



Le langage TypeScript

Mohamed ROMDANE

Septembre 2021

formation@ineotec.com

Présentation du cours

- Présentation
- Conseils pour suivre la formation
- Le parcours de formation

Formateur



- Mohamed Romdane est expert spécialisé en architecture Big Data. Au sein de Ineotec, il offre des solutions d'audit de base de données, code review, analyse de logs et développement de logiciels dans le monde open source et Full-Stack.
- Mohamed Romdane est diplômé en électronique de l'**Université de Paris XI** avec un DEA sur le traitement de l'information entre l'*Institut d'Électronique Fondamentale* et l'*Institut National des Sciences et Techniques Nucléaires*, puis en thèse de doctorat au *Laboratoire de Recherche en Informatique* sur les **bases de données déductives**.
- Il travaillera à l'*Institut Universitaire Technique - IUT Sceaux* et au *CNAM Saclay* en tant qu'enseignant en **Bases de données** avant de s'attaquer au milieu professionnel avec une expérience avec plusieurs entreprises, institutions et groupes à une échelle internationale: *Talan France, DGI Maroc, Gide, SMAI* en France et en Nouvelle-Calédonie, au *Ministère des Finances* en Polynésie française, *France Telecom, Sagep, Bouygues, Star et Sofrecom*. Les compétences et l'expertise de Mohamed Romdane au sein de Ineotec vont de la **Business Intelligence - Enterprise Service and Bus – Extract Transfer and Loading** aux **bases de données**, en passant par **JAVA et JEE**, les **solutions mobiles**, le **php**, plusieurs **langages de programmation**, **produits client/serveur** et **systèmes d'exploitation**.

Logiciels à installer

- Installation Visual Studio Code
 - <https://code.visualstudio.com/>
 - Eslint
 - Material Icon
 - Path Intellisense
- Installation node.js
 - <https://nodejs.org>
- Installation Typescript
 - <http://www.typescriptlang.org>
 - `npm install –global typescript` ou `npm i –g typescript`

Introduction à Typescript

- Qu'est ce que Typescript
- C'est un langage de programmation qui ajoute de nouvelles fonctionnalités à JavaScript comme la gestion des types et prise en charge de la Programmation Orientée Objet.
- Le Typescript n'est pas pris en charge par les navigateurs.
- Il doit subir des conversions vers JavaScript avant de pouvoir être exécuté par le navigateur
- Le Typescript permet l'écriture d'un code plus robuste car il est fortement typé et simple d'utilisation car il utilise la POO.
- Les problématiques du JavaScript
- Installation de TypeScript
- Premier programme TypeScript

Exemple de code

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>Additionner deux nombres :</h1>
    <div>
        <label for="nb1">Nombre 1 :</label>
        <input type="number" name="nb1" id="nb1">
    </div>
    <div>
        <label for="nb2">Nombre 2 :</label>
        <input type="number" name="nb2" id="nb2">
    </div>
    <input type="button" value="Calculer" id="calcul">
    <div class="resultat"></div>
    <script src="main.js"></script>
</body>
</html>
```

main.js

```
const nb1 = document.querySelector("#nb1");
const nb2 = document.querySelector("#nb2");

document.querySelector("#calcul").addEventListener("click", function(){
    let resultat = addition(nb1.value, nb2.value);
    document.querySelector(".resultat").innerHTML=resultat;
})

function addition(n1,n2){
    return n1 + n2;
}
```

Navigateur

Additionner deux nombres :

Nombre 1 :

Nombre 2 :

Compilateur Typescript

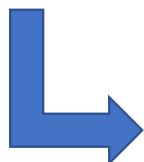
tsc: The TypeScript Compiler - Version 4.x.x

main.ts

```
const nb1 = document.querySelector("#nb1") as HTMLInputElement;
const nb2 = document.querySelector("#nb2") as HTMLInputElement;
const divResultat = document.querySelector(".resultat") as HTMLDivElement;

document.querySelector("#calcul").addEventListener("click", () => {
    let resultat = addition(+nb1.value, +nb2.value);
    divResultat.innerHTML = resultat.toString();
})

function addition(n1, n2){
    if( typeof(n1) === 'number' && typeof(n2) === 'number' ) {
        return n1 + n2;
    } else {
        return 'Les valeurs ne sont pas de type number !';
    }
}
```



tsc main.ts --watch

main.js

```
var nb1 = document.querySelector("#nb1");
var nb2 = document.querySelector("#nb2");
var divResultat = document.querySelector(".resultat");
document.querySelector("#calcul").addEventListener("click", function () {
    var resultat = addition(+nb1.value, +nb2.value);
    divResultat.innerHTML = resultat.toString();
});

function addition(n1, n2) {
    if (typeof (n1) === 'number' && typeof (n2) === 'number') {
        return n1 + n2;
    }
    else {
        return 'Les valeurs ne sont pas de type number !';
    }
}
```

Compilateur Typescript

```
const nb1 = document.querySelector("#nb1");
const nb2 = document.querySelector("#nb2");
const divResultat = document.querySelector(".resultat");
document.querySelector("#calcul").addEventListener("click", () => {
    let resultat = addition(+nb1.value, +nb2.value);
    divResultat.innerHTML = resultat.toString();
});
function addition(n1, n2) {
    if (typeof (n1) === 'number' && typeof (n2) === 'number') {
        return n1 + n2;
    }
    else {
        return 'Les valeurs ne sont pas de type number !';
    }
}
```

tsc main.ts -target es6 --watch

Les Types de Base avec TypeScript

```
let prenom: string = "Mohamed";
let age:number = 32;
let sexe:boolean = true;
const sports:string[] = ["foot","rugby"];
const adresse:{
    ligne:string;
    ville:string;
    cp:number;
} = {
    ligne : "rue des fleurs",
    ville : "Toulouse",
    cp : 31000
}
```

```
let afficherPersonne = (
    in_prenom:string,
    in_age:number,
    in_sexe:boolean,
    in_sports:string[],
    in_adresse:{
        ligne:string;
        ville:string;
        cp:number;
    }) => {
    console.log(`Prenom : ${in_prenom}`);
    console.log("Age : " + in_age);
    console.log("Sexe : " + (in_sexe ? "Homme" : "Femme"));
    for (let sport of in_sports){
        console.log(sport)
    }
    console.log("Adresse : ");
    console.log(in_adresse.ligne)
    console.log(in_adresse.cp + " " + in_adresse.ville);
}

afficherPersonne(prenom, age, sexe, sports, adresse);9
```

Typage implicite vs explicite

