

JavaScript Advanced

Objectifs et prérequis

- **Objectifs**
 - Consolider ses bases en JavaScript
 - S'initier à TypeScript
 - Adopter les bonnes pratiques, s'exercer
- **Prérequis**
 - JavaScript côté navigateur (syntaxe, DOM, évènements, etc.)
 - HTML / CSS

Plan de cours

Partie 1 : JS avancé

1) Mise en place (VSCode, NodeJS)

- 5) Paramètres par défaut
- 6) Template strings
- 7) Destructuring d'objets et d'arrays
- 8) Type Symbol

2) Retour sur certaines notions JS

- 1) Contextes d'exécution des fonctions
- 2) Closures
- 3) IIFE
- 4) Revealing Module Pattern
- 5) Intro à la programmation fonctionnelle

- 9) Iterateurs & Itérables
- 10) Générateurs
- 11) Promises
- 12) Spread operator

3) JavaScript ES6 (ES2015)

- 1) Let, const
- 2) Modules
- 3) Classes
- 4) Arrow functions

4) EcmaScript : versions suivantes

- 1) EcmaScript 7 (ES2016)
- 2) EcmaScript 8 (ES2017)
- 3) EcmaScript 9 (ES2018)

5) Fetch API

Plan de cours

Partie 2 : TypeScript

1) Introduction à TypeScript

- 1) Introduction
- 2) Mise en place, commande tsc
- 3) watch mode

- 4) Any
- 5) Union
- 6) Type littéral
- 7) Alias de type
- 8) Retours de fonctions
- 9) Function
- 10) Unknown
- 11) Never

2) Introduction à l'outil NPM

- 1) Introduction
- 2) Utilisation

4) Intro à la configuration du compilateur

- 1) Fichier de configuration
- 2) Répertoire racine et de destination
- 3) Inclusions / exclusions
- 4) Version de JS

3) Bases de TypeScript

- 1) Manipulation des types primitifs
- 2) Tuples
- 3) Énumérés

Plan de cours

- | | |
|---------------------------------------|---|
| 5) Programmation orientée objet en TS | 6) TS avancé |
| 1) Visibilité d'attributs | 1) Intersection de types |
| 2) Classes et constructeurs rapides | 2) Type Guard |
| 3) Lecture seule | 3) Types complexes |
| 4) Héritage | 4) L'union discriminante |
| 5) Getters & Setters | 5) Assertion de type |
| 6) Attributs et méthodes statiques | 6) Surcharge |
| 7) Classes abstraites | 7) Optional chaining |
| 8) Pattern singleton | 8) Null coalescing |
| 9) Interfaces | 9) Généricité |
| 10) Décorateurs | 7) Bonus : appels API avec Fetch avec notre propre mini webservice |

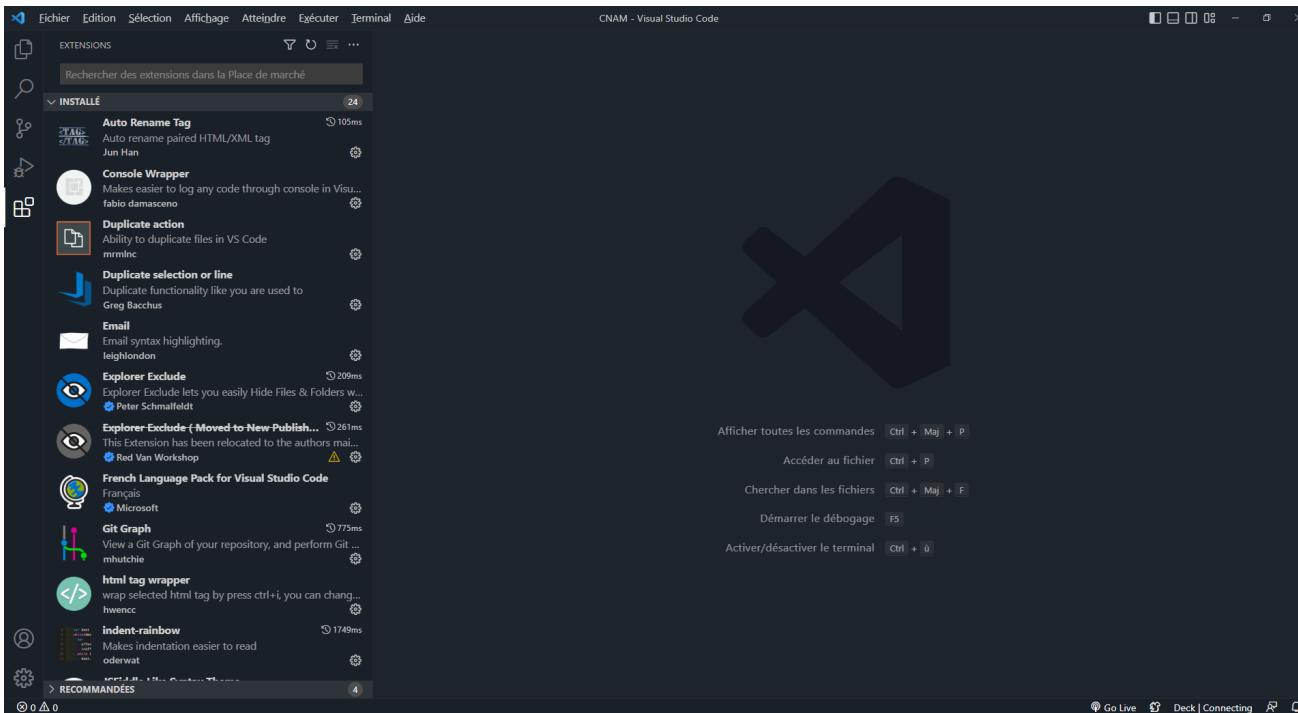
Partie 1 : JS avancé

Partie 1 - Chapitre 1

Mise en place

Mise en place de VSCode

- Nous aurons besoin d'un bon éditeur. Ex : <https://code.visualstudio.com>



- Extensions intéressantes :
 - HTML : auto rename tag
 - JS : console wrapper / code runner
 - Divers : duplicate action / duplicate selection or line / explorer exclude / jsfiddle-dark-theme / live server / prettier – Code formatter / french lang. pack

Mise en place de NodeJS

- Nous utiliserons Javascript en dehors du navigateur.
- C'est pourquoi nous aurons besoin de NodeJS : <https://nodejs.org/fr>



Partie 1 - Chapitre 2

Retour sur certaines notions JS

Contextes d'exécution des fonctions

- Il existe plusieurs façons d'invoquer une fonction :
 - comme fonction : `nomDeFonction();`
 - comme méthode : `objet.nomDeFonction();`
 - comme constructeur : `new nomDeFonction();`
 - Avec `apply()` ou `call()`
- Aussi, lorsque l'on appelle une fonction (quelque soit la méthode), le moteur JS ajoute automatiquement 2 paramètres : `this` (le contexte) et `arguments`, qui correspondent aux paramètres passés à la fonction

Invocation comme fonction

```
// invocation comme fonction
function saluer(){
    console.log(this);
    console.log(arguments);
    console.log("Bonjour");
}
saluer();
```

P1-C2/fonctions.js

- Dans ce cas :
 - `this` → objet window (contexte web) / objet global (contexte node)

Pratique

N'hésitez pas à lancer [P1-C2/fonctions.js](#) dans un navigateur en le chargeant avec une balise `<script>` depuis un fichier [index.html](#)

Testez ensuite une exécution dans [VisualStudioCode](#) avec l'extension [Code Runner](#) (icône play)

Invocation comme méthode

```
// objet JSON
let robot = {
    prenom:"C3PO",
    saluer:function(){
        console.log(this);
        console.log("Bonjour");
    }
}

// invocation comme méthode
robot.saluer();
```

P1-C2/fonctions.js

- Dans ce cas :
 - `this` → objet robot

Invocation comme constructeur

```
// invocation comme constructeur
function Robot(prenom_){
    this.prenom=prenom_,
    console.log(this);
    console.log("Bonjour");
}
let r1 = new Robot("C3PO");
```

P1-C2/fonctions.js

- Dans ce cas :
 - `this` → instance créé

L'invocation par constructeur est pratique et économique en lignes de code lorsque vous souhaitez créer de multiples objets très similaires. Cela évite de créer plusieurs objets similaires en JSON.

Invocation avec apply() ou call()

- Toute fonction en JS est un objet `Function` disposant de ses propres méthodes, comme `apply()` et `call()`
- Ces méthodes permettent d'appeler une fonction en surchargeant son contexte, c'est-à-dire la valeur de `this`. Elles diffèrent par leurs arguments.

```
function saluer(prenom){  
    console.log(this); // correspond à {}  
    console.log("Bonjour "+prenom)  
}  
// invocation avec apply - 2e param = tableau  
saluer.apply({},["Toto"]);  
  
// invocation avec call - sans tableau  
saluer.call({},"Toto");
```

P1-C2/fonctions.js

- Dans ce cas :
 - `this` → 1^{er} param (ici {})

Closures

- Dans le script ci-dessous, l'accès aux variables de la fonction externe est possible par la fonction interne, et ce même si cette fonction externe a terminé son exécution :

```
function fonctionExterne() {  
    var varFonctionExterne = "Demo";  
    function fonctionInterne() {  
        console.log(varFonctionExterne + " des closures");  
    }  
    setTimeout(fonctionInterne, 3000);  
}  
fonctionExterne(); // affiche : Demo des closures
```

P1-C2/closures.js

- `fonctionInterne()` sera ici exécutée dans 3 secondes, alors que l'exécution de `fonctionExterne()` sera terminée depuis plusieurs secondes
- Pour autant, le scope de variable est conservé et la fonction interne peut toujours accéder à la variable de la fonction externe
- C'est le principe des **closures** (fermetures).

Closures

- On peut par exemple exploiter ce mécanisme pour réaliser des compteurs indépendants via une unique fonction. Ici, la variable `count` est protégée de l'extérieur et modifiable seulement par la fonction anonyme interne

```
function compteur() {  
    var count = 0;      // indépendante et locale  
    return function() {  
        return count++; // en mesure de modifier count  
    };  
}  
  
var plusUn = compteur();  
console.log(plusUn()); // 0  
console.log(plusUn()); // 1  
console.log(plusUn()); // 2  
  
var plusUnBis = compteur();  
console.log(plusUnBis()); // 0  
console.log(plusUnBis()); // 1
```

P1-C2/closures.js

IIFE

- IIFE - Immediately Invoked Function Expression, est une fonction exécutée dès qu'elle est définie. Il s'agit d'une fonction anonyme définie dans des parenthèses (opérateur de groupement), suivie de 2 autres parenthèses permettant son appel immédiat

```
(function(prenom){  
    var salutation = "Bonjour "+prenom; // invisible depuis le scope global  
    console.log(salutation);  
})("Paul");
```

P1-C2/IIFE.js

- Dans certains cas, par exemple lorsque la fonction affecte une variable, on peut omettre l'opérateur de regroupement. Les variables au sein de la fonction seront invisible de l'espace global, mais il reste possible de récupérer le retour d'une IIFE en l'affectant à une variable (ici, laSalutation)

```
let laSalutation = function(prenom){  
    var salutation = "Bonjour "+prenom;  
    return salutation;  
}("Paul");  
console.log(laSalutation);
```

P1-C2/IIFE.js

Revealing module pattern

- On peut utiliser les IIFE pour organiser son code en **modules**, et ainsi définir des variables et méthodes publiques ou privées sans polluer l'espace global

```
var module = (function() {  
    var variablePrivee = "VARIABLE PRIVEE";  
    var variablePublique = "VARIABLE PUBLIQUE";  
    function _fonctionPrivee() { // il est pratique de préfixer son nom avec _  
        console.log("FONCTION PRIVEE");  
    }  
    function fonctionPublique(object) {  
        console.log("FONCTION PUBLIQUE : "+variablePrivee);  
    }  
    // on retourne les variables et fonctions que l'on souhaite rendre publiques  
    return {  
        laVariablePublique:variablePublique,  
        laFonctionPublique: fonctionPublique  
    };  
})();  
// UTILISATION  
console.log(module.laVariablePublique); // VARIABLE PUBLIQUE  
module.laFonctionPublique(); // FONCTION PUBLIQUE : VARIABLE PRIVEE  
console.log(module.variablePrivee); // undefined  
module.fonctionPrivee(); // erreur
```

P1-C2/IIFE.js

Intro à la prog. fonctionnelle

- Il s'agit d'un paradigme de programmation, une nouvelle évolution de la manière de coder, après la POO. Elle est centrée sur l'utilisation de fonctions.
Avantages : moins d'effets de bords, tests plus simples. *Inconvénients* : risque de code plus lourd, ou moins optimisé. Elle repose sur différents principes :
- Immutabilité** : il n'est pas accepté de muter les variables. `Object.freeze()`, `Object.assign()` ou des librairies telles qu'`ImmutableJS` peuvent se révéler utiles

```
var couleurs = ["bleu", "blanc"];
// Object.freeze(couleurs);           // interdit toute mutation
couleurs.push("rouge");             // KO - mutation interdite !
var couleurs2 = [...couleurs, "rouge"]; // OK - aucune mutation
```

P1-C2/PF.js

- Fonctions pures** : leur résultat ne dépend que de leurs arguments et non du contexte extérieur, sans effets de bords

```
var age = 20;
function ajouter1() { age += 1; return age; }    // fonction impure
function add1(n) { return ++n; }                  // fonction pure
console.log(ajouter1());   // 21
console.log(add1(20));    // 21
```

P1-C2/PF.js

Intro à la prog. fonctionnelle

- **Fonctions d'ordre supérieur** : désigne les fonctions qui prennent une ou plusieurs fonctions en paramètres, et/ou retournent une fonction.
`Array.forEach()`, `Array.filter()`, `Array.map()` en sont des exemples

```
var nombres = [1,2,3,4,5,6];
var nombresPairs = [];

/*
// nombres pairs, version impérative
for (var i=0; i < nombres.length; i++) {
    if (nombres[i] % 2 == 0) { nombresPairs.push(nombres[i]); }
}
*/

// nombres pairs, version prog fonctionnelle
var estPair = function(v) { return v % 2 == 0 };
nombresPairs = nombres.filter(estPair); // fonction d'ordre supérieur

console.log(nombresPairs); // [2,4,6]
```

Intro à la prog. fonctionnelle

```
function multiplicateur(coef) {  
    return function(nb) {  
        return nb*coef;  
    }  
}  
var multiplierPar10 = multiplicateur(10)  
console.log(multiplierPar10(5)); // 50
```

P1-C2/PF.js

- Ces concepts sont les plus connus, mais la programmation fonctionnelle invite à en utiliser d'autres (composition de fonctions, récursivité, etc.)
- Les règles de la programmation fonctionnelle sont souvent rappelées / mises en avant par les frameworks JS frontEnd (Angular / React avec Redux, VueJS, etc.)

