

---

# Langage typescript

(avec annexes  
es2015/es2017)

## Table des matières

I - Typescript (utilisation, bases , ....)	4
1. Typescript : présentation , utilisation	4
2. Bases syntaxiques du langage typescript (ts)	13
II - Typescript objet	16
1. Programmation objet avec typescript (ts)	16
III - Lambda , Generics , ... / typescript	27
1. Prog. fonctionnelle (arrow function / lambda, ...)	27
2. Generics de typescript (ts)	29

<b>IV - Modules "typescript" .....</b>	<b>30</b>
1. Namespaces et Modules typescript (ts).....	30
<b>V - Librairies de définitions (d.ts).....</b>	<b>38</b>
1. Modules/librairies de définitions (.d.ts).....	38
<b>VI - Aspects divers et avancés de typescript.....</b>	<b>46</b>
1. Aspects divers et avancés de typescript (ts).....	46
<b>VII - Annexe – Arrow functions, essentiel es2015.....</b>	<b>56</b>
1. Mots clefs "let" et "const" (es2015).....	56
2. "Arrow function" et "lexical this" (es2015).....	57
3. for...of (es2015) utilisant itérateurs internes.....	58
<b>VIII - Annexe – Objets es2015 (class , extends, ...).....</b>	<b>64</b>
1. Prog. orientée objet "es2015" (class, extends, ...).....	64
<b>IX - Annexe – Promise es2015 , async/await es2017.....</b>	<b>72</b>
1. Promise (es2015).....	72
2. async/await (es2017).....	79
<b>X - Annexe – Modules es2015.....</b>	<b>83</b>
1. Modules (es2015).....	83
<b>XI - Annexe – aspects divers et avancés es2015.....</b>	<b>91</b>
1. Aspects divers et avancés (es2015).....	91
<b>XII - Annexe – webPack.....</b>	<b>105</b>
1. Webpack.....	105
<b>XIII - Annexe – RxJs.....</b>	<b>114</b>
1. introduction à RxJs.....	114
2. Fonctionnement (sources et consommations).....	115
3. Réorganisation de RxJs (avant et après v5,v6).....	115
4. Sources classiques générant des "Observables".....	117
5. Principaux opérateurs (à enchaîner via pipe).....	119
6. Passerelles entre "Observable" et "Promise".....	121

---

XIV - Annexe – Références , TP.....	122
1. Références "typescript".....	122
2. TP "typescript".....	122

# I - Typescript (utilisation, bases , ....)

## 1. Typescript : présentation , utilisation

### 1.1. Rappel: Version de javascript (ES5 ou ES6/es2015)

Les versions standardisées/normalisées de javascript sont appelées ES (EcmaScript) .

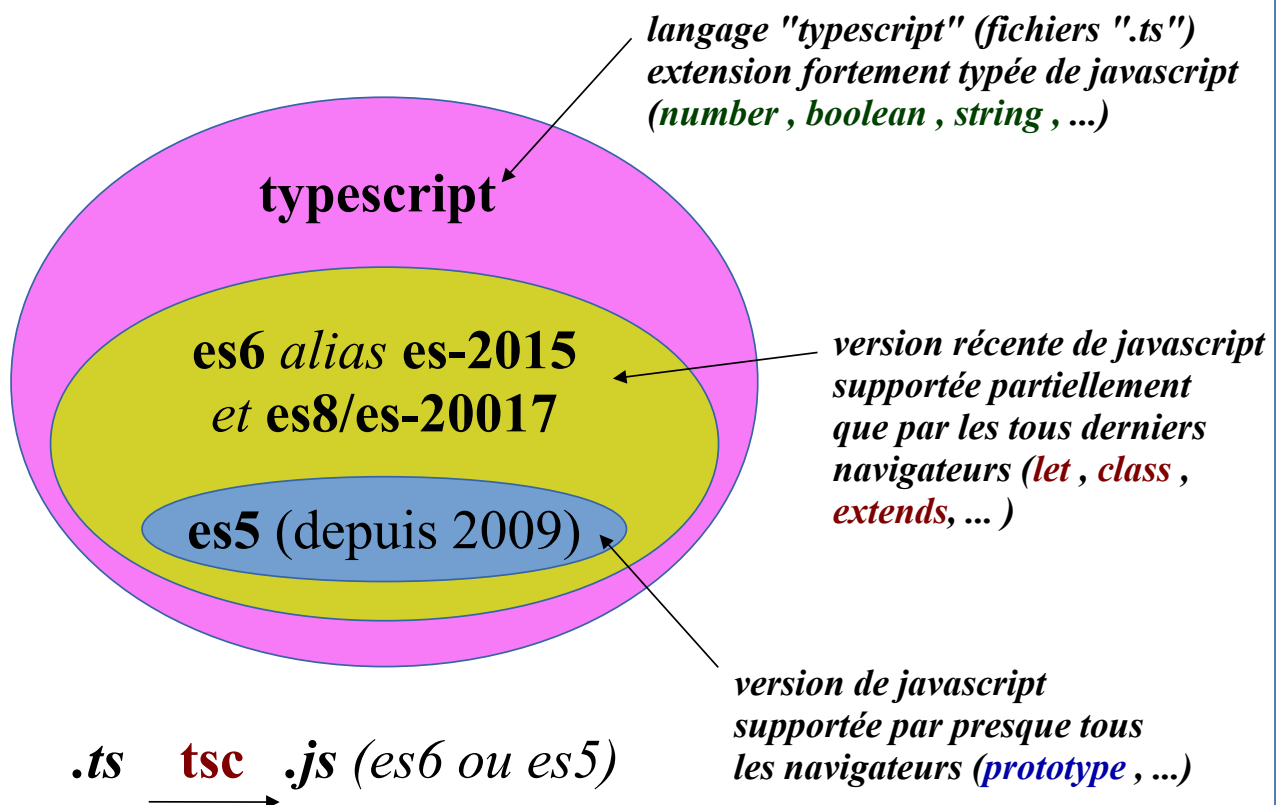
Les versions modernes sont :

- **ES5** (de 2009) – supporté par quasiment tous les navigateurs actuels ("mobiles" ou "desktop")
- **ES6** (renommé **ES2015** car normalisé en 2015) . ES6/es2015 n'est pour l'instant supporté que par quelques navigateurs "desktop" récents.  
ES6/ES2015 apporte quelques nouvelles syntaxes et mots clefs (**class** , **let** , ...) et gère des modules dits "statics" via "**import** { ComponentName } **from** 'moduleName' ;" et **export** .
- **ES2017** a apporté **async/await** en tant que nouvelles fonctionnalités importantes

En 2016, .... , 2019, .... , une application basée sur "typescript" doit être compilée/transpilée en ES5 de façon à pouvoir s'exécuter sur n'importe quel navigateur.

### 1.2. Présentation de TypeScript / ts

#### Fonctionnalités "es5" , "es6" et "typescript"



"Typescript" est un **langage de programmation libre et open-source** qui vise à améliorer la programmation javascript en apportant plus de rigueur ( typage fort optionnel) et une approche orientée objet .

"Typescript" est un **sur-ensemble de JavaScript** (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript)

TypeScript supporte les spécifications de **ES6** (ECMAScript 6 ) : il peut être vu comme une évolution de ES6/ES2015 et de ES8/ES2017 .

**Le langage "typescript" a été créé par Microsoft** et particulièrement par Anders Hejlsberg le principal inventeur de c# .

Il est depuis utilisé dans d'autres environnements et notamment par **Angular2+ (2,4,5,6,7,...) de Google**.

Ce langage s'utilise concrètement en écrivant des fichiers ".ts" qui sont généralement transformés en fichiers ".js" (ES5 ou ES6) via un pré-processeur "**tsc**" (typescript compiler) .

Cette phase de transcription (".ts → .js") s'effectue généralement durant la phase de développement.

Comportement de tsc: tant que l'on ajoute pas de spécificités "typescript" , le fichier ".js" généré est identique au fichier ".ts" .

Si par contre on ajoute des précisions sur les types de données au sein du fichier ".ts" alors :

- le fichier ".js" est bien généré (par simplification ou développement) si aucune erreur bloquante est détectée.
- des messages d'erreurs sont émis par "tsc" si des valeurs sont incompatibles avec les types des paramètres des fonctions appelées ou des affectations de variables programmées.

### **(!!!) Attention , attention :**

Pour l'instant la plupart des moteurs d'interprétation de javascript (des navigateurs , de nodeJs, ...) ne tiennent pas encore compte des types ( :number , :string , :boolean , ...) de l'extension "typescript" .

Les précisions de type ( :number , :string , :boolean , ...) des fichiers ".ts" sont ainsi "perdues" dans les fichiers ".js" générés et **des incohérences de types peuvent alors éventuellement avoir lieu au moment de l'exécution** .

Bien que déclarée de type "**:number**" , le contenu d'une variable *x* saisie numériquement dans un `<input />` sera quelquefois (selon le framework) récupérée comme la valeur **123** ou bien **"123"** .

Et *x* + 2 vaudra alors **125** ou **"1232"** .

Et **Number(x)** + 2 vaudra toujours **125** .

La **rigueur** supplémentaire apportée par typescript porte donc essentiellement sur des **contrôles effectués sur le code source par des outils de développement (éditeurs et/ou compilateurs sophistiqués)** . Des incohérences de types seront par exemple détectées au niveau des paramètres d'entrée et/ou des valeurs de retour lors d'un appel de fonction .

**Autre très grand intérêt de typescript:** dans le cadre (très fréquent) d'une approche orientée objet bien structurée , l'utilisation d'une variable dont le type "classe d'objet" est connu/déclaré mènera souvent l'éditeur à proposer spontanément une liste d'attributs ou méthodes possibles (*auto-complétion intelligente*) .

### 1.3. Historique/évolution de typescript (versions)

Version number	Release date	Significant changes
<b>0.8</b>	<b>1 October 2012</b>	
0.9	18 June 2013	
1.1	6 October 2014	performance improvements
1.3	12 November 2014	<b>protected</b> modifier, tuple types
1.4	20 January 2015	union types, <b>let</b> and <b>const</b> declarations, template strings, type guards, type aliases
<b>1.5</b>	<b>20 July 2015</b>	<b>ES6 modules</b> , <b>namespace</b> keyword, <b>for .. of</b> support, <b>decorators</b>
<b>1.7</b>	<b>30 November 2015</b>	<b>async and await</b> support,
2.0	22 September 2016	null- and undefined-aware types, control flow based type analysis, discriminated union types, <b>never</b> type, <b>readonly</b> keyword, type of <b>this</b> for functions
2.1	8 November 2016	<b>keyof</b> and lookup types, mapped types, object spread and rest,
<b>2.2</b>	<b>22 February 2017</b>	<b>mix-in classes</b> , <b>object type</b> ,
3.0	30 July 2018	project references, extracting and spreading parameter lists with tuples
<b>3.4</b>	<b>29 March 2019</b>	faster incremental builds, type inference from generic functions, <b>readonly</b> modifier for arrays, <b>const</b> assertions, type-checking <b>globalThis</b>

La version 1.7 (datant de fin 2015) est déjà très complète .  
Depuis : de simples éléments peaufinés (mix-in,...)