```
let prenom: string;    // Typage explicite
// ...
prenom = "Mohamed";

let prenom = "Mohamed" // Typage implicite
```

Les Fonctions et Type Function

```
function add(a: number, b: number): number {
    return a + b;
}

var add = function (a: number, b: number): number {
    return a + b;
}

let add = (a: number, b: number): number => {
    return a + b;
}

let ajouter : (a: number, b:number) => number;
ajouter = add;
let total = ajouter(5, 66)

let ajouter : Function;
ajouter = add;
let total = ajouter(5, 64);
```

Les Fonctions avec des paramètres par défaut

```
function add(a: number = 0, b: number = 0): number {
    return a + b;
}

var add = function (a: number = 0, b: number = 0): number {
    return a + b;
}

let add = (a: number = 0, b: number = 0): number => {
    return a + b;
}
```

Les Fonctions de Rappel (Callback)

```
function afficherAge(age:number) : void{
    console.log("L'age est de " + age);
}

function ajout(nb1:number, nb2:number, callback : (n : number) => void){
    let res = nb1 + nb2;
    callback(res);
}

ajout(10, 15, afficherAge);
```

Le Type "any"

- Le type `any` peut être utilisé pour obtenir une variable pouvant changer de type lors de l'exécution.
- Le type `any` est déconseillé avec Typescript
- Les paramètres non typés d'une fonction sont de type `any`.

```
const personne:any[] = ["Mohamed",31,true];

function afficherPersonne(perso:any[]){
    for(let valeur of perso){
        console.log(valeur);
    }
}
afficherPersonne(personne);
```

L'union de types

- Une variable peut être définie comme pouvant contenir des valeurs de type différent à l'aide du symbole « | »
- Une fonction peut prendre en paramètre des types différents et peut retourner des types différents.

```
var information : string | number | boolean;  
information = "Mohamed";  
console.log(information)  
information = 32;  
console.log(information)  
information = true;  
console.log(information)
```

```
function test (in_input : string | number | boolean){  
    if(typeof(in_input) === "string"){  
        console.log("Chaine de caractères");  
    } else if (typeof(in_input) === "number"){  
        console.log("Nombre");  
    } else if (typeof(in_input) === "boolean"){  
        console.log("Booléen");  
    }  
}
```

La surcharge de fonction

```
function ajout(e1:string,e2:number) : string;
function ajout(e1:number,e2:string) : string;
function ajout(e1:string, e2:string) : string;
function ajout(e1:number, e2:number) : number;
function ajout(e1 : number | string, e2: number | string) : number | string {
    if(typeof e1 === "number" && typeof e2 === "number"){
        return e1 + e2;
    }
    return e1.toString() + " " + e2.toString();
}

let calcul = ajout(5,15);
Math.floor(calcul);

let concat = ajout("Mohamed", "ROMDANE");
console.log(concat.toUpperCase());

let test = ajout(10,"Mohamed");
console.log(test);

let test2 = ajout("Mohamed",10);
console.log(test2);
```

Les Tableaux

```
const notes = [10, 15, 20]; // number[]
const prenoms = ["Mohamed", "Jean", "Li"]; // string[]
const personne:(string|number)[] = ["Mohamed", 31]; // (string | number)[]
personne[2] = "Test";

const perso:[string, number, boolean] = ["Mohamed", 31, true];
```

- Tableau de number
- Tableau de string
- Tableau de soit string soit number (union)
- Tableau fixe ou tuple
- Tableau de tableaux
- Tableau d'objets

```
const tab = [
  [10,12],
  [14,16],
  [18,20]
] // number[][]

const tab2: {
  x:number,
  y:number
}[] = [
  {x:1,y:2},
  {x:2,y:3},
  {x:1,y:3}
] // { x: number; y: number; }[]
```

Créer des Types Utilisateur

```
type Adresse = {  
    ligne : string;  
    ville : string;  
    cp : number;  
}  
  
const adresse:Adresse = {  
    ligne : "rue des fleurs",  
    ville : "Toulouse",  
    cp : 31000  
}
```

```
function afficherAdresse (in_adr:Adresse){  
    console.log(in_adr.ligne);  
    console.log(in_adr.cp + " " + in_adr.ville);  
}  
  
afficherAdresse(adresse);  
afficherAdresse({ligne:"rue des plantes",ville:"Paris",cp:75000})
```

```
type Point = {  
    x:number;  
    y:number;  
}  
  
const tab: Point[] = [  
    {x:1,y:2},  
    {x:2,y:3},  
    {x:1,y:3}  
]
```

Les interfaces

- Types Utilisateur et Interface sont identiques

```
interface Adresse {  
    ligne : string;  
    ville : string;  
    cp : number;  
}  
  
const adresse:Adresse = {  
    ligne : "rue des fleurs",  
    ville : "Toulouse",  
    cp : 31000  
}
```

```
interface Point {  
    x:number;  
    y:number;  
}  
  
const tab: Point[] = [  
    {x:1,y:2},  
    {x:2,y:3},  
    {x:1,y:3}  
]
```

L'union et les objets

```
interface Personnage {
    nom : string;
}

interface Humain extends Personnage{
    age : number;
}

interface Monstre extends Personnage{
    tribu : string;
}

type Perso = Humain | Monstre;
```

```
const p1: Humain = {
    nom : "Jean",
    age : 31
};

const p2 : Monstre = {
    nom : "Gael",
    tribu : "Orc vert"
}
```

```
function afficherPersonnage(perso : Perso){
    console.log("Nom : " + perso.nom);
    if("age" in perso){
        console.log("Age : " + perso.age);
    }
    if("tribu" in perso){
        console.log("Tribu : " + perso.tribu);
    }
}
```

L'intersection

```
interface Personnage {  
    nom : string;  
    age : number;  
}  
  
interface Humain {  
    classe : string;  
}  
  
interface Homme extends Personnage, Humain {  
    sports : string[],  
};
```

```
const test : Humain = {  
    classe : "voleur"  
}  
  
const matthieu:Homme ={  
    nom : "Matthieu",  
    age : 31,  
    classe : "Guerrier",  
    sports : ["foot",'rugby']  
}]
```

Le type énumération

```
enum CLASSE {GUERRIER = "Guerrier", VOLEUR = "Voleur", ARCHER = "Archer"};  
  
console.log(CLASSE.GUERRIER);
```

- Permet de définir des valeurs constantes dans une liste
- par défaut, les valeurs démarrent à 0, 1, 2
- Possibilité de spécifier des valeurs

