

Découverte de TypeScript



Présentation

Les origines de TypeScript :

TypeScript est un langage open source développé par **Microsoft** qui est un **surenement** du langage **JavaScript**.

La motivation initiale était issue d'insatisfactions sur le langage **JavaScript**, notamment dans le cadre de projets lourds, mais **JS** étant l'unique langage supporté par le standard **HTML5**, il n'était pas possible de le remplacer par un langage alternatif. Par contre l'utilisation d'un **surenement** permet d'apporter des améliorations substantielles au langage **JS** durant la phase développement. Pour la mise en production, un compilateur spécifique va compiler l'ensemble du code développé pour produire un code en pur **JavaScript** standard.

Les points d'insatisfaction sur **JavaScript** étaient principalement :

- L'absence de typage fort.
- La programmation orienté objet de **JavaScript** qui reposait initialement sur des objets prototypes et non sur des classes.

Pendant 2 ans **Microsoft** a développé **TypeScript**, puis l'a ouvert au public avec la version **0.8** en octobre 2021.

TypeScript étant un **surenement**, tout programme **JavaScript** est par défaut compatible **TypeScript** ! Il va apporter la programmation orientée objet par classe et un typage fort.

TypeScript répond aux limitations de **JavaScript**, sans compromettre la proposition de valeur clé de **JavaScript** : la possibilité d'exécuter votre code n'importe où et sur chaque plateforme, navigateur ou hôte.

Microsoft

Principe de fonctionnement de TypeScript :

Pour le programmeur, le développement sous **TypeScript** est relativement simple.

La démarche est la suivante :

- Le code est écrit dans des fichiers avec l'extension **.ts**.
- Puis les fichiers sont compilés avec un compilateur fourni avec **TypeScript**.

Le compilateur apporte 2 bénéfices :

- Transcompilation du code **TypeScript** vers **JavaScript**;
- Une partie des erreurs seront levées durant la phase compilation.

Pour information vous pourrez en savoir plus avec les liens ci-dessous :

Sources officielles :

- Site officiel de **TypeScript** : <https://www.typescriptlang.org>
- Documentation **TypeScript** : <https://www.typescriptlang.org/fr/docs/>
- Le playground de **Microsoft** pour **TypeScript** : <https://www.typescriptlang.org/play>

Quelques sources de formation :

- Formations **Microsoft** :
 - Formations pour apprentissage TypeScript
 - Bien démarrer avec TypeScript
 - Migration vers TypeScript à partir de JavaScript
 - Utilisation de TypeScript avec Visual Studio Code

Préparation du poste

Schéma général de la configuration du poste :

Nous utiliserons l'IDE **Visual Studio Code** pour produire le code **TypeScript**. Nous nous appuyerons donc sur les éléments suivants :

- **Visual Studio Code** : Pour l'éditeur de code
- **Node.JS** : Pour pouvoir exécuter le code **JavaScript** de nos programmes et applications.
- **TypeScript** : Nous aurons besoin du compilateur **TypeScript** pour compiler le code **TypeScript** en code **JavaScript** exécutable par **Node.JS** et/ou le moteur d'exécution **JavaScript** d'un navigateur (*Mozilla FireFox, Chrome, etc.*).

Note : Nous serons mené à installer les composants **Angular** quand nous aborderons l'étude de ce framework. **Angular** s'appuiera lui-même sur **Node.JS** et **TypeScript**.

Configuration de Visual Studio Code :

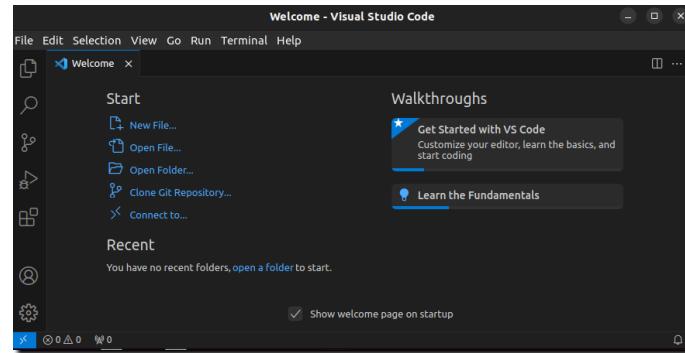
Avant-propos :

Sur l'année scolaire 2025-2026 vous devez normalement travailler sur des postes avec **Linux** (Ubuntu). Sur ces machines vous devez avoir **Visual Studio Code** qui est déjà installé. Vous devez donc normalement pas avoir besoin de procéder à l'installation de **Visual Studio Code**.

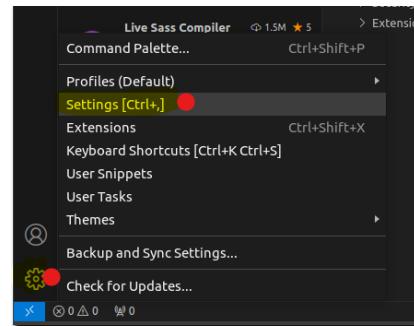
Configuration du proxy :

Si vous dépendez d'un proxy, il sera nécessaire de le renseigner.

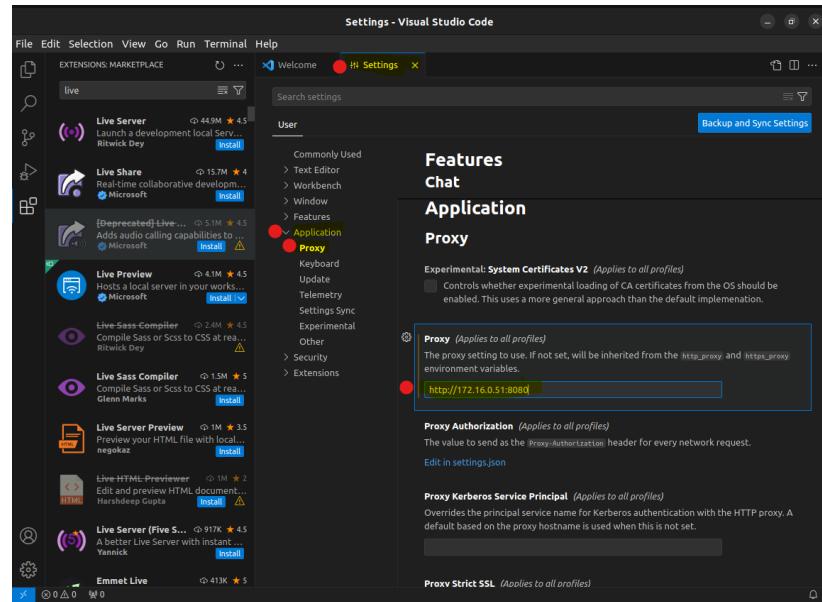
Ouvrir **VS Code** :



Accéder aux **paramètres (settings)** :