### 1.4. Environnement de développement (typescript)

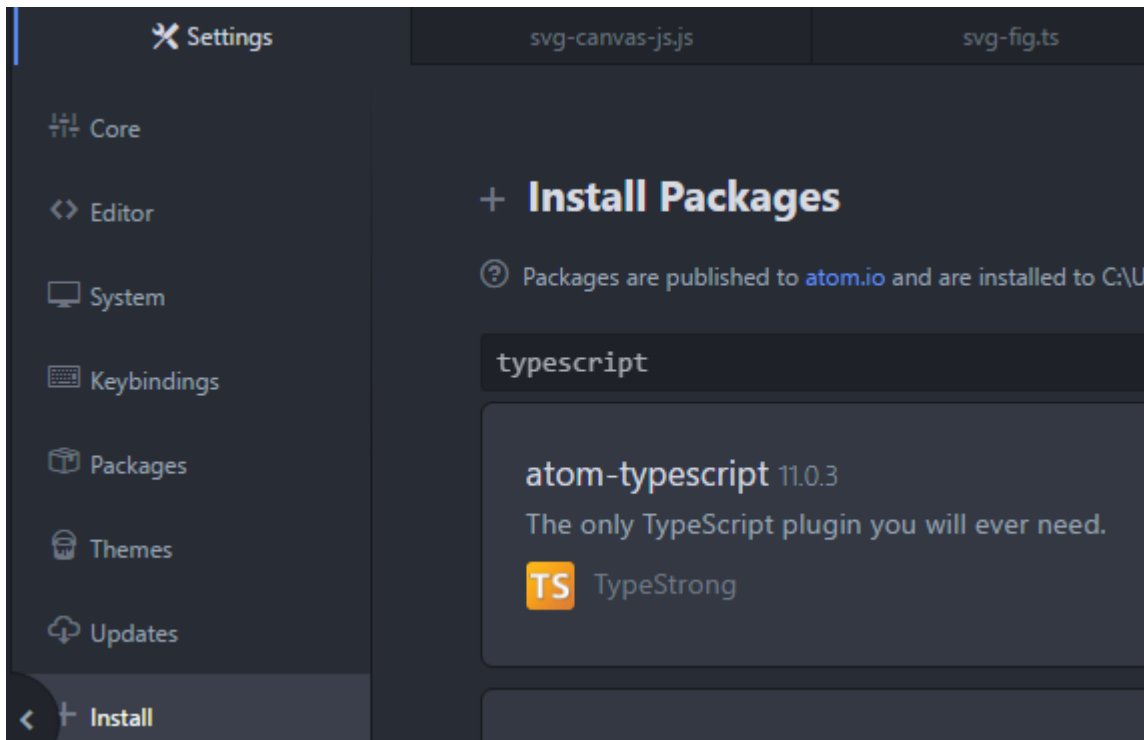
**Typescript** étant à l'origine créé par **Microsoft**, il est possible d'utiliser l'IDE "**Visual Studio**" ou "**Visual Studio Code**" de Microsoft (.net / c# , ...) de façon à programmer en "typescript" (éditeur et compilateur bien intégrés) .

D'autres environnement comportent des "plugins" pour travailler sur des fichiers ".ts" (en langage typescript) :

- plugin(s) typescript pour "**sublime text**" (installation et paramétrages complexes)
- plugin(s) typescript pour "**eclipse**"
- ....
- malheureusement : pas grand chose pour npp (nodepad++)

Actuellement, l'un des bons éditeurs "typescript" gratuit est "**Atom + plugin typescript**" .  
Cet éditeur (facile à installer) gère :

- la colorisation syntaxique (mot clefs , ...)
  - l'analyse des erreurs de syntaxes
  - l'auto-complétion
- et il fonctionne aussi bien sur *windows* et *linux* .



**webstorm** est également *un très bon éditeur "typescript" payant* . son grand frère *intelliJ* (dédié au monde java) est également *payant* et offre un très bon environnement "typescript + java" pour des développement "full stack" angular + java/spring .

Sur des machines "windows" , l'éditeur **gratuit "Visual Studio Code"** est très bien approprié pour effectuer des développement "typescript" (angular par exemple) .

De façon à télécharger et lancer le compilateur "tsc" , on pourra (entre autres possibilités) s'appuyer sur nodejs/npm .

```
npm install -g typescript
```

La version **2.3.2** est une des versions de **typescript** (mi 2017) .

```
npm install -g typescript@latest
```

version plus récente mai 2019 --> **3.4.5**

## 1.5. utilisation indirecte de tsc via npm

Exemple de projet "node" basé sur "typescript" :

**tp-typescript/package.json**

```
{
  "name": "tp-typescript",
  "version": "1.0.0",
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "tsc:tES5": "tsc -t ES5",
    "start": "npm run tsc:w"
  },
  "license": "ISC",
  "dependencies": {
  },
  "devDependencies": {
    "typescript": "^2.3.1"
  }
}
```

**cd tp-typescript; npm install**

Le déclenchement de la **transcription/compilation de "ts" vers "js"** peut s'effectuer de différentes manières :

- via une ligne de commande (éventuellement placée dans un script) :

**compile\_ts\_to\_js.bat**

```
REM npm run tsc hello_world.ts
REM      avec option -w signifiant "watch mode" (npm run tsc:w) , recompilation dès qu'une
REM      modification est détectée sur fichier .ts enregistré
npm run tsc:w hello_world.ts
```

- via le "task runner" Grunt.ts :
- ...

## 1.6. Générer le fichier tsconfig.json

**tsc --init**

---> cela génère un fichier de configuration avec plein de paramètres possibles en commentaires

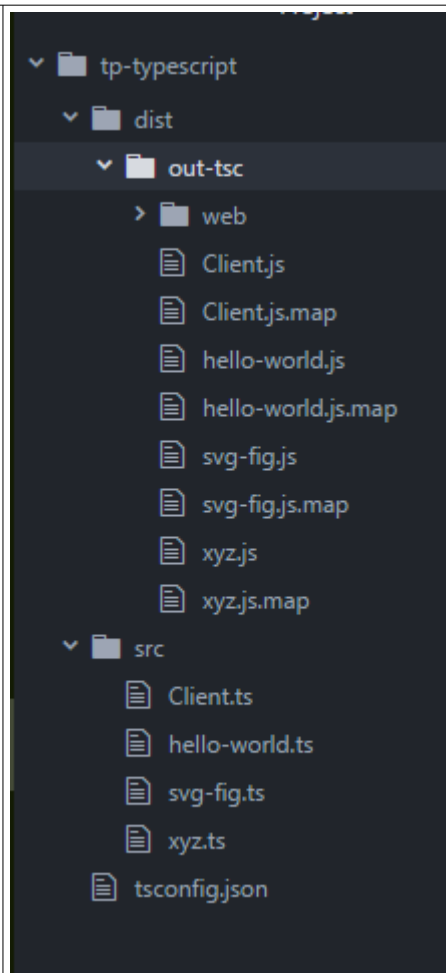
## 1.7. Lancement du mode watch depuis visual-studio-code

**Ctrl+Shift+B** et sélectionner **"tsc: watch - tsconfig.json"**



## 1.8. utilisation directe de tsc (paramétré par tsconfig.json)

Après une installation globale effectuée par `"npm install -g typescript"` , une ligne de commande en `"tsc"` ou `"tsc -w"` lancée depuis le répertoire d'un projet où est présent le fichier `tsconfig.json` suffit à lancer toute une série de compilations "typescript" :

<p><b>tsconfig.json</b></p> <pre>{   "compileOnSave": true,   "compilerOptions": {     "baseUrl": "",     "declaration": false,     "emitDecoratorMetadata": false,     "experimentalDecorators": false,     "outDir": "dist/out-tsc",     "sourceMap": true,     "target": "es5",     "noEmitOnError" : false   },   "include": [     "src/**/*"   ],   "exclude": [     "node_modules",     "**/*.spec.ts",     "dist"   ] }</pre>	
--	---

**NB :** l'option `"compileOnSave": true` est supportée par certains IDE sophistiqués (ex : Atom) et est équivalente à l'option `"-w"` (watch mode)

Lorsque l'option `"noEmitOnError" : true` est fixée , le fichier ".js" n'est pas généré en cas d'erreur (pourtant ordinairement non bloquante) soulevée par tsc .

## 1.9. Source map (.map)

Avec l'option `"sourceMap": true` , le compilateur/transpilateur "tsc" génère des fichiers `".map"` à coté des fichiers ".js".

Ces fichiers dénommés **"source map"** servent à effectuer des correspondances entre le code source original ".ts" et le code transformé ".js" .

Ces fichiers ".map" sont quelquefois chargés et interprétés par des débogueurs sophistiqués (ex : debug de VisualStudio , ...).

Ces fichiers ".map" ne sont pas indispensables pour l'interprétation du code ".js" généré et peuvent généralement être omis en production .

## 1.10. Quelques utilisations du code traduit en javascript

### launch\_nodeJs\_app.bat

```
REM node -v
node hello_world.js
REM node xyz.js
pause
```


→ Un **lancement via node (nodeJs)** est très **pratique** pour effectuer **quelques petits essais en mode texte**

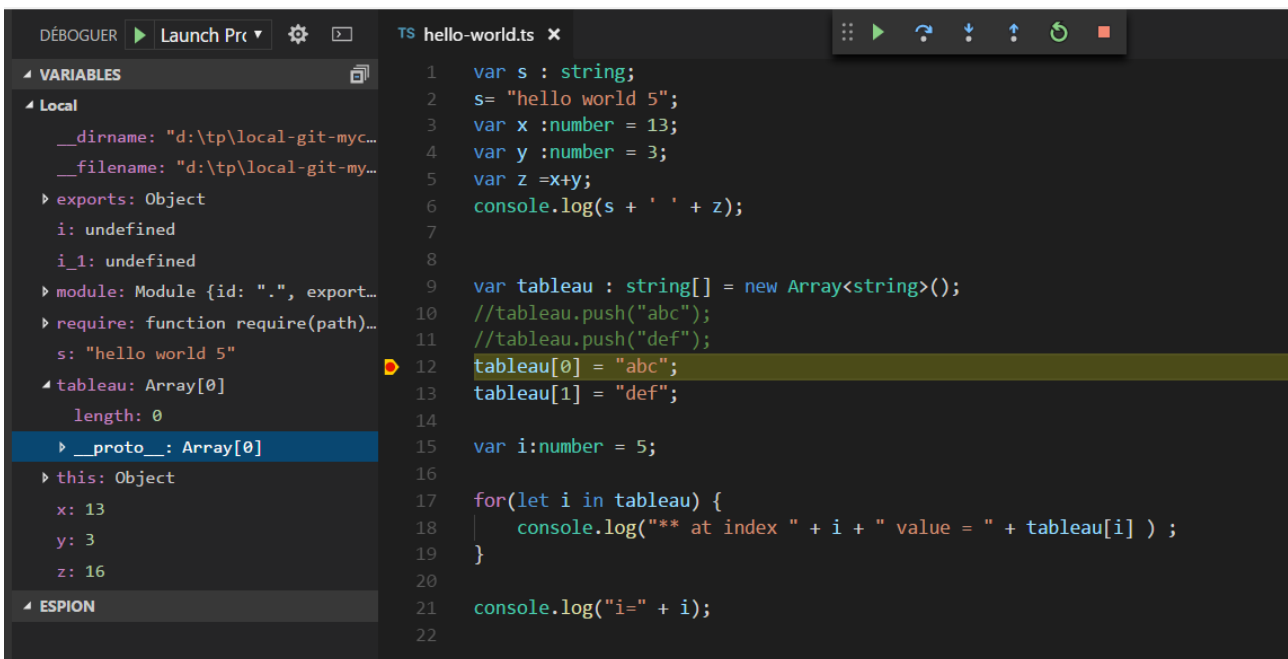
→ Une application "sérieuse" aura un cadre/contexte au cas par cas (ex : Angular 2+ et projet basé sur @angular/cli ) .