```
interface Personnage {  
    nom : string;  
}  
  
interface Guerrier extends Personnage {  
    classe : CLASSE.GUERRIER;  
}  
interface Voleur extends Personnage {  
    classe : CLASSE.VOLEUR;  
}  
  
const p1:Personnage & Guerrier = {  
    nom : "Matthieu",  
    classe : CLASSE.GUERRIER  
}  
  
console.log(p1);
```

Type "unknown" et "never"

```
let quoi : any;
quoi = 32;
quoi = "Mohamed";
var age : number = quoi;
console.log(age);
```

incohérence

```
let quoi : unknown;
quoi = 32;
quoi = "Mohamed"
if (typeof quoi === "string") {
    var nom : string = quoi;
}
if (typeof quoi === "number") {
    var age : number = quoi;
}
```

- Type never peut être retourné par une fonction à la place de void

```
function leverexception(msg: string, codeErreur : number) : never {
    throw {message : msg, errorCode : codeErreur};
}

leverexception("La page n'existe pas", 404);
```

Autres fonctionnalités

- « ! » pour indiquer qu'un élément ne peut pas être null.

```
const inputNom = document.querySelector("#nom")!;
```

- Pour transtyper « as »

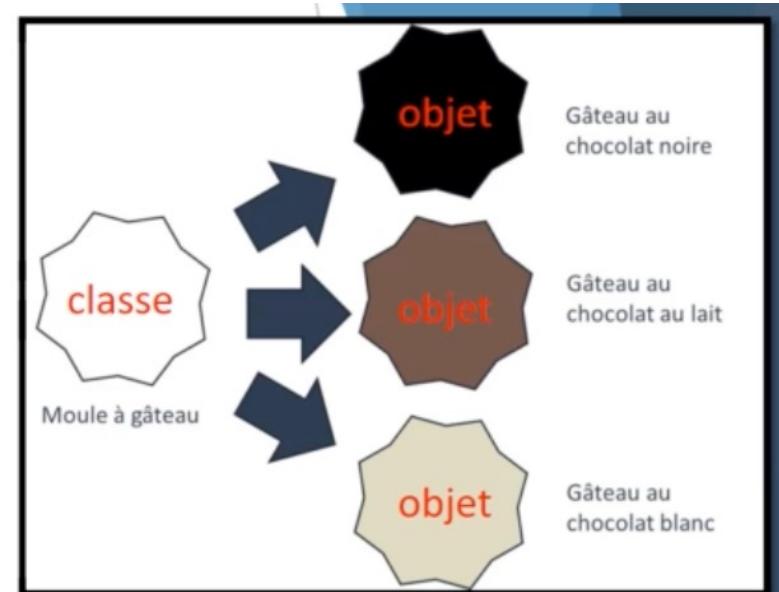
```
const titreh1 = document.querySelector("#titre1") as HTMLHeadingElement;
```

```
const titreh1 = <HTMLHeadingElement>document.querySelector("#titre1");
```

La Programmation Orientée Objet

Les Classes et Objets

- ▶ Rappel :
 - ▶ Un objet permet de représenter une entité propre (un livre, un homme, un animal) à l'aide d'informations :
 - ▶ Attribut : correspond aux données brutes (un nom, un âge, un libellé, un sexe, une liste de couleurs, une liste de passions...)
 - ▶ Méthode (fonction) : correspond aux fonctionnalités (traitements) que peut réaliser un objet
- ▶ JavaScript est un langage orienté Prototype. Les versions récentes du langage permettent (*plus ou moins*) d'écrire du code en utilisant la Programmation Orientée Objet (POO). De nombreux concepts sont manquant ou se mettent en œuvre de manière « particulière ».
- ▶ TypeScript permet une meilleur écriture de la POO et rajoute de nombreux éléments manquant à JavaScript, dont la gestion de la visibilité des informations (attributs et fonctions)



Le mot clef "this"

- Le mot clef « this » permet d'accéder à une information d'un objet à l'intérieur de lui-même

```
class Voiture{  
    marque:string;  
    modele:string;  
    nbPortes:number;  
  
    constructor(marque:string,modele:string,nbPortes:number){  
        this.marque=marque;  
        this.modele=modele;  
        this.nbPortes=nbPortes;  
    }  
  
    afficherVoiture(){  
        console.log("Marque :" + this.marque);  
        console.log("Modele :" + this.modele);  
        console.log("Nombre de portes :" + this.nbPortes);  
    }  
}
```

```
let v1 = new Voiture("Yotota","Riyas",3);  
v1.afficherVoiture();  
  
console.log("Marque :" + v1.marque);  
console.log("Modele :" + v1.modele);  
console.log("Nombre de portes :" + v1.nbPortes);
```

Marque :Yotota
Modele :Riyas
Nombre de portes :3

Marque :Yotota
Modele :Riyas
Nombre de portes :3

Le mot clef "this"

- ▶ Comme en JS « this » peut occasionner des incohérences, attention à son utilisation !

```
let affichages = {  
    afficherVoiture : v1.afficherVoiture,  
}  
affichages.afficherVoiture();
```



```
Marque :undefined  
Modele :undefined  
Nombre de portes :undefined
```

```
afficherVoiture(this:Voiture){  
    console.log("Marque :" + this.marque);  
    console.log("Modele :" + this.modele);  
    console.log("Nombre de portes :" + this.nbPortes);  
}
```

Le mot clef « this » dans la fonction « afficherVoiture » de l'objet « affichages » ne permet d'accéder aux informations « marque », « modele », « nbPortes ».

Pour que TypeScript indique que le code est invalide, on peut décrire « this » dans le paramètre de la fonction

```
let affichages = {  
    afficherVoiture : v1.afficherVoiture,  
}  
affichages.afficherVoiture();
```

Visibilité / accessibilité

- En POO, les informations (attributs et fonctions), disposent d'une visibilité, permettant de décrire leurs accessibilité.
 - « public » permet de rendre une information accessible de n'importe où
 - « private » permet de rendre une information inaccessible de l'extérieur d'un objet