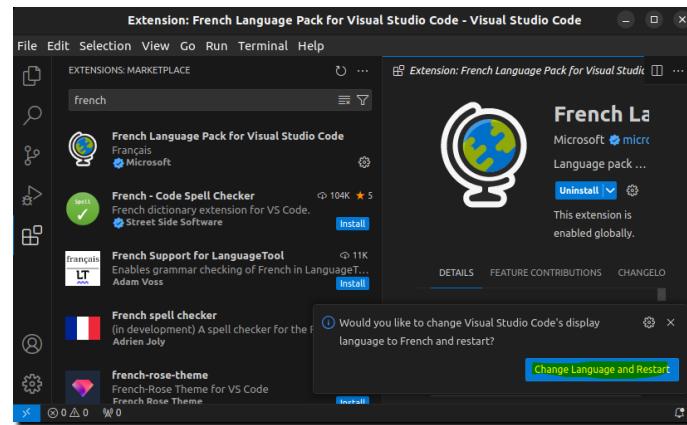
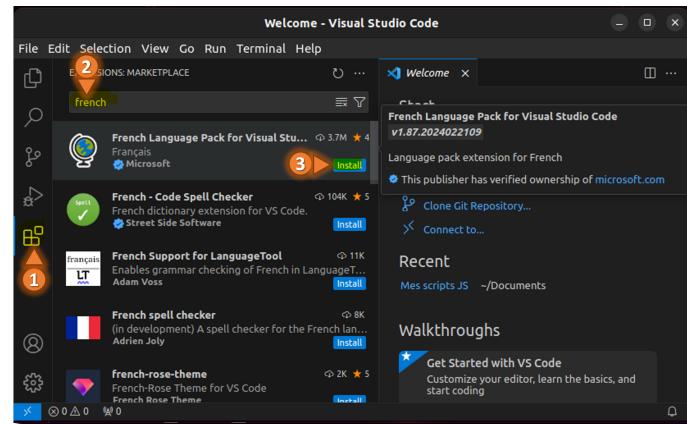


Aller dans **configuration** :



Installation de l'extension de langue d'interface Française :

Une bonne façon de tester le bon accès à Internet depuis VS Code est de procéder à l'installation d'une ou plusieurs extensions. Nous allons ici installer l'extension de langue française. Il suffit pour cela d'accéder au gestionnaire d'extension (étape 1), de taper le mot *french* dans la barre de recherche (étape 2) et de cliquer sur **Install** (étape 3).



Installation de

Choix de la stratégie de maintenance des versions :

Pour l'installation de **Node.js**, il existe 2 stratégies sur le type de version :

- Installation de la version dite **actual**, qui correspond à la dernière version **stable** disponible.
- Installation de la version dite **LTS**, qui est la dernière version **stable** disposant d'un support sur un long terme. C'est cette version qui est normalement installée sur votre poste.

C'est quoi Node.js et pourquoi l'installer ? :

Node.js est un environnement qui permet l'exécution de code **JavaScript** sur une machine. Typiquement **Node.js** sera installé sur des serveurs Web qui pourront ainsi utiliser du code **JavaScript** côté serveur (backend).

L'installation de **Node.js** est également utile pour une machine qui servira au développement d'applications de type **Web Front-End**. Il sera ainsi possible d'exécuter des tests à partir de l'**IDE** servant au développement.

Pour notre cas, l'installation de **Node.js** se fera avec son gestionnaire pour le langage **JavaScript**, à savoir **npm**.

Vérifions que **Node.js** est bien installé sur votre machine. Pour vérifier il suffit d'ouvrir le terminal (fenêtre d'invite de commande) et de taper la commande : **node -v** ou **node --version**

De même on pourra vérifier la présence du gestionnaire de paquet **npm** de **Node.js** avec les commandes :

- **npm --version** ou **npm -v**

Si **Node.js** est déjà présent, un message indiquera la version installée.

Configuration du proxy pour Node.js :

- **Etape 1 - Récupération de l'IP et numéro de port :**

Il faut d'abord récupérer l'adresse du proxy et son numéro de port. Pour ce faire, ouvrir un navigateur Web et saisir l'adresse suivante comme URL : <http://wpad.vinci-melun.org/wpad.dat> Vous obtiendrez une page avec une information de ce type :

```
function FindProxyForURL(url,host){  
    if( isPlainHostName(host) ||  
        isInNet(dnsResolve(host),"172.16.0.0","255.254.0.0") ||  
        isInNet(dnsResolve(host),"192.168.0.0","255.255.0.0") ||  
        isInNet(dnsResolve(host),"213.215.40.128","255.255.255.224")  
    ){  
        return "DIRECT";  
    }  
  
    return "PROXY 172.16.0.51:8080";  
}
```

Dans notre cas l'adresse IP du proxy est **172.16.0.51** avec un port en **8080**.

Voyons l'étape de configuration.

- **Etape 2 - Configuration de npm :**

- La commande de configuration du proxy est la suivante : **npm config set proxy http://172.16.0.51:8080**
- Il est possible de récupérer la configuration courante avec la commande : **npm config get proxy**

```
ljules@Vvm-ubu22:~$ npm config set proxy http://172.16.0.51:8080  
ljules@Vvm-ubu22:~$ npm config get proxy  
http://172.16.0.51:8080  
ljules@Vvm-ubu22:~$ █
```

Note Pour annuler le proxy on saisira la commande :

- **npm config set proxy null**

Installation du compilateur TS avec npm :

Une fois que **Node.js** est installé, nous allons pouvoir utiliser son gestionnaire de paquets **npm** pour installer le compilateur **TSC** pour **TypeScript**.

IMPORTANT : Si la connexion à Internet passe par un serveur proxy, il faudra alors configurer **npm** pour qu'il puisse accéder aux dépôts contenant les sources d'installation.

Commandes d'installation :

Procédons à l'installation du compilateur pour *TypeScript* qui s'appelle tout simplement **TSC**. Pour ce faire, exécuter dans le terminal la commande suivante :

- **sudo npm install -g typescript**

L'option **-g** indique que l'on désire procéder à une installation **générale**. On pourra ainsi utiliser **TypeScript** sur tout nos futur projets.

```
ljules@Vvm-ubu22:~$ sudo npm install -g typescript  
[sudo] Mot de passe de ljules :
```

Note : Si la commande **sudo npm install -g typescript** échoue, vous pouvez essayer la commande sans le **sudo** avec tout simplement : **npm install -g typescript**

Vérification de l'installation :

Nous pouvons vérifier l'installation de **TS** en tapant tout simplement la commande **tsc -v** :

```
ljules@Vvm-ubu22:~$ tsc -v  
Version 5.3.3  
ljules@Vvm-ubu22:~$ █
```

Premiers pas

Coder en JavaScript avec VS Code :

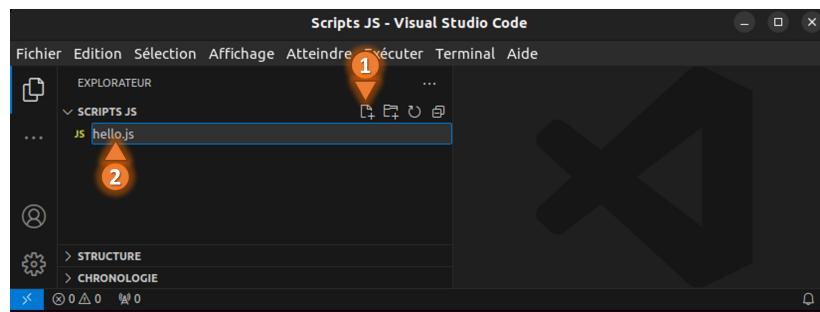
Avant d'aborder la programmation avec **TypeScript**, nous allons nous exercer à rédiger du code en pur **JS** avec l'éditeur **VS Code**.