→ Etant donné qu'un fichier typescript (.ts) est transformé en javascript (.js) , **il est possible de mixer du code "javascript" et "typescript"** (par exemple dans le cadre d'une application "web").

Par exemple , au sein d'un navigateur web (IE/Edge , Chrome, Firefox, ... ) , on peut charger une page HTML avec du code js/jQuery qui pilote certaines parties traduites de ".ts" en ".js" .


## 1.11. Configuration de Visual Studio code pour un debug pas à pas de code ".ts" traduit en ".js" s'exécutant dans un navigateur

Après avoir placé un point d'arrêt sur une ligne intéressante d'un fichier .js ou .ts (en cliquant dans la marge), le mode debug de "Visual Studio code" apparaît en cliquant sur l'icône  .



--> pas à pas classique avec "variables" et "watch expression" .

Visual studio code peut directement s'interfacer avec l'environnement d'exécution node (nodeJs) sans plugin supplémentaire .

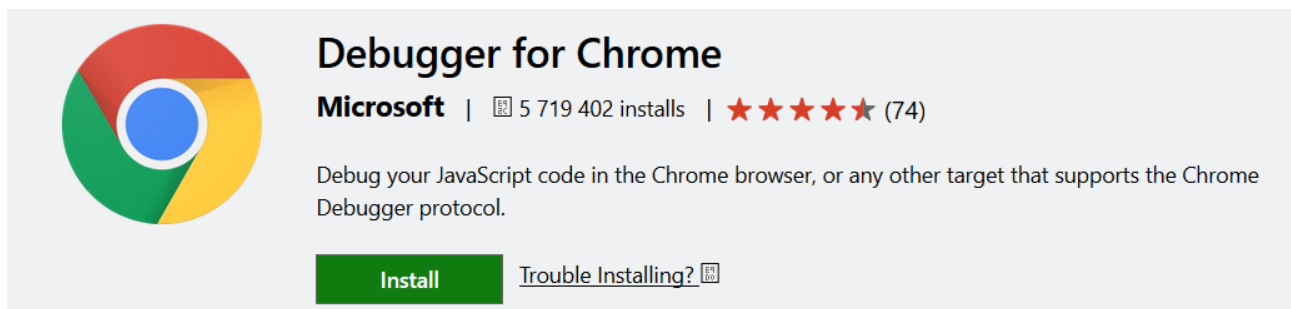
Pour bien paramétrer les chemins menant au code de l'application à exécuter et déboguer on peut ajuster le fichier suivant (quelquefois créé lors du premier lancement via l'icône  ).

### **.vscode/launch.json**

```
{
  // "${workspaceRoot}/out/**/*.js" --> "${workspaceRoot}/dist/out-tsc/**/*.js"
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${file}",
      "outFiles": [
        "${workspaceRoot}/dist/out-tsc/**/*.js"
      ]
    }
  ]
}
```

NB : les fichiers ".map" sont alors automatiquement utilisés pour établir des correspondances entre les lignes ".js" qui s'exécutent et les lignes ".ts" du code source .

De manière à utiliser le mode debug de Visual Studio code en mode web (exécution du code javascript/typescript dans un navigateur) , il faut installer le plugin suivant :



dans visual studio code .

Une fois le plugin installé (et après un éventuel arrêt/relance de **V**isual **S**tudio **C**ode) , on pourra soit :

- configurer VSC de façon à lancer automatiquement le navigateur "google Chrome" en mode debug
- configurer VSC pour communiquer avec une instance de "google Chrome" déjà démarrée.

De manière à configurer un lancement automatique du navigateur "google chrome" via le mode "debug" d'un projet VSC , on pourra ajuster les paramètres "url" et "webRoot" du fichier suivant :

#### **.vscode/launch.json**

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch localhost",
      "type": "chrome",
      "request": "launch",
      "url": "file:///D:/tp/tp_typescript/bases-typescript-web/dist/web/simple.html",
      "webRoot": "${workspaceFolder}/dist/web"
    },
    {
      "name": "Launch index.html (disable sourcemaps)",
      "type": "chrome",
      "request": "launch",
      "sourceMaps": false,
      "file": "${workspaceFolder}/index.html"
    }
  ]
}
```

NB : l'url pourra être en "http://" (avec un serveur HTTP configuré)

---

En mode "rattachement à une instance de google chrome déjà lancée" , il faut :

- que le navigateur "**google chrome**" soit lancé avec l'option **--remote-debugging-port=9222** (en configurant par exemple les "propriétés d'un raccourci windows" )
- placer **request : "attach"** plutôt que **request : launch** dans **.vscode/launch.json**

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "Attach",
      "type": "chrome",
      "request": "attach",
      "port": 9222,
      "url": "<url of the open browser tab to connect to>",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

## 2. Bases syntaxiques du langage typescript (ts)

### 2.1. précision des types de données

<b>boolean</b>	<code>var isDone: boolean = false;</code>
<b>number</b>	<code>var height: number = 6;</code> <code>var size : number = 1.83 ;</code>
<b>string</b>	<code>var name: string = "bob";</code> <code>name = 'smith';</code>
<b>array</b>	<code>var list1 : number[] = [1, 2, 3];</code> <code>var list2 : Array&lt;number&gt; = [1, 2, 3];</code>
<b>enum</b>	<code>enum Color {Red, Green, Blue}; // start at 0 by default</code> <code>// enum Color {Red = 1, Green, Blue};</code>  <code>var c: Color = Color.Green; //display as "1" by default</code> <code>var colorName: string = Color[1]; // "Green" if "Red" is at [0]</code>
<b>any</b>	<code>var notSure: any = 4;</code> <code>notSure = "maybe a string instead";</code> <code>notSure = false;</code>
<b>void</b>	<code>function warnUser(): void {</code> <code>alert("This is my warning message");</code> <code>}</code>
<b>object</b>	(objet quelconque : plus précis que "any" , moins précis qu'un nom de classe). <code>var obj : object = { id : 2 , label : "cahier" } ;</code> <code>obj = { prenom : "jean" , nom : "Bon" } ; //structure objet différente acceptée .</code>

#### hello\_world.ts

```
function greeterString(person : string) {
    return "Hello, " + person;
}
var userName = "Power User";
//i=0; //manque var (erreur détectée par tsc)

var msg = "";
//msg = greeterString(123456); //123456 incompatible avec type string (erreur détectée par tsc)
msg = greeterString(userName);
console.log(msg);
```

values: number[] = [1,2,3,4,5,6,7,8,9];

## 2.2. Quelques éléments essentiels issus de javascript "es5"

**Number**("123px") retourne NaN tandis que **parseInt**("123px") retourne 123 .

---

Remarque importante : une **variable non initialisée** est considérée comme "**undefined**" (notion proche de "null") et ne peut pas être utilisée en tant qu' objet préfixe .

---

L'opérateur **typeof** *variable* retourne une chaîne de caractère de type "string" , "number" , "boolean" , "undefined" , .... selon le type du contenu de la variable à l'instant t .

```
var vv ;  
if( typeof vv == "undefined" ) {  
    console.log("la variable vv n'est pas initialisée") ;  
}  
  
if( vv == null ) {  
    console.log("la variable vv est soit null(e) soit non initialisée") ;  
}
```

---

L'opérateur == (d'origine c/c++/java) retourne true si les 2 expressions ont des valeurs à peu près équivalente (ex 25 est une valeur considérée équivalente à "25") .

L'opérateur === (spécifique à javascript) retourne true si les 2 expressions ont à la fois les mêmes valeurs et le même type ("25" et 25 ne sont pas de même type) .

---

window.**setTimeout**(chExpr,n) permet d'interpréter l'expression chExpr en différé (n ms plus tard)

window.**setInterval**(chExpr,n) permet de lancer l'interprétation périodique de chExpr toutes les n ms;

---

```
var jsonString = JSON.stringify(jsObject) ;
```

```
var jsObject2 = JSON.parse(jsonString) ;
```

---

**delete** tab[i] ; //supprime la valeur de tab[i] qui devient **undefined** .

```
tab.splice(i , 2 , val1 , val2) ; //remplace tab[i] par val1 et tab[i+1] par val2 , etc  
tab.splice(i, 1) ; //remplace tab[i] par rien et donc supprime la case tab[i]  
//sans trou , certains autres éléments sont déplacés (changement d'indice)
```

## 2.3. Éléments essentiels issus de javascript "es6 / es2015"

- **arrow functions** (alias "*lambda expression*") ( exemple : `(p) => { console.log(p) ; }` )
- mots clefs "**let**" , "**const**" en plus de "**var**"
- nouvelle boucle **for (.. of )**
- *template string es2015* (avec *quotes inverses* et `${}`  )
- **Map** , **Set** , ...
- **Destructuring**
- ...

La plupart de ces éléments sont détaillés dans **l'annexe** "*arrow functions et essentiel es2015*" .

## 2.4. Syntaxes es5/es2015 améliorés par typage fort (typescript)

```
var tableau : string[] = new Array<string>();  
tableau.push("abc");  
....
```

## II - Typescript objet

### 1. Programmation objet avec typescript (ts)

#### 1.1. ressemblances et différences sur les objets "ts" et "es2015"

syntaxes orientées "objet" supportées de la même façon entre "typescript" et "es2015"	syntaxes orientées "objet" qui ne sont parfaitement supportées que par "typescript"
<ul style="list-style-type: none"> <li>• mots clefs <b>class</b> , <b>constructor</b></li> <li>• <b>static</b></li> <li>• mots clefs <b>get</b> , <b>set</b></li> <li>• héritage ( <b>extends</b> , <b>super</b> , ...)</li> <li>• <b>Object.assign(...)</b></li> </ul>	<ul style="list-style-type: none"> <li>• mots clefs <b>abstract</b> (classes abstraites) ,</li> <li>• <b>interface</b>, <b>implements</b>, ....</li> <li>• <b>public</b> , <b>private</b> , <b>protected</b> , ...</li> <li>• "public" , "private" ou "protected" au niveau des paramètres d'un constructeur pour définir automatiquement certaines variables d'instances (attributs)</li> </ul>

==> l'annexe "Objets es2015 (class , extends, ...)" montre une vision "es2015" de la programmation orientée objet .

Etant donné que tout code "javascript" s'interprète bien en "typescript" , tous les éléments de cette annexe sont exploitables tel quels en "typescript" .

Les principales extensions spécifiques "typescript" sont liées à la visibilité des membres d'une classe (private,public, protected) et aux types abstraits (classes abstraites , interfaces) .

La suite de ce chapitre sera basée sur une syntaxe "typescript" (avec typage fort).

#### 1.2. Classe et instances (typescript)