Présentation du projet fil rouge

Magnifilms
films magnifiques

Vote minimum : 0 50 100

Vote : 80%

A big setting in Green Hill, Sonic is eager to prove he has what it takes to be a true hero. His test comes when Dr. Robotnik returns, this time with a new partner, Knuckles, in search for an emerald that has the power to control reality. Sonic and his friends must find the emerald before it falls into the wrong hands.

retour

Copyright Magnifilms

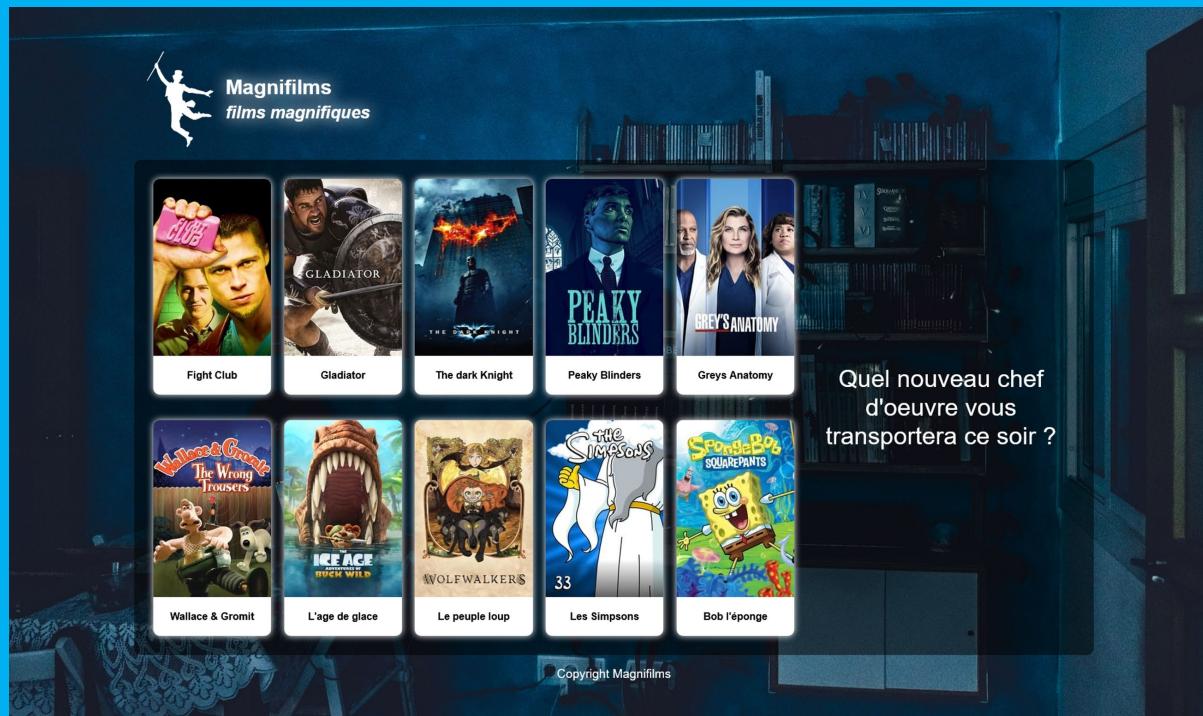
Quel nouveau Chef d'Oeuvre vous transportera ce soir ?

Copyright Magnifilms

| Movie | Score |
|-------------------------------------|-------|
| Spider-Man: No Way Home | 80 |
| Turning Red | 80 |
| Moonfall | 60 |
| Sonic the Hedgehog 2 | 80 |
| The Adam Project | 70 |
| Black Crab | 60 |
| Blacklight | 60 |
| Encanto | 80 |
| The Batman | 80 |
| Morbius | 60 |
| Pil's Adventures | 70 |
| The Grandmother | 60 |
| No Exit | 70 |
| Red Notice | 70 |
| Gold | 70 |
| Death on the Nile | 70 |
| Kimi | 60 |
| The Ice Age Adventures of Buck Wild | 70 |
| Restless | 60 |
| Hotel Transylvania: Transformania | 70 |

Fil rouge - étape 1

- Mise en place de la 1ère page (index.html) en HTML / CSS / JS :
 - mise en place d'un dossier de fil rouge
 - récupération des éléments : <https://tech-me-up.net/formation/jsadv/tp.zip>
 - structure : #haut, #gauche, #droite, #bas
 - allure correspondant à l'image
- Génération des vignettes dans #gauche, en JS, avec tableau d'objets JSON et un appel multiple d'une fonction capable de générer une vignette



Partie 1 - Chapitre 3

JavaScript ES6

var et let

- Par défaut, une variable déclarée avec `var` a pour portée :
 - la fonction qui contient la déclaration
 - le contexte global si la variable est définie en dehors de toute fonction
- Avec le mot clef `let` il est possible de définir une variable dont la portée sera limitée à celle du bloc dans lequel elle sera déclarée.

```
if (true) {  
  var x = 1;  
  let y = 2;  
  console.log(x,y); // 1 2  
}  
console.log(x); // 1  
console.log(y); // erreur
```

P1-C3/letConst.js

var et let

- C'est particulièrement important pour les boucles avec traitement évènementiel :

```
// CAS 1
for(var i=1;i<=3;i++) {
    setTimeout(function(){console.log(i);},1000);
} // 4 4 4

// CAS 2
for(let i=1;i<=3;i++){
    setTimeout(function(){console.log(i);},1000);
} // 1 2 3
```

P1-C3/letConst.js

var et let

- Une autre différence clé est qu'il n'est pas possible de déclarer 2 fois une même variable dans le même scope avec let

```
// CAS 1
var nom = "toto";
var nom = "tata";          // pas d'erreur car var

// CAS 2
let nom = "toto";
let nom = "tata";          // erreur car let et même scope

// CAS 3
let nom = "toto";
if(true) {
    let nom = "tata";    // pas d'erreur car scopes différents
}
```

P1-C3/letConst.js

const

- `const` permet de définir des constantes. On obtiendra des erreurs si on tente de changer leur valeur

```
const NOM = "toto";  
NOM = "tata";           // erreur car constante
```

P1-C3/letConst.js

- Par convention, on écrit les noms de constantes en **majuscules**
- Une constante est **mutable** ! C'est la référence d'une constante qui ne peut être changée. De fait on peut définir un tableau comme constante et le modifier sans provoquer d'erreur :

```
const t1 = [1,2,3];  
t1.push(4);          // pas d'erreur  
console.table(t1);
```

P1-C3/letConst.js

- En règle générale aujourd'hui on privilégié l'utilisation de `let` ou `const` par rapport à `var`

Modules

- Lorsque l'on développe, il est intéressant d'isoler du code indépendant et réutilisable. Les modules ES6 permettent de faire cela. Chaque module contient du code privé et du code exposé, utilisable dans d'autres fichiers. C'est un mécanisme d'import / export. Le code d'un module est par défaut en mode strict.
- Si on inclut un script en précisant `type="module"`, on peut bénéficier des mots clé import et export :

```
[...]
<script defer src="main.js" type="module"></script>
[...]
```

P1-C3/modules.html

- Les modules ES6 sont surtout utilisés côté navigateur même si NodeJS les supporte
- Veiller à tester les modules via un navigateur, et une URL liée au réseau

Modules

- V1 : exportations / importations multiples, avec renommage de "TITRE"

P1-C3/
config.js
(exporte)

```
const TITRE = "MON APP";
const COPYRIGHT = "Copyright";
export { TITRE , COPYRIGHT}
```

P1-C3/
main.js
(importe)

```
import { TITRE as LETITRE, COPYRIGHT } from "./config.js";
console.log(LETITRE, COPYRIGHT);
```

- V2 : importations multiples dans un objet "obj"

P1-C3/
config.js
(exporte)

```
const TITRE = "MON APP";
const COPYRIGHT = "Copyright";
export { TITRE , COPYRIGHT}
```

P1-C3/
main.js
(importe)

```
import * as obj from "./config.js";
console.log(obj,obj.TITRE,obj.COPYRIGHT);
```

Modules

- V3 : exportations / importations multiples, via un unique objet transmit

P1-C3/
config.js
(exporte)

```
let obj = { TITRE:"MON APP", COPYRIGHT:"Copyright"};  
export { obj }
```

P1-C3/
main.js
(importe)

```
import { obj } from "./config.js";  
console.log(obj.TITRE, obj.COPYRIGHT);
```

- V4 : importations / exportations via plusieurs mots clés `export`

P1-C3/
fonctions.js
(exporte)

```
export function saluer(){ console.log("Bonjour"); }  
export function auRevoir(){ console.log("Ciao"); }
```

P1-C3/
main.js
(importe)

```
import { saluer, auRevoir } from "./fonctions.js";  
saluer();  
auRevoir();
```

Modules

- V5 : exportation / importation par défaut avec `default`. On nomme dans ce cas l'importation comme on le souhaite. Il ne peut y avoir qu'un export par défaut. Pratique pour importer des fonctions anonymes par exemple

P1-C3/
User.js
(exporte)

```
export default class User{  
    constructor(nom_){  
        this.nom=nom_;  
    }  
}
```

P1-C3/
main.js
(importe)

```
import U from "./User.js"  
let user1 = new U("Alex");  
console.log(user1);
```

- Il est possible d'importer une export par défaut ainsi que des exports spécifiques (ici "Personne") ainsi :

P1-C3/
main.js
(importe)

```
import U, { Personne } from "./User.js"  
[...]
```

Modules CommonJS (NodeJS)

- Dans un contexte NodeJS, on peut utiliser les modules ES6, à condition d'utiliser les extensions **.mjs** pour vos fichiers

P1-C3/
User.mjs
(exporte)

```
export default class User{  
    constructor(nom_){ this.nom=nom_; }  
}
```

P1-C3/
main.mjs
(importe)

```
import U from "./User.mjs"  
let user1 = new U("Alex");  
console.log(user1);
```

- Mais en NodeJS on utilise plutôt la syntaxe **CommonJS** avec les mots clefs **module.exports** et **require** (importe tout dans un objet). Ici, utiliser l'extension **.js**

P1-C3/
User.js
(exporte)

```
module.exports.User = class User{  
    constructor(nom_){ this.nom=nom_; }  
}
```

P1-C3/
main.js
(importe)

```
const MonModule = require("./User");  
let user1 = new MonModule.User("Alex");  
console.log(user1);
```

Fil rouge - étape 2

- Créer un fichier de configuration config.js, contenant les constantes et variables de votre application.
- Créer un fichier de fonctions fonctions.js, et y placer les fonctions de votre application.
- Importer et utiliser ces fichiers depuis main.js

Fonctions comme classes

- En JS (<ES6) on ne dispose pas de classes, mais on peut s'en approcher en utilisant des fonctions

P1-C3/classes.js

```
// classe
function Cercle(r){
    this.rayon=r;                      // propriété
    this.circonference=function(){      // méthode
        return 2*this.rayon*3.14;
    };
}

var petit_cercle=new Cercle(5);
var grand_cercle=new Cercle(10);

console.log(petit_cercle,petit_cercle.circonference());
console.log(grand_cercle,grand_cercle.circonference());
```

Prototypes

- Chaque objet JS dispose d'un prototype qui permet de redéfinir son « type » depuis le programme principal. **On peut alors ajouter propriétés et méthodes sur toutes les instances d'un certain « type » d'objet**

```
function Cercle(r){  
    this.rayon=r;  
    this.circonference=function(){return 2*this.rayon*3.14;}  
}  
var petit_cercle=new Cercle(5);  
var grand_cercle=new Cercle(10);  
Cercle.prototype.aire=function(){return this.rayon*this.rayon*3.14;};  
console.log(petit_cercle,petit_cercle.circonference(),petit_cercle.aire());  
console.log(grand_cercle,grand_cercle.circonference(),grand_cercle.aire());
```

P1-C3/classes.js

- On peut s'en servir par ex. pour améliorer les types d'objets JS par défaut :

```
String.prototype.inverser = function(){  
    return this.split("").reverse().join("");  
}  
var nom = "toto";  
console.log(nom.inverser());
```

P1-C3/classes.js

class

- Il est possible de faire de la POO, et même de bénéficier d'un système d'héritage, en utilisant des fonctions et des prototypes (on parle de POO prototypale). Mais cette approche n'est pas très courante/intuitive et souvent peu appréciée des développeurs
- ES6 apporte un « syntaxic sugar » en proposant d'utiliser `class` plutôt que `function` lorsqu'il s'agit de créer un type d'objet :

```
class Chat {  
    constructor(nom) {  
        this.nom = nom;  
    }  
    decrire() {  
        console.log("Il s'agit d'un chat nommé " + this.nom);  
    }  
}  
let chat1 = new Chat("Félix");  
console.log(chat1.nom); // Félix  
chat1.decrire(); // Il s'agit...
```

extends

- ES6 apporte également le mot clé `extends` pour bénéficier de l'héritage de manière plus traditionnelle :

```
class Animal {  
    constructor(espece_) {  
        this.espece = espece_;  
    }  
}  
class Chat extends Animal {  
    constructor(nom) {  
        super("mammifère");  
        this.nom = nom;  
    }  
    decrire() {  
        console.log("Il s'agit d'un " + this.espece + " nommé " + this.nom);  
    }  
}  
let chat1 = new Chat("Félix");  
console.log(chat1.nom); // Félix  
chat1.decrire(); // Il s'agit...
```

Arrow functions

- On utilise souvent des fonctions anonymes en JS :

```
let getSalutation = function (prenom){  
    return "Bonjour "+prenom;  
};  
let salutation = getSalutation("Toto");  
console.log(salutation); // Bonjour Toto
```

P1-C3/arrow.js

- ES6 apporte une nouvelle syntaxe pour créer des fonctions anonymes plus rapidement. De la forme **(params) => { instructions }** :

```
let getSalutation = (prenom) => { return "Bonjour "+prenom; };
```

- Si la fonction n'a qu'une instruction et réalise un return, on peut omettre les {} et le mot return. Par ailleurs, si la fonction n'a qu'un paramètre, on peut omettre les (). Ici on obtient donc :

```
let getSalutation = prenom => "Bonjour "+prenom;
```