Création et ouverture du dossier projet :

La 1^{re} chose à faire avec **VS Code** et indépendamment du langage de programmation adopté, il faut commencer par créer un dossier de travail que nous allons ici tout simplement appeler **Scripts JS**. Vous placerez ce dossier l'emplacement de votre choix. Dans notre illustration nous allons créer le dossier *Scripts JS* dans le dossier **home**, pour ce faire on pourra tout simplement ouvrir un terminal (**[ALT] + [T]**) et saisir la commande : **mkdir "Scripts JS"**

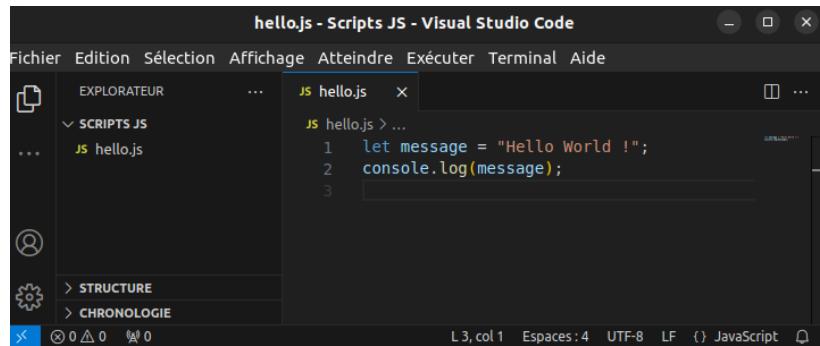
Ouvrir **VS Code** et ouvrir le dossier **Scripts JS** et créer un fichier **hello.js**

Création d'un fichier JS :



Saisir le code suivant dans le fichier **hello.js**

```
let message = "Hello World !";
console.log(message);
```



Exécution du script JS :

Nous allons maintenant voir comme exécuter notre code.

La méthode la plus simple est de faire exécuter le code par **node.js** via le terminal intégré à **VS Code**.

- Ouvrir le terminal si ce dernier n'est pas déjà présent avec le menu : **Terminal/Nouveau terminal**
- Saisir dans le terminal la commande : **node hello.js**

Votre code doit alors s'exécuter directement dans le terminal.

The screenshot shows the Visual Studio Code interface with a dark theme. In the center, there's a code editor window titled "hello.js - Scripts JS - Visual Studio Code" containing the following JavaScript code:

```
JS hello.js > ...
1 let message = "Hello World !";
2 console.log(message);
3
```

Below the code editor is a terminal window titled "TERMINAL". It displays the command "node hello.js" being run and the output "Hello World!". The terminal also shows the prompt "ljules@Vm-Ubu22:~/Scripts JS\$". At the bottom of the screen, there are status bars for "PROBLÈMES", "SORTIE", "TERMINAL", and "AIDE".

Installation de nodemon :

nodemon est un outil mis à disposition par **npm** (<https://www.npmjs.com/package/nodemon>). Cet outil nous permettra de lancer une seule fois notre script **JavaScript** et ce dernier sera automatiquement exécuté à chaque modification du votre code !

- Installer **nodemon** en saisissant dans le terminal la commande : **sudo npm install -g nodemon**
- Tester en exécutant votre script **hello.js** avec la commande : **nodemon hello.js**

The screenshot shows the Visual Studio Code interface with a dark theme. In the center, there's a code editor window titled "hello.js - Scripts JS - Visual Studio Code" containing the same JavaScript code as before. Below it is a terminal window titled "TERMINAL". The user has run the command "sudo npm install -g nodemon". The terminal output shows the process of installing nodemon, including prompts for a password ("[sudo] Mot de passe de ljules :") and a message about adding packages ("added 33 packages in 1s"). It also includes standard npm notices about funding and updates. The terminal prompt "ljules@Vm-Ubu22:~/Scripts JS\$" is visible at the end.

Vous pourrez constater que votre code est automatiquement exécuté à chaque modification :

The screenshot shows a Visual Studio Code interface with a dark theme. In the center is a terminal window titled 'hellojs - Scripts JS - Visual Studio Code'. It displays the following content:

```
l jules@vmm-ubu22:~/Scripts JS$ nodemon hello.js
[nodemon] 3.1.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting 'node hello.js'
Hello World !
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting 'node hello.js'
Bonjour à tous !
[nodemon] clean exit - waiting for changes before restart
o ^C l jules@vmm-ubu22:~/Scripts JS$
```

The terminal shows the execution of the 'nodemon' command on a file named 'hello.js'. The output includes the message 'Hello World !' and 'Bonjour à tous !' printed to the console. The terminal window has tabs for 'PROBLÈMES', 'SORTIE', and 'TERMINAL'. At the bottom, there are status indicators for 'L3, col 1', 'Espaces:4', 'UTF-8', 'LF', and 'JavaScript'.

Pour stopper **nodemon** il suffira de taper la commande **[CTRL] + [C]** dans le terminal.

Utilisation des commandes de VS Code pour exécuter vos scripts :

Vous pouvez également exécuter vos scripts en utilisant les commandes de **VS Code**. Il existe 2 options :

- **Exécuter sans débogage** : Le script est exécuté sans débogage même si des points d'arrêts ont été placés au niveau de la fenêtre d'édition du code.
 - L'exécution sans débogage peut se faire soit :
 - Avec le menu dédié : **Exécuter/Exécuter sans débogage**
 - Utiliser le raccourci clavier dédié : presser la combinaison de touches **[CTRL] + [F5]**
- **Démarrer le débogage** : Lance le débogage du script.
 - L'exécution sans débogage peut se faire soit :
 - Avec le menu dédié : **Exécuter/Démarrer le débogage**
 - Utiliser le raccourci clavier dédié : presser la touche **[F5]**

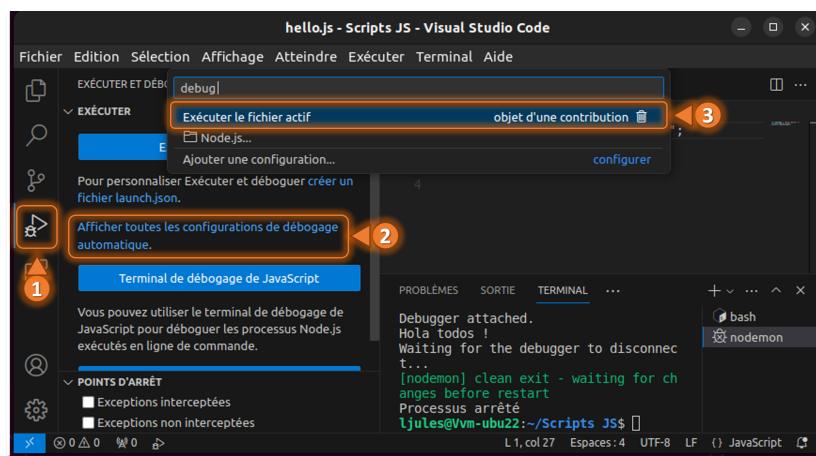
Configuration d'un fichier `launch.json` :

VS Code utilise des fichiers au format **.json** pour insrire différentes configurations. Il utilise notamment un fichier **launch.json** pour définir les actions pour réaliser une exécution/débogage de nos scripts. Le fichier **launch.json** sera enregistré dans un dossier **.vscode** qui lui même contenu dans le dossier projet.