```
class Voiture{
    marque:string;
    public modele:string;
    nbPortes:number;

    constructor(marque:string,modele:string,nbPortes:number){
        this.marque=marque;
        this.modele=modele;
        this.nbPortes=nbPortes;
    }

    afficherVoiture(this:Voiture,){
        console.log("Marque :" + this.marque);
        console.log("Modele :" + this.modele);
        console.log("Nombre de portes :" + this.nbPortes);
    }

    let v1 = new Voiture("Yotota","Riyas",3);
    v1.afficherVoiture();
    console.log("-----")
    console.log("Marque :" + v1.marque);
    console.log("Modele :" + v1.modele);
    console.log("Nombre de portes :" + v1.nbPortes);
}
```

Par défaut, si aucun mot clef n'est positionné devant une information, c'est la visibilité « public » qui est définie

```
class Voiture{
    private marque:string;
    private modele:string;
    private nbPortes:number;

    public constructor(marque:string,modele:string,nbPortes:number){
        this.marque=marque;
        this.modele=modele;
        this.nbPortes=nbPortes;
    }

    public afficherVoiture(this:Voiture,){
        console.log("Marque :" + this.marque);
        console.log("Modele :" + this.modele);
        console.log("Nombre de portes :" + this.nbPortes);
    }

    let v1 = new Voiture("Yotota","Riyas",3);
    v1.afficherVoiture();
    console.log("-----")
    console.log("Marque :" + v1.marque);
    console.log("Modele :" + v1.modele);
    console.log("Nombre de portes :" + v1.nbPortes);
}
```

Rappel : En règle général, les attributs sont définis en « private » et des fonctions « Getter » et « Setter » permettent de les gérer.

Getters et Setters

- Méthode POO classique :

```
/**  
 * GETTERS  
 */  
public getMarque():string { //Getter  
    return this.marque;  
}  
public getModele():string {return this.modele;}  
public getNbPortes():number {return this.nbPortes;}  
  
/**  
 * SETTERS  
 */  
public setMarque(marque:string) : void {  
    this.marque = marque;  
}  
public setModele(modele:string) : void {  
    this.modele = modele;  
}  
public setNbPortes(nbPortes:number) : void {  
    this.nbPortes = nbPortes;  
}
```

```
let v1 = new Voiture("Yotota","Riyas",3);  
v1.afficherVoiture();  
console.log("-----");  
v1.setMarque("GePo");  
v1.setModele("803")  
v1.setNbPortes(5)  
console.log("Marque : " + v1.getMarque());  
console.log("Modele : " + v1.getModele());  
console.log("Nombre de Portes : " + v1.getNbPortes());
```

```
Marque :Yotota  
Modele :Riyas  
Nombre de portes :3  
-----  
Marque : GePo  
Modele : 803  
Nombre de Portes : 5
```

```
public setMarque(marque:string) : Voiture {  
    this.marque = marque;  
    return this;  
}  
public setModele(modele:string) : Voiture {  
    this.modele = modele;  
    return this;  
}  
public setNbPortes(nbPortes:number) : Voiture {  
    this.nbPortes = nbPortes;  
    return this;  
}
```

Permet le chaînage
des fonctions !

```
v1.setMarque("GePo").setModele("803").setNbPortes(5);
```

Le mot clef "readonly"

- ▶ Il n'est pas possible de définir une constante de classe dans TypeScript
- ▶ Le mot clef « readonly » permet d'indiquer qu'un attribut **ne peut pas être modifié** après son initialisation.
- ▶ A la différence d'une constante, cette information peut-être définie par le constructeur et être spécifique à un objet !

« constante d'objet » identique

```
class Voiture{  
    public readonly annee:number;  
}
```

```
let v1 = new Voiture("Yotota","Riyas",3);  
let v2 = new Voiture("test","test",5);
```

```
console.log("Année de constructeur : " + v1.annee);  
console.log("Année de constructeur : " + v2.annee);
```

```
Année de constructeur : 2020  
Année de constructeur : 2020
```

« constante d'objet » spécifique

```
class Voiture{  
    public readonly annee:number;  
  
    public constructor(  
        protected _marque:string,  
        private _modele:string,  
        protected _nbPortes:number,  
        annee:number){  
            this.annee = annee;  
        }  
}
```

```
console.log("Année de constructeur : " + v1.annee);  
console.log("Année de constructeur : " + v2.annee);
```

```
Année de constructeur : 2021  
Année de constructeur : 2019
```

Static

- ▶ La mot clef « static » permet d'indiquer qu'une information (attribut ou fonction) est accessible directement depuis la classe elle-même (et non depuis l'objet)

- ▶ Cette fonctionnalité est intéressante pour définir :

- ▶ Des attributs communs aux objets (mettre en œuvre le pattern singleton, définir des « constantes de classe », ...)
- ▶ Créer des listes d'objets provenant d'une classe (tableaux)
- ▶ Des fonctions communes aux objets (de récupération de data de la BD par exemple, ou des fonctions utilitaires)
- ▶ Regrouper des fonctions utilitaires dans une classe (boîte à outil)

```
class Toolbox{
    public static afficherTab(tab : any[]){
        for(let element of tab){
            console.log(element);
        }
    }
    public static calculMoyenne(...notes:number[]){
        let moyenne = 0;
        for(let note of notes){
            moyenne+=note;
        }
        return moyenne/notes.length;
    }
}
```

```
public static ajouterVoitureListe(v : Voiture){
    Voiture.listeVoiture.push(v);
}
```

```
let v1 = new Voiture("Yotota","Riyas",3,2021);
let v2 = new Voiture("test","test",5,2019);
Voiture.ajouterVoitureListe(v1);
Voiture.ajouterVoitureListe(v2);
console.log(Voiture.listeVoiture);
```

```
let v1 = new Voiture("Yotota","Riyas",3,2021);
let v2 = new Voiture("test","test",5,2019);
Voiture.ajouterVoitureListe(v1);
Voiture.ajouterVoitureListe(v2);
Toolbox.afficherTab(Voiture.listeVoiture);
```

```
console.log("Moyenne : " + Toolbox.calculMoyenne(12,13,15,10,20));
```

```
Moyenne : 14
```

Héritage

- ▶ L'héritage permet de créer une classe héritant des propriétés d'une autre classe. On parle de classe mère et de classe fille. Une classe fille ne peut avoir qu'une seule mère mais plusieurs classes filles peuvent hériter de la classe mère.
- ▶ Exemple :

```
class Voiture{  
    public static listeVoiture:Voiture[] = [];  
    public static readonly TVA:number = 20;  
    public readonly annee:number;  
  
    public constructor(protected _marque:string,  
                      private _modele:string,  
                      protected _nbPortes:number,  
                      annee:number){  
        this._marque=_marque;  
        this._modele=_modele;  
        this._nbPortes=_nbPortes;  
        this.annee = annee;  
    }  
}
```

Les informations « protected » sont accessible par les classes filles, ce qui n'est pas le cas avec les informations « private »