Paramètres par défaut

- De base, si un paramètre n'est pas transmis lors de l'appel d'un fonction, celui-ci vaudra `undefined`. **On peut donc tester les paramètres** dans la fonction avant utilisation :

```
function saluer(nom){  
    if (nom == undefined){var nom = "";}  
    console.log("Bonjour "+nom);  
}  
saluer();      // Bonjour  
saluer("Toto"); // Bonjour Toto
```

P1-C3/params.js

- Depuis ES6 **on peut passer des valeurs par défaut en signature de fonction** :

```
function saluer(nom=""){  
    console.log("Bonjour "+nom);  
}  
saluer();      // Bonjour  
saluer("Toto"); // Bonjour Toto
```

P1-C3/params.js

Template Strings

- En fonction des cas, utiliser l'opérateur + pour concaténer des chaînes peut être fastidieux... Depuis ES6, les template strings nous permettent :
 - d'injecter des variables, ou d'évaluer des expression JS
 - de prendre en compte les sauts de ligne

```
let prenom="toto";
let messages=1;

console.log(`Bonjour ${prenom}
Comment allez-vous ?
Vous avez ${++messages} message(s).`);
```

P1-C3/templatestrings.js

- Exemple d'utilisation de l'opérateur ternaire au sein d'une template string :

```
console.log(`${
  messages == 0 ? "Vous n'avez aucun message" :
  `Vous
avez ${messages} message(s).`}`);
```

P1-C3/templatestrings.js

Destructuring d'objet

- Le destructuring d'objet, de la forme `let {prop1,prop2}=obj` permet de copier les propriétés `prop1` et `prop2` d'un objet `obj` dans de nouvelles variables de même nom. On peut aussi renommer les variables créées et leur attribuer une valeur par défaut si les props ne sont pas présentes sur l'objet

```
let voiture={  
    marque:"Fiat",  
    modele:"500",  
    annee:2010  
};  
  
let {marque,modele}=voiture;          // simple extraction  
console.log(marque+" "+modele);  
  
let {annee:anneeDeSortie}=voiture;    // extraction + renommage  
console.log(anneeDeSortie);  
  
let {couleur="rouge"}=voiture;        // extraction et valeur par défaut  
console.log(couleur);
```

Destructuring d'Array

- Le destructuring d'array, de la forme `let [var1,var2]=tab`, permet de créer 2 variables `var1` et `var2` correspondant au 2 premières valeurs du tableau `tab`

```
let notes=[10,11,12];
let [note1,note2]=notes;
console.log(note1+" "+note2); // 10 11
```

P1-C3/destructurings.js

Fil rouge - étape 3

- Modifier le tableau d'objets JSON en tableau d'objets Film, qui comprend :
 - Propriétés : id / titre / petitelimage / grandelimage
 - Méthodes : getVignette() : retourne une vignette (remplace la fonction précédente)
- Utiliser lorsque c'est possible les template strings pour générer du code HTML

Symboles

- `Symbol` est un nouveau type primitif au même titre que `String`, `Number`, `Boolean`, `null`, ou `undefined`
- Il représente une valeur (grand chiffre) unique et immutable
- Il sert d'identifiant unique pour créer des propriétés d'objets. C'est là son rôle
- On le crée avec `Symbol()` qui permet aussi de lui associer une description

```
// SYMBOLE SANS PRECISION DE CLE
// s1 est une valeur unique. "MONSYMBOLE" est ici juste sa description
const s1 = Symbol("MONSYMBOLE"); // ex : 19856156156121651956
// infos sur ce symbole. Ici, pas de clé.
console.log(typeof(s1),s1,s1.description,Symbol.keyFor(s1));
// s2 est une autre valeur unique, malgré la même description que s1
const s2 = Symbol("MONSYMBOLE"); // ex : 65161519891321561512
// false car il s'agit de 2 symboles différents
console.log(s1 == s2);
```

P1-C3/symbol.js

Symboles globaux

- Il est possible de préciser une clé de symbole avec `Symbol.for()`. Si le symbol lié à cette clé n'est pas trouvé, il est créé. Autrement, il est retourné.
- Les symboles liés à des clés sont dans le registre global des symboles et sont donc des symboles globaux
- `Symbol.keyFor()` permet de retourner la clé d'un symbole passé en param

```
// SYMBOLE AVEC PRECISION DE CLE (dans registre global des symboles)
// ci-dessous, on tente de trouver le symbole de clé "MONSYMBOLE".
// ici, il ne le trouve pas et donc le crée
const symbolGlobal1 = Symbol.for("MONSYMBOLE");

// cette fois, il le trouve
const symbolGlobal2 = Symbol.for("MONSYMBOLE");

// retourne la clé du symbole
console.log(Symbol.keyFor(symbolGlobal1));

// true car il s'agit en fait du même symbole
console.log(symbolGlobal1 == symbolGlobal2);
```

Symboles comme props d'objets

- On peut utiliser les symboles comme **props d'objets**, en utilisant des []:

```
// SYMBOLE EN TANT QUE PROPRIETE D'OBJET
```

P1-C3/symbol.js

```
// nouveau symbole avec description
const IDENTITE = Symbol("Son identité");
```

```
// notation 1 : ajout direct de symbole sur objet avec []
let personne = {prenom:"Bruce",nom:"Wayne",[IDENTITE]:"Bruce Wayne"};
```

```
// OU notation 2 : ajout en 2 temps
```

```
// let personne = {prenom:"Bruce",nom:"Wayne"};
// personne[IDENTITE]="Bruce Wayne";
```

```
// Attention ! Un ajout sans les [] crée une autre propriété !
// personne.NOM_COMPLET="autre nom";
```

```
console.log(personne);           // voyons notre objet
```

```
console.log(personne[IDENTITE]); // voyons la valeur de la prop
```

Symboles comme props d'objets

- En cas de **serialisation** ou de **boucle for in**, les propriétés liées à des Symboles ne seront pas prises en compte. Elles se comportent dans ce cas là comme des props privées et nous évitent de « polluer » nos objets par l'affichage de props non-utiles au prog principal

```
// perte des symboles lors d'une serialisation
console.log(JSON.stringify(personne));

// symbole non inclus dans une boucle for in
for (let prop in personne) { console.log(prop); }
```

P1-C3/symbol.js

- Il est cependant possible de lister les Symboles d'un objet avec **Object.getOwnPropertySymbols()**

```
// Liste des Symboles de l'objet personne
console.log(Object.getOwnPropertySymbols(personne));
```

P1-C3/symbol.js

Itérateurs & Itérables

- Un cas courant d'utilisation des **Symbols** est lié à l'utilisation d'**Itérateurs** et d'**Itérables**
- Un **Itérateur** est un objet ayant une méthode **next()** qui retourne à chaque appel un objet ayant une propriété **value** et une propriété **done** (à **false** si d'autres values sont disponibles / **true** autrement)
- Les **Itérables** sont des structures de données comprenant un **Itérateur** récupérable par leur méthode **Symbol.iterator()**. JS définit plusieurs structures de données **Itérables** comme les **Array**, les **String**, etc.

```
// ITERATIONS DE TABLEAUX
const arr = [1, 2 ,3];
const iterator = arr[Symbol.iterator]();
```

```
// ITERATIONS DE CHAINES
const str = 'hello';
const iterator2 = str[Symbol.iterator]();
```

P1-C3/iterateurs.js

Parcours d'Itérables

- Il est possible d'itérer sur (de parcourir) les structures de données **Itérables** au moyen de boucles **for... of** auxquelles on passe l'**Itérateur** souhaité. La boucle **for... of** accepte aussi un tableau à parcourir directement

```
// ITERATION 1 : on utilise l'iterator dans une boucle for of
for(var v of iterateur){ console.log(v); };
```

P1-C3/iterateurs.js

- On peut également appeler la méthode **next()** de cet **Iterateur**

```
// ITERATION 2 : on utilise la méthode next() de l'iterator
console.log(iterateur.next());
console.log(iterateur.next());
console.log(iterateur.next());
console.log(iterateur.next());
```

P1-C3/iterateurs.js

Protocole d'itération

- Javascript définit un « protocole » pour itérer sur des structures de données :
 - On peut itérer sur un objet `Iterable`
 - Un objet est `Iterable` s'il implémente la clé `Symbol.iterator`
 - La clé `Symbol.iterator` doit retourner un `Iterator`
 - Un `Iterator` est un objet qui implémente une méthode `next()`
 - La méthode `next()` d'un `Iterator` doit retourner un objet de la forme :
`{value:(valeur),done:(boolean)}`. Sa valeur correspond à chaque appel à un nouvel élément de la structure de données
 - L'itération est considérée terminée lorsque `next()` renvoie une propriété `done` à `true`

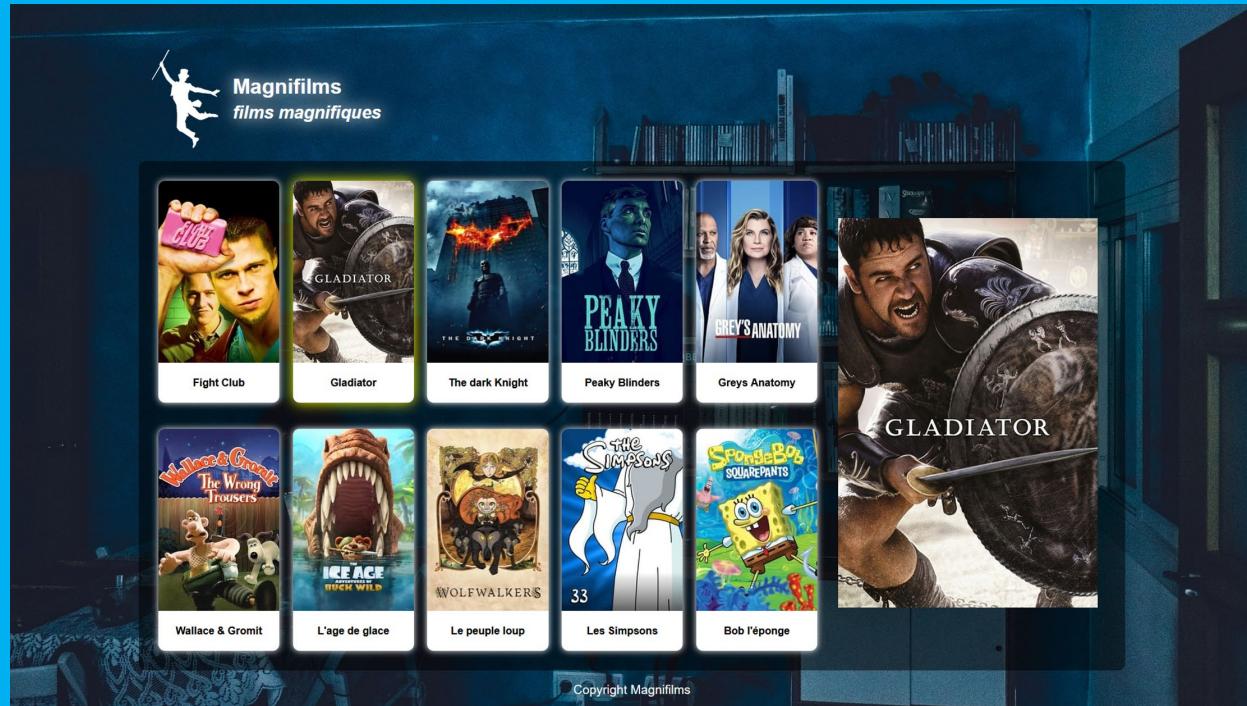
Itérateurs personnalisés – v1

- On peut donc suivre le protocole pour définir son propre `Iterator` :

```
// création d'un type d'objet Phrase
function Phrase(laPhrase){
    let mots = laPhrase.split(" ");
    // ici, on implémente l'itérateur en suivant le protocole
    this[Symbol.iterator]=function(){
        let i = 0;
        return {
            next: () => {
                if(i<mots.length){
                    return {value:mots[i++],done:false};
                }
                return {value:undefined,done:true};
            }
        };
    }
}
let phr = new Phrase("Voici une belle phrase.");
for(let mot of phr){ console.log(mot); }
```

Fil rouge - étape 4

- Créer un objet Filmotheque, permettant de gérer une liste de Film
- On initialise Filmotheque en lui transmettant un tableau de Film
- Implémenter le Symbol Iterator de telle manière à ce qu'une boucle for of sur une variable de type Filmotheque permette d'obtenir les prochains Films
- Faire en sorte qu'un survol sur chaque vignette affiche celle-ci en grand dans #droite et qu'aucun survol rétablisse la phrase précédente



Générateurs

- Une **fonction génératrice** permet de réaliser, de manière indirecte, plusieurs **returns**. Pour l'utiliser, on place une ***** après **function**, et on utilise les mots clés **yield** au lieu de **return**
- Chaque fonction génératrice retourne un **GeneratorObject** indépendant avec une méthode **next()**. Chaque appel de **next()** retourne un objet avec une propriété **value** (correspondant aux yields) et une propriété **done** (true/false)

```
function* fctGeneratrice(){  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
let objGenerateur = fctGeneratrice();  
  
console.log(objGenerateur.next()); // { value: 1, done: false }  
console.log(objGenerateur.next()); // { value: 2, done: false }  
console.log(objGenerateur.next()); // { value: 3, done: false }  
console.log(objGenerateur.next()); // { value: undefined, done: true }
```

P1-C3/generateurs.js

Générateurs

- Un cas pratique d'utilisation peut être un générateur d'IDs

```
// GENERATION D'ID
function* generatriceId(){
    let id=1;
    while(true){
        yield id;
        id++;
    }
}
let generateurId = generatriceId();
console.log(generateurId.next()); // { value: 1, done: false }
console.log(generateurId.next()); // { value: 2, done: false }
console.log(generateurId.next()); // { value: 3, done: false }
```

P1-C3/generateurs.js

Générateurs

- On peut aussi parcourir un tableau avec une syntaxe alternative

```
// PARCOURT DE TABLEAU EN UTILISANT UN GENERATEUR
function* generatriceArray(tab){
    for(let i=0;i<tab.length;i++){
        yield tab[i];
    }
}
let generateurArray = generatriceArray(["A","B","C"]);
console.log(generateurArray.next()); // { value: "A", done: false }
console.log(generateurArray.next()); // { value: "B", done: false }
console.log(generateurArray.next()); // { value: "C", done: false }
console.log(generateurArray.next()); // { value: undefined, done: true }
```

P1-C3/generateurs.js

Itérateurs personnalisés – v2

- On peut utiliser un générateur pour simplifier l’itération précédente :

```
// EXEMPLE D'ITERATEUR PERSONNALISÉ (AVEC GENERATEUR)

// création d'un type d'objet Phrase
function Phrase(laPhrase){
    this.mots = laPhrase.split(" ");
    // ici, on implémente avec une fonction génératrice
    this[Symbol.iterator]=function*(){
        for(let mot of this.mots){
            yield mot;
        }
    }
}

// utilisation
let phr = new Phrase("Voici une belle phrase.");
for(let mot of phr){ console.log(mot); }
```

Itérateurs personnalisés – v3

- Ou encore, en ES6

```
// EXEMPLE D'ITERATEUR PERSONNALISÉ (AVEC GENERATEUR, EN ES6)

// création d'un type d'objet Phrase
class Phrase{
    constructor(laPhrase){
        this.mots = laPhrase.split(" ");
    }
    // ici, on implémente avec une fonction génératrice
    *[Symbol.iterator](){
        for(let mot of this.mots){
            yield mot;
        }
    }
}

// utilisation
let phr = new Phrase("Voici une belle phrase.");
for(let mot of phr){ console.log(mot); }
```

Fil rouge - étape 4 (suite)

- Modifier la Filmotheque en utilisant un générateur

Promises

- Nouvel objet ES6 pour le développement asynchrone : la **promise** (promesse)

Pour rappel, le **développement asynchrone** consiste en l'utilisation **d'opérations asynchrones** : instructions dont on n'attend pas l'aboutissement avant de passer à la suite de notre programme. Leur aboutissement déclenche en revanche une fonction dite de "callback". Ex : écoute d'évènement, setInterval(), appel API, etc.

