Web Developer
Scripts Clients

07 – Objets

Plan du chapitre

7.1 Généralités

- Un **objet-type** : représentation d'un concept, d'une chose, ... (niveau abstraction).
- Un **objet** est l'implémentation d'un objet-type (niveau concret)
- Permettent de stocker et de manipuler des informations inhérentes à ce qui est représenté
- Parfois appelés « tableaux associatifs »

7.1 Généralités

- Constructeur
  - « Fonction » appelée à la création d'un objet
  - Permet d'initialiser des propriétés, d'ouvrir une connexion à une DB, etc.
- Propriétés
  - Données qui décrivent l'objet
  - Le nom d'une personne, sa taille, son adresse, etc.
  - Appel à une propriété d'un objet :
    - `nomObjet.nomPropriété`
    - `nomObjet['nomPropriété']`
  - La clé est toujours une chaîne de caractères
  - La valeur peut être de n'importe quel type

7.1 Généralités

- Méthodes
  - Fonctions de manipulation des propriétés
  - Obtenir les cours suivis par un élève, ajouter les points obtenus dans un cours particulier, etc.
  - Appel à une méthode d'un objet :
    - `nomObjet.nomMéthode()`

7.2 Littéraux

- Création d'un objet vide :

```
const monObjet = {};
```

7.2 Littéraux

- Ajout d'une propriété (Accès aux données via une clé et pas par un index comme avec les tableaux) :

```
monObjet.cle1 = 'test1';
// Ou
monObjet['cle2'] = 'test2';
```

7.2 Littéraux

- Accès en lecture aux propriétés :

```
console.log(monObjet.cle1);
// Ou
console.log(monObjet['cle2']);
// Attention aux simples quotes !
```

7.2 Littéraux

- Création et définition d'un objet :

```
const monObjet = {
    cle1: 'valeur1',
    cle2: 'valeur2'
};
```

7.2 Littéraux – boucle

- Parcourir les clés des propriétés d'un objet :

```
for (const cle in monObjet) {
    console.log(cle);
}
```

7.2 Littéraux – boucle

- Parcourir les valeurs des propriétés d'un objet, essayons :

```
for (const valeur of monObjet) {
    console.log(valeur);
}
```

7.2 Littéraux – boucle

- Attention : le `for…of` ne fonctionne pas avec les objets.
- Il faut utiliser un `for…in` pour parcourir les clés afin d'accéder indirectement aux valeurs des propriétés :

```
for (const cle in monObjet) {
    console.log(monObjet[cle]);
}
```

7.2 Littéraux – boucle

La méthode moderne consiste en l'utilisation de méthodes de l'objet… `Object` pour ensuite les récupérer dans un `for… of…` :

- `keys` : permet de récupérer un tableau avec les clés (les labels)
- `values` : permet de récupérer un tableau avec les valeurs
- `entries` : permet de récupérer un tableau avec les paires clé-valeur

7.2 Littéraux – boucle

Récupérer un tableau avec les clés (les labels) :

```
for (const cle of Object.keys(monObjet)) {
    console.log(cle);
}
```

7.2 Littéraux – boucle

Récupérer un tableau avec les valeurs :

```
for (const valeur of Object.values(monObjet)) {
    console.log(valeur);
}
```

7.2 Littéraux – boucle

Récupérer un tableau avec les paires clé-valeur :

```
for (const [cle, valeur] of Object.entries(monObjet)) {
    console.log(cle, valeur);
}
```

7.2 Littéraux

- Permettre à une fonction de retourner plusieurs valeurs :

```
function testRetourValeurs() {
    const monObjet = {
        cle1: 'valeur1',
        cle2: 'valeur2',
        cle3: 'valeur3'
    };
    return monObjet;
}
```

7.2 Littéraux

- Testons :

```
const monObjetResultat = testRetourValeurs();

console.log(monObjetResultat);
```

7.2 Littéraux

- Pour obtenir un tableau avec toutes les clés d'un objet :

```
const keys = Object.keys(monObjet);
```

7.2 Get/Set

- Il est possible de définir des accesseurs (getter) et des mutateurs (setter) :

```
const monObjet = {
    cle1: 'VIDE',
    get getCle1() {
        return this.cle1;
    },
    set setCle1(val1) {
        this.cle1 = val1;
    }
};
```