```
class VoitureCourse extends Voiture{  
    public readonly couleur:string;  
  
    constructor(marque:string, modele:string, couleur:string){  
        super(marque,modele,3,2019);  
        this.couleur = couleur;  
    }  
  
    public afficherVoitureCourse(){  
        console.log("Marque : " + this._marque);  
        console.log("Modele : " + this._modele);  
    }  
    public afficherVoitureCourse2(){  
        console.log("Marque : " + this._marque);  
        console.log("Modele : " + this.modele);  
    }  
}
```

Le mot clef « super » permet de faire référence à la classe mère (ici d'appeler son constructeur)

L'information est en « private » dans la classe mère

Fait appel au Getter de modèle de la classe mère

Abstract

- Le mot clef « abstract » permet de définir une fonction dans une classe mère sans pour autant écrire son implémentation. Ainsi, les classes filles devront impérativement définir et implémenter la fonction abstraite
- Si une classe contient une fonction abstraite, alors elle doit elle-même être définie comme une classe abstraite. Ainsi, elle ne pourra plus être instanciée directement !

```
abstract class Voiture{
    public static listeVoiture:Voiture[] = [];
    public static readonly TVA:number = 20;
    public readonly annee:number;

    public constructor(protected _marque:string,
                      private _modele:string,
                      protected _nbPortes:number,
                      annee:number){
        this._marque=_marque;
        this._modele=_modele;
        this._nbPortes=_nbPortes;
        this.annee = annee;
    }

    abstract afficherVoiture(this:Voiture);
}
```

```
class VoitureCourse extends Voiture{
    public readonly couleur:string;

    constructor(marque:string, modele:string, couleur:string){
        super(marque,modele,3,2019);
        this.couleur = couleur;
    }

    public afficherVoiture(this:VoitureCourse){
        console.log("Marque : " + this.modele);
        console.log("Modele : " + this.modele);
        console.log("Nombre de portes : " + this.nbPortes);
        console.log("Couleur : " + this.couleur);
    }
}

let v1 = new Voiture("Yotota", "Riyas", 3, 2021);
let v2 = new Voiture("test", "test", 5, 2019);
let v3 = new VoitureCourse("Pors", "359", "bleue");
console.log(v1);
console.log(v2);
console.log(v3);
```

Il faut créer une nouvelle classe fille pour gérer les voitures « normales »

Interface

- ▶ Comme les classes abstraites, les interfaces permettent de définir une structure d'une classe et ne peuvent pas être instanciées. Une classe peut implémenter plusieurs interfaces.
- ▶ La différence entre les classes abstraites et les interfaces est que l'interface ne définit aucune implémentation de code.

```
interface Vehicule{  
    masse:number;  
    calculerPoids(): number;  
}
```

```
abstract class Voiture implements Vehicule{  
    public static listeVoiture:Voiture[] = [];  
    public static readonly TVA:number = 20;  
    public readonly annee:number;  
    public masse:number;  
  
    public constructor(protected _marque:string,  
        private _modele:string,  
        protected _nbPortes:number,  
        annee:number,  
        masse:number){  
        this._marque=_marque;  
        this._modele=_modele;  
        this._nbPortes=_nbPortes;  
        this.annee = annee;  
        this.masse = masse;  
    }  
  
    public calculerPoids(){  
        return this.masse * 9,81;  
    }  
}
```

Force l'implémentation de l'attribut « masse » et de la fonction « calculerPoids ».
Attention, il n'est pas possible de définir un attribut « private » provenant d'une interface

```
class VoitureCourse extends Voiture{  
    public readonly couleur:string;  
  
    constructor(marque:string, modele:string, couleur:string){  
        super(marque,modele,3,2019,500);  
        this.couleur = couleur;  
    }  
}
```

```
let v3 = new VoitureCourse("Pors", "359", "bleue");  
console.log("Masse de la voiture : " + v3.masse);  
console.log("Poids de la voiture : " + v3.calculerPoids());
```

Les bases des « Generic Type »



- ▶ Les « Generic Type » permettent de définir des composants pouvant travailler avec plusieurs types afin de les rendre réutilisables.
- ▶ Les fonctions peuvent accepter en argument des informations de plusieurs types et en retourner.
- ▶ Le type « Array » est un type « generic » pouvant fonctionner avec plusieurs autres types. Exemple :

```
const notes:Array<number> = [15,19,20];
const noms:Array<string> = ["Matthieu","Tya","Milo"];
const personne:Array<string|number|boolean> = ["Matthieu",31,true];
```

Equivalent



```
const notes:number[] = [15,19,20];
const noms:string[] = ["Matthieu","Tya","Milo"];
const personne:(string|number|boolean)[] = ["Matthieu",31,true];
```

Les fonctions

- Il est possible de créer des fonctions polyvalentes pouvant récupérer des informations de plusieurs types simples :

```
function maFonction<Type>(param:Type) : void{  
    console.log(typeof param);  
}  
  
maFonction("bonjour");  
maFonction(15);  
maFonction(true);
```

Capture le type du paramètre de fonction

string
number
boolean

- Lorsque des tableaux sont récupérés en paramètre de fonction, TypeScript est capable d'identifier le type d'informations qu'ils contiennent :

```
function maFonction<Type>(param:Array<Type>) : void{  
    console.log(typeof param[0]);  
}  
  
maFonction([10,20,30]);  
maFonction(["matthieu","paul"]);  
maFonction([true,true]);
```

number
string
boolean

Les Classes

- Une classe peut utiliser le type « generic » et devenir plus polyvalente.
- Exemple de classe :

```
class Personnages<T extends {}> {  
    private liste = [];  
  
    ajouterPersonnage(perso:T){  
        this.liste.push(perso);  
    }  
    afficherPersonnages(){  
        for(let element of this.liste){  
            console.log(element);  
        }  
    }  
}
```

Cette classe permet de conserver une liste de personnages de n'importe quel type d'objet



