantes.

Objet	Propriétés	
	Nom	**Valeur**
p3	`Texte`	"Y'a rien de plus con qu'un ordinateur."
	`auteur`	"Louis Schuffenecker"
	`couleur`	"blue"
	`gras`	true

Entrez ensuite la définition suivante dans la console Javascript (via un copier/coller).

```
let propriétésP4 = {texte: "On voit bien que tu n’as pas rencontré
certains de mes étudiants, Louis !", auteur: "enseignant anonyme",
couleur: "olive", gras: false};
```

Définissez maintenant un objet `p4` héritant de `protoPara` grâce à `Object.create`. Puis utilisez `Object.assign` pour ajouter à `p4` toutes les propriétés se trouvant dans `propriétésP4`.

Exécutez la méthode `affiche()` sur les objets `p3` et `p4`.

Exercice 2 : carrés et constructeurs

Cet exercice est similaire au précédent, si ce n'est qu'ici, on va utiliser une fonction constructrice pour créer les objets et donc gérer automatiquement l'héritage prototypal.

Étape 1 : une page web comme cadre de travail

Commencez par construire un fichier HTML contenant le code suivant.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <style>
      #cadre {
        background-color: #EFFFEF;
        border: 1px solid blue;
        width: 800px;
        height: 600px;
        position: relative;
        box-sizing: border-box;
      }
      #cadre div {
        box-sizing: inherit ;
      }
    </style>
  </head>
  <body>
    <div id="cadre">
    </div>
  </body>
</html>
```

Dans cet exercice, vous allez définir des objets correspondant à des carrés à afficher à l'intérieur du cadre défini dans la page HTML. Un carré sera représenté par un `<div>` positionné de manière absolue (c'est-à-dire par rapport au cadre qui le contient) avec une certaine couleur de fond et une certaine épaisseur de bordure.

Pour voir comment cela fonctionne, entrez les lignes suivantes dans la console Javascript.

```
let carréHTML = "<div style='position:absolute; left:20px; top:50px;
  width: 30px; height: 30px; border: 1px solid black;
  background-color: yellow'></div>";
document.getElementById("cadre").innerHTML += carréHTML;
```

Elles devraient afficher un carré jaune de 30 pixels de côté avec une bordure de 1 pixel d'épaisseur à 20 pixels du bord gauche du cadre et 30 pixels du bord supérieur du cadre.

Étape 2 : une fonction constructrice

Pour chaque carré, on va donc devoir retenir les informations suivantes : sa couleur, la longueur de son côté, l'épaisseur de sa bordure, son écart par rapport au bord gauche et son écart par rapport au bord supérieur.

Ces informations seront conservées respectivement dans les propriétés `couleur`, `côté`, `bordure`, `écartGauche` et `écartHaut`.

Créez une fonction constructrice `Carré` recevant ces 5 informations et les plaçant dans les propriétés correspondantes de l'objet `this`. Contentez-vous de copier aveuglément les valeurs données sans vous préoccuper de leur validité.

Utilisez ensuite la fonction pour créer un premier carré :

```
let carré1 = new Carré ("green", 50, 2, 100, 100);
```

Affichez la valeur de `carré1` dans la console puis cliquez sur `Object` pour voir son contenu. Outre les cinq propriétés définies par la fonction constructrice, on retrouve bien sûr une propriété `__proto__` qui référence un objet.

Cliquez sur le triangle à gauche de `__proto__` pour voir le contenu de cet objet. Vous devriez apercevoir deux propriétés : `constructor` et `__proto__`. La première référence la fonction constructrice `Carré` ; la seconde pointe vers l'objet-père `Object` (vous pouvez le vérifier en « déballant » la propriété `__proto__` en question : vous retrouvez alors les propriétés vues dans l'exercice 1).

Le prototype de `carré1` est donc un objet possédant une propriété `constructor` qui pointe vers la fonction `Carré`. Il s'agit de `Carré.prototype`, le prototype que la fonction `Carré` associe automatiquement à tous les objets qu'elle crée. Vérifiez-le en évaluant dans la console

```
Carré.prototype
```

puis en déballant le contenu de cet objet.

Vous pouvez également évaluer l'expression suivante.