```
class Compte{
    numero : number;
    label : string;
    solde : number;

    debiter(montant : number) : void {
        this.solde -= montant; // this.solde = this.solde - montant;
    }

    crediter(montant : number) : void {
        this.solde += montant; // this.solde = this.solde + montant;
    }
}
```

```
var c1 = new Compte(); //instance (exemplaire) 1
```



```
console.log("numero et label de c1: " + c1.numero + " " + c1.label);  
console.log("solde de c1: " + c1.solde);  
var c2 = new Compte(); //instance (exemplaire) 2  
c2.solde = 100.0;  
c2.crediter(50.0);  
console.log("solde de c2: " + c2.solde); //150.0
```

NB: Sans initialisation explicite (via constructeur ou autre) , les propriétés internes d'un objet sont par défaut à la valeur **"undefined"** .

## 1.3. constructor

Un constructeur est une méthode qui sert à initialiser les valeurs internes d'une instance dès sa construction (dès l'appel à new) .

En langage typescript le constructeur se programme comme la méthode spéciale "**constructor**" (mot clef du langage) :

```
class Compte{
    numero : number;
    label: string;
    solde : number;

    constructor(numero:number, libelle:string, soldeInitial:number){
        this.numero = numero;
        this.label = libelle;
        this.solde = soldeInitial;
    }

    //...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
c1.crediter(50.0);
console.log("solde de c1: " + c1.solde);
```

NB: il n'est pas possible d'écrire plusieurs versions du constructeur :

```
constructor(numero:number, libelle:string, soldeInitial:number){
    this.numero = numero;
    this.label = libelle;
    this.solde = soldeInitial;
}
```

```
constructor(){
    this.=0;
    this.label="?";
    this.=0.0;
}
```

Il faut donc quasi systématiquement utiliser la syntaxe = *valeur\_par\_defaut* sur les arguments d'un constructeur pour pouvoir créer une nouvelle instance en précisant plus ou moins d'informations lors de la construction :

```
class Compte{
    numero : number;
    label: string;
    solde : number;

    constructor(numero:number=0, libelle:string="?", soldeInitial:number=0.0){
        this.numero = numero;
```

```

        this.label = libelle;
        this.solde = soldeInitial;
    } //...
}

```

```

var c1 = new Compte(1,"compte 1",100.0);
var c2 = new Compte(2,"compte 2");
var c3 = new Compte(3);
var c4 = new Compte();

```

Remarque : en plaçant le mot clef (récent) "**readonly**" devant un attribut (propriété) , la valeur de celui ci doit absolument être initialisée dès le constructeur et ne pourra plus changer par la suite.

## 1.4. Propriété "private"

```

class Animal {
    private _size : number;
    name:string;
    constructor(theName: string = "default animal name") {
        this.name = theName;
        this._size = 100; //by default
    }
    move(meters: number = 0) {
        console.log(this.name + " moved " + meters + "m." + " size=" + this._size);
    }
}

```

```
var a1 = new Animal("favorite animal");
```

**a1.\_size=120;** //erreur détectée ' \_size' est privée et seulement accessible depuis classe 'Animal'.

```
a1.move();
```

### Remarques importantes :

- **public par défaut .**
- En cas d'erreur détectée sur "private / not accessible" , le fichier ".js" est (par défaut) tout de même généré par "tsc" et l'accès à ".size" est tout de même autorisé / effectué au runtime.  
⇒ private génère donc des messages d'erreurs qu'il faut consulter (pas ignorer) !!!

## 1.5. Accesseurs automatiques `get xxx()` / `set xxx()`

```
class Animal {
    private _size : number;
    public get size() : number { return this._size;
    }
    public set size(newSize : number){
        if(newSize >=0) this._size = newSize;
        else console.log("negative size is invalid");
    }
    ...} //NB : le mot clef public est facultatif devant "get" et "set" (public par défaut)
```

```
var a1 = new Animal("favorite animal");
a1.size = -5; // calling set size() → negative size is invalid (at runtime) , _size still at 100
a1.size = 120; // calling set size()
console.log("size=" + a1.size) ; // calling get size() → affiche size=120
```

## 1.6. Mixage "structure & constructor" avec public ou private

```
class Compte{
    numero : number;
    label: string;
    solde : number;

    constructor(public numero : number=0,
                public label : string="?",
                public solde : number=0.0){
        this.numero = numero;
        this.label = label; this.solde = solde;
    } //...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
c1.solde = 250.0; console.log(c1.numero + ' ' + c1.label + ' ' + c1.solde );
```

Remarque importante : via le mot clef "**public**" ou "**private**" ou "protected" (au niveau des paramètres du constructeur) , **certaines paramètres passés au niveau du constructeur sont automatiquement transformés en attributs/propriétés de la classe** .

Autrement dit, toutes les lignes "barrées" de l'exemple précédent sont alors générées implicitement (automatiquement) .

## 1.7. mot clef "static"

De la même façon que dans beaucoup d'autres langages orientés objets (c++, java, ...) , le mot clef **static** permet de déclarer des variables/attributs de classes (plutôt que des variables/attributs d'instances).

La valeur d'un attribut "static" est partagée par toutes les instances d'une même classe et l'accès s'effectue avec le préfixe "NomDeClasse." plutôt que "this." .

Exemple:

```
class CompteEpargne {
    static taux: number = 1.5;

    constructor(public numero: number, public solde: number = 0){
    }

    calculerInteret(){
        return this.solde * CompteEpargne.taux / 100;
    }
}

var compteEpargne = new CompteEpargne(1,200.0);
console.log("interet="+compteEpargne.calculerInteret());
```

## 1.8. héritage et valeurs par défaut pour arguments:

```
class Animal {
    name: string;
    constructor(theName: string = "default animal name") { this.name = theName; }
    move(meters: number = 0) {
        console.log(this.name + " moved " + meters + "m.");
    }
}
```

```
class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 5) {
        console.log("Slithering...");
        super.move(meters);
    }
}
```

```
class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(meters = 45) {
        console.log("Galloping...");
        super.move(meters);
    }
}
```

```
var a = new Animal(); //var a = new Animal("animal");

var sam = new Snake("Sammy the Python"); //var sam = new Snake();

var tom: Animal = new Horse("Tommy the Palomino");

a.move() ; // default animal name moved 0m.
sam.move(); // Slithering... Sammy the Python moved 5m.

tom.move(34); //avec polymorphisme (for Horse)
// Galloping... Tommy the Palomino moved 34m.
```

NB: depuis la version 1.3 de typescript , le mot clef "**protected**" peut être utilisé dans une classe de base à la place de private et les méthodes des sous classes (qui hériteront de la classe de base) pourront alors accéder directement au attributs/propriétés "*protected*".

## 1.9. Classes abstraites (avec opérations abstraites)

```
...
// classe abstraite (avec au moins une méthode abstraite / sans code):
abstract class Fig2D {
  constructor(public lineColor : string = "black",
    public lineWidth : number = 1,
    public fillColor : string = null){
  }
  performVisit(visitor : FigVisitor) : void {}
  abstract performVisit(visitor : FigVisitor) : void ;
}

// classe concrète (avec du code d'implémentation pour chaque opération):
class Line extends Fig2D{
  constructor(public x1:number = 0 , public y1:number = 0 ,
    public x2:number = 0 , public y2:number = 0,
    lineColor : string = "black",
    lineWidth : number = 1){
    super(lineColor,lineWidth);
  }
  performVisit(visitor : FigVisitor) : void {
    visitor.doActionForLine(this);
  }
}
```

```
var tabFig : Fig2D[] = new Array<Fig2D>();
tabFig.push( new Fig2D("blue") ); //impossible d'instancier une classe abstraite
tabFig.push( new Line(20,20,180,200,"red") ); //on ne peut instancier que des classes concrètes
```

## 1.10. Interfaces

person.ts

```
interface Person {
  firstname: string;
  lastname: string;
}

function greeterPerson(person : Person) {
  return "Hello, " + person.firstname + " " + person.lastname;
}

//var user = {name: "James Bond", comment: "top secret"};
//incompatible avec l'interface Person (erreur détectée par tsc)

var user = {firstname: "James", lastname: "Bond", country: "UK"};
//ok : compatible avec interface Person

msg = greeterPerson(user);
console.log(msg);

class Student {
  fullname : string;
  constructor(public firstname, public lastname, public schoolClass) {
    this.fullname = firstname + " " + lastname + "[" + schoolClass + "]";
  }
}

var s1 = new Student("cancre", "Ducobu", "Terminale"); //compatible avec interface Person
msg = greeterPerson(s1);
console.log(msg);
```

Rappel important : via le mot clef "**public**" ou "**private**" (au niveau des paramètres du constructeur) , certains paramètres passés au niveau du constructeur sont automatiquement transformés en attributs/propriétés de la classe (ici "*Student*") qui devient donc compatible avec l'interface "*Person*" .

Au sein du langage "typescript", une **interface** correspond à la notion de "structural subtyping".  
Le véritable objet qui sera compatible avec le type de l'interface pourra avoir une structure plus grande.

### Interface simple/classique :

```
interface LabelledValue {  
  label: string;  
}
```

```
function printLabel(labelledObj: LabelledValue) {  
  console.log(labelledObj.label);  
}  
  
var myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);
```

### Interface avec propriété(s) facultative(s) (suffixée(s) par?)

