Complément Web – CM3 : Utiliser des objets et implémenter des formulaires

Objectif

Aborder sommairement les objets en JavaScript, et apprendre les méthodologies de base pour gérer des formulaires.

I/ Les objets « basiques »

Jusqu'à maintenant, vous avez essentiellement utilisé des types dit « primitifs », tels que des nombres ou des strings, mais vous n'avez que très peu pu manipuler des objets. Un objet en JavaScript est une structure de données différente qui permet de stocker des entités plus complexes.

Mais vous avez utilisé des objets en JavaScript sans forcément le savoir pour le moment. Par exemple, l'objet « document » que vous utilisez pour faire des « document.querySelector » est un objet. Un autre exemple est l'objet « Event » que vous avez pu manipuler avec les « addEventListener() »

A) Les bases

Un objet permet d'associer une quantité indéfinie de clés/valeurs, en sachant que la valeur peut être n'importe quoi (un string, un nombre, null, un tableau, un autre objet, une fonction...). De façon un peu plus visuelle, un objet peut être vu comme un classeur. Chaque donnée stockée dans ce classeur est identifiée par une clé. Cette clé permet d'accéder à la donnée qu'elle contient. Il est facile d'ajouter/supprimer/modifier des clés.

Ce chapitre va principalement s'intéresser sur la création d'objets « simples », mais sachez que des objets plus complexes peuvent être définis, tel que les Arrays, Events, Error... qui sont des objets plus structurés.

B) Utiliser des objets simples en JS

Un objet vide peut être initialisé de la façon suivante :

```
const myObj = {};
```

Ce nouvel objet ne contient aucune propriété. Mais un objet peut être adapté à volonté après sa déclaration.

On peut directement initialiser un objet avec des propriétés avec cette syntaxe :

```
const user = {
  name: "Jean-Michel",
  age: 42,
};
```

Ainsi, une propriété de l'objet se définit en tant que « clé: valeur » ; la clé est écrite avant les « : » et la valeur est écrite après. Chaque propriété d'un objet est séparée par une virgule afin de les délimiter.

Cette notation prend aussi le nom de JSON, qui est un acronyme pour « JavaScript Object Notation »

1) Lecture/Ecriture/Suppression simple des valeurs dans un objet

La façon la plus classique pour accéder aux propriétés d'un objet est l'utilisation de la notation « . ». En reprenant l'exemple ci dessus, voici ce qu'on peut en faire ;

```
console.log(user.name); // "Jean-Michel"
console.log(user.age); // 30
console.log(user.country); // undefined (La propriété n'existe pas dans l'objet user)
```

Cette façon d'accéder aux propriétés peut également être utilisée pour affecter des valeurs. Par exemple

```
const user = {
  name: "René",
  age: 30,
};
console.log(user.isAdmin); // undefined
user.isAdmin = true;
console.log(user.isAdmin); // true
```

Pour supprimer une propriété, l'instruction « delete » peut être utilisée

```
delete user.age;
```

2) Pour aller plus loin...

Une clé dans un objet peut prendre virtuellement n'importe quelle valeur (à une exception près), y compris des clés avec espaces, caractères spéciaux, ou encore mots réservés :

```
const user = {
  "nom nominatif complet et exhaustif à souhait":
    "Jean-Michel Baudouin de la Panaque",
  age: 60,
  if: [10, 42, 100],
  "(/◕ヮ◕)/*:･ﾟ✧": true,
};
```

Petite nuance à noter, les propriétés qui possèdent des espaces ne peuvent être accédées avec la notation « . ». Il faut à la place utiliser des brackets :

```
console.log(user.nom nominatif complet et exhaustif à souhait); // Erreur
console.log(user["nom nominatif complet et exhaustif à souhait"]); // "Jean-Michel
Baudouin de la Panaque"
```

Cette notation avec des brackets a aussi d'autres avantages. Elle permet par exemple d'accéder dynamiquement à une propriété d'un objet :

```
const user = {
  name: "René",
  age: 30,
};
const property = prompt("Which property would you like to access?");
alert("The property '" + property + "' has the value '" + user[property] + "'");
```

3) « Boucler » sur un objet

Un objet n'est pas un simple tableau. Mais il existe tout de même des façons de faire des boucles dessus, ce qui peut s'avérer pratique :

```
const user = {
  name: "John",
  age: 30,
  isAdmin: true,
};

for (let key in user) {
  // Log des clés
  console.log(key); // name, age, isAdmin
  // Log des valeurs
  alert(user[key]); // John, 30, true
}
```

A noter qu'il existe d'autres façons de parcourir des objets avec des APIs plus avancées.