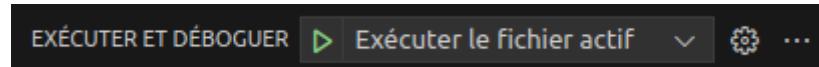
Le plus simple pour générer ce fichier est d'utiliser l'assistant de **VS Code**. Cet assistant s'exécute automatiquement au 1er lancement d'un script s'il n'existe pas de fichier **launch.json**

Voici la marche à suivre :

- Lancer la commande **exécuter et déboguer** soit en cliquant sur  ou en utilisant la combinaison de touches **[CTRL] + [MAJ] + [D]**
- Cliquer sur l'option : **Afficher toutes les configurations de débogage automatique**.
- Choisir : **Exécuter le fichier actif**

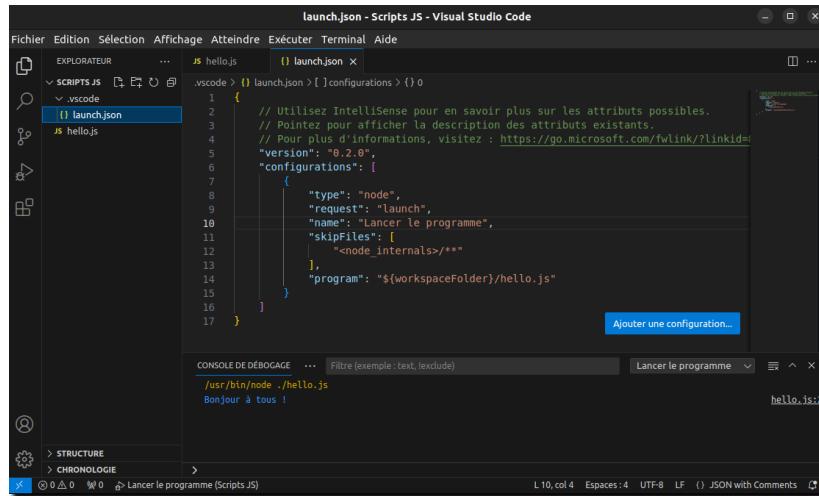


Maintenant vous avez accès à un menu permettant de choisir vos à vos différentes configurations d'exécution/débogage :



Cependant le fichier **launch.json** n'a pas été généré si vous avez bien choisi **Exécuter le fichier actif**. Pour générer le fichier **launch.json** vous pouvez cliquer sur le menu déroulant ou sur le bouton de configuration . Choisir alors la proposition **Node.js**

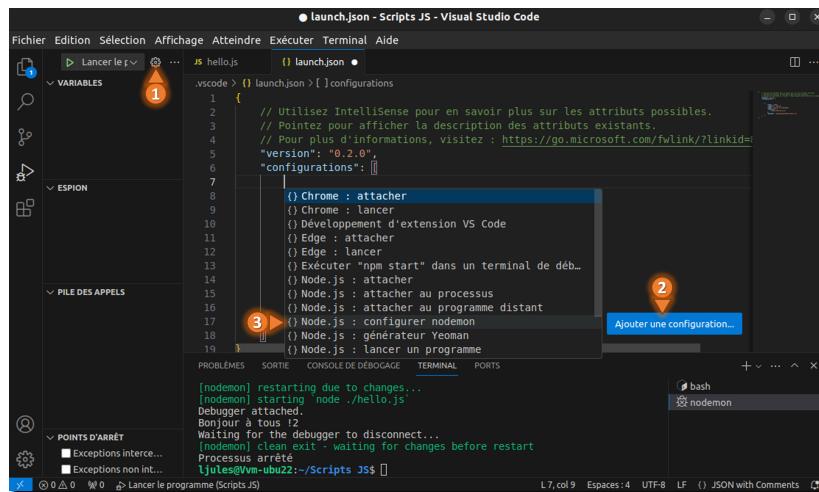
Le fichier **launch.json** est généré et s'ouvre dans l'éditeur



Nous pouvons l'enrichir avec d'autres configurations, pour ce faire :

- Cliquer sur le bouton de configuration et cliquer sur **Ajouter une configuration...**
- Choisir dans le menu déroulant : **Node.js : configurer nodemon**
- Remplacer l'entrée "**program**": " **\${workspaceFolder}/app.js**" par "**program**": " **\${workspaceFolder}/hello.js**"

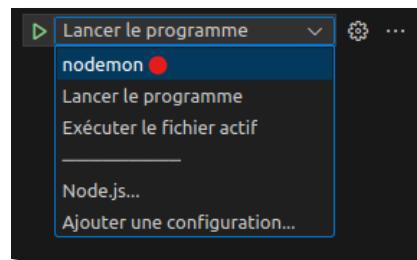
Création d'une nouvelle configuration :



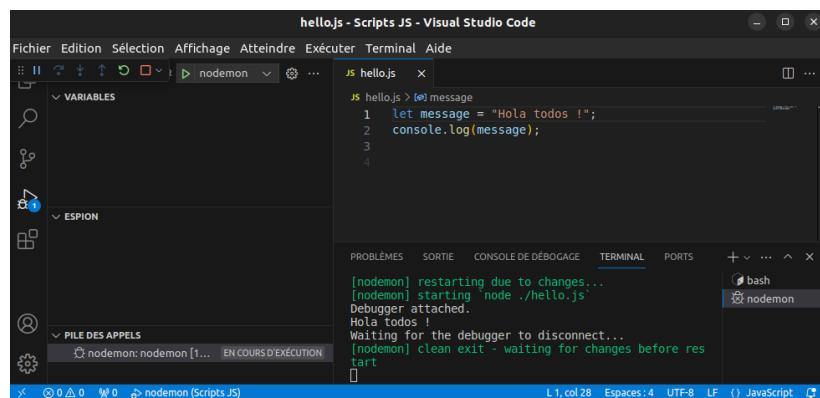
Mettre à jour le nom du script JS concerné par la configuration :

```
.vscode > { } launch.json > [ ] configurations > {} 0 > program
1 ~ {
2   // Utilisez IntelliSense pour en savoir plus sur les attributs possibles.
3   // Pointez pour afficher la description des attributs existants.
4   // Pour plus d'informations, visitez : https://go.microsoft.com/fwlink/?linkid=1130365
5   "version": "0.2.0",
6   "configurations": [
7     {
8       "console": "integratedTerminal",
9       "internalConsoleOptions": "neverOpen",
10      "name": "nodemon",
11      "program": "${workspaceFolder}/hello.js", hello.js
12      "request": "launch",
13      "restart": true,
14      "runtimeExecutable": "nodemon",
15      "skipFiles": [
16        "/**"
17      ],
18      "type": "node"
19    }
]
```

Votre nouvelle configuration est maintenant accessible via une entrée dans la barre de débogage :



Vous pouvez sélectionner la configuration et l'exécuter et l'arrêter :



Nous ne pousserons pas plus loin l'investigation pour le développement en pur **JavaScript** durant ce TP, l'objectif de cette partie était de vous donner une vue d'ensemble pour créer, exécuter/déboguer des scripts **JS** avec **VS Code** et **Node.js**.

Dans la partie suivante nous allons voir comment procéder pour un projet **TypeScript**.