```
Carré.prototype.isPrototypeOf(carré1) True
Carré.prototype.constructor == Carré   True
```

Étape 3 : bien ranger ses méthodes

Si on désire ajouter aux objets créés par Carré une méthode `toString` qui renvoie une chaîne de caractères similaire à

```
carré green de 50 px de côté
```

(où les parties en gras italique sont les parties variables), il faut placer cette méthode dans leur prototype commun, à savoir dans `Carré.prototype`.

Faites-le (utilisez un gabarit de chaînes de caractères) puis testez son fonctionnement en évaluant `carré1.toString()`, ce qui devrait donner le résultat présenté ci-dessus.

Étape 4 : afficher des carrés

L'objectif de cette étape est d'ajouter une méthode `affiche()` qui permet d'afficher un carré. Cela peut se faire directement en écrivant

```
Carré.prototype.affiche = function () { … }
```

mais, pour utiliser une syntaxe différente, cette fois-ci, définissez la tout d'abord sous la forme d'une simple fonction `affiche` (se référant toujours à `this` comme étant le carré).

```
function affiche () { … this.côté … this.écartGauche … }
```

Notez que vous ne pouvez pas (encore) tester cette fonction via `carré1.affiche()` vu qu'elle n'est pas encore placée dans le prototype, ni même via `affiche(carré1)` vu qu'elle n'est pas prévue pour recevoir un argument et qu'elle applique à `this`.

Pour la tester, vous pouvez utiliser la commande suivante

```
affiche.call(carré1);
```

qui sera revisitée plus tard dans le cours. La méthode `call` est une méthode dont toutes les fonctions héritent et qui permet d'exécuter leur code en précisant qui est l'objet `this` (dans ce cas-ci, `this` est `carré1`).

Note. Si, au cours des tests, vous avez besoin d'effacer certains des carrés dessinés dans le cadre, vous pouvez effectuer un clic droit sur les carrés en question, choisir l'option « Inspecter l'élément », puis supprimer les qui correspondent aux carrés en trop (via un clic droit sur le code HTML du puis « Delete Node »).

Une fois que vous serez satisfait du code de `affiche`, entrez la commande suivante.

```
Object.assign(Carré.prototype, { affiche });
```

Comprenez-vous ce qu'elle fait ? Et ce qu'est le deuxième argument entre accolades (indice : il s'agit d'une version raccourcie pour une syntaxe normalement plus longue) ?

Notez que, désormais, vous pouvez utiliser `carré1.affiche()`.

Définissez un second carré en utilisant la fonction constructrice et testez son affichage. Ce second carré, `carré2`, sera bleu, situé à 150px du bord gauche et 50px du bord supérieur, aura une bordure de 1 pixel et un côté de 100 pixels.

Exercice 3 : autoboxing et constructeurs prédéfinis

Au cours des séances précédentes, vous avez vu qu'il existait plusieurs types de données scalaires (ou élémentaires ou simples) en Javascript : les nombres, les chaînes de caractères, les booléens, et le type `undefined` (dont la seule valeur est `undefined`).

Javascript permet de traiter ces valeurs élémentaires comme des objets. Dans ce cas-là, la valeur élémentaire est automatiquement transformée en un objet qui représente la valeur. C'est ce qu'on appelle de l'autoboxing (la valeur simple est automatiquement mise dans une « boîte » qui est un objet). L'objet en question est parfois appelé « wrapper » (littéralement, l'emballeur).

L'objectif de cet exercice est de montrer ce phénomène d'autoboxing et de voir que les constructeurs prédéfinis (ceux utilisés pour construire les wrappers et les autres) peuvent être modifiés comme n'importe quels constructeurs.

Étape 1 : J'autoboxe, tu autoboxes, il autoboxe, nous autoboxons, …

Initialisez les variables suivantes dans la console de Firefox.

```
let nombre = 3.1415;
let chaîne = "Bonjour !";
```

Ensuite, évaluez les expressions suivantes une par une.