C) Références et copie d'objets

1) Le concept des références

Les objets ont une différence fondamentale sur leur fonctionnement comparé aux valeurs primitives. Par exemple :

```
let message = "Bonjour";
let phrase = message;
```

Le résultat est deux variables message et phrase complètement indépendants qui chacune stockent leur string "Bonjour".

Les objets à l'inverse ne fonctionnent pas comme ça. Une variable assignée à un objet ne stocke pas l'objet lui-même, mais une adresse dans la mémoire, ou en d'autres termes, une référence.

Par exemple :

```
const initialObject = { name: "John" };
const newObject = initialObject;
newObject.name = "Lenon";
console.log(newObject); // { name: "Lenon" }
console.log(initialObject); // { name: "Lenon" }
```

Effectivement, le fait d'avoir modifié « newObject » a également changé « initialObject », car ces deux variables stockent la référence vers le même objet en mémoire. Ainsi, la modification de l'un modifie l'autre.

2) Égalités d'objets

Lorsqu'on vérifie l'égalité d'objets avec la notation « == », c'est la référence qui est comparée, et non pas sa valeur !

Application :

```
const object1 = {}; // Objet vide
const object2 = {}; // Autre objet vide
const object1Copy = object1; // Copie de la référence de object 1
console.log(object1 == object2); // false -> les deux objets, bien que identiques par leur valeur, n'ont pas la même référence
console.log(object2 == objectCopy1); // false -> Même idée, les deux objets ont la même valeur, mais pas la même référence.
console.log(object1 == object1Copy); // true -> il s'agit ici de la même référence, et cette égalité est donc vérifiée
```

Ainsi, si on veut tester une égalité d'objet par rapport à sa valeur, il n'est donc pas possible de tester en faisant une comparaison simple entre les variables. Il existe plusieurs façons de tacler ce problème, mais généralement, l'idée consiste à vérifier la valeur associée à chaque clé de l'objet, et confronter l'égalité à la valeur dans l'autre objet.

Vous pouvez trouver des fonctions sur internet qui répondent à vos besoins en fonction du contexte.

Il est tout de même important de garder en tête ce fonctionnement particulier par rapport aux objets. Si vous comprenez le fonctionnement, il sera beaucoup plus simple de résoudre des problèmes lié à ça dans vos futurs développements et projets.

N'obliez pas non plus qu'un objet peut contenir aussi des sous objets dans ses clés ; ces sous-objets sont aussi affectés par référence et non pas par valeur !

3) Copier un objet sans garder sa référence

La plupart du temps, utiliser les références des objets est suffisant, mais il y a des situations où on a besoin de travailler avec un clone de l'objet en question. Par exemple :

```
const user = {
  name: "John",
};
const addPassword = (user) => {
  user.password = "123";
  return user;
};
console.log(addPassword(user)); // { name: "John" , password: "123" }
console.log(user); // { name: "John" , password: "123" }
```

Dans l'exemple ci-dessus, on a passé l'utilisateur en paramètre de la fonction « addUser », mais celle-ci ayant modifié le paramètre directement, l'objet initial est modifié avec. Pour éviter de muter l'objet initial, il convient de cloner cet objet au préalable.

Une façon simple est d'utiliser la fonction « spread » de JavaScript qui permet un clone sur un niveau. Cette fonction s'utilise avec un « … » comme sur l'exemple suivant :

```
const user = {
  name: "John",
};
const addPassword = (user) => {
  const userCopy = { ...user };
  userCopy.password = "123";
  return userCopy;
};
console.log(addPassword(user)); // { name: "John" , password: "123" }
console.log(user); // { name: "John" }
```

Les détails du spread seront traités lors d'un futur cours. Mais gardez en tête que cette fonction permet dans la plupart des cas de cloner fiablement un objet simple.

II/ Constructeur d'objets et classes

Dans la plupart des cas, une déclaration « simple » d'objets comme décrit ci-dessus est suffisant pour gérer nos cas d'usages. Cependant, le JavaScript est en mesure de faire usage de constructeurs d'objets pour gérer un template d'objet, et des méthodes contrôlées pour interagir avec.