7.2 Get/Set

- On peut les utiliser comme des propriétés (pas comme des méthodes :

```
console.log(monObjet.getCle1);

monObjet.setCle1 = 10;
```

7.2 Suppression

- On peut supprimer une propriété :

```
delete monObjet.getCle1;
delete monObjet.cle1;
```

7.2 Copie

- Essayons de copier un objet :

```
const objet1 = {
  nom: 'hamilton',
  prenom: 'lewis'
}
const objet2 = objet1;

objet1.nom = 'vettel';
console.log(objet2);
```

7.2 Copie

- Nous pouvons observer qu'une modification sur l'objet1 impacte l'objet2
- La raison est que nous avons copié la référence de l'objet1 dans le 2 et pas les valeurs
- Pour réaliser cela, la version moderne est d'utiliser le spread operator :

```
const objet1 = {
  nom: 'hamilton',
  prenom: 'lewis'
}
const objet2 = {...objet1};


objet1.nom = 'vettel';
console.log(objet2);
```

7.3 Les objets inclus dans JS

- String
- Date
- Number et Math
- Array
- Image

7.3 String

- Traitement des chaînes de caractères
- Propriété(s) :
  - length qui représente la longueur de la chaîne de caractères
- Méthode(s) :
  - charAt(), charCodeAt() et fromCharCode()
  - indexOf() et lastIndexOf()
  - toLowerCase() et toUpperCase()
  - trim()
  - substring(début, fin) et slice()
  - split()

7.3 String - Exercices

7.3.1 : vous devez :

- extraire les codes ASCII de tous les caractères d'une chaîne de test
- les afficher
- les stocker dans un tableau
- passer le tableau dans une boucle
- pour chacun des codes ASCII, afficher le caractère correspondant

7.3.2 : créez une fonction qui mettra en majuscule la première lettre d'une chaîne de caractères

7.3 Date

- Traitement des dates et du temps
- Référence : timestamp en ms depuis le 01/01/1970 (voir UNIX)
- Constructeur(s) :

```
const date1 = new Date(); // « maintenant »

// un timestamp, par exemple, hier
const date2 = new Date(date1 - 24 * 60 * 60 * 1000);


// Attention, inversion mois/jour : new Date("mois/jour/année")
const date3 = new Date("01/13/2014");


// Attention, le mois commence à 0 !!!
const date4 = new Date(2014, 0, 1);
```

7.3 Date

- Méthode(s) :
  - set/getFullYear()
  - set/getMonth()
  - set/getDate()
  - set/getHours()
  - etc.

7.3 Date – Exercice

7.3.3 : Affichez le nombre de jour(s) séparant aujourd'hui d'une date encodée par l'utilisateur

7.3 Number

- Représentation des nombres
- Propriété(s) :
  - `MAX_VALUE` : plus grande valeur d'un nombre en JS
  - `MIN_VALUE` : plus petite valeur d'un nombre en JS (valeur la plus proche de zéro)

Rem : la plus grande valeur négative est `-MAX_VALUE`

7.3 Math

- Bibliothèques de fonctions mathématiques
- Propriété(s) :
  - Diverses constantes mathématiques (dont PI)
- Méthode(s) :
  - abs()
  - max() et min()
  - round(), floor() et ceil(). Différences ?
  - pow() et sqrt()
  - random()
  - Fonctions trigonométriques

7.3 Math

- random

```
// Par exemple, valeur "aléatoire" comprise entre 1 et 10
const MIN = 1;
const MAX = 10;

const aleatoire = MIN + Math.random() * (MAX - MIN);

console.log(aleatoire);
```

7.3 Array

- Gestion des tableaux
- Propriété(s) :
  - length : taille du tableau
- Constructeur(s) :

```
const array1 = new Array();
// ou avec initialisation
const array1 = new Array(1,2,3);

const array2 = [ ];
// ou avec initialisation
const array2 = [1,2,3];
```

7.3 Array

- Méthode(s) :
  - concat()
  - indexOf() et lastIndexOf()
  - sort() et reverse()
  - pop() et push() : ajout/suppression en fin de tableau
  - unshift() et shift() : ajout/suppression en début de tableau

Rem : Attention aux performances de (un)shift car décalage complet nécessaire (ok si length<10000)

7.3 Image

- Objet natif de gestion des images
- Propriété(s) :
  - width et height
  - src et alt
  - Etc.
- Événement(s)
  - onload, onmouseover, etc.