```
interface Personnage {nom : string;}  
interface Guerrier extends Personnage {classe : "guerrier"}  
interface Archer extends Personnage {classe : "archer", type : "distance"}  
  
const g1:Guerrier = {nom:"Matthieu",classe:"guerrier"};  
const g2:Guerrier = {nom:"Gael",classe:"guerrier"};  
const a1:Archer = {nom : "Tya",classe:"archer",type:"distance"};  
  
const listeGuerrier = new Personnages<Guerrier>();  
listeGuerrier.ajouterPersonnage(g1);  
listeGuerrier.ajouterPersonnage(g2);  
console.log("Liste des guerriers : ");  
listeGuerrier.afficherPersonnages();  
  
const listeArcher = new Personnages<Archer>();  
listeArcher.ajouterPersonnage(a1);  
console.log("Liste des Archers : ");  
listeArcher.afficherPersonnages();
```

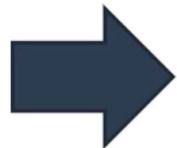
Liste de guerriers

Liste d'archers

« Partial » type

- ▶ Pour créer des objets partiels d'un type donné, il faut utiliser le mot clef « Partial »

```
interface Personnage {  
    nom : string;  
    age : number;  
    sexe : boolean;  
};
```



```
let p1:Partial<Personnage> = {};  
p1.nom = "Matthieu";  
p1.age = 31;  
p1.sex = true;  
p1.force = 13; |  
console.log(p1.age);
```

Il n'est pas possible de rajouter des propriétés non prévues par le type !



```
function creerHomme(nom,age) : Personnage {  
    let perso : Partial<Personnage> = {};  
    perso.nom = nom;  
    perso.age = age;  
    perso.sex = true;  
    return perso as Personnage;  
}
```

Permet d'indiquer que la variable « perso » est de type « Personnage » mais est initialisée de manière incomplète.

Le type « generic » Readonly

- ▶ Tableau :

```
const noms : string[] = ["Matthieu", "GASTON"];
noms[0] = "toto";
noms.push("bernard");
```

Les valeurs présentes dans le tableau peuvent être modifiées et il est possible d'ajouter ou de diminuer le nombre de cases

- ▶ Tableau contenant des valeurs fixes :

```
const noms : Readonly<string[]> = ["Matthieu", "GASTON"];
noms[0] = "toto";|
noms.push("bernard");
```

Le tableau n'est plus modifiable

- ▶ Objet contenant des valeurs fixes :

```
type Personnage = {
    nom : string;
    age : number;
}

let p1: Readonly<Personnage> = {
    nom : "Tya",
    age : 15
}
```

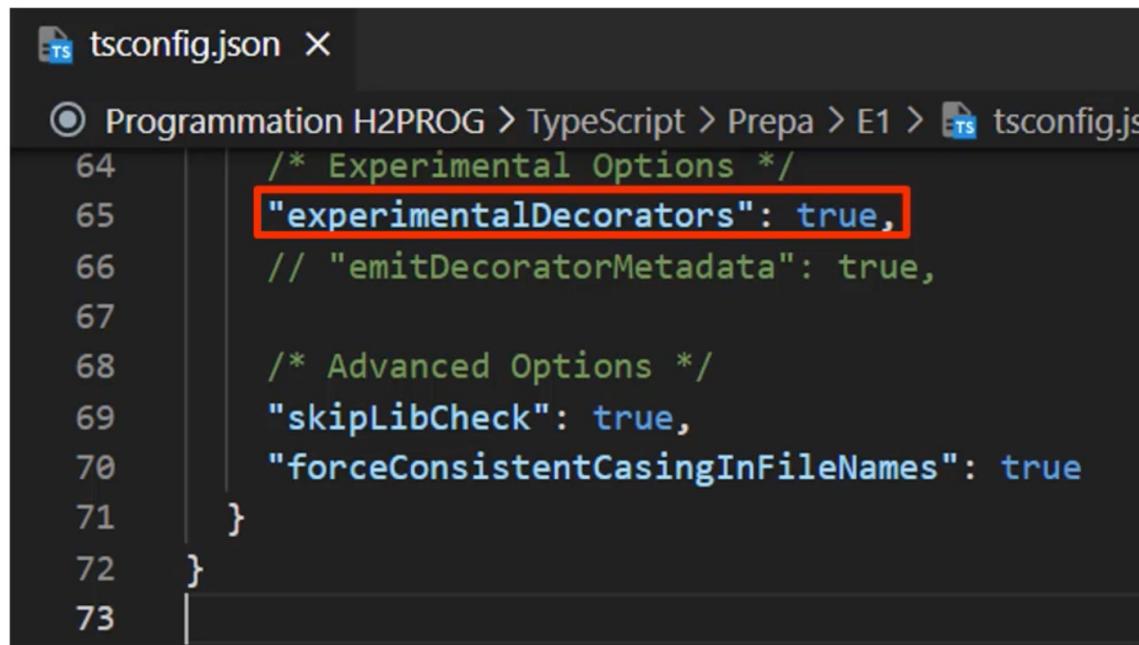
Les décorateurs

- ▶ Les décorateurs sont des fonctions qui s'utilisent avec des classes.
- ▶ Ces fonctions peuvent être en lien avec la classe elle-même, d'autres fonctions de la classe (méthode), des accesseurs, des attributs et des paramètres
- ▶ Elles permettent de décrire les éléments d'une classe et s'exécute lors l'intégration de la classe, au lancement du programme.
- ▶ Un décorateur s'utilise avec le symbole « @ » suivi de son nom. Cette syntaxe doit précéder l'élément concerné.
- ▶ Les décorateurs constituent des fonctionnalités expérimentales de TypeScript. Leur utilisation est susceptible de changer. Ils ne sont que très peu utiles pour écrire du code logique mais sont utilisés par certaines librairies, dans des cas très spécifiques.
- ▶ **Je ne recommande donc pas leur maîtrise, mais simplement de connaître leurs existence.**

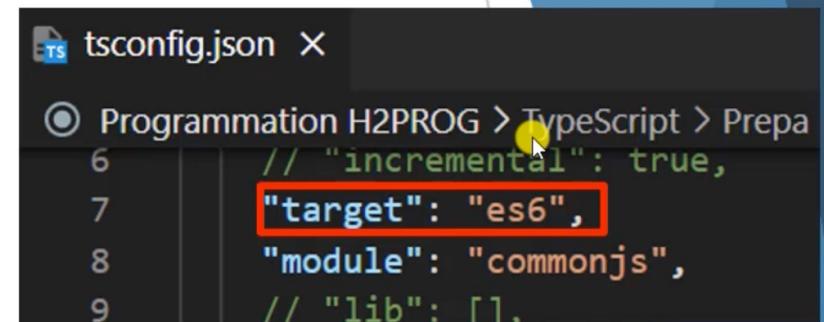
```
function decoClass(constructor : Function){
```