```
nombre.toString() "3.1415"
nombre.toFixed(6)  "3.141500"
nombre.toFixed(2) "3.14"
chaîne.length 9
chaîne.charAt(2) "n"
chaîne.charAt(5) "u"
chaîne.startsWith("Bon") true
chaîne.startsWith("b") false
```

Observez que chacune des expressions précédentes a utilisé les variables `nombre` et `chaîne` comme s'il s'agissait d'objets. La traduction (du nombre ou de la chaîne) en un objet s'est faite automatiquement. C'est le phénomène de l'autoboxing.

Étape 2 : Les prototypes des wrappers

Continuez avec les lignes suivantes…

```
let autreNombre = 7;
let autreChaîne = "oui";
```

puis examinez la valeur des expressions ci-dessous.

```
Object.getPrototypeOf(nombre) == Object.getPrototypeOf(autreNombre) true
Object.getPrototypeOf(chaîne) == Object.getPrototypeOf(autreChaîne) true
Object.getPrototypeOf(nombre) == Object.getPrototypeOf(chaîne)  false
```

Ainsi, tous les nombres (ou, plutôt, tous leurs wrappers) partagent le même prototype. Et il en va de même pour les chaînes de caractères.

On peut également voir que c'est dans ces prototypes que sont définies les méthodes utilisées plus haut.

```
let protoNombre = Object.getPrototypeOf(nombre);
protoNombre.toString // indique que c'est une fonction (qui existe)
protoNombre.toFixed  // indique que c'est une fonction (qui existe)
let protoString = Object.getPrototypeOf(chaîne);
protoString.charAt
protoString.startsWith
```

Étape 3 : les constructeurs des wrappers

On a déjà utilisé les fonctions `Number`, `String` et `Boolean` pour convertir des valeurs en nombres, chaînes de caractères ou booléens. Mais ces fonctions sont plus que de simples convertisseurs : il s'agit en fait de fonctions constructrices, comme le montre l'exemple suivant.

```
let porteBonheurChinois = new Number (8);
```

Si vous évaluez la valeur de `porteBonheurChinois`, la console indique

```
Number { 8 }
```

ce qui représente un wrapper pour le nombre 8, c'est-à-dire un objet représentant le nombre 8.

Testez un exemple concernant une chaîne de caractères, comme

```
new String ("Coucou")
```

et observez le résultat dans la console. Cliquez sur `String` pour voir les propriétés de l'objet en question et notez que ces propriétés sont associées à des clefs numériques, ce qui permet d'écrire par exemple

```
(new String ("Coucou"))[2] « u »
```

Comme les fonctions `Number`, `String` et `Boolean` sont des constructeurs, tous les objets qu'elles créent sont automatiquement liés à un prototype. Et le prototype en question est un objet tel que `Number.prototype`, `String.prototype` ou `Boolean.prototype`.

Vous pouvez le tester en évaluant l'expression

```
Object.getPrototypeOf(porteBonheurChinois) == Number.prototype
```

Et c'est ce même prototype qui est utilisé pour les wrappers.

```
Object.getPrototypeOf(nombre) == Number.prototype
```

Étape 4 : quelques autres constructeurs prédéfinis

Aux trois fonctions constructrices mises en évidence à l'étape précédente, on peut encore en ajouter au moins deux qui concernent des objets que vous avez déjà utilisés : le constructeur `Function` pour les fonctions et le constructeur `Array` pour les tableaux.

Notez que vous avez déjà utilisé ces fonctions comme des constructeurs :

```
let estPair = new Function ("x", "return (x % 2 == 0);");
let fibo = new Array (1, 1, 2, 3, 5, 8, 13);
```

Qui est le prototype de tous les tableaux ? Et qui est le prototype de toutes les fonctions ?

Vérifiez vos réponses en complétant les expressions suivantes et en vous assurant qu'elles s'évaluent bien à `true`.

```
Object.getPrototypeOf([1,2,3]) == …  Array.prototype
Object.getPrototypeOf(isNaN) == …  Number.prototype
```

Saurez-vous également compléter les lignes suivantes pour qu'elles s'évaluent à `true` ?