- Une **promise** représente une valeur qui peut être disponible maintenant, dans le futur, ou jamais. Il s'agit d'un objet réalisant en général une **opération asynchrone** (ex : appel API) et qui peut avoir 3 états :
 - **pending** (en attente), tant que l'opération n'a pas abouti
 - **fulfilled** (promesse tenue) si l'opération se passe bien
 - **rejected** (promesse rompue) s'il y a eu une erreur lors de l'opération

Promises - exécuteur

- Chaque promise prend une fonction en paramètre (appelée « exécuteur »), qui elle-même attend 2 paramètres :
 - un nom (librement choisi) correspondant à une « fonction de callback » à appeler en cas de succès de son opération asynchrone, si la promesse est tenue (ici, resolve)
 - un nom (librement choisi) correspondant à une « fonction de callback » à appeler en cas d'erreur, si la promesse n'est pas tenue (ici, reject)
- On peut utiliser ces noms de fonctions au sein de la promise pour définir que la promesse **est tenue ou pas**, tout en transmettant une valeur

```
// Déclaration : promesse directement résolue/rejetée
var promesse = new Promise((resolve,reject) => {
    resolve(1); // OU BIEN : reject(1);
});
```

```
// Utilisation
console.log(promesse); // Promise {<state>:"fulfilled", <value>:1}
```

P1-C3/promises.js

Promises – then()

- Après avoir définit une promise, on l'utilise en appelant sa méthode `then()`, qui permet de **décrire les actions à faire en cas de promesse tenue**
- `then()` prend ainsi une fonction dont le paramètre correspond à la valeur renournée par `resolve()`

```
// Déclaration : promesse directement résolue/rejetée
var promesse = new Promise((resolve,reject) => {
    resolve(1);
});

// Utilisation
console.log(promesse); // Promise {<state>:"fulfilled", <value>:1}
promesse.then( v => { console.log(v); }); // 1
```

P1-C3/promises.js

Promises – catch()

- Lors de l'utilisation de la promise, sa méthode `catch()` permet de **décrire les actions à faire en cas de promesse non-tenu**
- `catch()` prend une ainsi **une fonction dont le paramètre correspond à la valeur renournée par `reject()`**

```
// Déclaration : promesse directement résolue/rejetée
var promesse = new Promise((resolve,reject) => {
    reject(1);
});
// Utilisation
console.log(promesse);
// promesse.then( (v) => { console.log(v); });
promesse.catch( (v) => { console.log(v); }) // 1
```

P1-C3/promises.js

- Comme `then()` et `catch()` retournent la promesse, on peut les chainer :

```
promesse
    .then( (v) => { console.log(v); })
    .catch( (v) => { console.log(v); });
```

P1-C3/promises.js

Promises – exemple

- Exemple de récupération d'une salutation au bout d'une seconde :

```
// définition de la promesse : une salutation à venir (dans 1s)
var getSalutation = new Promise(function (resolve, reject) {
    // reject("Aucune salutation"); // si on veut forcer l'erreur
    setTimeout(function () { resolve("Bonjour"); }, 1000);
});

// utilisation : dans 1s, salutation (Bonjour) sera ici affichée
getSalutation
    .then(function (salutation) { console.log(salutation); })
    .catch(function (erreur) { console.error(erreur); })
```

P1-C3/promises.js

Chainage de then() & finally()

- Si on utilise plusieurs `then()`, la valeur retournée par la fonction du 1er `then()` est passée en tant que paramètre à la fonction du `then()` suivant
- `finally()` (es9) peut être ajoutée afin de préciser les actions à faire lorsque la promesse à abouti, qu'il s'agisse d'un succès ou d'une erreur

```
// utilisation : dans 1s, salutation (Bonjour) sera ici affichée
getSalutation
  .then(function (salutation) {
    console.log(salutation+" 1");
    return salutation;
})
  .then(function (salutation) {
    console.log(salutation+" 2");
})
  .catch(function (erreur) {
    console.error(erreur);
})
  .finally(function(){
    console.log("promesse aboutie");
});
```

Agrégation de promesses

```
let p1 = new Promise((succes,echec)=>{setTimeout(()=>{succes("OK1")},1000);});  
let p2 = new Promise((succes,echec)=>{setTimeout(()=>{succes("OK2")},1000);});  
let p3 = new Promise((succes,echec)=>{setTimeout(()=>{succes("OK3")},1000);});  
  
// agrégation de 3 promesses en une, résolu lors les 3 le seront  
// si l'une des promesses échoue, la promesse aggregée échoue immédiatement  
Promise.all([p1,p2,p3]).then((valeurs) => { console.log(valeurs); });  
  
// agrégation de 3 promesses en une, résolu lors les 3 le seront  
// le résultat indiquera pour chaque promesse si elle a réussi ou échoué,  
// ainsi que sa valeur  
Promise.allSettled([p1,p2,p3]).then((valeurs) => { console.log(valeurs); });  
  
// agrégation de 3 promesses en une, résolu dès que l'une le sera  
// erreur si aucune ne l'est  
Promise.any([p1,p2,p3]).then((valeurPlusRapide) => {  
  console.log(valeurPlusRapide); });  
  
// agrégation de 3 promesses en une, résolu dès qu'une abouti (succes/echec)  
Promise.race([p1,p2,p3]).then((valeurPlusRapide) => {  
  console.log(valeurPlusRapide); });
```

Spread operator

- En JS, les types primitifs sont copiés par valeurs, et les objets par référence.
Si on modifie un tableau, sa copie est affectée :

```
let tab1=[1,2,3];
let tab2=tab1;
tab1[0]=0; // impact sur tab2
console.log(tab2); // [0,2,3]
```

P1-C3/spread.js

- Le **spread operator**, qui s'écrit “`...`” s'applique sur tout objet itérable (tableau, chaîne de caractères) et permet de décomposer tous ses éléments. On peut l'utiliser, par exemple, pour créer une copie indépendante :

```
let tab1=[1,2,3];
let tab2=[...tab1];
tab1[0]=0; // aucun impact sur tab2
console.log(tab2); // [1,2,3]
```

P1-C3/spread.js

Spread operator

- Attention, s'il s'agit de tableaux d'objets, leur références restent conservées. Toute modification d'un objet de tab1 sera visible dans tab2

```
let tab1 = [{x:1},{x:2}];  
let tab2 = [...tab1,{x:3}];  
tab1[0].x=4;  
console.log(tab2); // [{x:4},{x:2},{x:3}]
```

P1-C3/spread.js

- On peut aussi utiliser le spread operator directement sur des objets (es9). Mais il s'agit d'une copie superficielle (shallow copy) et non en profondeur (deep copy) : seuls les premiers niveaux d'un objets sont copiés par valeur, les autres par référence

```
const obj = { w: { a: 0 }, x: 1, y: 2 };  
// let clone = Object.assign({},obj); // avant ES9  
let clone = {...obj}; // depuis ES9  
console.log(clone); // {w: { a: 0 }, x: 1, y: 2 }  
obj.x=5; // pas d'impact sur clone (copie par valeur)  
obj.w.a=4; // impact sur clone (copie par réf.)  
console.log(clone); // {w: { a: 4 }, x: 1, y: 2, z: 3}
```

P1-C3/spread.js

Spread operator

- On peut aussi l'utiliser pour définir une fonction au nombre indéfini de paramètres (appelé rest param) :

```
function additionner(...nombres){  
    // nombres est vu comme un tableau  
    return nombres.reduce(function(resultat,nbCourant){  
        return resultat+nbCourant;  
    });  
  
console.log(additionner(1,1,1));  
console.log(additionner(1,1,1,1,1));
```

P1-C3/spread.js

Partie 1 - Chapitre 4

Ecmascript : versions suivantes

EcmaScript 7 (ES2016)

- Il y a 2 principales nouveautés :
 - l'opérateur de puissance
 - la méthode include du prototype Array
- L'**opérateur de puissance** s'écrit ****** et est utilisable comme suit :

```
console.log(2**2); // 4  
console.log(2**3); // 8
```

P1-C4/ES7.js

- La méthode **includes()** permet de savoir si un tableau contient une valeur :

```
let t = [1,2,3];  
console.log(t.includes(2)); // true  
console.log(t.includes(4)); // false
```

P1-C4/ES7.js

EcmaScript 8 (ES2017)

- Il y a 3 principales nouveautés :
 - String padding
 - Nouvelles propriétés du prototype Object
 - Async / await
- Les **String paddings** permettent de garantir une taille de chaîne minimum.
On peut les appliquer avec **padStart()** et **padEnd()** :

```
let ref = "57";
console.log(ref.padStart(5, "0"));    // 00057
console.log(ref.padEnd(5, "0"));     // 57000
```

P1-C4/ES8.js

- **Object.values()** retourne les valeurs d'un objet en tableau :

```
let langues = {FR:"Francais",IT:"Italien"};
console.log(Object.values(langues)); // [ 'Francais', 'Italien' ]
```

P1-C4/ES8.js

EcmaScript 8 (ES2017)

- `Object.entries()` retourne les propriétés et valeurs d'un objet en tableau :

```
let langues = {FR:"Francais",IT:"Italien"};
console.log(Object.entries(langues));//[['FR','Francais'], ['IT','Italien']]
```

P1-C4/ES8.js

- `Object.getOwnPropertyDescriptors()` permet d'avoir des infos sur ses props

```
let langues = {FR:"Francais",IT:"Italien"};
console.log(Object.getOwnPropertyDescriptors(langues));
```

P1-C4/ES8.js

- Les **trailing commas** nous évitent des erreurs de syntaxes si un virgule termine la liste des paramètres d'une fonction

```
function maFonction(arg1,arg2,){}
```

P1-C4/ES8.js

ES8 – async

- Principales nouveautés ES8, les mots clés `async` et `await` simplifient la prog. asynchrone. Ils « masquent » l'utilisation de Promises tout en maintenant une exécution asynchrone
- `async` transforme une fonction synchrone en fonction asynchrone. Celle-ci retournera alors implicitement une `Promise`. Tout `return` équivaut alors à un `resolve()`, et tout `throw new Error` équivaut à un `reject()`. Il n'est cependant pas obligatoire que la fonction asynchrone retourne une valeur

P1-C4/ES8.js

```
// fonction asynchrone
async function getSalutationAsync(){
    // throw new Error("erreur"); // équivaut à Promise.reject("erreur");
    return "Hello";           // équivaut à Promise.resolve("Hello");
}

console.log(getSalutationAsync()); // Promise { [...] <value>: "Hello" }

getSalutationAsync()
    .then( salutation => {console.log(salutation);} )
    .catch( erreur => {console.error(erreur);} );
```

ES8 – await

- `await` n'est utilisable que dans une fonction **asynchrone**, et permet d'attendre l'aboutissement d'opérations asynchrones. Il s'agit d'un sucre syntaxique qui équivaut à une utilisation de `then()`, mais en plus lisible. On peut dire qu'`await` écrit un `then()` à notre place, en y plaçant la suite des instructions de sa fonction

```
async function helloAsync(){
    console.log("début");    // affiché avant Hello
    const hello = await new
Promise((resolve,reject)=>{setTimeout(()=>{resolve("Hello");},2000);});
    console.log(hello);
    console.log("fin");      // affiché après Hello
}
helloAsync();

// EQUIVALENT DE :
function hello(){
    new Promise((resolve,reject)=>{setTimeout(()=>{resolve("Hello");},2000);})
        .then((hello) => {
            console.log(hello);
            console.log("fin");      // affiché après hello
        });
    console.log("début");      // affiché avant hello
}
hello();
```

P1-C4/ES8.js

ES8 – gestion des erreurs avec await

- Il est possible de gérer les erreurs avec un simple try catch

```
async function helloAsync(){
  try{
    console.log("début");    // affiché avant Hello
    const hello = await new
Promise((resolve,reject)=>{setTimeout(()=>{reject("Erreur");},2000);});
    console.log(hello);
    console.log("fin");      // affiché après Hello
  }catch(e){
    console.error("ERREUR");
  }
}
helloAsync();
```

P1-C4/ES8.js

ES8 – await

- La simplicité de lecture de la syntaxe avec `await` est encore plus claire lors de l'appel de multiples opérations asynchrones. On évite le « callback hell »

```
async function helloWorldAsync(){
    const hello = await new Promise((resolve,reject)=>{setTimeout(()=>{resolve("Hello")},2000);});
    console.log(hello);
    const world = await new Promise((resolve,reject)=>{setTimeout(()=>{resolve("world")},2000);});
    console.log(world);
}
helloWorldAsync();

// EQUIVALENT DE :
function helloWorld(){
    new Promise((resolve,reject)=>{setTimeout(()=>{resolve("Hello")},2000);})
        .then((hello) => {
            console.log(hello);
            return new Promise((resolve,reject)=>{setTimeout(()=>{resolve("world")},2000);});
        })
        .then((world)=>{
            console.log(world);
        });
}
helloWorld();
```