Création d'un projet :

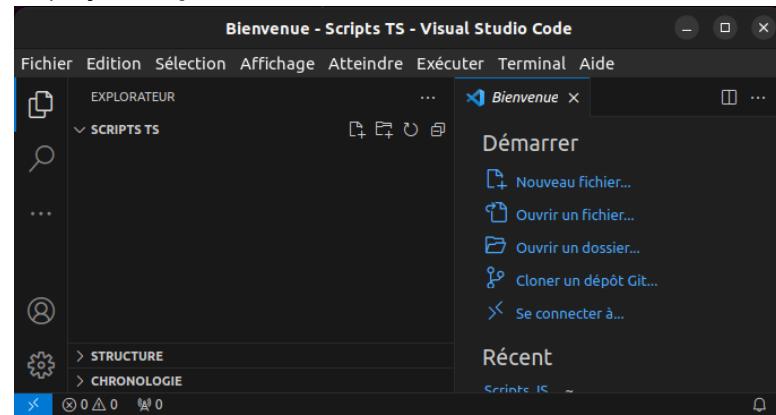
Création du dossier de projet :

Créer un dossier **Scripts TS** (attention ne pas confondre avec le dossier **Scripts JS** de la section précédente).

Génération du fichier de configuration :

Tout projet développé avec **TypeScript** s'appuie sur un fichier de configuration **tsconfig.json**. Ce fichier peut naturellement être générer manuellement, mais **TypeScript** peut nous aider à générer ce fichier.

- Ouvrir votre dossier de projet **Scripts TS** dans **VS Code** :



- Ouvrir le terminal (**Terminal/Nouveau terminal**) :
- Taper dans le terminal la commande : **tsc -init**

```

1 {
2   "compilerOptions": {
3     /* Visit https://aka.ms/tsconfig to read more about this */
4     /* Projects */
5     // "incremental": true, /* Se */
6     // "composite": true, /* Er */
7     // "tsBuildInfoFile": "./tsbuildinfo", /* Sp */
8     // "disableSourceOfProjectReferenceRedirect": true, /* Dj */
9     // "disableSolutionSearch": true. /* Or */
10    /* Options par défaut */
11    /* You can learn more at https://aka.ms/tsconfig */
12  }
13}

```

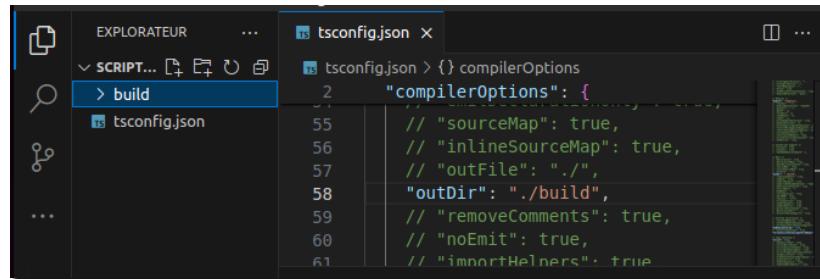
Nous vous invitons à ouvrir le fichier **tsconfig.json**, vous pourrez constater qu'il comporte de nombreuses options qui sont dans la grande majorité désactivées (mise en commentaire avec **//**)

Personnalisation du fichier de configuration :

Nous allons apporter quelques modifications au fichier de configuration :

Choix d'un dossier de sorti pour les fichiers transcompilés :

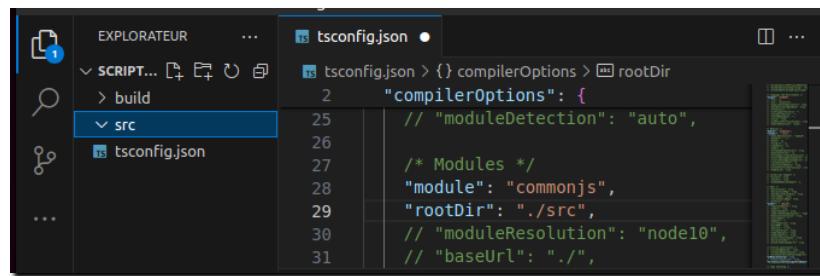
- Créer un dossier **build** dans le dossier de projet.
- Modifier l'option **outdir** en l'activant (suppression du **//**) et renseigner l'option avec la valeur **build**.



```
tsconfig.json
{
  "compilerOptions": {
    "outDir": "./build",
    // "removeComments": true,
    // "noEmit": true,
    // "importHelpers": true
  }
}
```

Choix d'un dossier d'entrée pour les fichiers sources :

- Créer un dossier **src** dans le dossier de projet
- Activer et renseigner l'entrée pour le dossier source (option **rootDir**)



```
tsconfig.json
{
  "compilerOptions": {
    "rootDir": "./src",
    // "moduleDetection": "auto",
    /* Modules */
    "module": "commonjs",
    "rootDir": "./src",
    // "moduleResolution": "node10",
    // "baseUrl": "./"
  }
}
```

Astuce :

Pour les plus curieux, vous pourrez en apprendre plus sur les options du fichier de configuration en consultant la page dédiée : <https://www.staging-typescript.org/fr/tsconfig>

Création d'un fichier TypeScript :

Créer dans le dossier **src** un fichier **hello.ts**, placer dans ce fichier le script **JavaScript** :

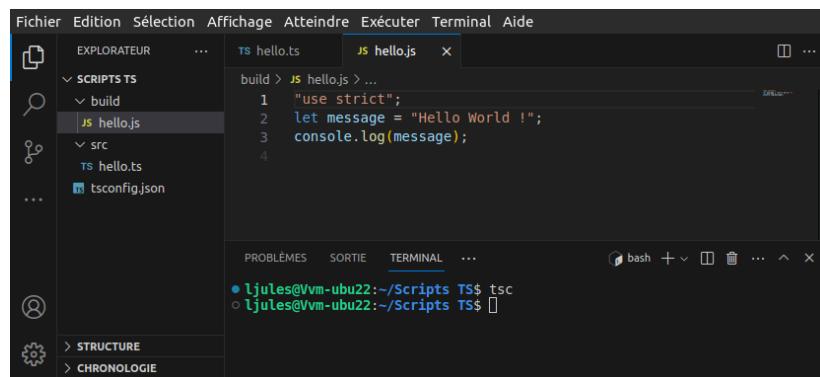
Code à saisir dans le fichier **hello.ts** :

```
let message = "Hello World !";
console.log(message);
```

Note Nous avons saisi du code purement **JavaScript** pour le moment, car pour le moment nous allons nous familiariser avec le processus de création/compilation de **TypeScript**.

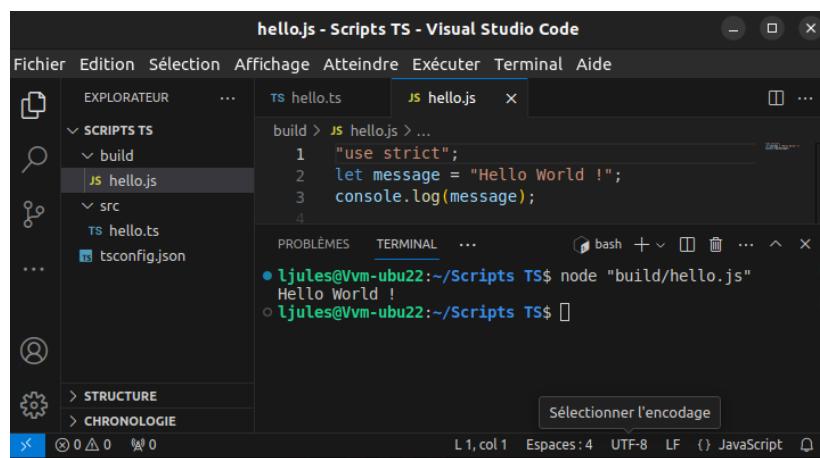
Pour lancer la compilation, il faut exécuter l'une des commandes suivantes :

- **tsc** : Compilera *tous* les fichiers **.ts** du dossier en *JavaScript* et les placera dans le dossier **build**;
- **tsc hello.ts** : Pour compiler uniquement le fichier **hello**, par contre le fichier ne sera pas placé dans le dossier **build**.