7.3 Globales

- Fonctions globales (cas particuliers)
  - parseInt() et parseFloat()
  - isNaN() et isFinite()

7.4 Les objets créés par le développeur

- Problème : créer un objet pour gérer des élèves
- Propriétés :
  - Nom
  - Prénom
  - Date de naissance
  - Cours suivis
  - Notes obtenues

7.4 Les objets créés par le développeur

- 3 types d'implémentation :
  1) Via littéral à la déclaration
  2) Via littéral après la déclaration
  3) Via constructeur

7.4 Les objets créés par le développeur

1) Via littéral à la déclaration :

```
const eleve1 = {
    nom: 'Dubois 1',
    prenom: 'Yvan 1',
    dateNaissance: '17/10/1961',
};
```

7.4 Les objets créés par le développeur

2) Via littéral après déclaration :

```
const eleve2 = {};
eleve2.nom = 'Dubois 2';
eleve2.prenom = 'Yvan 2';
eleve2.dateNaissance = '17/10/1962';
```

7.4 Les objets créés par le développeur

3) Via Constructeur :

```
function Eleve(nom, prenom, dateNaissance){
    this.nom = nom.toLowerCase();
    this.prenom = prenom.toLowerCase();
    this.dateNaissance = new Date(dateNaissance);
    this.cours = [];
    this.notes = {};
}
```

7.4 Les objets créés par le développeur

- Instanciation :

```
const eleve1 = new Eleve('Lewis','Hamilton','07/01/1985');
```

7.4 Les objets créés par le développeur

- Ajoutons à notre objet une méthode de présentation de l'élève :

```
function Eleve(nom, prenom, dateNaissance){
   …
     this.presenterEleve = function () {
         return `Elève ${this.nom} ${this.prenom}
            né(e) le
            ${this.dateNaissance.toDateString()}`;
     };
}
```

7.5 Modification d'un objet existant

- Il est possible de modifier un objet existant, pour lui ajouter une méthode par exemple
- C'est possible pour les objets que nous créons, mais également pour les objets « JavaScript »
- Pour cela, nous devons utiliser l'objet prototype
- C’est l’approche historique : remplacée par les classes ES6 dans le code moderne en VanillaJS, mais utile pour comprendre le moteur JS
- Dans les frameworks modernes (ex : React), on privilégie la composition de fonctions plutôt que l’héritage de classes
- Les deux approches sont donc utiles et complémentaires

7.5 Modification d'un objet existant

- Ajoutons une méthode `afficherNomPrenom` à notre objet `Eleve` :

```
Eleve.prototype.afficherNomPrenom = function () {
    console.log(this.nom + ' ' + this.prenom);
};
```

7.5 Modification d'un objet existant

- Ajoutons une méthode `rightTrim` à l'objet `String` :

```
String.prototype.rightTrim = function () {
    let idx_max = this.length - 1;
    let new_string = '';
    while (this[idx_max] === ' ') {
        idx_max--;
    }
    for (let i = 0, max = idx_max + 1; i < max; i++) {
        new_string += this[i];
    }
    return new_string;
};
```

7.5 Modification d'un objet existant

- Idéalement, il faut tester que la méthode que nous voulons ajouter n'existe pas :

```
if(!String.prototype.rightTrim){
    String.prototype.rightTrim = function () {
        ...
    };
}
```

7.5 Modification d'un objet existant

- Revenons à notre objet afin de lui ajouter une méthode de présentation de l'élève via prototype :

```
Eleve.prototype.presenterEleve = function () {
  return 'Elève ' + this.nom
+ ' ' + this.prenom
+ ' né(e) le ' + this.dateNaissance.toDateString();
};
```

- L'avantage de cette méthode par rapport à l'ajout par le constructeur est que toutes les fonctions ajoutées seront **disponibles** pour tous les objets **déjà instanciés**.