Par convention, une classe se déclare en PascalCase (par exemple : User, AppUser, AdminUser…)

Il existe deux façons de faire en JavaScript ; une méthode « ancienne » à base de fonction qui peut être invoqué avec le mot clé « new », et une plus moderne avec le mot clé « class ».

A) Les fonctions constructeurs

1) Déclarer une fonction constructeur simple

Dans son état le plus simple, un objet peut être construit à l'aide d'une simple déclaration de fonction qui set des paramètres dans « this » qui est ensuite implicitement retourné à la construction de l'objet. Par exemple :

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

console.log(user.name); // Jack
console.log(user.isAdmin); // false
```

Voici une syntaxe très simple pour une création d'objet. Implicitement, voici ce que ça fait lorsqu'on l'invoque avec un « new »

```
function User(name) {
  // this = {}; (implicitly)
  // add properties to this
  this.name = name;
  this.isAdmin = false;
  // return this; (implicitly)
}
```

En pratique, cette construction est équivalente à faire :

```
let user = {
  name: "Jack",
  isAdmin: false,
};
```

Mais ceci peut être lourd lorsqu'on doit créer plein de fois le même objet. Et encore plus lorsqu'on veut rajouter des méthodes communes à chacun de ces objets.

2) Rajouter des méthodes dans le constructeur

Un des gros intérêts d'un objet c'est de pouvoir créer des méthodes pour interagir directement avec un l'objet, ce qui permet de répéter des fonctions pour des objets différents sans avoir à les dupliquer. Par exemple :

```
function User(name) {
  this.name = name;
  this.sayHi = function () {
    return "Hello! My name is " + this.name + " !";
  };
}
let john = new User("John");
console.log(john.sayHi()); // Hello! My name is John !
```

Ici, chaque construction d'un objet « User » aura accès à la méthode user.sayHi(), ce qui permet de centraliser une fonction, tout en restant générique via l'accès à « this » qui référence l'objet en question.

3) Pour aller plus loin…

Tous ces concepts font partie du cœur JavaScript dans sa façon de gérer les objets, où il porte le nom de « prototyping-based Object Oriented Programming language ». Ce prototype contient toutes les propriétés propres à un objet en JS. Ce prototype peut être utilisé dans des contextes plus avancés, tel que de l'héritage. Vous pouvez creuser ce thème par vous-même si vous voulez mais cela ne sera pas nécessaire pour comprendre le reste du cours.

Ces concepts restent relativement avancés et sont tout de même assez rarement utilisés au profit de la notation plus avancée des classe qui sont plus lisibles et avec une écriture plus « traditionnelle » (relative à d'autres langages tel que le Java, C#...).

B) La notation moderne des classes

1) La notation de base

Aujourd'hui, il est possible d'écrire des fonctions constructeurs sous forme de classes afin de rendre la notation plus simple et analogue à d'autres langages de programmation.

Par exemple, la fonction « User » présentée dans la section précédente peut s'écrire ainsi sous forme de classe :

```
class User {
  name;
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    return "Hello ! My name is " + this.name + " !";
  }
}

// Usage:
let user = new User("John");
console.log(user.sayHi()); // Hello ! My name is John !
```

2) L'héritage de classes

L'héritage de classes est une façon d'ajouter de la fonctionnalité à une autre classe. Voici un exemple :

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  move(speed) {
    this.speed = speed;
    alert(this.name + " moves with the speed " + this.speed + ".");
  }
  stop() {
    this.speed = 0;
    alert(this.name + " stands still.");
  }
}

let animal = new Animal("Ecco the Dolphin");
```

Cette fonction "Animal" declare simplement une nom et une vitesse pour un animal. Nous pouvons à partir de cette base, déclarer plusieurs autres animaux qui se construisent autour de cette base ainsi :

```
class Cat extends Animal {
  sleep() {
    this.speed = 0;
    alert(this.name + " stops and sleeps!");
  }
}

let cat = new Rabbit("White Rabbit");

cat.run(4); // White Rabbit runs with speed 4.
cat.sleep(); // White Rabbit stops and sleeps!
```

Ou encore

```
class Rabbit extends Animal {
  earLength;

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  hide() {
    this.speed = 0;
    alert(this.name + " stops and hides!");
  }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit stops and hides!
```

Ces deux nouvelles classes “Rabbit” et “Cat” héritent de la classe “Animal” ; et donc, héritent des propriétés speed et name, ainsi que les méthodes move et stop.