P1-C4/ES8.js

EcmaScript 9 (ES2018)

- ES9 apporte :
 - le spread operator pour **le destructuring d'objets** (déjà abordé)
 - la méthode **finally()** pour le traitement des promises (déjà abordée)
 - divers **ajouts liés aux RegExp** (abordés plus tard)
 - l'**itération asynchrone**
- « protocole » d'itération asynchrone :
 - On peut itérer sur un objet **Asynchronous Iterable**
 - Un objet est **Asynchronous Iterable** s'il implémente la clé **Symbol.asyncIterator**
 - La clé **Symbol.asyncIterator** doit retourner un **AsyncIterator**
 - Un **AsyncIterator** est un objet qui implémente une méthode **next()**
 - La méthode **next()** d'un **AsyncIterator** **doit retourner une Promise qui lors de son resolve renverra un objet de la forme : {value:(valeur),done:(booleen)}**. Sa valeur correspond à chaque appel à une nouvelle donnée
 - L'itération est considérée terminée lorsque **next()** renvoie une propriété **done** à **true** à travers sa promise

Itération asynchrone

- Modification de l'itérateur précédent. `For...await...of` boucle sur un `async Iterable`

P1-C4/ES8.js

```
class Phrase{
    constructor(laPhrase){
        this.mots = laPhrase.split(" ");
    }
    // ici, on implémente avec une fonction génératrice asynchrone
    async *[Symbol.asyncIterator](){
        for(let mot of this.mots){
            // ici on marque une pause (ex. d'opération asynchrone)
            ajout // ici on marque une pause (ex. d'opération asynchrone)
            await new Promise((resolve,reject) => {
                setTimeout(() => { resolve() }, 1000);
            });
            yield mot;
        }
    }
}
// utilisation
(async function(){ ajout : fonction async auto executée pour utiliser for await of
    let phr = new Phrase("Voici une belle phrase.");
    for await(let mot of phr){ console.log(mot); } for await of
})();
```

Ajout
d'async
et modif
de prop

ajout

P1-C4/ES8.js

for await of

Partie 1 - Chapitre 5

Fetch API

Fetch API

- Traditionnellement, pour construire des requêtes HTTP en JavaScript on utilise un objet **XMLHttpRequest**. Mais cet objet est globalement peu apprécié des développeurs (fastidieux, pb de compatibilité, nombreux callbacks), qui lui ont préféré la surcouche offerte par des librairies (jQuery, axios, etc.)
- Désormais, tous les navigateurs (sauf IE) proposent une nouvelle méthode pour réaliser des appels AJAX avec des promises : l'**API Fetch**. Il s'agit d'une API Web, et non du JS. Voir le package **node-fetch** pour le support node
- **fetch(url)** retourne une promise. L'appel est donc de la forme :

```
// SIMPLE APPEL AVEC FETCH
// (https://www.boredapi.com/api/activity => idée activité)
fetch("https://www.boredapi.com/api/activity")
  .then(reponse => { console.log(reponse); })
  .catch(erreur => { console.error(erreur); });
```

P1-C5/fetch.js

- Si toute ce passe bien, nous obtenons en console un objet **Response** dont le body est un **ReadableStream**, mais qui n'est pas directement exploitable

Fetch API – gestion des erreurs

- Si nous coupons la connexion, `catch()` sera appelée. Mais elle ne gère que les erreurs réseaux, et `then()` reste appelée si le serveur retourne une 404 !

```
// SIMPLE APPEL FETCH AVEC GESTION DES ERREURS
fetch("https://httpstat.us/404")                                // erreur 404
  .then(reponse => { console.log(reponse); })                  // mais exécutée
  .catch(erreur => { console.error(erreur); });
```

P1-C5/fetch.js

- Il faut donc aussi vérifier la propriété `reponse.ok` dans `then()` pour détecter des erreurs HTTP. S'il y en a, on appelle `Promise.reject()` pour forcer `catch()`

```
// SIMPLE APPEL FETCH AVEC GESTION DES ERREURS
fetch("https://httpstat.us/404")
  .then(reponse => {
    if(!reponse.ok) return Promise.reject("ERREUR"); // si erreur
    console.log(reponse); // si pas d'erreur
  })
  .catch(erreur => { console.error(erreur); });
```

P1-C5/fetch.js

Fetch API – conversion de données

- La réponse peut être convertie en plusieurs formats au moyen des méthodes `reponse.json()`, `reponse.text()`, etc. en fonction du type de donnée souhaitée
- On peut donc passer la réponse convertie et exploitable à un second `then()`

```
// SIMPLE APPEL FETCH AVEC GESTION DES ERREURS ET CONVERSION
fetch("https://www.boredapi.com/api/activity")
  .then(reponse => {
    if(!reponse.ok) return Promise.reject("ERREUR"); // si erreur
    else return reponse.json(); // si pas d'erreur, conversion
  })
  .then(json => console.log(json)) // réponse convertie / exploitable
  .catch(erreur => { console.error(erreur); });
```

P1-C5/fetch.js

Fetch API – appel en GET

- Il est possible d'utiliser des API tierces pour faire des tests, comme {JSON} Placeholder : <https://jsonplaceholder.typicode.com>
- Il existe une liste de ressources que l'on peut tester
- Et pour chaque ressource plusieurs routes en fonction de l'opération à réaliser (récupérer une liste, un élément, certaines propriétés d'un élément, ou bien ajouter un élément, modifier un élément, supprimer un élément, etc.) :

| | | | |
|---------------------------|--------------|--------|------------------------------------|
| /posts | 100 posts | GET | /posts |
| /comments | 500 comments | GET | /posts/1 |
| /albums | 100 albums | GET | /posts/1/comments |
| /photos | 5000 photos | GET | /comments?postId=1 |
| /todos | 200 todos | POST | /posts |
| /users | 10 users | PUT | /posts/1 |
| | | PATCH | /posts/1 |
| | | DELETE | /posts/1 |

Fetch API – appel en GET

- Suivant les verbes utilisés, on effectue par convention des actions diverses :
 - **GET** : sélection / récupération de ressource(s)
 - **POST** : ajout de ressource(s)
 - **PUT / PATCH** : modification de ressource(s)
 - **DELETE** : suppression de ressource(s)
- Effectuons un appel GET pour récupérer une ressource, sans gérer ici les erreurs. On obtient alors un objet représentant la ressource TODO d'id 1 :

```
// APPEL GET, VERS UNE RESSOURCE D'ID 1
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(reponse => reponse.json())
  .then(json => console.log(json))
```

P1-C5/fetch.js

- Pour obtenir une liste de plusieurs TODOs, modifions l'URL :

```
// APPEL GET, VERS UNE RESSOURCE D'ID 1
fetch('https://jsonplaceholder.typicode.com/todos')
  .then(reponse => reponse.json())
  .then(json => console.log(json))
```

P1-C5/fetch.js

Fetch API – appel en POST

- Par défaut `fetch()` envoie en `GET`, mais il est possible d'indiquer d'autres verbes, d'envoyer des données dans le body, ou de préciser des entêtes HTTP via un objet options passé en second paramètre
- Ajoutons une ressource TODO à l'aide du verbe POST :

```
// APPEL EN POST, POUR AJOUTER UNE RESSOURCE TODO
fetch('https://jsonplaceholder.typicode.com/todos',
{
  method: 'POST',
  body: JSON.stringify({userId:'1',title:'faire les courses',completed:0}),
  headers: { 'Content-type': 'application/json; charset=UTF-8' }
})
.then((response) => response.json())
.then((json) => console.log(json));
```

P1-C5/fetch.js

- On remarque que la ressource ajoutée est retournée en réponse, avec son id

Fetch API – PATCH & DELETE

- Effectuons une modification de ressource TODO à l'aide du verbe PATCH :

```
// APPEL EN PATCH, POUR MODIFIER UNE RESSOURCE TODO
fetch('https://jsonplaceholder.typicode.com/todos/201', {
  method: 'PATCH',
  body: JSON.stringify( { completed: '1' } ),
  headers: { 'Content-type': 'application/json; charset=UTF-8' }
})
.then((response) => response.json())
.then((json) => console.log(json));
```

P1-C5/fetch.js

- Supprimons la ressource :

```
// APPEL EN DELETE, POUR SUPPRIMER UNE RESSOURCE TODO
fetch("https://jsonplaceholder.typicode.com/todos/1", {method: "DELETE"});
```

P1-C5/fetch.js

Compte d'API TMDB

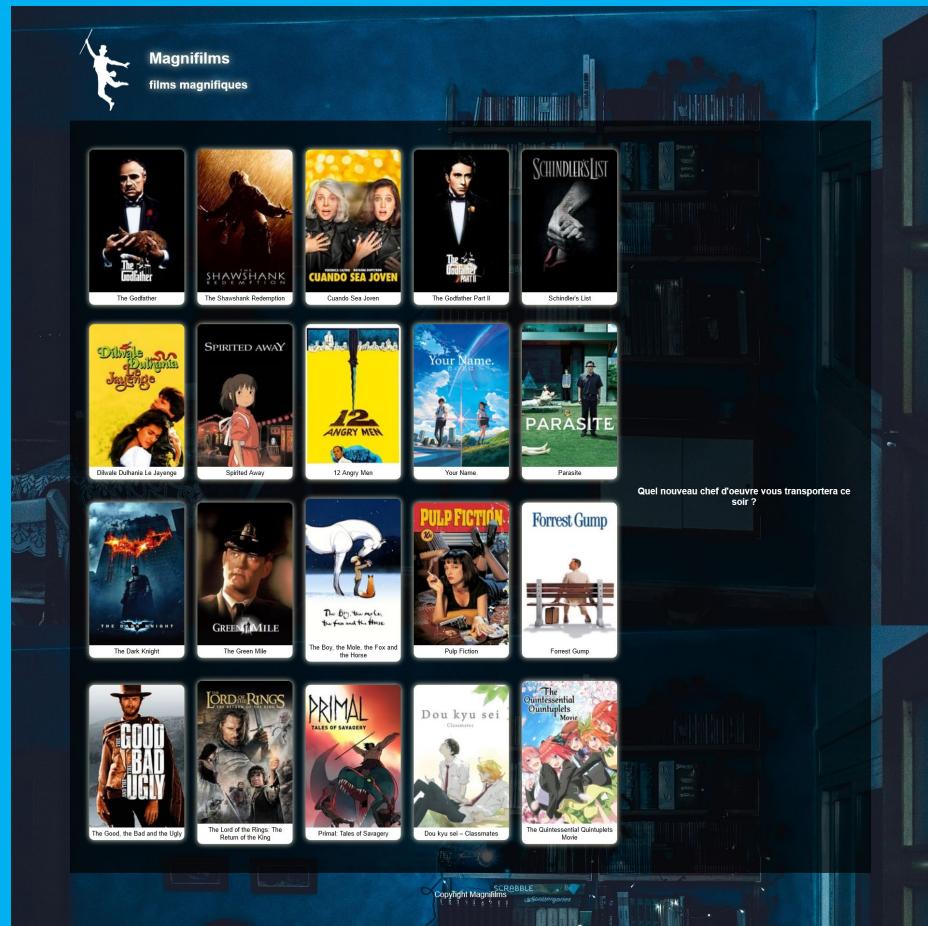
- Créons un accès à l'API TMDB : The MovieDataBase
- Créez un compte sur <https://www.themoviedb.org>, identifiez-vous, puis cliquez sur votre avatar / paramètres / API / Créer
- Type d'utilisation : éducation
- Nom de l'application : Magnifilms
- Adresse internet de l'application : localhost:3000
- Documentation : <https://developers.themoviedb.org/3/getting-started/introduction>

Infos sur l'API TMDB

- Routes `/movie/popular` et `/movie/top_rated`
 - Permet d'obtenir les films les plus populaires ou les mieux notés
 - https://api.themoviedb.org/3/movie/popular?api_key=CLE_API
- Route `/configuration` :
 - permet d'obtenir des infos de configuration, comme la `base_url`, les `file sizes` disponibles, etc.
 - https://api.themoviedb.org/3/configuration?api_key=CLE_API
- Routes `/movie/{id}` et `/tv/{id}`
 - permet d'obtenir les infos spécifique à un film ou une série tv, et ses `files path` d'images
 - https://api.themoviedb.org/3/movie/550?api_key=CLE_API
- L'affichage d'une image de film est de la forme : `base_url/file_size/file_path`
 - ex : <https://image.tmdb.org/t/p/w185/pB8BM7pdSp6B6lh7QZ4DrQ3PmJK.jpg>

Fil rouge - étape 5

- Faites des requêtes vers l'API TMDB :
 - un appel vers `/configuration` pour récupérer la base URL des images, ainsi que les tailles disponibles, utiles pour reconstituer chaque URL complète d'image
 - un appel vers `/movie/popular` afin de récupérer les films populaires
- Suite à ces 2 appels AJAX, et en utilisant ces données, modifier l'application pour que les films proviennent d'appels AJAX vers The Movie Database
- Ajoutez à ces films les propriétés : vote, annee, resume



Partie 2 : TypeScript

Partie 2 - Chapitre 1

Introduction à TypeScript

Introduction

- JavaScript est un langage :
 - **Interprété** : sans compilation, le code n'est pas vérifié avant son exécution
 - **Faiblement typé** : une variable n'est pas conditionnée à un seul type de données, et il est possible de changer son type lors de l'exécution du programme
- Ces caractéristiques ont des avantages mais aussi des inconvénients :
 - Nécessité de **tester** très rigoureusement son code
 - **Pas d'aides** lors du développement
 - Apparition de **potentiels bugs** au runtime
- Pour pallier à ces inconvénients, il est possible d'utiliser des librairies comme vérificateur de type comme **Flow** ou un linter comme **ESLINT** afin d'avoir des remontée d'alertes sous forme de warning
- **Mais une solution plus complète consiste à développer en TypeScript**

Introduction

- **TypeScript** est une surcouche du langage JavaScript développée par Microsoft. Le principe global du TypeScript est de typer toutes les variables
- Néanmoins, utiliser du TypeScript ne se résume pas simplement à cela. Le langage vise à apporter un certain nombre d'améliorations vis à vis du JavaScript, de la **programmation orientée objet** à la **programmation générique** par exemple
- Lorsque l'on développe en TypeScript (fichiers **.ts**) on doit utiliser un **compilateur** qui traduira finalement notre code en JavaScript.
- On parle d'ailleurs plutôt de **transcompilation**, et non de **compilation**, puisque l'on traduit un code écrit dans un langage en un code d'un langage du même niveau (à l'inverse de la compilation où l'on passe d'un langage lisible humainement à un langage bas niveau)