7.5 Modification d'un objet existant

- Il est également possible de réécrire (surcharger) les méthodes d'un objet via prototype. Exemple pour la méthode `toString` :

```
Eleve.prototype.toString = function () {
  return 'Elève ' + this.nom
+ ' ' + this.prenom
+ ' né(e) le ' + this.dateNaissance.toDateString();
};
```

7.5 Modification d'un objet existant

- Pour ajouter plusieurs méthodes via prototype, on peut le faire en passant par une notation « objet » :

```
Eleve.prototype = { // écrase tout le prototype existant
  toString: function(){
    return 'Elève ' + this.nom
+ ' ' + this.prenom
+ ' né(e) le ' + this.dateNaissance.toDateString();
  },
  getNom: function(){
    return this.nom;
  }
};
```

7.5 Exercice

- Ajoutez une méthode `arrondi`, via prototype, à l'objet `Math`
- Celle-ci prendra en paramètre le nombre à arrondir (n) et le nombre de décimales (d)

7.5 Solution

```
Math.prototype.arrondi = function (n, d) {
    // code qui calcule l'arrondi
};
```

7.5 Solution

- Lorsqu'on exécute cette méthode, on obtient l'erreur suivante :

```
TypeError: Cannot set property 'arrondi' of undefined
```

- Pourquoi ?

7.5 Solution

Il faut distinguer deux cas de figure dans la création d'objet en JS :

1) Les objets statiques
2) Les constructeurs d'objets (auxquels on peut ajouter des méthodes pour les utiliser comme des objets statiques)

Afin d'illustrer ces différences, nous allons utiliser les objets JS `String` et `Math`.

7.5 Solution

1) Les objets statiques (comme `Math`)

- Ne permettent pas l'instanciation d'objets via `new`
- On les crée et on leur ajoute des méthodes comme ceci :

```
// Objet statique
const TestStatique = {};
TestStatique.triple = function (a) {
    return 3 * a;
}
// Test de la méthode
const tri = TestStatique.triple(11);
console.log(tri);
```

7.5 Solution

2) Les constructeurs d'objets (comme `String`)

- Permettent l'instanciation d'objets via `new`
- On les crée et on leur ajoute des méthodes comme ceci :

```
// Constructeur d'objet
function TestConstructeur() {
    // Code du constructeur
}
TestConstructeur.prototype.double = function (a) {
    return 2 * a;
};
const t = new TestConstructeur();
const dou = t.double(10);
console.log(dou);
```

7.5 Solution

2) Les constructeurs d'objets (comme `String`)

- Il est également possible de leur ajouter des méthodes statiques (attention, sans le prototype !) :

```
// Constructeur d'objet
function TestConstructeur() {
    // Code du constructeur
}
TestConstructeur.triple = function (a) {
    return 3 * a;
};
const tri = TestConstructeur.triple(11);
console.log(tri);
```

7.5 Solution

- Comment savoir quelle méthode utiliser selon l'objet ?

```
console.log(typeof Math);
// --> object
// objet statique

console.log(typeof String);
// --> function
// signifie constructeur dans ce cas-ci
```

7.5 Solution

- Pour ajouter la méthode arrondi à `Math`, nous devons donc procéder sans le prototype :

```
Math.arrondi = function (n, d) {
    let result = Math.trunc(n * Math.pow(10, d) + 0.5);
    result /= Math.pow(10, d);

    return result;
};
```

7.6 ES6 – Objet

- Avec ES6, lors de la déclaration des objets, il est possible d'utiliser une forme raccourcie dans l'écriture des paires clé/valeur :

```
const nom = 'Hamilton';
const prenom = 'Lewis';
const age = 26;

const personne = {nom, prenom, age};
// Équivalent à {nom: nom, prenom: prenom…
console.log(personne.nom);
```

7.6 ES6 – Objet

- À l'inverse, il est possible de déstructurer un objet en plusieurs variables :

```
const personne = {
    nom: 'Hamilton',
    prenom: 'Lewis',
    age: 33,
    email: 'lewis.hamilton@mercedes.com'
};
const {age, nom, prenom} = personne;
console.log(prenom);
```