Petite remarque sur ces deux nouvelles classes. La classe « Cat » ne spécifie aucune nouvelle propriété, mais la classe « Rabbit » rajoute la propriété « earLength ». Pour la rajouter au constructeur, il faut impérativement passer par la déclaration d’un nouveau constructeur, et l’invocation du mot clé « super » qui va appeler le constructeur parent avec les paramètres spécifiés. Si on omet ce mot clé, la construction d’objet ne fonctionnera pas.

3) Autres notions propres aux classes…

Il existe d’autres notions propres aux classes dont vous avez certainement entendu parlé auparavant, à savoir les propriétés et fonctions statiques, ou encore les propriétés privées ou protégées, les getters et setter… Ces notions ne seront pas abordées en détail dans ce cours, mais sachez qu’elles existent.

III/ La gestion de formulaires

C'est l'heure de dynamiser nos formulaires ! Dans l'ensemble, un formulaire et ses éléments qui le composent possèdent tout un lot d'évènements intéressants auquel on peut s'accrocher.

Les avantages de dynamiser nos formulaires sont nombreux. Ils permettent de rajouter du dynamisme dans les champs, d'effectuer des contrôles sur la validité des saisies avant même que les données ne touchent le serveur. En plus de ces contrôles, il est possible de gérer de façon facile des champs conditionnels. Par exemple, afficher un champ de texte libre lorsque dans une select box, on sélectionne la valeur « autre », ce qui permet à l'utilisateur de spécifier sa valeur dans un champ libre.

Cette section va aborder les évènements principaux qui sont intéressants à utiliser.

A) Evènements du formulaire entier

En principe, jusqu'à maintenant, dans vos formulaires, vous avez spécifié un attribut « action », et spécifie la page qui sera ouverte lorsque vous soumettez votre formulaire. Un évènement intéressant à gérer ici serait d'empêcher l'envoi du formulaire lorsqu'un des champs du formulaire par exemple n'est pas correct.

L'évènement à utiliser ici est le onSubmit :

```
const form = document.querySelector("form");

const validateForm = () => {
  // Logique pour vérifier le formulaire ici
  return true;
};

form.addEventListener("submit", (event) => {
  if (!validateForm()) {
    event.preventDefault();
  }
});
```

B) Évènements par rapport à des inputs

Vous savez déjà comment gérer des évènements pour la plupart. Voici une liste de ceux qui seront importants pour les formulaires :

- L'évènement « blur » est lancé lorsqu'un élément d'input perd le focus. C'est généralement utile pour savoir quand l'utilisateur a terminé de saisir un texte dans un input texte.
- L'évènement « change » sert à indiquer quand il y a un changement, mais ne fonctionne pas pour tous les types d'inputs. Par exemple, il n'est pas utilisable pour les inputs type text. Cependant, il reste très utile pour les autres formes d'input, tels que les checkbox, les radios, ou les select.

Pour le reste, il n'y a pas de magie, ça reste des évènements comme vous commencez à avoir l'habitude maintenant !

C) Limitations de vos vérifications

Imposer des restrictions à vos formulaires en front-end avec du JavaScript est quelque chose que l'on pourrait qualifier d'important dans notre contexte de développement d'applications d'aujourd'hui. Cependant, il ne faut jamais s'appuyer exclusivement sur le JavaScript en front-end pour vérifier/limiter la saisie utilisateur.

Le code JavaScript reste un code exécuté côté client, de plus en clair, de façon non compilée. Par définition, il est possible de le désactiver ou de l'altérer ce qui aurait comme conséquence de court-circuiter complètement vos vérifications.

Traitez donc vos contrôles de formulaires en JS comme une aide pour l'utilisateur, et jamais comme une couche de sécurité. Si le JavaScript permet d'empêcher un utilisateur lambda de submit des données erronées, cela ne reste qu'une barrière superficielle pour un utilisateur qui s'y connait, ou un utilisateur malicieux. Vous devez donc TOUJOURS complémenter vos vérifications côté serveur qui va récupérer le formulaire envoyé.