Vous pouvez ensuite exécuter votre code soit avec la commande :

- **node "build/hello.js"**



La syntaxe TypeScript

Typage des variables :

Syntaxe de déclaration des variables :

La syntaxe reprend celle de **JavaScript** avec le mot clé **let** ou **const** qui est complétée par une partie supplémentaire qui permet de préciser le type. On retrouve une syntaxe assez proche de la syntaxe du langage **Kotlin**.

```
let idVar: typeVar;
```

Il est également possible de déclarer des constantes (l'initialisation doit se faire à la volée) :

```
const idConstante: typeVar = valeur;
```

Les principaux types et leurs déclaration :

Le principal apport de **TypeScript** est le typage statique des variables. Nous allons dans cette partie jeter un aperçu sur les principaux types.

Les types primitifs :

Nous retrouvons les 3 types primitifs du langage **JavaScript** :

- **number** : Pour toutes les valeurs numériques (entiers relatifs et décimaux).
- **string** : Pour les chaînes de caractères.
- **boolean** : Pour les variables booléennes **true** et **false**

Exemples de déclarations et d'initialisations :

```
// Déclarations sans initialisation :
let nombre: number;
let texte: string;
let varlog: boolean;

// Affectations de valeurs :
nombre = 33.7;
texte = "TypeScript est un langage avec un typage statique.";
varlog = true;

// Vérification des types :
console.log(`Type de nombre -> ${typeof(nombre)} `);
console.log(`Type de texte -> ${typeof(texte)} `);
console.log(`Type de varlog -> ${typeof(varlog)} `);
```

Résultat en sortie de console :

```
Type de nombre -> number  
Type de texte -> string  
Type de varlog -> boolean
```

TypeScript joue bien son rôle de garde fou par rapport au respect des types des variables, dans l'exemple suivant, si nous essayons d'affecter à la variable **nombre**, cela génère une erreur dans l'éditeur et dans le compilateur.

Exemple d'une affectation non conforme au type :

```
Impossible d'assigner le type 'string' au type 'number'. ts(2322)  
let nombre: number  
Voir le problème (Alt+F8) Aucune solution disponible dans l'immédiat  
nombre = "43" // Refusé car "43" est considéré comme un string.
```

Le mécanisme d'inférence :

Il est possible d'affecter le type d'une variable en utilisant le mécanisme d'inférence.

ATTENTION :

Ici c'est bien un mécanisme d'inférence qui est mis en oeuvre et non pas un typage dynamique. En effet il est impossible de modifier le type comme le typage dynamique une fois que la variable a été initialisée (typage statique).

Exemple d'utilisation du mécanisme d'inférence :

```
// Déclarations avec mécanisme d'inférence :  
let nombre = 33.7;  
let texte = "TypeScript est un langage avec un typage statique.";  
let varlog = true;  
  
// Vérification des types :  
console.log(`Type de nombre -> ${typeof(nombre)}`);  
console.log(`Type de texte -> ${typeof(texte)}`);  
console.log(`Type de varlog -> ${typeof(varlog)}`);
```

Résultat en sortie de console :

```
Type de nombre -> number  
Type de texte -> string  
Type de varlog -> boolean
```

Gestion des types **null** et **undefined** :

TypeScript est un langage avec une approche **null safety**, c'est-à-dire qu'il encouragera la non utilisation des types **null** et **undefined** dans votre code.

Astuce :

Pour rappel en **JavaScript** **null** et **undefined** sont 2 types. Le type **null** est explicitement affecté à une variable pour indiquer quelle comporte la valeur **null**. Le type **undefined** est appliqué à une variable qui n'a jamais été initialisée, il ne comporte donc pas de valeur (comme pour **null**) mais aucun type n'a pu être déterminé (à cause de son absence de valeur).

Par défaut **TypeScript** n'autorise pas la déclaration de variables avec les types **null** et **undefined**.

The screenshot shows a code editor with the following TypeScript code:

```
s 3 Impossible d'assigner le type 'null' au type 'number'. ts(2322)
.ts > ...
let nombre: number
// D Voir le problème (Alt+F8) Aucune solution disponible dans l'immédiat
let nombre: number = null;
let texte: string = undefined;
let varlog: boolean = null;
```

An error message is displayed: "Impossible d'assigner le type 'null' au type 'number'. ts(2322)". A tooltip "Voir le problème (Alt+F8)" is shown above the error message, with the note "Aucune solution disponible dans l'immédiat". The identifiers `nombre`, `texte`, and `varlog` are underlined in red, indicating they are invalid.

Il est tout de même possible de déclarer une variable en leurs attribuant le type **null** ou **undefined**, mais l'affectation de valeur autre que respectivement **null** et **undefined** n'est pas possible.

```
let nombre: undefined;
let texte: null;
let varlog: null

nombre = undefined // Accepté
nombre = 33;        // Refusé
texte = null;      // Accepté
texte = "33";       // Refusé
varlog = null;     // Accepté
varlog = true;      // Refusé
```

The code editor shows valid assignments for variables declared with `undefined` and `null` types. The assignments `nombre = undefined`, `texte = null`, and `varlog = null` are marked as accepted (green). The assignments `nombre = 33` and `varlog = true` are marked as refused (red), indicating they are invalid.

Cependant il est possible de déclarer une variable avec le type `null` ou `undefined` en utilisant l'opérateur `|` associé à un type alternatif.

```
let nombre: undefined | number;
let texte: null | string;
let varlog: null | boolean;

nombre = undefined // Accepté
nombre = 33;        // Accepté
texte = null;        // Accepté
texte = "33";        // Accepté
varlog = null;        // Accepté
varlog = true;       // Accepté
```

L'opérateur ?? :

TypeScript prend en charge l'opérateur de coalescence de null (`??`), qui permet de définir une valeur par défaut lorsqu'une variable est **null** ou **undefined**. Cela permet de simplifier le traitement des valeurs **null** dans le code.

```
let x: number | null = null;
let y = x ?? 10;    // Si x est null, y prendra la valeur par défaut 10
console.log(y);    // affiche: 10
```

Le type NaN :

Comme en **JS** nous retrouvons également le type **NaN** dans **TypeScript**.

- Type de **NaN**: En termes de type, **NaN** est considéré comme un **number** en **TypeScript**.
- Caractéristiques de **NaN** :
 - **NaN** n'est jamais égal à aucune autre valeur, y compris lui-même. Cela signifie que `NaN !== NaN`.
 - Pour vérifier si une valeur est **NaN**, vous ne pouvez pas utiliser l'opérateur d'égalité simple (`==` ou `===`). Au lieu de cela, vous devez utiliser la fonction **isNaN()**.
- Utilisation de **isNaN()** : La fonction **isNaN()** prend une valeur en argument et retourne **true** si la valeur est **NaN**, sinon elle retourne **false**. Cependant, il convient de noter que **isNaN()** tente de convertir la valeur en un nombre avant de vérifier si elle est **NaN**, donc elle peut donner des résultats inattendus pour certaines valeurs non numériques.