7.6 ES6 – Objet

- Les méthodes peuvent également être plus concises
- Avec les template strings (délimitées par des backquotes – accents graves), vous pouvez utiliser des variables directement dans la chaîne et également la répartir sur plusieurs lignes

```
const personne = {
    nom: 'Hamilton',
    prenom: 'Lewis',
    toString(){
        return `Je suis ${this.nom} ${this.prenom}`;
    }
};
console.log("" + personne);
```

7.6 ES6 – Objet

- Pour passer des options à vos fonctions et utiliser des valeurs par défaut pour enfin configurer vos fonctions, nous allons utiliser trois objets distincts. Ces trois objets seront « mixés » grâce à la méthode `Object.assign()`
- Cette méthode prend autant de paramètres que nous voulons

7.6 ES6 – Objet

- Exemple avec trois paramètres :

```
const options = {
    p1: "OPTION 1.1",
    p2: "OPTION 1.2"
};
const defaults = {
    p2: "DEFAULT 2.2",
    p3: "DEFAULT 2.3"
};
const settings = {};
Object.assign(settings, defaults, options);
console.log(settings);
```

7.6 ES6 – Objet

- Ou :

```
const options = {
    p1: "OPTION 1.1",
    p2: "OPTION 1.2"
};
const defaults = {
    p2: "DEFAULT 2.2",
    p3: "DEFAULT 2.3"
};
const settings = Object.assign({}, defaults, options);
console.log(settings);
```

7.6 ES6 – Objet

- Résultat :

```
{
  p1: "OPTION 1.1",
  p2: "OPTION 1.2",
  p3: "DEFAULT 2.3"
}
```

7.6 ES6 – Objet

- Exemple avec une fonction :

```
function testSettings(options = {}){
    const defaults = {
      p1: 'DEF1',
      p2: 'DEF2'
    };
    const settings = Object.assign({}, defaults, options);
    return settings;
}
const final_settings = testSettings({p2: 'OPT2', p3: 'OPT3'});
console.log(final_settings);
```

7.6 ES6 – Classe

- Depuis ES6, la notion de classe a été introduite en JS
- On peut dès lors définir une nouvelle classe comme ceci :

```
class Personne{
// Méthode appelée implicitement lors de l'instanciation
// Sert principalement aux initialisations
    constructor(nom, prenom){
        this.nom = nom;
        this.prenom = prenom;
    }
}
```

7.6 ES6 – Instanciation

- Pour déclarer un objet, on instancie la classe (les accès aux méthodes et propriétés sont similaires aux objets vus précédemment) :

```
const personne1 = new Personne('Hamilton', 'Lewis');
```

7.6 ES6 – Ajout de méthodes

- Pour ajouter une méthode, on utilise la version courte des objets :

```
class Personne{
    constructor(nom, prenom){
        this.nom = nom;
        this.prenom = prenom;
    }
    bonjour(){
        return `Bonjour, je m'appelle ${this.prenom}`;
    }
}
const p1 = new Personne('Hamilton', 'Lewis');
console.log(p1.bonjour());
```

7.6 ES6 – Ajout de propriétés

- Les propriétés sont des méthodes qui sont appelées comme des attributs :

```
class Personne{
    constructor(nom, prenom){
        this.nom = nom;
        this.prenom = prenom;
    }
    get hello(){
        return `Hello, je m'appelle ${this.prenom}`;
    }
}
const p1 = new Personne('Hamilton', 'Lewis');
console.log(p1.hello);
```

7.6 ES6 – Ajout de propriétés

- Les propriétés sont principalement utilisées comme accesseurs ou mutateurs des attributs
- Elles permettent de procéder à des tests avant modification des valeurs ou à des mises en forme
- Par convention, elles débutent par une majuscule
- Afin de les distinguer aisément des attributs, on préfixe les noms de ceux-ci avec un caractère souligné _

7.6 ES6 – Ajout de propriétés

```
class Personne{
    constructor(nom){
        this._nom = nom;
    }
    get Nom(){
        return this._nom;
    }
    set Nom(nom){
        this._nom = nom;
    }
}
```