```
Object.getPrototypeOf(fibo).constructor == …
Object.getPrototypeOf(estPair(3)) == …
Object.getPrototypeOf("true").constructor == …
fibo instanceof …
isFinite instanceof …
```

Étape 5 : modifier des prototypes prédéfinis

Les prototypes prédéfinis, comme `Number.prototype`, peuvent être modifiés au même titre que ceux qui sont définis dans un script. Et ces modifications ont un effet rétroactif sur tous les objets qui possèdent ce prototype.

Par exemple, on pourrait définir une nouvelle méthode, sur les nombres, qui affiche une ligne composée d'étoiles, dont la longueur est le nombre en question. Le but est donc de faire en sorte que le code

```
let nombre = 4;
nombre.afficheÉtoiles();
```

affiche « **** » dans la console.

Ajoutez au prototype des nombres (à savoir `Number.prototype`) une méthode appelée `afficheÉtoiles()` qui réalise cette tâche puis testez la sur diverses valeurs.

Créez ensuite les méthodes suivantes.

- Une méthode `bégaie`, portée par toutes les chaînes de caractères, qui renvoie la chaîne de caractères obtenue en doublant chacun des caractères de la chaîne de départ.
- Une méthode `présente`, portée par toutes les fonctions, qui affiche dans la console le message « La fonction *nomfonction* attend *nbArguments* argument(s). » (évaluez un

nom de fonction dans la console et ouvrez le détail dans la partie droite de la console pour voir quelles propriétés permettent d'accéder à ces informations)

- Une méthode `somme`, portée par tous les tableaux, qui affiche la somme des éléments numériques qui se trouvent dans le tableau.
- Une fonction `ajouteSiPasPrésent`, portée par tous les tableaux, qui ajoute son argument comme élément du tableau si celui-ci n'est pas déjà présent dans le tableau.

Testez ces fonctions (entre autres) avec les exemples suivants.

```
"Javascript".bégaie()       // renvoie JJaavvaassccrriipptt
isNaN.présente()            // La fonction isNaN attend 1 argument(s).
parseInt.présente()         // … parseInt attend 2 argument(s).
[1, 3, 5].somme()           // 9
[1, true, 3, "44", 5, [6]].somme()
                            // également 9
[1, true, {}].ajouteSiPasPrésent(2)
                            // donne [1, true, {}, 2]
[1, true, {}].ajouteSiPasPrésent(true)
                            // donne [1, true, {}]
```

Exercice 4 : à vous de corriger !

Critiquez et corrigez chacun des bouts de code suivants relatifs à l'orienté objet en Javascript.

Bout de code 1 : fonction constructice pour des cercles

```
function cercle (rayon, couleur) {
  this.rayon = rayon;
  this.couleur = couleur;
  this.surface = function () {
    return Math.PI * rayon * rayon;
  }
}
```

Bout de code 2 : fonction constructrice pour des pions sur un damier

```
let protoPion = {
   droite () { this.positionX++;   },
   gauche () { this.positionX--; },
   haut () { this.positionY++; },
   bas () { this.positionY--; }
};

function Pion (positionX, positionY) {
  this.positionX = positionX;
  this.positionY = positionY;
  Object.setPrototypeOf(this, protoPion);
}
```

Dans le cas de ce bout de code, expliquez le comportement des lignes suivantes.

```
let p = new Pion (3, 4);
protoPion.isPrototypeOf(p)        // donne true
p instanceof Pion                 // donne false !
```

Bout de code 3 : attributs « statiques » dans le prototype ?

```
function Fenêtre (titre) {
   this.titre = titre;
   this.nbFenêtresCréées++;
}

Fenêtre.prototype.nbFenêtresCréées = 0;

Fenêtre.prototype.toString = function () {
   return `fenêtre ${this.titre}`;
}
```

```
Fenêtre.prototype.compte = function () {
   console.log (`${this.nbFenêtresCréées} fenêtre(s) créée(s)`);
}
```

Pouvez-vous expliquer le problème suivant ?

```
let f1 = new Fenêtre("principale");
f1 + ""                // affiche bien « fenêtre principale »
f1.compte()            // affiche bien « 1 fenêtre(s) créée(s) »
let f2 = new Fenêtre("annexe");
f2 + ""                // affiche bien « fenêtre annexe »
f2.compte()            // affiche « 1 fenêtre(s) créé(s) » !
                       // pourquoi f2 n'a-t-elle pas été comptée
```