Les collections :

Nous nous concentrerons sur 3 types structurés et plus particulièrement 3 collections :

- Les **Array**
- Les **Map**
- Les **Set**

Les Array :

Les **Array** sont des tableaux mutables, indexés, itérables et dynamiques. Il existe une multitudes de façons pour les déclarer/initialiser :

```
// Déclaration d'un tableau vide de number :
let tableauVide_1: number[] = [];
let tableauVide_2: Array<number> = [];

// Déclaration d'un tableau vide de number et string :
let tabStrNumVide_1: (number | string)[] = [];
let tabStrNumVide_2: Array<number | string> = [];

// Déclaration d'un tableau avec des éléments initiaux :
let tabInitNumber = [0, 1, 2, 3];
let tabInitMixte = ["0", 1, "2"];
let tabDefInit: Array<string | number> = [1, "2"];

// Déclaration d'un tableau de tableaux :
let matrice: number[][] = [[0, 1, 2], [10, 11, 12], [20, 21, 22]];

// Tableau initialisé avec des éléments vides :
let tableauTailleFixe: number[] = new Array<number>(5);
```

Les Map :

Les **Map** ou dictionnaires ou tableaux associatifs sont des conteneurs de paires *clés/valeurs*. Ils sont mutables, itérables et dynamiques. Ils conservent l'ordre d'insertion des éléments, mais n'est pas indexable.

```
// Déclaration d'une Map vide :
let mapVide: Map<string, string> = new Map<string, string>();

// Déclaration d'une Map avec des éléments initiaux :
let mapInit: Map<string, string> = new Map<string, string>([
  ["nom", "Bob"],
  ["age", "27"],
  ["ville", "Lyon"]
]);
```

Les Set :

Les **Set** que nous pourrions aussi appeler tuples ou ensembles sont des collections mutables, itérables mais non indexables.

```
// Déclaration d'un Set vide :
let ensemble: Set<string | number> = new Set<string | number>();

// Déclaration et initialisation d'un Set :
let ensembleMixte: Set<string | number> = new Set<string | number>([1, "deux", 3, "quatre"]);
```

Définition des fonctions :

On retrouve la syntaxe de base de **JavaScript** enrichie des types pour les paramètres et le retour de la fonction.

Les fonctions régulières :

```
function idFonction(idPara: type): type {
    // Corps de la fonction
}
```

Exemples :

Une fonction prenant 2 arguments et ne comportant pas de retour :

```
// Définition de la fonction somme(a: number, b: number): void
function somme(a: number, b: number): void {
    console.log(`\$ {a} + \$ {b} = \$ {a + b}`);
}

// Appel de la fonction :
somme(2, 3);
```

Variante de la fonction précédente qui retourne la somme :

```
// Définition de la fonction somme(a: number, b: number): number
function somme(a: number, b: number): number {
    return a + b;
}

// Appel de la fonction :
console.log(`2 + 3 = \$ {somme(2, 3)} `);
```

Les fonctions anonymes :

D'un point de vue syntaxique, une fonction anonyme est très similaire à la fonction régulière ne présentant pas d'identifiant.

```
// Définition de la fonction (a: number, b: number): number
let somme = function (a: number, b: number): number {
    return a + b;
}

// Utilisation de la fonction :
console.log(`2 + 3 = \$ {somme(2, 3)} `);
```

Les fonctions fléchées :

Dernière variante de fonction, les fonctions fléchées :

```
// Définition de la fonction (a: number, b: number): number
let somme = (a: number, b: number): number => {
    return a + b;
}

// Appel de la fonction :
console.log(`2 + 3 = ${somme(2, 3)}`);
```

La P.O.O. avec TS :

TypeScript s'appuie la syntaxe orientée classe de *JavaScript* avec évidemment l'ajout des types pour les attributs/propriétés et méthodes (arguments et retours).

Elément de base pour définir une classe :

Un exemple sur une classe *Personne* sera plus explicite :

```
class Personne {
    // Déclaration des propriétés/attributs :
    nom: string;
    prenom: string;
    age: number

    // Constructeur :
    constructor(nom: string, prenom: string, age: number) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
    // Méthode :
    presenter(): string {
        return `Bonjour, mon nom est ${this.prenom} ${this.nom} et j'ai ${this.age} ans.`;
    }
}
```

Définitions des getters et setters (accesseurs & mutateurs) :

Voici un second exemple avec une classe *Rectangle*. Cet exemple permet de voir comment procéder pour réaliser des *getters* et *setters*. On utilise également le modificateur d'accessibilité **private** pour rendre inaccessible les propriétés/attributs endehors du corps de la classe/objet.

```
class Rectangle {
    // Déclaration des propriétés/attributs :
    private _longueur: number;
    private _largeur: number;

    // Constructeur :
    constructor(longueur: number, largeur: number) {
        this._longueur = longueur;
        this._largeur = largeur;
    }
    // Getters & setters :
    get longueur() {
        return this._longueur;
    }
    set longueur(longueur) {
        this._longueur = longueur;
    }
    get largeur() {
        return this._largeur;
    }
    set largeur(largeur) {
        this._largeur = largeur;
    }
    get surface(): number {
        return this._longueur * this._largeur;
    }
    get perimetre(): number {
        return 2 * (this._longueur + this._largeur);
    }
}
```

L'héritage :

Pour mettre en place une relation d'héritage nous aurons besoin de 2 mots réservés à cet usage :

- **extends** : Pour établir la relation d'une classe fille vers sa classe mère
- **super** : Pour accéder au constructeur, méthodes et attributs de la classe mère :
 - **super()** : Accès au constructeur de la classe mère.
 - **super.idMethode()** : Accès à une méthode de la classe mère.
 - **super.idAttribut** : Accès à un attribut de la classe mère.

Exemple : Une classe **Etudiant** qui hérite de la classe **Personne**

```
class Etudiant extends Personne {
    filiere: string;
    niveau: number;

    constructor(nom: string, prenom: string, age: number, filiere: string, niveau: number) {
        super(nom, prenom, age); // Appel du constructeur de la classe mère
        this.filiere = filiere;
        this.niveau = niveau;
    }

    // Surcharge de La méthode presenter :
    presenter(): string {
        return super.presenter() + ` Je suis en année ${this.niveau} de la filière ${this.filiere}`;
    }
}
```

Les interfaces :

Les interfaces nous permettent de définir les *contrats* des classes à développer en stipulant la structure de base des classes (méthodes et attributs à implémenter).

Pour illustrer la démarche nous allons reprendre la classe *Rectangle* en la faisant implémenter une interface *FormeGeo*.