Mise en place et 1^{er} test

- Installons typescript avec la ligne de commande suivante :

```
npm install -g typescript
```

CMD, dans P2-C1

- Créer un fichier [/P2-C1/test.ts](#) :

```
function calculerSomme(a:number,b:number){ return a+b; }
let nb1 = prompt("NB 1");
let nb2 = prompt("NB 2");
console.log(calculerSomme(nb1,nb2));
```

P2-C1/test.ts

- Dans un Terminal CMD, transpiler le fichier [test.ts](#) afin de générer le fichier [test.js](#). Utiliser la commande [tsc test.ts](#) (ou bien juste [tsc](#)) qui fait appel au compilateur Typescript et génère un fichier [.js](#) à partir du fichier [.ts](#) s'il n'y a pas d'erreurs TypeScript

```
tsc test.ts
```

CMD, dans P2-C1

Observations

- TypeScript nous alerte sur le fait que notre fonction prend en paramètre 2 **nombres**, et que les valeurs récupérées sont des **chaînes**. Il nous force donc à faire la conversion, ce qui nous prémunit d'un potentiel bug à l'exécution
- Dans VSCode, pour éviter l'erreur **duplicate function implementation**, utiliser la commande **tsc --init** qui génère une fichier de config **tsconfig.json**
- Le code sera transpilé en le modifiant comme ceci :

```
function calculerSomme(a:number,b:number):number{ return a+b; }
let nb1:number = Number(prompt("NB 1"));
let nb2:number = Number(prompt("NB 2"));
console.log(calculerSomme(nb1,nb2));
```

P2-C1/test.ts

- C'est un exemple simple, mais dans une application plus complexe, le fait de typer explicitement nos variables limitera beaucoup les bugs

Watch mode

- Le **watch mode** permet d'être à l'écoute permanente des changements côté Typescript, et de générer un fichier **.js** à chaque sauvegarde du fichier **.ts**
- On peut lancer le **watch mode** avec le flag **-w**

```
tsc -w test.ts
```

CMD, dans P2-C1

- Ou bien sans spécifier le fichier :

```
tsc -w
```

CMD, dans P2-C1

Partie 2 - Chapitre 2

L'outil NPM

Introduction

- Npm (Node Package Manager) est un gestionnaire de dépendances. Historiquement utilisé sur des projets NodeJS, il l'est désormais également sur beaucoup de projets de développement Web, et tous les frameworks JS front l'utilisent
- NPM est en 3 parties : une base de données, un site, une CLI
- NPM met à disposition une gigantesque base de données (le registry npm) contenant des librairies JS publiques ou privées
- Chaque librairie doit suivre le versioning semver. Chaque numéro de version est de la forme MAJOR.MINOR.PATCH. Suivant les changements, on incrémente :
 - MAJOR lorsque l'on a un changement d'API qui introduit des incompatibilités avec le code utilisant la librairie dans sa version précédente
 - MINOR lorsque l'on ajoute de nouvelles fonctionnalités rétro-compatibles avec la version précédente ;
 - PATCH lorsque l'on corrige des bugs qui n'introduisent pas d'incompatibilités

NPM en pratique

- Dans un projet utilisant NPM, on retrouve deux fichiers principaux, et un dossier :
 - `package.json`
 - `package-lock.json`
 - `/node_modules`
- Le fichier `package.json` contient :
 - Des informations générales sur notre projet
 - Des scripts (alias de commandes)
 - La liste des dépendances de notre projet (on peut indiquer des dépendances de production, de développement, ou des dépendances transitives)
- Le fichier `package-lock.json` contient des métadonnées issues de l'installation des packages. Il permet de figer l'arbre de dépendances d'un projet
- Il existe également un répertoire `node_modules` qui contiendra l'ensemble des librairies téléchargées via `npm` sur un projet. Ce dossier doit être ignoré par votre gestionnaire de sources

Utilisation

- Les commandes principales de `npm` sont :
 - `npm init [--yes]` : crée un fichier `package.json`
 - `npm install <nom d'un package npm>` : installe un package
 - `npm remove <nom d'un package npm>` : supprime un package
 - `npm pack / npm publish` : pour publier un package dans le registre
 - `npx <nom d'un package npm>` : exécute une commande de package
- Exemples d'installation de dépendances :
 - `npm install [--production]` : installe toutes les dépendances d'un projet
 - `npm install @angular/core` : installe le package `@angular/core` dernière version
 - `npm install @angular/core@9.1.0` : permet d'obtenir une version spécifique
 - `npm install -g @angular/cli` : installation d'un package au niveau OS (global)
 - `npx create-react-app helloworld =>` exécution d'un package (ici init. react) sans l'installer
 - `npm install typescript --save-dev =>` installation d'un package qui servira uniquement côté développement (ne sera pas inclus lors d'un packaging de production)

Utilisation

- Lorsque l'on ajoute une dépendance à notre projet, on peut contrôler la plage de versions autorisées lors de l'installation des dépendances sur un autre poste / répertoire (indispensable lorsque l'on travaille en équipe).
- Voici différentes options de configuration dans `package.json` :
 - `"@angular.core": "9.1.0"` : fixe une version
 - `"@angular.core": "^9.1.0"` : fixe le numéro de version majeure en autorisant les versions 9.1.0 à 9.X.X
 - `"@angular.core": "~9.1.0"` : fixe les numéros de version majeure & mineure : en autorisant les versions 9.1.0 à 9.1.X
 - il existe des variantes ("`<x.x.x`", "`x.x.x - x.x.x`")
- Cela fonctionne dans le meilleur des mondes si les développeurs des packages que l'on utilise respectent bien la norme semver

Partie 2 - Chapitre 3

Bases de TypeScript

Manipulation des types primitifs

- En TypeScript (TS), le type d'une variable est **déclaré** ou **inféré** :

```
// TYPES DECLARES OU INFERES
let nb1:number=10;    // type déclaré
let nb2 = 20;         // type inféré
nb2="test";          // erreur car son type est toujours number !
```

P2-C3/tests.ts

- Généralement on utilise l'**inférence de types** lorsque l'initialisation est immédiatement consécutive à la déclaration de la variable. On prendra pour bonne habitude de typer explicitement toutes les variables non initialisées immédiatement (exemple : paramètres d'une fonction)
- Pour le reste, et en ce qui concerne les **number**, **string**, et **boolean**, JS et TS fonctionnent de la même manière

Manipulation des objets & tableaux

- Par défaut, **on ne peut ajouter des propriétés aux objets que lors de leur initialisation**. Par ailleurs, **leurs types sont inférés** lors de la déclaration :

```
// MANIPULATION DES OBJETS
let personne = {nom:"Statham"};           // unique prop. nom de type string
personne.prenom="Jason";                  // erreur car prop. inexiste
personne.nom=2000;                       // erreur car prop. string
```

P2-C3/tests.ts

- On peut préciser **le type de valeurs contenues** dans un tableau ainsi :

```
// MANIPULATION DES TABLEAUX
let couleurs:string[] = ["blue","red","green"];
couleurs.push(000000);                   // erreur car tableau de chaines
for(let v of couleurs){                // TS "sait" que v est une chaine
    console.log(v);
}
```

P2-C3/tests.ts

Tuples

- TS permet de créer des **tuples** : tableaux dont le nombre et les types de valeurs sont prédéfinis :

```
// TUPLES
var adresse : [number, string, number, string, string];

// OK
adresse = [1, "place des Grand Hommes", 69000, "Lyon", "France"];

// erreur de type (1er élément)
// adresse = ["1", "place des Grand Hommes", 69000, "Lyon", "France"];

// TS reconnaît le type number et sa méthode
let voieAbregee = adresse[1].substring(0,9)+"...";
console.log(voieAbregee);

// erreur : seulement 5 valeurs
// adresse[5]="Europe";
```

P2-C3/tests.ts

Énumérés

- Les **enums** sont des groupes de constantes (read-only)
- Il y a des **enums numériques** (constantes liées à des nombres) et des **enums de chaînes** (constantes liées à des chaînes)

```
// ENUMS NUMERIQUES
enum COULEURS {
    PRIMAIRE, // 0
    SECONDAIRE, // 1
    CONTRASTEE // 2
};
console.log(COULEURS.PRIMAIRE); // 0
// COULEURS.PRIMAIRE=5;      // erreur
```

```
// ENUMS NUMERIQUES TOTALEMENT
INITIALISES
enum CODES {
    NOT_FOUND = 404, // 404
    SUCCESS = 200    // 200
};
console.log(CODES.SUCCESS); // 200
```

```
// ENUMS NUMERIQUES INITIALISES
enum COULEURS {
    PRIMAIRE = 5, // 5
    SECONDAIRE, // 6
    CONTRASTEE // 7
};
console.log(COULEURS.PRIMAIRE); // 5
```

```
// ENUMS DE CHAINES
enum DBCONFIG {
    DB_USER = "root",
    DB_PWD = "",
    DB_HOST = "db.mysupersite.org",
    DB_NAME = "siteDB",
}
console.log(DBCONFIG.DB_HOST);
```

Any

- Il est possible d'utiliser le mot clef `any` si l'on ne souhaite pas préciser de type. Cela revient à développer en JS natif donc l'intérêt reste limité.
- `any` est surtout utilisé lorsque les déclarations de type ne sont pas connues (exemple une librairie qui ne les fournit pas).

```
// ANY
let n:any = 1;
n = "salut"; // autorisé seulement si n est de type any

let user:any = { nom:"Statham" };
user.nom=1; // non autorisé, sauf si user est de type any
```

P2-C3/tests.ts

Union

- Il est possible de préciser qu'une variable peut avoir plusieurs types avec `|`.
Cette union de types autorise la variable à être d'un type ou d'un autre

```
// UNION
let x:string|number;
x = "toto"; // OK
x=56;       // OK
x=true;     // erreur : type non autorisé
```

P2-C3/tests.ts

Alias de type

- Afin d'éviter de répéter une définition de type, on peut créer un **alias de type**

```
// ALIAS DE TYPE
type ChaineOuNombre = string | number;
let x:ChaineOuNombre;
x = "toto"; // OK
x=56;        // OK
x=true;      // erreur : type non autorisé
```

P2-C3/tests.ts

- Il est utilisable sur des objets :

```
type Produit = {titre:string,prix:number};
const p1:Produit = {titre:"Banane",prix:3.70};
const p2:Produit = {titre:"Pomme",prix:"1.70/Kg"}; // erreur !
```

P2-C3/tests.ts

Type littéral

- Le **type littéral** permet de définir des valeurs exactes. Cela peut être pratique en combinaison avec **l'union de types**, pour limiter les valeurs possibles

```
// TYPE LITERAL
let res:"OK"|"KO"|200|500;
res="OK";    // OK
res=200;    // OK
res=404;    // erreur ! valeur non autorisée
```

P2-C3/tests.ts

Retours de fonctions

- Le type de retour d'une fonction peut être inféré ou défini. Lorsqu'une fonction ne retourne rien, le type inféré par TypeScript est void

```
// RETOUR DE FONCTION
function saluer():string{ // erreur : retour chaine attendue
    let salutation:string="Bonjour";
    console.log(salutation);
}

saluer();
```

P2-C3/tests.ts

Type Function

- On peut utiliser le type **Function**

```
// TYPE FUNCTION
function salutationSimple(){
    console.log("Bonjour");
}
let salutation:Function;
salutation=salutationSimple;
salutation="Ciao"; // erreur !
```

P2-C3/tests.ts

Signature de fonctions

- On peut préciser la signature attendue d'une fonction

```
// SIGNATURE DE FONCTION
function salutationSimple(prenom:string){
    console.log("Bonjour "+prenom);
}
function salutationAvancee(prenom:string,nom:string):string{
    return "Bonjour "+prenom+" "+nom;
}

let salutation:(prenom:string)=>void;      // signature attendue
salutation=salutationSimple;
salutation=salutationAvancee;                  // erreur !
```

P2-C3/tests.ts

Unknown

- Le type `unknown` est à utiliser lorsque l'on ne connaît pas le type d'une variable à l'avance
- Il est différent de `any` dans le sens où il ne désactive pas la vérification de types et nous oblige à faire des transtypages ou des vérifications de types

```
// UNKNOWN
let variableDeTypeInconnu:unknown;
let n:number;
variableDeTypeInconnu=10;

// n = variableDeTypeInconnu;           // erreur ! car type non vérifié
n = Number(variableDeTypeInconnu);    // OK

if(typeof variableDeTypeInconnu==="number")
    n=variableDeTypeInconnu;          // OK, car type vérifié
```

P2-C3/tests.ts

Never

- Le type `never` est utilisé pour définir le fait qu'une fonction ne se terminera jamais. Décrit par exemple des fonctions destinées à lancer des exceptions, ou qui contiennent une boucle infinie :

```
//NEVER
function lancerErreur():never{
    throw new Error("Erreur");
}
// LancerErreur();

function boucleInfinie():never{
    while(true){
        console.log("test");
    }
}
// boucleInfinie();
```

P2-C3/tests.ts

Partie 2 - Chapitre 4

Introduction à la configuration du compilateur

Fichier de configuration

Merci de créer un répertoire `/P2-C4` puis de pointer dessus avec un terminal CMD

- Nous avons vu que la commande `tsc --init` génère une fichier de config `tsconfig.json`. On peut le modifier pour configurer le compilateur TypeScript

```
cd P2-C4  
tsc --init
```

CMD, dans P2-C4

Répertoires racine / destination

- Par défaut, le compilateur transpile les fichiers `.ts` en `.js` dans le même répertoire
- Préciser le répertoire des `.ts` avec `rootDir`, et des `.js` avec `outDir`
- Lancer `tsc` : la structure du dossier source sera conservée dans le dossier de sortie

```
// [...]
"rootDir": "./src",
// [...]
"outDir": "./dist",
// [...]
```

P2-C4/tsconfig.json

Inclusions / exclusions

- On peut ajouter "exclude" pour exclure des fichiers / répertoires de la compilation
- On peut ajouter "include" pour inclure des fichiers / répertoires de la compilation

```
// [...]
// inclu les .ts de /src sauf src/main.ts
"exclude": ["src/main.ts"],
"include": ["src/*.ts"]
}
```

P2-C4/tsconfig.json

```
// inclu les .ts de /src ET de /assets, sauf src/main.ts
"exclude": ["src/main.ts"],
"include": ["src/*.ts", "assets/*.ts"]
}
```