7.6 ES6 – Ajout de propriétés

```
const p1 = new Personne('Hamilton');

p1.Nom = 'Vettel';

console.log(p1.Nom);
```

7.6 ES6 – Attributs privés

- Il est possible de déclarer les attributs en mode privé afin de contraindre l'utilisateur de passer par les propriétés pour les modifier, ce qui est plus sûr
- Pour cela, on les liste simplement avant le constructeur et on les préfixe d'un #
- Les attributs privés sont apparus avec ES2022 et ne sont pas accessibles aux sous-classes

7.6 ES6 – Attributs privés

```
class Personne {
  #nom;

  constructor(nom) {
    this.#nom = nom;
  }

  get Nom() {
    return this.#nom;
  }

  set Nom(nom) {
    this.#nom = nom;
  }
}
```

7.6 ES6 – Attributs privés

```
const personne1 = new Personne('hamilton');

// L'instruction suivante déclenche une erreur :
// SyntaxError: reference to undeclared private field or method #nom
personne1.#nom = 'vettel';

// Seule façon de modifier l'attribut nom
personne1.Nom = 'vettel';
```

7.6 ES6 – Attributs privés

```
class Personne {
  #nom;

  constructor(nom, prenom) {
    this.#nom = nom;
    this.prenom = prenom;
  }
// …
}

const personne1 = new Personne('hamilton');
console.log(Object.keys(personne1)); // #nom n'apparaît pas
```

7.6 ES6 – Attributs privés

- Un attribut privé est néanmoins accessibles aux méthodes de la classe :

```
class Personne {
  #nom;

  constructor(nom) {
    this.#nom = nom;
  }

  nomMajuscules() {
    this.#nom = this.#nom.toUpperCase();
  }
//…
}

const personne1 = new Personne('hamilton');
personne1.nomMajuscules();
console.log(personne1.Nom);
```

7.6 ES6 – Méthodes statiques

- Pour ajouter une méthode statique, on utilise le mot-clé `static`. Cette méthode peut être appelée via la classe et pas via un objet instancié :

```
class Personne{
    …
    static nom_classe(){
        return `Je suis la classe Personne`;
    }
}
console.log(Personne.nom_classe());
```

7.6 ES6 – Héritage

- Il est possible de déclarer une nouvelle classe qui sera basée sur une autre classe (héritage).
- `extends` permet de préciser la classe qu'on va dériver
- `super` permet d'appeler le constructeur de la classe « parent »

```
class Eleve extends Personne{
    constructor(nom, prenom, email){
        super(nom, prenom);
        this.email = email;
    }
}
```

7.6 ES6 – Héritage

```
const eleve1 = new Eleve('hamilton', 'lewis', 'l.h@ferrari.it');


console.log(eleve1);
```

7.7 Bonnes pratiques

- Nommage et structure
  - Utiliser des noms explicites pour les propriétés et méthodes (dateNaissance, calculerPrixTVAC())
  - Préfixer les attributs internes par _ si non destinés à être manipulés directement
  - Grouper les fonctions liées dans un même objet (ou classe) plutôt que d'utiliser des variables globales
- Modernité du code
  - Toujours utiliser const ou let (jamais var)
  - Privilégier les classes ES6 et la composition fonctionnelle (React, Node.js) plutôt que les prototypes manuels
  - Copier un objet avec le spread operator {...obj} pour éviter les effets de bord

7.7 Bonnes pratiques

- Encapsulation et sécurité
  - Utiliser les getters / setters pour contrôler les modifications
  - Employer les attributs privés # (ES2022+) si le contexte le permet
  - Ne jamais exposer de données sensibles via Object.keys() (ou JSON.stringify())
- Lisibilité et maintenance
  - Une classe = un concept clair
  - Des méthodes courtes et cohérentes (une seule responsabilité)
  - Éviter les chaînes de dépendances trop longues (obj.prop1.prop2.prop3)
  - Documenter brièvement les propriétés et méthodes (commentaire au-dessus)
- Bon réflexe
  - Avant de créer un nouvel objet, vérifier s'il n'existe pas déjà un objet natif (Date, Map, Set, etc.) adapté au besoin

7.8 Exercice

- Créez un objet selon les deux méthodes avec, au minimum, les attributs et méthodes suivantes :
  - Prix HTVA
  - Taux de TVA
  - Calcul du prix TVAC