Définition de l'interface :

```
interface FormeGeo {
    perimetre: number;
    surface: number;
}
```

Implémentation de l'interface avec la classe Rectangle :

```
class Rectangle implements FormeGeo {
    // Déclaration des propriétés/attributs :
    private _longueur: number;
    private _largeur: number;

    // Constructeur :
    constructor(longueur: number, largeur: number) {
        this._longueur = longueur;
        this._largeur = largeur;
    }
    // Getters & setters :
    // Voir l'implémentation réalisée précédemment ...

    get surface(): number {
        return this._longueur * this._largeur;
    }
    get perimetre(): number {
        return 2 * (this._longueur + this._largeur);
    }
}
```

Implémentation de l'interface avec la classe Cercle :

```
class Cercle implements FormeGeo {
    private _rayon: number;

    constructor(rayon: number) {
        this._rayon = rayon;
    }
    set rayon(rayon) {
        this._rayon = rayon;
    }
    get rayon() {
        return this._rayon;
    }
    get perimetre() {
        return 2 * Math.PI * this._rayon;
    }
    get surface() {
        return Math.PI * this._rayon ** 2;
    }
}
```

Travaux pratiques

Mise en situation et objectifs :

Nous allons réaliser une ébauche d'application bancaire. Le principal objectif est de vous exercer à rédiger du code typé avec **TypeScript**. Pour cette ébauche nous nous contenterons d'un fichier unique que vous pourrez par exemple nommer **gestionBancaire.ts**.

La classe CompteCourant :

Ebauche de la classe CompteCourant :

C	CompteCourant
+numCompte:	number
-solde:	number
+limiteDecouvert:	number
+crediter(montant: number):	void
+debiter(montant: number):	boolean

Travail à faire :

- Implémenter la classe **CompteCourant**, l'attribut **solde** doit être privé (modificateur **private**). Il faudra donc penser à implémenter un **getter** (accesseur) pour accéder au solde. Le constructeur initialisera des différents attributs dont le solde initial (somme versée à l'ouverture du compte).
- Le solde ne comportera pas de **setter** (modificateur), son évolution dépendra des méthodes **crediter(montant: number)** et **debiter(montant: number): boolean**.
- La méthode **debiter(montant)** autorisera le débit qu'à la condition que le solde du compte reste supérieur ou égal à la limite de découvert autorisé.
- La méthode **debiter(montant)** retourne un booléen qui sera à **true** si l'opération de débit est effective et retournera **false** si l'opération de débit n'a pu être réalisée en raison d'un solde suffisant.
- Instancier un compte dans votre code et tester le comportement de l'objet compte et tout particulièrement des méthodes **crediter** et **debiter**.

Les classes Personne et Client :

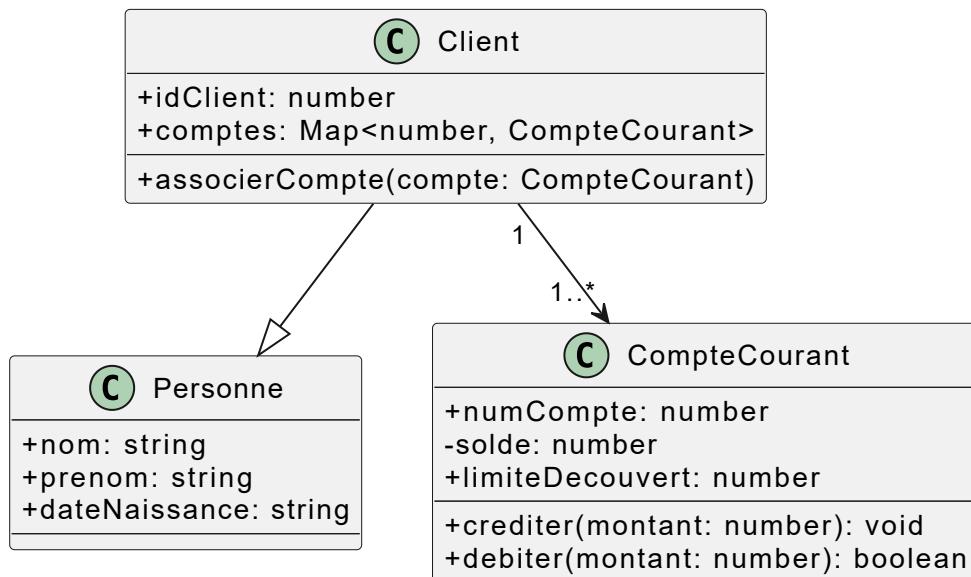
Vous allez maintenant implémenter la classe **Client** et **Personne**. La classe **Client** est une classe fille de la classe mère **Personne** (relation de généralisation). Nous mettons ici en oeuvre le mécanisme d'héritage, car dans un développement ultérieur, nous pouvons imaginer d'autres classes qui hériterait de la classe **Personne** : **Guichetier**, **GestionnaireCompte**, etc.

Travail à faire :

- Implémenter les classes **Personne** et **Client** selon le modèle ci-dessous.
- L'attribut **comptes** est un **Map** qui contiendra des paires où la clé est le numéro du compte (attribut **numCpt**) et la valeur la référence objet du compte.
- La méthode **associerCompte** prend en charge l'ajout d'un compte à l'attribut **comptes**.

Remarque : Ici on fait l'hypothèse qu'un client peut disposer de plusieurs comptes courants. Par contre on considère qu'un compte n'appartiendra qu'à un seul compte (pas de notions de *comptes joints* ou des comptes collectifs pour une entreprise, association, etc.).

Classes Personne et Client :



Enrichissement de la classe Personne :

L'âge du client peut influencer les conditions tarifaires de la gestion des comptes. Par exemple les mineurs ou jeûne (exemple âge maxi de 21 ans pour avoir le statut jeûne) sont exonérés de frais de gestion de compte.

Ajouter à la classe **Client** la méthode **age(): number** qui calcule l'âge courant du client à partir de la date du système et de la date de naissance. Nous vous laissons faire vos recherches pour trouver la solution...

Classe Operation :

Dans le domaine du métier bancaire nos désignons par **opération** toute action de **débit** ou **crédit** d'un compte. L'opération est caractérisée par une **date**, un **montant**. Le signe du montant caractérise la nature de l'opération (montant positif correspond à un crédit sur le compte, alors qu'un montant négatif caractérise un débit).

Travail à faire :

Implémenter (pour le moment sans aucune relation/association avec les autres classes) la classe **Operation** :

Classes Operation :

C	Operation
+date:	string
+montant:	number
+description:	string

Maintenant nous aimerons pour établir l'historique des opérations (**débits** et **crédits**) et réaliser sur les comptes. L'historique pourrait correspondre à une entité à part entière, mais pour le moment nous allons prendre le parti d'ajouter à la classe **CompteCourant** un nouvel attribut : **historique: Array<Operation>** qui est un tableau (**Array**) qui l'ensemble de références d'objets de type **Operation**. Le choix d'un tableau semble pertinent, car ce dernier respecte la chronologie des opérations. Les objets **Operation** contenu dans le tableau/attribut **historique** est pris en charge par les méthodes **crediter(montant: number)** et **debiter(montant: number)**.

Travail à faire :

Mettre à jour la classe **CompteCourant** en ajoutant l'attribut **historique** et en apportant les modifications nécessaires aux méthodes **debiter(montant: number)** et **crediter(montant: number)**

Ebauche de la classe CompteCourant :

C	CompteCourant
+numCompte:	number
-solde:	number
+limiteDecouvert:	number
+historique:	Array<Operation>
+crediter(montant: number):	void
+debiter(montant: number):	boolean

Note : La relation que nous venons d'établir entre les classes *Operation* et *CompteCourant* est désignée par relation de *composition*. On dit que cette relation est une relation forte, car l'ensemble des références des objets (composant) *Operation* sont uniquement référencé par les objets (composites) *CompteCourant*

Elaborer et implémenter un scénario de test pour vérifier le bon fonctionnement de votre code.