P2-C4/tsconfig.json

```
// inclu les .ts de chaque dossier, sauf src/main.ts
"exclude": ["src/main.ts"],
"include": ["**/*.ts"]
}
```

P2-C4/tsconfig.json

Version de JS, et noEmitOnErrors

- Le code TypeScript est traduit en JavaScript
- Vous avez la possibilité de choisir la version du standard EcmaScript utilisée !
- Pour cela il faut modifier la propriété target :

```
// [...]
"target": "es2016",
// [...]
```

P2-C4/tsconfig.json

- Les valeurs possibles sont : es3 / es5 / es6 (es2015) / es2016 / es2017 / es2018 / es2019 / es2020 / es2021 / es2022 / esnext (dernière version supportée par la version utilisée de TypeScript)
- L'instruction **noEmitOnErrors** permet d'accepter ou d'éviter de compiler lorsque le fichier TS comporte des erreurs

Partie 2 - Chapitre 5

Programmation orientée objet en TypeScript

Visibilité d'attributs

- Chaque attribut ou méthode d'une classe est créé avec une visibilité.
 - **public** : l'attribut / la méthode est accessible en dehors de la classe
 - **private** : seul un accès interne est autorisé (elle est “encapsulée”)
 - **protected** : l'attribut / la méthode n'est pas accessible directement en dehors de la classe, sauf pour les classes filles

```
class Chien{  
    private id:number = 1;  
    public nom:string = "Rex";  
    public race:string = "Labrador";  
}  
  
let monChien = new Chien();  
console.log(monChien.nom);  
console.log(monChien.id); // erreur ! (id privé)
```

tsc --init puis P2-C5/main.ts

Constructeur ES6

- On peut ajouter un constructeur pour personnaliser notre instance :

```
class Chien{  
    private id:number;  
    public nom:string;  
    public race:string;  
  
    constructor(id:number,nom:string,race:string){  
        this.id=id;  
        this.nom=nom;  
        this.race=race;  
    }  
  
}  
  
let monChien = new Chien(1,"Rex","Labrador");  
console.log(monChien.nom,monChien.race);
```

P2-C5/main.ts

Constructeurs rapides TS

- Pour gagner du temps lors de l'écriture d'une classe, TypeScript nous propose un **constructeur rapide** qui crée et initialise les champs de nos instances automatiquement
- Ses paramètres précisent la visibilité et le type de chacun d'entre eux :

```
class Chien {  
    constructor(private id: number, public nom: string, public race: string){  
    }  
}  
let monChien = new Chien(1, "Rex", "Labrador");  
console.log(monChien.nom, monChien.race);
```

P2-C5/main.ts

Lecture seule - readonly

- Un champ d'une classe peut être en `readonly` afin qu'il ne soit pas modifiable depuis le programme principal
- Comme il n'est pas possible d'avoir un attribut de classe déclaré comme une constante (on utilise ni `var`, ni `let`, ni `const`), déclarer cet attribut en `readonly` est une bonne alternative
- Un champ `readonly` doit être affecté `via un constructeur`. Il peut l'être à plusieurs reprises, mais uniquement dans le constructeur.

```
class Chien {  
    constructor(  
        private id: number,  
        public nom: string,  
        public race: string,  
        public readonly espece:string) {}  
}  
let monChien = new Chien(1, "Rex", "Labrador","mammifère");  
console.log(monChien.nom, monChien.race, monChien.espece);  
// monChien.espece="reptile"; // erreur car readonly
```

Héritage

- TypeScript apporte peu de spécificités au niveau de l'héritage
- Une classe fille sans constructeur propre hérite de son constructeur parent :

```
class Chien {  
    constructor(  
        private id: number,  
        public nom: string,  
        public race: string,  
        public readonly espece:string) {}  
}  
class Labrador extends Chien{}  
let monChien = new Labrador(1, "Rex", "Labrador","mammifère");  
console.log(monChien.nom, monChien.race, monChien.espece);
```

P2-C5/main.ts

Héritage

- Si une classe fille définit son constructeur, elle doit appeler son constructeur parent avec `super()`. Penser alors à modifier `id` en `protected` :

```
class Chien {  
    constructor(  
        protected id: number,  
        public nom: string,  
        public race: string,  
        public readonly espece:string) {}  
}  
class Labrador extends Chien{  
    constructor(protected id:number, public nom:string){  
        super(id,nom,"labrador","mammifère");  
    }  
}  
let monChien = new Labrador(1, "Rex");  
console.log(monChien.nom, monChien.race, monChien.espece);
```

P2-C5/main.ts

Getters & setters

- On peut bien entendu ajouter des getters et setters « traditionnels » à nos classes :

```
class Chien {  
    constructor(  
        protected id: number,  
        public nom: string,  
        public race: string,  
        public readonly espece:string) {}  
    public getID():number{ return this.id; }  
    public setID(newId:number){ this.id=newId; }  
}  
class Labrador extends Chien{  
    constructor(protected id:number, public nom:string){  
        super(id,nom,"labrador","mammifère");  
    }  
}  
let monChien = new Labrador(1, "Rex");  
console.log(monChien.getID());
```

Getters & setters TS

- Typescript propose une syntaxe particulière pour les getters et setters. Elle n'est utilisable que si la propriété est préfixée de `_` et permet de manipuler une propriété privée ou protégée comme si elle était publique !

```
class Chien {  
    constructor(  
        protected _id: number,  
        public nom: string,  
        public race: string,  
        public readonly espece:string) {}  
    public get id():number{ return this._id; }  
    public set id(newId:number){ this._id=newId; }  
}  
class Labrador extends Chien{  
    constructor(protected _id:number, public nom:string){  
        super(_id,nom,"labrador","mammifère");  
    }  
}  
let monChien = new Labrador(1, "Rex");  
monChien.id=5;          // utilisation du setter  
console.log(monChien.id); // utilisation du getter
```

Attributs et méthodes statiques

- Comme en POO classique (existe aussi en ES6), il est possible de définir des attributs et méthodes **statiques**. Cela signifie que ces attributs ou méthodes ne sont pas directement rattachés à l'instance mais à la classe

```
class Chien {  
    static decrire(){  
        return "Un chien, meilleur ami de l'Homme.";  
    }  
    // [...]  
}  
// [...]  
console.log(Chien.decrire());
```

P2-C5/main.ts

Classes abstraites

- Une méthode d'une classe peut être abstraite, c'est à dire non implémentée
- Toute classe possédant une ou plusieurs méthodes abstraites est elle-même abstraite et ne peut être instanciée

```
abstract class Animal{
    constructor(){}
    abstract crier():void;
}

class Chien extends Animal{
    crier(): void {
        console.log("WAF WAF !");
    }
    // [...]
}
let monChien = new Labrador(1, "Rex");
monChien.crier();
new Animal() // erreur ! abstraite
```

P2-C5/main.ts

Interfaces

- TypeScript utilise les interfaces qui n'existent pas en JS, mais bien en POO
- L'interface est assez proche de la classe abstraite, mais **aucune méthode n'est implémentée dedans**. Il y a juste la déclaration de propriétés ou de méthodes qui doivent être publiques dans la classe qui les implémente
- Une classe peut **implémenter** une ou plusieurs interfaces (et non en hériter)
- Il est possible de définir une propriété `readonly` dans une interface. Une interface peut étendre une autre interface. On peut utiliser `?` Pour définir un champ optionnel

```
interface Deplacable{
    vitesse?:number; // champ optionnel
    déplacer():void;
}

class Chien extends Animal implements Deplacable {
    public déplacer(): void {
        console.log("Ce chien court.");
    }
    // [...]
```

P2-C5/main.ts

Classe abstraite ou interface ?

- Une **classe abstraite** peut contenir des **méthodes implémentées**, d'autres non
Une **interface** ne peut contenir que des **méthodes non implémentées**
- Une **classe abstraite** peut contenir des **méthodes/attributs publics/privés/protégés**
Un **interface** ne peut contenir que des **méthodes/attributs publics**
- Une classe peut hériter d'**une seule classe abstraite**, mais peut implémenter **une ou plusieurs interfaces**
- Une **classe abstraite** permet de **factoriser du code**. Elle permet de décrire une entité dont le comportement **est en partie connu, et en partie inconnu**. Elle sert de **base** à d'autres classes
- Un **interface** est en quelques sorte une **classe totalement abstraite** qui représente une sorte de contrat. Elle décrit souvent un comportement, qui **peut s'appliquer à des objets de types très différents**. Par exemple, un animal et une voiture peuvent se déplacer et donc implémenter la même interface

Pattern singleton

- Pour implémenter le pattern singleton (une classe qui ne peut être instanciée qu'une seule fois), on peut utiliser l'astuce suivante :

```
P2-C5/main.ts  
class Timeline{  
    public temps = 0;  
    private static singleton:Timeline;  
    private constructor(){}
    static getInstance():Timeline{
        if(!this.singleton){
            this.singleton=new Timeline();
        }
        return this.singleton;
    }
}  
  
let laTimeLine = Timeline.getInstance(); // récupération du singleton  
// let laTimeLine = new Timeline(); // erreur car constructor privé !  
console.log(laTimeLine.temps);
```

Decorators

- Les **décorateurs** sont des **fonctions** que l'on peut utiliser sur des **classes**, **propriétés**, **méthodes**, **accesseurs**, et **paramètres**. Il permettent de prendre en compte, réagir, et éventuellement modifier les classes ou éléments de classes auxquels ils sont appliqués
- Les décorateurs sont **expérimentaux**. On peut les utiliser en compilant ainsi :

```
tsc --experimentalDecorators
```

CMD

- On peut aussi ajouter une ligne au fichier **tsconfig.json** :

```
"compilerOptions": {  
    "experimentalDecorators": true,  
    // [...]
```

P2-C5/tsconfig.json

Decorators

- Les décorateurs utilisent la forme `@expression` ou `@expression()` (**decorator factory**). `expression` fait référence à une fonction qui prend des paramètres différents suivant l'élément sur lequel le décorateur est appliqué

```
function Console(target:any){
    console.log(target);
}

@Console
class Chat{
    public nom:string;
    constructor(nom:string){
        this.nom=nom;
    }
}

let monChat = new Chat("Félix");
```

P2-C5/decorators.json

- Référence : <https://www.typescriptlang.org/docs/handbook/decorators.html>

Decorator factories & property decorators

- Les **decorator factories** permettent de passer des paramètres. Ils prennent des parenthèses. On peut les utiliser par exemple sur des **propriétés** :

```
function PropertyDecorator(){
    return function(target: any, propertyKey:string){ // class parente / nom de propriété
        let valeur = target[propertyKey];
        const getter = () => {
            return valeur;
        };
        // permet de définir la modification à appliquer à chaque nouvelle valeur
        const setter = (next:any) => {
            valeur = `🐱 ${next} 🐱`; // ici on entour d'Emoji le nom du chat
        };
        // permet de redéfinir la propriété en utilisant les getters / setters ci-dessus
        Object.defineProperty(target, propertyKey,
            {get: getter, set: setter, enumerable: true, configurable: true});
    }
}
class Chat{
    @PropertyDecorator()
    public nom:string;
    constructor(nom:string){
        this.nom=nom;
    }
}
let monChat = new Chat("Félix"); console.log(monChat.nom);
```

Method decorators

- On parle de **method decorators** lorsqu'un décorateur est appliqué à une méthode. Il y a alors 3 paramètres récupérables. Le **PropertyDescriptor**, qui décrit la méthode, peut être utilisé pour la redéfinir :

```
function MethodDecorator(){
    return function(target: any, propertyKey: string, descriptor: PropertyDescriptor){
        descriptor.value = function(){
            console.log(`Le chat fait : miaou !`);
        };
    }
}

class Chat{
    @PropertyDecorator()
    public nom:string;
    constructor(nom:string){
        this.nom=nom;
    }
    @MethodDecorator()
    public miauler(){
        console.log("Miaou !");
    }
}

let monChat = new Chat("Félix");
monChat.miauler();
```

Partie 2 - Chapitre 6

TypeScript avancé

Intersection de types

- L'intersection de types permet de définir un type combiné. On peut aussi définir un type combiné avec des interfaces

```
type Admin = {privileges:string[]}
type Employe = {nom:string,salaire:number}
type EmployeAdmin = Admin & Employe;

// OU
// interface Admin {privileges:string[]}
// interface Employe {nom:string;salaire:number}
// type EmployeAdmin = Admin & Employe;

let employe1:EmployeAdmin = {
    nom:"Alex",
    salaire:2000,
    privileges:["section serveur","section reservee"]
};
```

P2-C6/main.ts

Type guard

- Le **type guard** est un concept et non un type en soi
- C'est le principe de tester le type d'une variable dans une condition, qui permet de lever d'éventuelles erreurs
- Reprenons l'exemple cité pour unknown, qui comprend un type guard :

```
// UNKNOWN
let variableDeTypeInconnu:unknown;
let n:number;
variableDeTypeInconnu=10;

// n = variableDeTypeInconnu;           // erreur ! car type non vérifié
n = Number(variableDeTypeInconnu);    // OK

if(typeof variableDeTypeInconnu==="number")
  n=variableDeTypeInconnu;            // OK, car type vérifié
```

P2-C6/main.ts

Type guard appliqué aux objets

- Les tests d'existence de propriétés d'objets sont spécifiques en TypeScript :

```
let obj = {uneProp:"uneValeur"};
// if(obj.uneProp){ // si uneProp = "", console non exécutée
//   console.log(obj.uneProp);
// }
if("uneProp" in obj){ // console du moment que uneProp existe
  console.log(obj.uneProp);
}
```

P2-C6/main.ts

L'union discriminante

- Si un type à un membre **littéral**, ce membre peut être utilisé dans un type guard

P2-C6/main.ts

```
// UNION DISCRIMINANTE
type Carre = {
    type:"Carre"; // membre littéral (valeur imposée)
    taille:number;
}
type Rectangle = {
    type:"Rectangle"; // membre littéral (valeur imposée)
    largeur:number;
    longueur:number;
}
type Forme = Carre | Rectangle; // UNION
function calculerAire(f:Forme){
    if(f.type==="Carre") return f.taille*f.taille; // il s'agit d'un Carré
    else return f.largeur*f.longueur; // il s'agit forcément d'un Rectangle
}
let f1:Carre={type:"Carre",taille:5};
let f2:Rectangle={type:"Rectangle",largeur:5,longueur:10};
console.log(calculerAire(f1),calculerAire(f2)) // 25 50
```