```
interface LabelledValue {  
  label : string;  
  size? : number ;  
}
```

```
function printLabel(labelledObj: LabelledValue) {  
  console.log(labelledObj.label);  
  if( labelledObj.size ) {  
    console.log(labelledObj.size);  
  }  
}  
  
var myObj = {size: 10, label: "Size 10 Object"};  
printLabel(myObj);  
  
var myObj2 = { label: "Unknown Size Object"};  
printLabel(myObj2);
```



**Interface pour type précis de fonctions :**

```
interface SearchFunc {  
  (source: string, subString: string): boolean;  
}
```

→ deux paramètres d'entrée de type "string" et valeur de retour de type "boolean"

```
var mySearch: SearchFunc;  
  
mySearch = function(src: string, sub: string) {  
  var result = src.search(sub);  
  if (result == -1) {  
    return false;  
  }  
  else {  
    return true;  
  }  
} //ok
```

**Interface pour type précis de tableaux :**

```
interface StringArray {  
  [index: number]: string;  
}
```

```
var myArray: StringArray;  
myArray = ["Bob", "Fred"];
```

**Interface pour type précis d'objets :**

```
interface ClockInterface {  
  currentTime: Date;  
  setTime(d: Date);  
}
```

```
class Clock implements ClockInterface {  
  currentTime: Date;  
  setTime(d: Date) {  
    this.currentTime = d;  
  }  
  //...  
}
```

**Héritage (simple ou multiple) entre interfaces :**

```
interface Shape {  
    color: string;  
}
```

```
interface PenStroke {  
    penWidth: number;  
}
```

```
interface Square extends Shape, PenStroke {  
    sideLength: number;  
}
```

```
var square = <Square>{};  
square.color = "blue";  
square.sideLength = 10;  
square.penWidth = 5.0;
```

```
var square2: Square = { "color": "blue" , "sideLength": 10, "penWidth": 5.0 } ;
```

## III - Lambda , Generics , ... / typescript

### 1. Prog. fonctionnelle (arrow function / lambda, ...)

#### 1.1. Points communs avec es2015

La partie "*Arrow function et lexical this*" de l'**annexe** "*Arrow functions, essentiel es2015*" (à lire préalablement) montre la vision "es2015" des lambdas (arrow functions) .

La suite de ce chapitre montrera essentiellement les apports "fortement typés" liés aux lambdas.

#### 1.2. Typage fort associés aux lambdas (alias arrow functions)

Versions avec paramètres et valeur de retour typés (typescript) :

```
function add(x: number, y: number): number {
    return x+y;
}

var myAdd = function(x: number, y: number): number { return x+y; };
```

Type complet de fonctions :

```
var myAdd : (a:number, b:number) => number =
    function(x: number, y: number): number { return x+y; };
```

**NB :** les noms des paramètres (ici "a" et "b" ) ne sont pas significatifs dans la partie "type de fonction". Ils ne sont renseignés que pour la lisibilité .

Le type de retour de la fonction est préfixé par **=>** . Si la fonction de retourne rien alors **=> void** .

Inférences/déduction de types (pour paramètres et valeur de retour du code effectif):

```
var myAdd: (a:number, b:number)=>number =
    function(x, y) { return x+y; };
```

Paramètre de fonction optionnel (suffixé par ?)

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
```

```
var result1 = buildName("Bob"); //ok
var result2 = buildName("Bob", "Adams", "Sr."); //error, too many parameters
var result3 = buildName("Bob", "Adams"); //ok
```

### Valeur par défaut pour paramètre de fonction

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

var result1 = buildName("Bob"); //ok : Bob Smith
var result2 = buildName("Bob", "Adams", "Sr."); //error, too many parameters
var result3 = buildName("Bob", "Adams"); //ok
```

### Derniers paramètres facultatifs (... [])

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

var employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

### Lambda expressions

Une "**lambda expression**" est syntaxiquement introduite via **() => { }**

Il s'agit d'une syntaxe épurée/simplifiée d'une fonction anonyme où les parenthèses englobent d'éventuels paramètres et les accolades englobent le code.

Subtilité de "typescript" et "es2015":

La valeur du mot clef "this" est habituellement évaluée lors de l'invocation d'une fonction .  
Dans le cas d'une "lambda expression" , le mot clef this est évalué dès la création de la fonction.

Exemples de "lambda expressions" :

```
var myFct : ( tabNum : number[]) => number ;
myFct = (tab) => { var taille = tab.length; return taille; }
//ou plus simplement:
myFct = (tab) => { return tab.length; }

//ou encore plus simplement:
myFct = (tab) => tab.length;

//ou encore plus simplement:
myFct = tab => tab.length;
```

```
var numRes = myFct([12,58,69]);
console.log("numRes=" + numRes);
```

```
var myFct2 : ( x : number , y: number ) => number ;
myFct2 = (x,y) => { return (x+y) / 2; }
//ou plus simplement:
myFct2 = (x,y) => (x+y) / 2;
```

NB : la technologie "RxJs" utilisée par angular2+ utilise beaucoup de "lambda expressions" .

## 2. Generics de typescript (ts)

### 2.1. Fonctions génériques :

```
function identity<T>(arg: T) : T {  
    return arg;  
}
```

T sera remplacé par un type de données (ex : string , number , ...) selon les valeurs passées en paramètres lors de l'invocation ( Analyse et transcription "ts" → "js" ) .

```
var output = identity<string>("myString"); // type of output will be 'string'
```

```
var output = identity("myString"); // type of output will be 'string' (par inférence/déduction)  
var output2 = identity(58.6); // type of output will be 'number' (par inférence/déduction)
```

### 2.2. Classes génériques :

```
class GenericNumber<T> {  
    zeroValue: T;  
    add: (x: T, y: T) => T;  
}  
  
var myGenericNumber = new GenericNumber<number>();  
myGenericNumber.zeroValue = 0;  
myGenericNumber.add = function(x, y) { return x + y; };  
  
var stringNumeric = new GenericNumber<string>();  
stringNumeric.zeroValue = "";  
stringNumeric.add = function(x, y) { return x + y; };
```

```
interface Lengthwise {  
    length: number;  
}  
  
function loggingIdentity <T extends Lengthwise>(arg: T): T {  
    console.log(arg.length); // we know it has a .length property, so no error  
    return arg;  
}
```

## IV - Modules "typescript"

### 1. Namespaces et Modules typescript (ts)

#### 1.1. Namespace et Modules externes

Le langage typescript gère deux sortes de modules "namespaces logiques" et "modules externes" :

<b>namespace</b> (anciennement <b>modules</b> internes/logiques)	Via mot clef <b>namespace</b> <i>ModuleName</i> { ... } englobant plusieurs " <b>export</b> ...." (sémantique de "namespace" et utilisation via préfixe " <b>ModuleName</b> .")	Dans un seul fichier ou réparti dans plusieurs fichiers (à regrouper) , peu importe.
<b>modules</b> (externes) (selon norme es2015)	Via (au moins un) mot clef <b>export</b> au premier niveau d'un fichier et utilisation via mot clef <b>import</b>	Toujours un fichier par module ( <b>nom du module = nom du fichier</b> )  Selon contexte (nodeJs ou ...)

#### 1.2. Namespace

NB : depuis la version **1.5** de typescript ,  
la terminologie a changé (ancien "**module** interne" --> **namespace**)  
Ceci dit , l'ancien mot clef "**module**" (équivalent au nouveau mot clef "**namespace**") est encore  
utilisable au sein des versions récentes de typescript .  
L'emploi de "**namespace**" est cependant conseillé .

**namespace** (avec utilisation locale dans même fichier):

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }

  var lettersRegex = /^[A-Za-z]+$/; // volontairement non exporté (détail interne)
  var numberRegex = /^[0-9]+$/; // volontairement non exporté (détail interne)

  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegex.test(s);
    }
  }

  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegex.test(s);
    }
  }
}
```

```

}

// échantillon de valeurs :
var strings = ['Hello', '98052', '101'];
// tableau de validateurs
var validators: { [s: string]: Validation.StringValidator; } = {};
validators['ZIP code'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();
// Validation de chaque échantillon par chaque validateur :
strings.forEach(s => {
    for (var name in validators) {
        console.log('"' + s + '"' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ')
                    + name);
    }
});

```

```

"Hello"  does not match ZIP code
"Hello"  matches Letters only
"98052"  matches ZIP code
"98052"  does not match Letters only
"101"    does not match ZIP code
"101"    does not match Letters only

```

--> au sein du code transpilé en javascript (.js) on s'aperçoit que les namespaces sont transformés en du code basé sur des IIFE (*Immediately-Invoked Function Expression*) et le pattern "module" (voir fin d'annexe sur les modules es2015).

--> limité à un seul fichier (ou bien plusieurs fichiers référencés), les namespaces (utilisés seuls) sont un peu moins intéressants que les modules es2015/typescript et finalement assez peu utilisés.

### 1.3. (rare) directive "Triple-slash"

La directive "Triple-slash" permet d'indiquer au compilateur une référence vers un autre fichier en indiquant son emplacement physique (en relatif).

```

///

```

Ce type de directive est essentiellement utilisé pour référencer des fichiers Typescript externes à un projet. La plupart du temps, avec les fichiers de configuration `tsconfig.json`, elle n'est plus nécessaire puisqu'il est possible d'indiquer dans ce fichier les emplacements des fichiers à compiler.

Par exemple, pour ne pas utiliser de directives *triple-slash*, on peut utiliser les éléments de configuration "files" ou "include" dans un fichier `tsconfig.json`:

```

{
    "files": [
        "fichierA.ts",
        "fichierB.ts",
    ],
    "include": [
        "src/**/*"
    ],
}

```

```
...
}
```

## 1.4. Modules (externes) normalisés "es2015"

La langage typescript reprend à fond la syntaxe (et les comportements) des modules normalisés "es2015".

==> annexe "Modules es2015"

## 1.5. Quelques syntaxes "typescript" pour les modules :

*validateurs.ts*

```
export interface StringValidator {
  isAcceptable(s: string): boolean;
}

var lettersRegexp = /^[A-Za-z]+$/;
var numberRegexp = /^[0-9]+$/;

export class LettersOnlyValidator implements StringValidator {
  isAcceptable(s: string) {
    return lettersRegexp.test(s);
  }
}

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

Utiliser la syntaxe **import** { Xxxx, Yyyy } **from** './xxyy' ;  
pour utiliser des éléments (classes , interfaces , ....) exportés dans le fichier "xxyy.ts"

**test-import-validateur.ts**

```
import { StringValidator , ZipCodeValidator , LettersOnlyValidator } from './validateurs';

// échantillon de valeurs :
var strings = ['Hello', '98052', '101'];
// tableau de validateurs :
var validators: { [s: string]: StringValidator; } = {};
```



```

validators['ZIP code'] = new ZipCodeValidator();
validators['Letters only'] = new LettersOnlyValidator();
// Validation de chaque échantillon par chaque validateur :
strings.forEach(s => {
  for (var name in validators) {
    console.log('"' + s + '"' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ')
              + name);
  }
});

```

Via l'option "module": "commonjs" (de *tsconfig.json*)

**ou bien**

Soit l'alias "tsc:mcjs": "tsc --module commonjs" défini dans package.json (sachant que commonjs correspond à la technologie historique des modules de nodeJs )

```
npm run tsc:mcjs validateurs.ts test-import-validateur.ts
```

## 1.6. Eventuelle importation "commonjs / .js" pour env. "nodeJs"

test-validateurs.js

```

var validateurs = require('./validateurs');

// Some samples to try :
var strings = ['Hello', '98052', '101'];
// Validators to use :
var validators = {};
validators['ZIP code'] = new validateurs.ZipCodeValidator();
validators['Letters only'] = new validateurs.LettersOnlyValidator();
// Show whether each string passed each validator
strings.forEach(s => {
  for (var name in validators) {
    console.log('"' + s + '"' + (validators[name].isAcceptable(s) ? ' matches ' : ' does not match ') +
name);
  }
});

```

## 1.7. nodeJs et les modules "es2015" via -r esm

NB : les versions récentes de nodeJs (10.x) supportent maintenant partiellement les modules au format "es2015" via la configuration suivante :

npm-init.bat

```
REM npm init
```

```
REM @std/esm permet de lancer des modules "es2015" (.mjs) depuis node
```

*REM et d'importer des modules ".mjs" dans des vieux modules ".js / cjs"*

**npm install esm**

*REM ceci est utile que pour lancer des modules ".ts" ==> ".js" (en mode module=es2015)*

*REM via node -r esm main.js*

*REM pas besoin de tout cela pour modules ".ts" ==> ".js" (en mode module=commonjs)*

**tsconfig.json**

```
...  
    "module": "es2015",  
//    "module": "commonjs",  
...
```

**test-import-validateur.ts** (--> .js ou .mjs)