```
@decoClass  
class Perso {
```

Activation



```
tsconfig.json X
Programmation H2PROG > TypeScript > Prepa > E1 > tsconfig.json
64      /* Experimental Options */
65      "experimentalDecorators": true,
66      // "emitDecoratorMetadata": true,
67
68      /* Advanced Options */
69      "skipLibCheck": true,
70      "forceConsistentCasingInFileNames": true
71  }
72 }
73 |
```



```
tsconfig.json X
Programmation H2PROG > TypeScript > Prepa
6      // "incremental": true,
7      "target": "es6",
8      "module": "commonjs",
9      // "lib": [].
```

Décorateur de classe

```
@decoClass
class Perso {
    private _nom:string;
    private _age:number;

    constructor(nom:string,age:number){
        console.log("Construteur de la classe")
        this._nom = nom;
        this._age = age;
    }

    get nom(){return this._nom}
    set nom(newNom:string){this._nom = newNom}

    get age(){return this._age}

    afficherPerso(){
        console.log("Nom : " + this.nom + " - Age : " + this.age);
    }

    modifierAge(nombre:number){
        this._age += nombre;
    }
}
```

```
function decoClass(constructor : Function){
    console.log("Décorateur de la classe")
}
```

```
const p1 = new Perso("Matthieu",31);
const p2 = new Perso("Tya",18);
const p3 = new Perso("Milo",20);
```

```
Décorateur de la classe
Constructeur de la classe
Constructeur de la classe
Constructeur de la classe
```

Le décorateur n'est exécuté qu'une seule fois, lors de l'intégration de la classe dans le programme

Import de fichier

- ▶ Avec TypeScript, il existe 3 façons pour utiliser plusieurs fichiers et les intégrer dans les pages web :

- ▶ Ajout d'une balise « script » par fichier JS généré
 - ▶ Avantage : permet d'intégrer manuellement chaque fichier et d'assurer le fonctionnement
 - ▶ Désavantage : lourd à mettre en place et risque d'erreur
- ▶ Utilisation des modules JavaScript (ne fonctionne que sur des navigateurs modernes acceptant la syntaxe ES6)
 - ▶ Avantage : permet d'assurer la bonne utilisation des informations récupérées dans les fichiers TypeScript (auto-completion)
 - ▶ Désavantage : ne fonctionne pas sur tous les navigateurs. Il faudra passer par un serveur local pour éviter une « CORS error »
- ▶ Utilisation des Namespaces TypeScript
 - ▶ Avantage : le code écrit fonctionnera dans toutes les versions de JS (le code généré n'utilise pas la syntaxe des Namespace, car elle n'existe pas en JS)
 - ▶ Désavantage : ne permet pas l'auto-completion, risque d'erreur et fonctionnalité spécifique à TypeScript
- ▶ Recommandation : Utiliser la syntaxe des modules JavaScript si le projet doit fonctionner en ES6 (ou supérieur)

Les modules ES6



P3-base.zip

- ▶ Exporter tous les éléments qui doivent être utilisés dans d'autres fichiers
 - Les différences
- ▶ Importer les éléments nécessaires dans les fichiers

```
ts main.ts ×
● Programmation H2PROG > TypeScript > Prepa > P3 - Namespace > js > src > ts main.ts > ...
1 import { ClasseAliment, Aliment } from "./classes/Aliment.class.js";
2 import { Fruit } from "./classes/Fruit.class.js";
3 import { Charcuterie } from "./classes/Charcuterie.class.js";
4
5 new Fruit("Pomme",52,0.2,14,0.3,"pomme.png");
6 new Fruit("Poire",57,0.1,15,0.4,"poire.png");
7
8 new Charcuterie("Saucisson",416,33.1,1.9,27.6,"saucisson.png");
9 new Charcuterie("Cervelas",292,26.5,1.3,12,"salami.png");
```

ts Aliment.class.ts ×

● Programmation H2PROG > TypeScript > Prepa > P3 - Namespace > js > src > classe

```
1 export enum ClasseAliment {MAUVAIS="C",MOYEN="B",BON="A"};
2
3 export abstract class Aliment{
4     public static listeAliments:Aliment[]=[];
5 }
```

ts Charcuterie.class.ts ×

● Programmation H2PROG > TypeScript > Prepa > P3 - Namespace > js > src > classes

```
1 import { ClasseAliment,Aliment } from "./Aliment.class.js";
2
3 export class Charcuterie extends Aliment{
4     public static listeCharcuterie:Charcuterie[] = [];
```

ts Fruit.class.ts ×

● Programmation H2PROG > TypeScript > Prepa > P3 - Namespace > js > src > classes

```
1 import { ClasseAliment,Aliment } from "./Aliment.class.js";
2
3 export class Fruit extends Aliment{
4     public static listeFruit:Fruit[] = [];
```

Attention ce sont les fichiers JS
qu'il faut importer !

Les Namespaces



P3-base.zip

► Transformer le projet 3 en utilisant les Namespaces :

- Le code qui fonctionne « ensemble » doit être ajouté dans un même Namespace. De plus, pour utiliser un élément (classe, fonction, variable...) à l'extérieur d'un fichier, il est nécessaire de l'exporter. Certains éléments pourront ainsi être utilisés que dans un seul fichier (si non exporté).

```
namespace App {
    export enum ClasseAliment {MAUVAIS="C",MOYEN="B",BON="A"};

    export abstract class Aliment{
        public static listeAliments:Aliment[]=[];

        constructor(
            protected _nom:string,
            protected _sante:ClasseAliment,
            public readonly calorie:number,
            public readonly lipide:number,
            public readonly glucide:number,
            public readonly protéine:number,
            protected _image:string){
                Aliment.listeAliments.push(this);
            }

            public get nom() : string {return this._nom;}
            public get sante() : ClasseAliment {return this._sante;}
            public get image() : string {return this._image;}

            public set nom(newNom:string) {this._nom = newNom;}
            public set sante(newSante:ClasseAliment) {this._sante=newSante;}
            public set image(newImage:string) {this._image=newImage}
        }
    }
}
```

```
/// <reference path="Aliment.class.ts" />
namespace App {
    export class Fruit extends Aliment{
        public static listeFruit:Fruit[] = [];