Assertion de type

- On peut assertir un type en utilisant `as` ou bien la notation `<>`
- L'assertion de type permet d'indiquer au compilateur que vous êtes sûr qu'une variable est d'un type particulier à ce stade de votre programme
- Cela peut aider le compilateur à détecter les erreurs de type et à fournir une assistance au développement.
- Cependant, si la variable n'est pas du type précisé à l'exécution, cela entraînera une erreur

P2-C6/main.ts

```
let x:unknown = "hello";
// console.log(x.length);           // ERREUR DU COMPILATEUR
console.log((x as string).length); // x précisé comme chaîne, donc on peut
utiliser .length
console.log(<string>x.length);    // idem

let y:unknown = 4;
console.log(String(y).length);     // ce cast permet d'accéder à .length
console.log(<string>y.length);    // cette assertion est erronée :
autorisée par le compilateur mais produit un bug car pas de .length
```

Surcharge

- Le polymorphisme en TypeScript est géré en définissant pour chaque fonction ses signatures possibles

P2-C6/main.ts

```
// SURCHARGE
function dernierElement(liste:number[]):number;
function dernierElement(liste:string[]):string;
function dernierElement(liste:string[]|number[]){
    return liste[liste.length-1];
}
console.log(dernierElement(["a","b","c"]));      // c
console.log(dernierElement([1,2,3]));           // 3
```

Optional chaining

- L'**optional chaining** est arrivé en ES2020 et permet d'accéder de manière “sécurisée” aux attributs d'une propriété si l'on a un doute sur le fait que la variable cible soit **undefined**. Si une partie d'expression est **undefined**, toute l'expression sera **undefined** mais ne provoquera **pas d'erreur**
- Dans l'exemple ci-dessous, si personne devient **undefined**, la récupération du nom V1 provoque une erreur, la V2 est verbeuse, et la V3 est plus courte

P2-C6/main.ts

```
// OPTIONAL CHAINING
let personne:any = {nom:"toto"};

personne=undefined;

// let nom = personne.nom;           // V1 : erreur !
// let nom=(personne==null || personne==undefined)?undefined:personne.nom;
//                                         // V2 : verbeux / undefined

let nom = personne?.nom;            // V3 : optional chaining / undefined

console.log(nom);
```

Null coalescing

- Le **null coalescing** (“opérateur de coalescence des null”), qui existe depuis ES2020, noté `variable??valeur` permet de retourner une variable si elle n'est ni `null` ni `undefined`, ou bien de retourner la valeur associée
- Il s'agit presque du raccourci d'écriture correspondant à `variable?variable:valeur` à la différence près que **null coalescing** ne retourne la valeur que pour une variable à `null` ou `undefined`, toute autre valeur étant considérée comme valide

P2-C6/main.ts

```
// NULL COALESCING
let age:any = undefined;
// age=age?age:-1; // -1 si age = undefined, null, false, 0, '', NaN
age = age??-1;    // -1 si age = null, undefined
console.log(age);
```

Généricité

- Le **typage générique** permet d'écrire des définitions (de classes, interfaces, fonctions, types, etc.) paramétriques
- Il permet par exemple de s'assurer que des paramètres sont de même types, même si celui-ci n'est pas clairement défini

P2-C6/main.ts

```
// GENERICITE
function fonctionGenerique<T>(arg1:T,arg2:T){
    console.log(arg1,arg2);
}
fonctionGenerique<string>("chaine1","chaine2"); // OK
fonctionGenerique("chaine1","chaine2");           // OK
fonctionGenerique<number>(1,2);                  // OK
fonctionGenerique(1,2);                          // OK
// fonctionGenerique(1,"chaine2");                // erreur ! types différents
```

Généricité multiple

- On peut aussi préciser de multiples types génériques :

```
function genererTableau<T,U>(arg1:T,arg2:U):[T,U]{  
    return [arg1,arg2];  
}  
console.log(genererTableau<string,number>("ABC",1));  
console.log(genererTableau<string,number>("ABC",1));
```

P2-C6/main.ts

Utilisation de keyof dans les génériques

- Dans l'exemple ci-dessous, que l'on passe une valeur **chaine** ou **tableau**, on peut récupérer la lettre ou la case d'une position donnée

```
P2-C6/main.ts
function recuperer<T>(valeur:T,position:number){
    if((typeof valeur==="string" || Array.isArray(valeur)) && position< valeur.length){
        return valeur[position as keyof T]; // keyof T => union des clés 0/1/2/...
    }
    return false;
}
console.log(recuperer("Bonjour",3));    // j
console.log(recuperer([0,1,2,3,4],3));  // 3
```

- keyof**, appliqué à un type d'objet, permet de retourner une **union de ses clés**. La partie **as keyof T** est nécessaire en **mode strict** TS, pour restreindre les valeurs possibles de **position**

Contraintes génériques

- On peut éviter d'utiliser `as keyof`, en précisant avec `extends` les types valides acceptés par notre fonction :

P2-C6/main.ts

```
type TypesValides = string | Array<any>;
function recuperer<T extends TypesValides>(valeur:T,position:number){
    if(position < valeur.length)
        return valeur[position];
    else
        return false;
}
console.log(recuperer("Bonjour",3));    // j
console.log(recuperer([0,1,2,3,4],3)); // 3
console.log(recuperer(true,3));         // Erreur !
```

Fil rouge – étape 6

- Modifier l'application, pour qu'elle soit en TypeScript !

Bonus :
appels API avec Fetch
avec notre propre mini webservice

Mini serveur Express

- Cr ons un fichier `serveur.js` et mini-serveur Node. Pour package.json :

```
npm init
```

TERMINAL, dans P1-C5

- Puis indiquez la config suivante :

```
{  
  "name": "serveur",  
  "version": "1.0.0",  
  "description": "",  
  "main": "serveur.js",  
  "scripts": {  
    "start": "nodemon serveur.js"  
  },  
  "author": "",  
  "license": "ISC"  
}
```

TERMINAL, dans P1-C5

- Installons le serveur `express` et `nodemon` :

```
npm i express
```

```
npm i -g nodemon
```

TERMINAL, dans P1-C5

Mini serveur Express

- Dans `serveur.js` mettre le code suivant :

```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
    res.send('Hello World')
})

app.listen(3000)
```

P1-C5/serveur.js

- Procéder au démarrage du serveur :

```
npm run start
```

TERMINAL, dans P1-C5

- Dans le navigateur, `localhost:3000` devrait afficher : « Hello, World ». Un appel en GET depuis `fetch.js` aussi :

```
fetch('http://localhost:3000')
.then(reponse => reponse.text())
.then(json => console.log(json))
```

P1-C5/fetch.js

Mini serveur Express

- Dans `serveur.js` mettre le code suivant :

```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
    res.send('Hello World')
})

app.listen(3000)
```

P1-C5/serveur.js

- Procéder au démarrage du serveur :

```
npm run start
```

TERMINAL, dans P1-C5

- Dans le navigateur, `localhost:3000` devrait afficher : « Hello, World ». Un appel en GET depuis `fetch.js` aussi :

```
fetch('http://localhost:3000')
.then(reponse => reponse.text())
.then(json => console.log(json))
```

P1-C5/fetch.js

Mini serveur Express

- Résolvez temporairement les pb de cross-origin (**seulement en dev.**), puis ajoutez un tableau de salutations ainsi :

```
const express = require('express')
const app = express()

// temporaire : résout les pb de cross-origin requests pour nos tests
app.use((req, res, next) => {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Methods", "GET, POST, OPTIONS");
    res.header("Access-Control-Allow-Headers", "Content-Type");
    if (req.method === "OPTIONS") return res.sendStatus(204);
    next();
});

let salutations = [
    {id:1,langue:"FR",salutation:"Bonjour"}, 
    {id:2,langue:"IT",salutation:"Ciao"}, 
    {id:3,langue:"EN",salutation:"Hello"}, 
];

app.get('/', function (req, res) {
    res.send('Hello World')
})
app.listen(3000)
```

P1-C5/serveur.js

Routes GET

- Dans `serveur.js`, entre les `require` et le `app.listen(3000)`, définir les routes GET :

```
// GET : LECTURE DES SALUTATION
app.get(`/api/salutations`, (req, res) => {
    res.json(salutations)
})

app.get(`/api/salutations/:id`, (req, res) => {
    const id = Number(req.params.id);
    const salutation = salutations.find(salutation => salutation.id === id);
    if (!salutation) return res.status(404).send('Non trouvée.')
    res.json(salutation)
})

app.get(`/api/salutations/langue/:langue`, (req, res) => {
    const langue = req.params.langue;
    const salutation = salutations.find(salutation => salutation.langue ===
langue);
    if (!salutation) return res.status(404).send('Non trouvée.')
    res.json(salutation)
})
```

P1-C5/fetch.js

Appels en GET

- Dans `fetch.js` réaliser les appels suivants :

```
// API INTERNE

// APPEL GET, POUR LIRE UNE LISTE DE RESSOURCES
fetch('http://localhost:3000/api/salutations')
  .then(reponse => reponse.json())
  .then(json => console.log(json));

// APPEL GET, POUR LIRE LA RESSOURCE D'ID 1
fetch('http://localhost:3000/api/salutations/1')
  .then(reponse => reponse.json())
  .then(json => console.log(json));

// APPEL GET, POUR LIRE LA RESSOURCE DE LANGUE IT
fetch(`http://localhost:3000/api/salutations/langue/${encodeURI("IT")}`)
  .then(reponse => reponse.json())
  .then(json => console.log(json));
```

P1-C5/fetch.js

Routes POST

- Dans `serveur.js`, ajouter la route POST :

```
// à ajouter en haut, pour parser tout contenu JSON de body
app.use(express.json())

// [...]

// POST : AJOUT D'UNE SALUTATION
app.post('/api/salutations', (req, res) => {
  const nouvelleSalutation = {
    id: salutations.length+1,
    langue: req.body.langue,
    salutation: req.body.salutation,
  }
  salutations.push(nouvelleSalutation)
  res.status(201).json(nouvelleSalutation)
})
```

P1-C5/fetch.js

Appels en POST

- Et effectuer l'appel suivant en POST dans `fetch.js` pour ajouter une langue :

```
// APPEL EN POST, POUR CREER UNE RESSOURCE DE LANGUE RU
let data = { langue: 'RU', salutation: 'Priwét' };

fetch(`http://localhost:3000/api/salutations`,
{
  method: 'POST',
  body: JSON.stringify(data),
  headers: {'Content-type': 'application/json; charset=UTF-8'}
})
.then(reponse => reponse.json())
.then(json => console.log(json))
// liste des ressources mises à jour
.then(()=>{
  fetch('http://localhost:3000/api/salutations')
    .then(reponse => reponse.json())
    .then(json => console.log(json))
})
.catch(erreur => { console.error(erreur); });
```

Routes PUT

- Dans `serveur.js`, ajouter la route PUT :

```
// PUT : MODIFICATION D'UNE SALUTATION
app.put('/api/salutations/:id', (req, res) => {
  const id = Number(req.params.id);
  const index = salutations.findIndex(salutation => salutation.id === id)
  if (index === -1) return res.status(404).send('Inconnu');
  const salutationMiseAJour = {
    id: salutations[index].id,
    langue: salutations[index].langue,
    salutation: req.body.salutation
  }
  salutations[index] = salutationMiseAJour
  res.status(200).json('Mise à jour OK');
})
```

P1-C5/fetch.js

Appels en PUT

- Et effectuer l'appel suivant en PUT dans `fetch.js` pour modifier une langue :

```
// APPEL EN PUT, POUR MODIFIER UNE RESSOURCE D'ID 2
let data = { salutation: 'Hi' };
fetch(`http://localhost:3000/api/salutations/2`,
{
  method: 'PUT',
  body: JSON.stringify(data),
  headers: {'Content-type': 'application/json; charset=UTF-8'}
})
.then(reponse => reponse.json())
.then(json => console.log(json))
// liste des ressources mises à jour
.then(()=>{
  fetch('http://localhost:3000/api/salutations')
    .then(reponse => reponse.json())
    .then(json => console.log(json))
})
.catch(erreur => { console.error(erreur); });
```

Routes DELETE

- Dans `serveur.js`, ajouter la route **DELETE** :

```
// DELETE : SUPPRESSION D'UNE SALUTATION PAR ID
app.delete('/api/salutations/:id', (req, res) => {
  const id = Number(req.params.id)
  const index = salutations.findIndex(salutation => salutation.id === id)
  if (index === -1) return res.status(404).send('Inconnu');
  salutations.splice(index, 1)
  res.status(200).json('Suppression OK')
})
```

P1-C5/fetch.js

Appels en DELETE

- Effectuer l'appel suivant en DELETE dans `fetch.js` pour supprimer une langue :

```
// APPEL EN DELETE, POUR SUPPRIMER UNE RESSOURCE D'ID 2
fetch(`http://localhost:3000/api/salutations/2`,
{
    method:"DELETE",
    headers: {'Content-type': 'application/json; charset=UTF-8'}
})
.then(reponse => reponse.json())
.then(json => console.log(json))
// liste des ressources mises à jour
.then(()=>{
    fetch('http://localhost:3000/api/salutations')
        .then(reponse => reponse.json())
        .then(json => console.log(json))
})
.catch(erreur => { console.error(erreur); });
```

P1-C5/fetch.js

Appels avec async / await

- On pourrait réaliser le même DELETE avec une autre syntaxe :

```
// APPEL EN DELETE, POUR SUPPRIMER UNE RESSOURCE D'ID 2 V2
(async function(){
    try{

        // suppression
        let reponseDEL = await fetch(
            `http://localhost:3000/api/salutations/2`, {method:"DELETE", headers:
{'Content-type': 'application/json; charset=UTF-8'}});
        let JSONreponseDEL = await reponseDEL.json();
        console.log(JSONreponseDEL);

        // affichage des salutations mises à jour
        let reponseGET = await fetch('http://localhost:3000/api/salutations')
        let JSONreponseGET = await reponseGET.json();
        console.log(JSONreponseGET);

    }catch(e){
        console.log("ERREUR "+e);
    }
})();
```

Webliographie

- M. Nicolas Amini-Lamy
- <https://www.w3schools.com/>
- <https://developer.mozilla.org/fr/>
- <https://www.themoviedb.org/>
- <https://www.pierre-giraud.com/javascript-apprendre-coder-cours/>
- <https://grafikart.fr/tutoriels>
- <https://buzut.net/programmation-fonctionnelle-en-javascript/>
- <https://www.programiz.com/javascript>
- Chaines youtube :
 - @WebDevSimplified
 - @grafikart
 - @ColtSteeleCode
 - @DevTheory
 - @LiorCHAMLA
 - @codeursenior