```
import { StringValidator , ZipCodeValidator , LettersOnlyValidator} from "./validateurs";  
.....
```

**node -r esm dist/out-tsc/test-import-validateur.js** (ou .mjs)

## 1.8. Packaging de "bundle" via rollup/es2015 ou webPack

L'un des principaux atouts de la structure des "modules es2015" tient dans les imports statiques et précis qui peuvent ainsi être analysés pour une génération optimisée des bundles à déployer en production.

La technologie de packaging "rollup" qui est spécialisée "es2015" peut ainsi exploiter cette optimisation via la chaîne de transformation suivante :

```
myXy.ts      →      myXy.es2015.js
  tsc (target=es2015) ...      → rollup → myBundle.es2015.js → myBundle.es5.js
myZzt.ts     →      myZzt.es2015.js      (*)
```

(\*) **es2015-to-es5** via **typescript -target:es5 et allowJs** ou bien via **babel (presets : es2015)** permet d'obtenir un bundle interprétable par quasiment tous les navigateurs.

NB : si après rollup + es2015-to-es5 il persiste une erreur de type "require(...)" , c'est qu'une des dépendances n'a pas été résolue (pas transformée en include ajusté de code ou d'appel vers d'autres bundle) . Une meilleur configuration de rollup.config.js (ou équivalent) est alors nécessaire.

Rappel : rollup nécessite en entrée du es2015. Il faut que tsconfig.json soit en target : 'es2015' et module : 'es2015' .

Avantages et inconvénients de cette approche :

- il faut mettre au point des scripts pour automatiser la chaîne de production (npm + grunt ou gulp ou ...) et les maintenir/ajuster.
- + la structure du projet est assez libre/ouverte

Scripts "npm" directs de la chaîne "rollup + es2015-to-es5" (sachant qu'une autre version est possible en utilisant gulp) :

**package.json**

```
{
  "name": "tp-ts-es2015-rollup-web",
  "version": "1.0.0",
  "scripts": {
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "rollup-to-es2015": "rollup --config rollup.config.js",
    "es2015-to-es5": "tsc --out ./dist/build-es5/app-bundle.js --target es5
                    --allowJs dist/build-es2015/app-bundle.js"
  },
  ...
}
```

Avant de lancer rollup , il faut préparer une compilation des fichiers de notre projet au format "es2015" attendu par rollup.

Le compilateur typescript (tsc) doit donc être configuré via un fichier tsconfig.json de ce type :

**tsconfig.json (ou tsconfig-es2015.json) :**

```
{
  "compilerOptions": {
```

```

"target": "es2015",
"module": "es2015",
"moduleResolution": "node",
"declaration": false,
"removeComments": true,
"noLib": false,
"emitDecoratorMetadata": true,
"experimentalDecorators": true,
"lib": ["es6", "es2015", "dom"],
"sourceMap": true,
"pretty": true,
"allowUnreachableCode": false,
"allowUnusedLabels": false,
"noImplicitAny": true,
"noImplicitReturns": true,
"noImplicitUseStrict": false,
"noFallthroughCasesInSwitch": true,
"outDir": "./dist/out-tsc",
"rootDir": "src",
"typeRoots": [
  "./node_modules/@types",
  "./node_modules"
],
"types": [
]
},
"files": [
  "src/main.ts"
],
"exclude": [
  "node_modules",
  "dist",
  "src"
],
"compileOnSave": false
}

```

#### rollup.config.js

```

export default {
  input: 'dist/out-tsc/main.js',
  output: {
    file: 'dist/build-es2015/app-bundle.js',
    format: 'iife',
    name: "myapp"
  }
};

```

Dans la configuration très importe de *rollup.config.js* ,

input : .../main.js correspond au point d'entrée (.ts --> .js) autrement dit la racine d'un arbre import/export entre différents fichiers (.ts ou es6) qui seront analysés et gérés par rollup.

Il peut quelquefois y avoir plusieurs input/output dans le fichier `rollup.config.js`.

**main.ts** (ou `.js` / es6/es2015) doit idéalement comporter une ligne **`export default mainFct ;`** en tant que point d'entrée analysé via `rollup`.

*Rien n'interdit cependant un autre export parallèle de type `export mainFct { .... } ;`  
pour une future utilisation de type `myapp.mainFct()` ; plutôt que `myapp["default"]()` ;*

le **format** doit être **"cjs"** pour une future interprétation via `node/nodeJs`

ou **"iife"** pour une future interprétation via `html/js` (navigateur)

output/name : `"myapp"` correspond au nom logique du module qui sera construit via `rollup`.

Au sein d'un code js accompagnant une page `html` , les appels seront de type **`myapp["default"]()`** ;

ou **`myapp.fct_xy()`**;

# V - Librairies de définitions (d.ts)

## 1. Modules/librairies de définitions (.d.ts)

### 1.1. Notion de "ambient ts"

On appelle "**ambient ts**" une déclaration de choses "js" externes potentiellement appelables (ex : jQuery) .

Ceci permet d'appeler des fonctions "javascript" ( "jQuery" ou autres) depuis le code typescript de notre application tout en ayant la possibilité d'utiliser un typage fort.

Intérêts potentiels : debug plus aisé , auto-complétion, ...

Les parties

**"lib"**: ["es6", "es2015", "dom"],

et

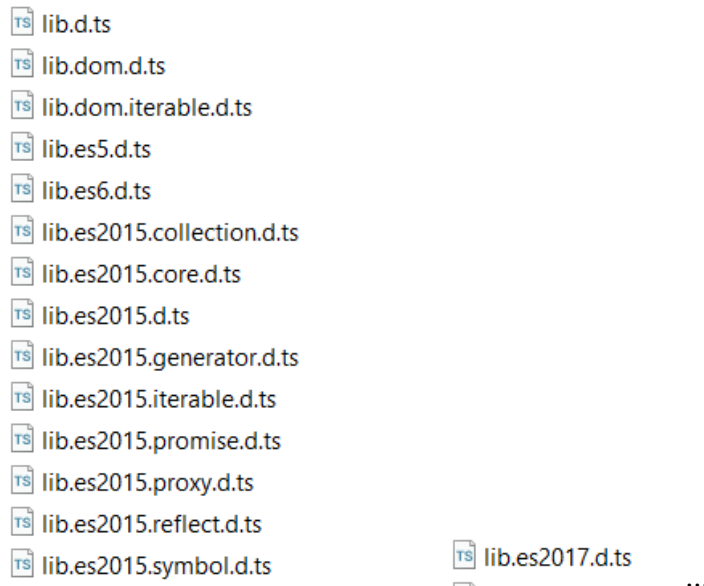
**"typeRoots"**: [  
     "./node\_modules/@types/"  
 ]

de **tsconfig.json** vont dans ce sens et il vaut mieux éviter les doublons de déclaration.

Pour infos , @types est plus récent que "definitivedTyped" (quasi obsolète aujourd'hui) et les librairies "es6" ou "es2015" sont souvent plus précises que @types/...

### 1.2. Librairies prédéfinies

Sur une machine windows, si **npm install -g typescript** a été au moins préalablement lancé une fois, le répertoire **C:\Users\username\AppData\Roaming\npm\node\_modules\typescript\lib** comporte plein de fichiers **lib.xyz.d.ts** dont on peut s'inspirer !!!



L'extension **".d.ts"** signifie "librairie de **d**éfinitions" .

Un tel fichier ne comporte pas le code des fonctions mais simplement toutes les **signatures des fonctions** appelables de la librairie :

- noms et types des paramètres d'entrée
- types des valeurs en retour
- Type de données spécifiques (interfaces , ...)
- éventuelles déclarations de variables globales
- ....

### 1.3. Librairies de définitions personnalisées (classiques)

Exemple :

Soit **lib/js/my-js-lib.js** une librairie simple de fonctions "javascript" :

```
function add_js_lib(a,b){
    return a+b;
}

function prefix_string_js_lib (prefix,str){
    return prefix+"."+str;
}

//-----

var unite="m";

/*
exemples:
entries1=[{"x":3,"y":5},{ "x":5,"y":7},{ "x":7,"y":9}]
stats1=[{"label":"statX unite=cm","sum":15,"average":5},
```

```

    {"label":"statY unite=cm","sum":21,"average":7}]
*/

function buildStats(entries){
    var sx=0,ax=0,n=0,sy=0,ay=0;
    var withY=false;
    var stats=[];
    for(var i in entries){
        n++;
        var vx=entries[i].x;
        var vy=entries[i].y;
        sx+=vx; ax=(vx+ax*(n-1))/(n*1.0);
        if(vy!=null){
            withY=true;
            sy+=vy; ay=(vy+ay*(n-1))/(n*1.0);
        }
    }
    stats.push({
        label:'statX unite='+unite,
        sum: sx,
        average:ax
    });
    if(withY){
        stats.push({
            label:'statY unite='+unite,
            sum: sy,
            average:ay
        });
    }
    return stats;
}

function Line(x1,x2,y1,y2){
    this.x1=x1;
    this.y1=y1;
    this.x2=x2;
    this.y2=y2;
    this.length = function(){
        return Math.sqrt( Math.pow(this.x2 - this.x1,2) + Math.pow(this.y2 - this.y1,2) ) ;
    }
}

//permettant d'écrire var l1=new Line(10,10,50,40) ; console.log(l1.length()) ;

```

alors , un fichier de définition des signatures des fonctions peut s'écrire de la façon suivante :

**lib/d.ts/my-js-lib.d.ts**

```

declare function add_js_lib(a:number, b:number) :number;
declare function prefix_string_js_lib(prefix :string,str :string) :string;

//-----

interface Entry{

```



```

x:number;
y?:number;
z?:number;
}

interface Stat {
  label: string;
  sum: number | undefined;
  average: number | undefined;
}

declare type Unite = "mm" | "cm" | "m" | "km" ;
declare var unite : Unite | undefined;

declare function buildStats(entries :Array<Entry>) :Array<Stat>

interface Line{
  x1:number;
  y1:number;
  x2:number;
  y2:number;
  length():number;
}

declare var Line: {
  prototype: Line;
  new(x1?:number,y1?:number,x2?:number,y2?:number): Line;
};

```

Finalement d'utilisation de cette librairie "javascript" et de son fichier de définitions "typescript" peut s'effectuer de la façon suivante :

#### *essai-classical-ambient.ts*

```

/// <reference path = "../lib/d.ts/my-js-lib.d.ts" />

function add_delegating_to_js_lib(x:number, y:number){
  let res= add_js_lib(x,y);
  console.log(prefix_string_js_lib("resulat",res.toString()));
  log_essai_stats(); log_essai_Line() ;
  return res;
}

function log_essai_stats(){
  unite='cm';
  var statEntries1 :Entry[] = [ { x:3,y:5} , { x:5,y:7}, { x:7,y:9} ];
  var stats1 :Stat[] = buildStats(statEntries1);
  console.log("statEntries1="+JSON.stringify(statEntries1));
  console.log("stats1="+JSON.stringify(stats1));
}

```