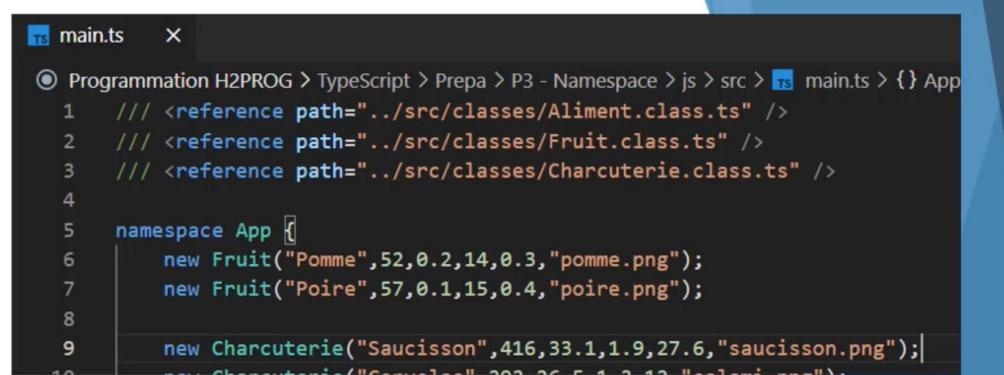
        constructor(
            _nom:string,
            _calorie:number,
            _lipide:number,
            _glucide:number,
            _proteine:number,
            _image:string){
                super(_nom,ClasseAliment.BON,_calorie,_lipide,_glucide,_proteine,_image);
                Fruit.listeFruit.push(this);
            }
    }
}

/// <reference path="Aliment.class.ts" />
namespace App {
    export class Charcuterie extends Aliment{
        public static listeCharcuterie:Charcuterie[] = [];

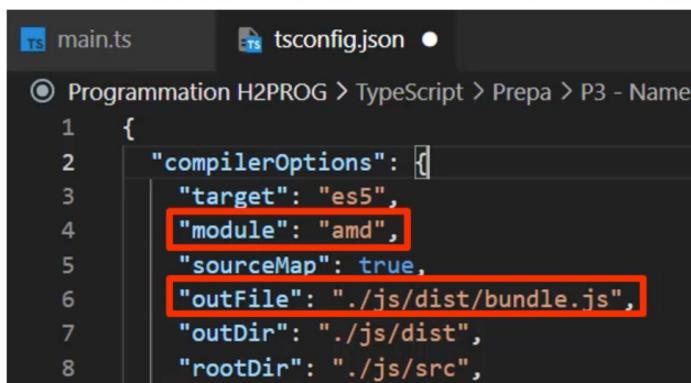
        constructor(
            _nom:string,
            _calorie:number,
            _lipide:number,
            _glucide:number,
            _proteine:number,
            _image:string){
                super(_nom,ClasseAliment.MAUVAIS,_calorie,_lipide,_glucide,_proteine,_image);
                Charcuterie.listeCharcuterie.push(this);
            }
    }
}
```

Les Namespaces

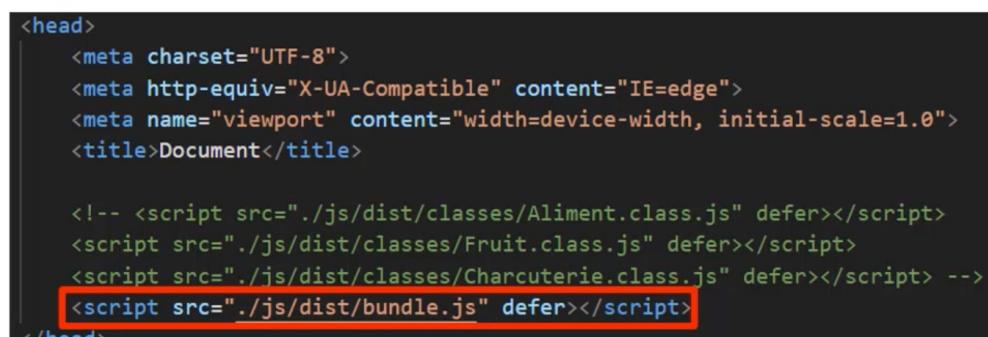
- Le fichier principal TypeScript doit également être ajouté dans le Namespace
- Pour que le code fonctionne, il faudra modifier le paramétrage du fichier de configuration de TypeScript. Après compilation, un fichier « bundle » sera généré
- Dans le fichier HTML il suffira d'importer le fichier « bundle » généré :



```
main.ts
1  /// <reference path="../src/classes/Aliment.class.ts" />
2  /// <reference path="../src/classes/Fruit.class.ts" />
3  /// <reference path="../src/classes/Charcuterie.class.ts" />
4
5  namespace App {
6    new Fruit("Pomme",52,0.2,14,0.3,"pomme.png");
7    new Fruit("Poire",57,0.1,15,0.4,"poire.png");
8
9    new Charcuterie("Saucisson",416,33.1,1.9,27.6,"saucisson.png");
10   new Charcuterie("Gastral",202,36.5,1.3,12,"gastral.png");
11 }
```



```
tsconfig.json
1  {
2    "compilerOptions": {
3      "target": "es5",
4      "module": "amd",
5      "sourceMap": true,
6      "outFile": "./js/dist/bundle.js",
7      "outDir": "./js/dist",
8      "rootDir": "./js/src",
9    }
10 }
```



```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>

  <!-- <script src="./js/dist/classes/Aliment.class.js" defer></script>
  <script src="./js/dist/classes/Fruit.class.js" defer></script>
  <script src="./js/dist/classes/Charcuterie.class.js" defer></script> -->
  <script src="./js/dist/bundle.js" defer></script>
</head>
```

Présentation et Structure d'un projet

- Présentation du module
- Création du dossier de base
- Initialisation du projet
- Débogage avec Chrome
- Structurer le projet

Les Aliments

- Présentation du projet
- La partie HTML
- La classe "Aliment"
- Les classes filles
- Finalisation

Projet parc Auto

- Présentation du projet
- Les classes véhicules
- Listes de véhicules
- Le parc auto
- Le DOM
- Finalisation

Convertisseur de Devises

- Présentation du projet
- La partie HTML
- Le type DeviseType et les objets
- Les listes déroulantes
- Récupération des valeurs
- Finalisation du projet

Drapeaux Pays

- Présentation du projet
- Le fichier HTML et la structure du projet
- La récupération des données avec l'API RestCountries
- Le traitement des données
- Le jeu
- Rechargement du jeu

Urlographie

- <https://ts.chibicode.com/todo/>
- <https://www.typescriptlang.org/docs/handbook/>