```

var statEntries2 :Entry[] =[ { x:3} , { x:4}, { x:2} ];
var stats2 :Stat[] = buildStats(statEntries2);
console.log("statEntries2="+JSON.stringify(statEntries2));
console.log("stats2="+JSON.stringify(stats2));
}

function log_essai_Line(){
    let l1,l2 : Line;
    l1 = new Line();
    l1.x1=10;
    l1.x2=12;
    l1.y1=20;
    l1.y2=24;
    console.log("l1.length()" + l1.length());
    l2=new Line(25,30,10,15);
    console.log("l2.length()" + l2.length());
}

```

Exemple de page html utilisant tout cela :

```

<html>
  <script src="lib/js/my-js-lib.js"></script>
  <script src="dist/essai-classical-ambient.js"></script> <!-- .ts transpilé en .js -->
  <script>
    window.addEventListener("load", function(){
      document.querySelector('#btnAdd').addEventListener('click',function(){
        var x= Number(document.querySelector('#x').value);
        var y= Number(document.querySelector('#y').value);
        document.querySelector('#resultat').innerHTML
          =add_delegating_to_js_lib(x,y);    *= add_js_lib(x,y);    *
      });
    });
  </script>
<body>
  x:<input value="2" id="x" /> <br/>
  y:<input value="3" id="y" /> <br/>
  <input type="button" value="action addition" id="btnAdd" /> <br/>
  res: <span id="resultat" ></span>
</body>
</html>

```

NB : Il est éventuellement possible, lors d'un prototype à livrer pour la veille de définir des versions temporaires sans aucune précision de type:

*my-any-lib.d.ts*

```
declare function mult_js_lib(...args:any):any;  
declare function getBasicInfo(...args:any):any;
```

## 1.4. Librairies de définitions pour modules (ex:iife)

soit *lib/js/my-iife-js-lib.js* une librairie javascript packagée sous forme de fichier "*IIFE*" (design pattern "module") :

```
//begin of IIFE module myLifeJsLib
var myLifeJsLib = (function () {
//-----
function internal_divide_in_iife_js_lib(a,b){
    return a/b;
}

function internal_mult_in_iife_js_lib (x,y){
    return x*y;
}

function internal_carre_in_iife_js_lib (x){
    return internal_mult_in_iife_js_lib(x,x);
}

var selfModuleWithPublicMethods = {
    divide : internal_divide_in_iife_js_lib,
    carre : internal_carre_in_iife_js_lib
};
return selfModuleWithPublicMethods;

//end of IIFE module myJsLib
})();
//-----
```

alors un fichier de définitions "typescript" peut s'écrire de la façon suivante :

*lib/d.ts/my-iife-js-lib.d.ts*

```
declare module myLifeJsLib {
    export function divide(a:number,b:number):number;
    export function carre(x:number):number;
}
```

Et l'utilisation depuis un fichier "typescript" peut s'effectuer de la manière ci-après :

```
/// <reference path = "../lib/d.ts/my-iife-js-lib.d.ts" />

function div_delegating_to_js_lib(x:number, y:number){
    let res= myLifeJsLib.divide(x,y);
    console.log(""+x+"/"+y+"="+res.toString());
    return res;
}
```

```
function carre_delegating_to_js_lib(x:number){  
  let res= myLifeJsLib.carre(x);  
  console.log(""+x+"/"+x+"="+res.toString());  
  return res;  
}
```

# VI - Aspects divers et avancés de typescript

## 1. Aspects divers et avancés de typescript (ts)

### 1.1. Mixins entre classes typescript (proche héritage multiple)

NB : le terme "mixin" est utilisé dans tous un tas de contextes informatiques :

- mixin entre styles css (ex : scss)
- mixin entre objets/instances javascript (ex : rassembler méthodes d'une instance avec données d'un autre objet via Object.assign(obj1,obj2) )
- mixin entre types de données "typescript" (proche de héritage multiple)

Dans la suite de ce paragraphe , on s'intéressera au mixin entre types de données "typescript" .

NB : l'héritage multiple n'est pas (directement) supporté par le langage typescript :

```
class Hydravion extends Avion, ObjetFlottant { ... }
```

Pour contourner la limitation de l'héritage multiple direct impossible, typescript 2.2+ a introduit la notion de **fonction mixin** qui :

- **prend une fonction constructeur en entrée,**
- **créer une classe qui étend/enrichi ce constructeur en ajoutant de nouvelles fonctionnalités**
- **retourne la nouvelle classe**

Exemple :

```
class Avion {
  constructor(public type: string="avion" ,
    public altitude:number=0){
  }
}

let a1 = new Avion();
console.log('a1='+JSON.stringify(a1));

//Type générique de constructeur (de base) à enrichir
//Type d'un point d'entrée de tous les "mixins":
type Constructor<T = {}> = new (...args: any[]) => T;

//Mixin ajoutant la fonctionnalité "flotter sur l'eau":
function Flottant<TBase extends Constructor>(Base: TBase) {
  return class extends Base {
    nbFlotteurs : number = 2; //default value=2
    glisser(){

```

```

        console.log("glisser sur l'eau avec nbFlotteurs="+this.nbFlotteurs);
    }
};
}

//Mixin ajoutant la fonctionnalité générique "timestamped":
function Timestamped<TBase extends Constructor>(Base: TBase) {
    return class extends Base {
        timestamp = Date.now(); //default value=now
        logTimestamp(){
            console.log("object created at timestamp="+this.timestamp);
        }
    };
}

//nouvelle classe AvionFlottant basée sur classe Avion et enrichie via Mixin Flottant
const AvionFlottant = Flottant(Avion);

//nouvelle classe ActionFlottantWithTimestamp basée sur classe AvionFlottant et enrichie
//via Mixin Timestamped (résultat d'une double application de mixins)
const ActionFlottantWithTimestamp=Timestamped(AvionFlottant);

//test de AvionFlottant
let a2 = new AvionFlottant();
console.log('a2='+JSON.stringify(a2));
a2.glisser();

//test de ActionFlottantWithTimestamp
let a3 = new ActionFlottantWithTimestamp();
console.log('a3='+JSON.stringify(a3));
a3.glisser();
a3.logTimestamp();
==>
a1={"type":"avion","altitude":0}
a2={"type":"avion","altitude":0,"nbFlotteurs":2}
glisser sur l'eau avec nbFlotteurs=2
a3={"type":"avion","altitude":0,"nbFlotteurs":2,"timestamp":1557231328404}
glisser sur l'eau avec nbFlotteurs=2
object created at timestamp=1557231328404

```

## 1.2. Décorateurs "typescript"

Les "décorateurs typescript" correspondent à une **fonctionnalité expérimentale** de typescript et de es7 mais qui est *déjà utilisée par Angular >=2*.

Syntaxiquement introduit via la syntaxe **@décorateurXy()**, un décorateur "typescript" correspond à la fois :

- à une *variante du design pattern "décorateur"* (fonctionnalités/comportements enrichis)
- à une *variante des "annotations java" (programmation déclarative)*

Plus concrètement, un décorateur "typescript" :

- se code comme une fonction technique ayant une signature très particulière
- s'applique de manière déclarative en plaçant **@décorateurXy** ou **@décorateurXy()** ou **@décorateurXy(valParam1,valParam2)** au dessus d'un élément à enrichir ou paramétrer (classe, méthode, propriété, argument d'une méthode).
- s'exécute au runtime comme une fonction de pré-traitement permettant d'altérer par méta-programmation/réflexion le code ordinaire (constructeur, méthode, ...) appelé.

Exemple d'application de décorateurs :

```
@myLogClassDecorator()
class Cercle {

    @myLogPropertyDecorator
    public unite:string | undefined;

    constructor(public xC :number =0, public yC :number =0,
                public rayon :number =0){
    }

    @myLogMethodDecocator(true)
    description() : string{
        return "Cercle de centre (" +this.xC+", "+this.yC+") et de rayon " + this.rayon;
    }

    @myLogMethodDecocator()
    aire() : number {    return Math.PI * this.rayon * this.rayon;
    }

    @myLogMethodDecocator()
    perimetre() : number {    return Math.PI * this.rayon * this.rayon;
    }

    @myLogMethodDecocator()
    carre(x :number):number{    return x*x;
    }

    @myLogMethodDecocator()
    addition(a :number, b:number):number{    return a+b;
    }
}
```



```
//Cercle.prototype.unite="m";
```

```
let c1=new Cercle(100,80,50);
```

```
//>>> myLogClassDecorator , New: Cercle is created
```

```
//>>> myLogPropertyDescriptor , set newValue of propertyName=unite : ?
```

```
let descriptionC1= c1.description();
```

```
console.log("descriptionC1="+descriptionC1);
```

```
* >>myLogMethodDecocator intercept call of description() returnValue is CERCLE DE CENTRE (100,80) ET DE RAYON 50 *
```

```
//descriptionC1=CERCLE DE CENTRE (100,80) ET DE RAYON 50
```

```
c1.unite='cm';
```

```
//>>> myLogPropertyDescriptor , set newValue of propertyName=unite : cm
```

```
//>>> myLogPropertyDescriptor , get value of propertyName=unite : cm
```

```
console.log("unite de c1="+c1.unite); //unite de c1=CM
```

```
let perimetreC1= c1.perimetre();
```

```
//>>myLogMethodDecocator intercept call of perimetre() returnValue is 7853.981633974483
```

```
console.log("perimetre de c1 =" +perimetreC1); //perimetre de c1 =7853.981633974483
```

```
let aireC1= c1.aire();
```

```
//>>myLogMethodDecocator intercept call of aire() returnValue is 7853.981633974483
```

```
console.log("aire de c1 =" +aireC1); //aire de c1 =7853.981633974483
```

```
let troisAuCarre=c1.carre(3);
```

```
//>>myLogMethodDecocator intercept call of carre(3) returnValue is 9
```

```
console.log("troisAuCarre =" +troisAuCarre); //troisAuCarre =9
```

```
let deuxPlusTrois=c1.addition(2,3);
```

```
//>>myLogMethodDecocator intercept call of addition(2,3) returnValue is 5
```

```
console.log("deuxPlusTrois =" +deuxPlusTrois); //deuxPlusTrois =5
```

NB :

En tant que fonctionnalité expérimentale (et pas encore définitivement normalisée) , l'utilisation des décorateurs nécessite l'option **--experimentalDecorators** de **tsc**

ou bien **"experimentalDecorators": true** dans **tsconfig.json**

D'autre part, **"noImplicitThis": false** dans **tsconfig.json** est souvent utile pour éviter des erreurs dans le code générique des décorateurs

Finalement, **"lib" : [ "es2017", "dom" ]** peut être utile pour pouvoir utiliser **Reflect.construct(...)** ou d'autres spécificités de es2015 ou es2017.

Généralité : 1 à 3 niveaux de code imbriqué pour un décorateur:

```
//niveau quelquefois facultatif "decorator factory" pour déclenchement au runtime
//avec @decoXy() @decoXy(valP1,valP2)
//inutile sur décorateur de méthode ou de propriété si @decoXy sans paramètre NI PARENTHESE
function decoXy(p1, p2) {
```

```
//niveau principal/obligatoire "décorateur" (META-PROGRAMMATION) :
```

```
return function (target [, propertyKey [, descriptor_or_other]]) {
```

```
  //META-READAPTATION (descriptor, ...) du code qui sera déclenché
```

```
    descriptor.value = function(...args: any) {
```

```
      return Value = ....apply(this, args);
```

```
    OU BIEN
```

```
    updateProperty = Object.defineProperty(...
```

```
    OU BIEN ...
```

```
      //niveau facultatif (mais souvent le plus utile): réadaptation du code qui sera exécuté
```

```
      //...
```

```
    }
```

```
  }
```

```
}
```

### Décorateur de méthode :

```
//decorator factory [to apply it at runtime]
```

```
//when declared with @myLogDecorator() or @myLogDecorator(true) above a method
```

```
function myLogMethodDecocator(paramMaj :boolean =false) {
```

```
  //the decorator (internal):
```

```
  return function (target:any,
```

```
    methodNameAspropertyKey: string,
```

```
    descriptor: PropertyDescriptor) {
```

```
    let originalMethodFunction= descriptor.value;
```

```
    //console.log(">> originalMethodFunction (code)="+originalMethodFunction);
```

```

descriptor.value = function(...args: any[]) {
  //for methodDecorator , target = prototype object of current object (ex: Cercle.prototype)
  //console.log(">>> in myLogDecorator, target (current obj) prototype as json string="+JSON.stringify(target));
  let returnValue = originalMethodFunction.apply(this, args);
  if(paramMaj && typeof returnValue === 'string') {
    returnValue=returnValue.toUpperCase(); //little change/adaptation
  }
  const params = args.map(arg => JSON.stringify(arg)).join();
  console.log(`>>>myLogMethodDecocator intercept call of ${methodNameAspropertyKey}(${params})`
    + `returnValue is ${returnValue}`);
  return returnValue;
}
return descriptor;
}

```

## property decorator:

```

function myLogPropertyDecorator(target: Object, propertyName: string) {
  //target is prototype objet of current object
  let value: any; //will be used in subscooped get / set arrow functions
  const updateProperty = Object.defineProperty(
    target,
    propertyName,
    {
      configurable: true,
      enumerable: true,
      get: () => {
        console.log(">>>> myLogPropertyDecorator , get value of propertyName="+propertyName + " : " + value );
        if(value && typeof value === 'string')
          return value.toUpperCase();// little adaptation
        else
          return value;
      },
      set: (newValue: number) => {
        console.log(">>>> myLogPropertyDecorator , set newValue of propertyName="

```

```

        +propertyName + " : " + newValue );
    value = newValue ;
}
},
);
}

```

### class decorator:

```

function myLogClassDecorator(){
    return function (target: Function) {
        // save a reference to the original constructor
        const originalConstructor = target;
        //originalConstructor.prototype.additionalProperty="abc"; //syntax ok but not a very good practice
        //NB: use mixin in class decorator is not a very good idea (the constructed objet may be too different)

        // the new constructor behaviour
        const c: any = function (...args :any) {
            console.log(`>>> myLogClassDecorator , New: ${originalConstructor['name']} is created`);
            return Reflect.construct(originalConstructor,args)
        }

        // copy prototype so instanceof operator still works
        c.prototype = originalConstructor.prototype;

        // return new constructor (will override original)
        return c;
    }
}

```

Autres décorateurs possibles : sur accesseur get ou set ...

sur un paramètre d'une méthode

### 1.3. Options du compilateur "tsc" et tsconfig.json

#### Exemple1:

##### tsconfig.json

```
{
  "compilerOptions": {
    "baseUrl": "",
    "declaration": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": ["es6", "dom"],
    "mapRoot": "./",
    "module": "es6",
    "moduleResolution": "node",
    "outDir": "../dist/out-tsc",
    "sourceMap": true,
    "target": "es5",
    "typeRoots": [
      "../node_modules/@types"
    ]
  }
}
```

#### Exemple2 :

##### tsconfig.json

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "baseUrl": "",
    "declaration": false,
    "emitDecoratorMetadata": false,
    "experimentalDecorators": false,
    "outDir": "dist/out-tsc",
    "sourceMap": true,
    "target": "es5",
    "noEmitOnError" : false
  },
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules",
    "**/*.spec.ts",
    "dist"
  ]
}
```

```
}
```

Référence sur options du compilateur →

<https://www.typescriptlang.org/docs/handbook/compiler-options.html>

Référence sur tsconfig.json →

<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

### 1.4. Surcharge très limitée

#### **Fonctions à retour variable ou bien surchargées (overload) :**

La prise en charge des fonctions surchargées (avec différents types de paramètres d'entrées) est assez limitée en "typescript" .

*Exemple (seulement intéressant pour la syntaxe) :*

```
function displayColor( tabRgb : number[] ) : void;
function displayColor(c: string) : void;

function displayColor(p:any) : void{
  if(typeof p == "string" ){
    console.log("c:" + p);
  }
  else if(typeof p == "object" ){
    console.log("r:" + p[0] + ",g:" + p[1] + ",b="+p[1]);
  }
  else{
    console.log("unknown color , typeof =" + (typeof p));
  }
}

displayColor([125,250,30]);
displayColor("red");
```

# ANNEXES

## VII - Annexe – Arrow functions, essentiel es2015

### 1. Mots clefs "let" et "const" (es2015)

Depuis longtemps (en javascript) , le mot clef "**var**" permet de déclarer explicitement une variable dont la portée dépend de l'endroit de sa déclaration (globale ou dans une fonction ).

Sans aucune déclaration, une variable (affectée à la volée) est globale et cela risque d'engendrer des effets de bords (incontrôlés) .

Introduits depuis es6/es2015 et typescript 1.4 , les mots clefs **let** et **const** apportent de nouveaux comportements :

- Une variable déclarée via le mot clef **let** a une *portée limitée au bloc local* (exemple boucle for) . Il n'y a alors pas de collision avec une éventuelle autre variable de même nom déclarée quelques ligne au dessus du bloc d'instructions ( entre {} , de la boucle).
- Une variable déclarée via le mot clef **const** *ne peut plus changer de valeur après la première affectation*. Il s'agit d'une **constante** .

Exemple :

```
const PISur2 = Math.PI / 2;
//PISur2=2; // Error, can't assign to a `const`
console.log("PISur2 = " + PISur2);

var tableau = new Array();
tableau[0] = "abc";
tableau[1] = "def";

var i = 5;
var j = 5;

//for(let i in tableau) {
for(let i=0; i<tableau.length; i++) {
    console.log("*** at index " + i + " value = " + tableau[i] );
}

//for(j=0; j<tableau.length; j++) {
for(var j=0; j<tableau.length; j++) {
    console.log("### at index " + j + " value = " + tableau[j] );
}

console.log("i=" + i); //affiche i=5
console.log("j=" + j); //affiche j=2
```



## 2. "Arrow function" et "lexical this" (es2015)

Rappels (2 syntaxes "javascript" ordinaires) valables en "javascript/es5" :

```
//Named function:
function add(x, y) {
    return x+y;
}

//Anonymous function:
var myAdd = function(x, y) { return x+y; };
```

### Arrow functions (es2015) (alias "Lambda expressions" )

Une "**Arrow function**" en javascript/es2015 ( à peu près équivalent à une "lambda expression" de java >8) est syntaxiquement introduite via **() => { }**

Il s'agit d'une syntaxe épurée/simplifiée d'une fonction anonyme où les parenthèses englobent d'éventuels paramètres et les accolades englobent le code.

Subtilité du "**lexical this**" :

La valeur du mot clef "this" est habituellement évaluée lors de l'invocation d'une fonction .

Dans le cas d'une "lambda expression" , le mot clef this est évalué dès la création de la fonction et correspond au this du niveau englobant (classe ou fonction).

Exemples de "lambda expressions" :

```
myFct = (tab) => { var taille = tab.length; return taille; }
//ou plus simplement:
myFct = (tab) => { return tab.length; }
//ou encore plus simplement:
myFct = (tab) => tab.length;
//ou encore plus simplement:
myFct = tab => tab.length;
```

```
var numRes = myFct([12,58,69]);
console.log("numRes=" + numRes); //affiche 3
```

```
myFct2 = (x,y) => { return (x+y) / 2; } //with statement body in { }
//ou plus simplement:
myFct2 = (x,y) => (x+y) / 2; //with simple expression
```

**NB** : les "**Promise**" (es2015) et la technologie "**RxJs**" utilisée par angular>=2 utilisent beaucoup de "Arrow Functions" .

Exemple de "arrow function" combiné ici avec `arrayXy.forEach(callback)` de "javascript/es5" :

```
let array1 = [ 1 , 2 , 3 , 4 , 5 , 6 ];
let eltPairs = [];
array1.forEach( (e) => { if( (e % 2) === 0 )
                        eltPairs.push(e);
                      }
                );
console.log(eltPairs); // affiche [2,4,6]
```

Suite de l'exemple utilisant `nouveauTableau = arrayXy.map(transform_callback)` de es5 :

```
let eltImpairs = eltPairs.map( v => v-1 );
console.log(eltImpairs); // affiche [1,3,5]
```

Exemple montrant "lexical this" (arrow function utilisant `this` de niveau englobant) :

```
var toto = {
  _name: "toto",
  _friends: [ 'titi' , 'tata'],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " est ami avec " + f));
  }
};
toto.printFriends();
```

Autre comportement à connaître:

Si une "fonction fléchée" / "arrow fonction" est à l'intérieur d'une autre fonction, elle partage alors les arguments/paramètres de la fonction parente .

### 3. for...of (es2015) utilisant itérateurs internes

```
var tableau = new Array();
```

```
//tableau.push("abc");
```

```
//tableau.push("def");
```

```
tableau[0] = "abc";
tableau[1] = "def";
```

Au moins 3 parcours possibles via boucle **for**:

```
var n = tableau.length;
for(let i = 0; i < n; i++) {
  console.log(">> at index " + i + " value = " + tableau[i] );
}
```

```
for(let i in tableau) {
  console.log("** at index " + i + " value = " + tableau[i] );
}
```

*//for( index in ...) existait déjà en es5*

*//for(...of ...) au sens "for each ... of ..." est une nouveauté de es2015*

```
for( let s of tableau){
  console.log("## val = " + s );
}
```

**NB** : la boucle for...of est prédéfinie sur un tableau . il est cependant possible de personnaliser son comportement si l'on souhaite la déclencher sur une structure de données personnalisée. On peut pour cela mettre en oeuvre des itérateurs (et éventuels générateurs de bas niveaux) ---> dans chapitre ou annexe "éléments divers et avancés de es2015" .

### 3.1. "template string" es2015 (avec quotes inverses et \${})

```
var name = "toto";
var year=2015;
// ES5
//var message = "Hello " + name + " , happy " + year; // Hello toto , happy 2015
// ES6/ES2015 :
const message = `Hello ${name} , happy ${year}`; // Hello toto , happy 2015
//attention: exception "ReferenceError: name is not defined" si name est undefined
console.log(message);
```

**\${}** peut éventuellement englober des expressions mathématiques ou bien des appels de fonctions.

```
let x=5 , y=6;
let carre = (x) => x*x ;
console.log(`pour x=${x} et y=${y} , x*y=${x*y} et x*x=${carre(x)}`);
//affiche pour x=5 et y=6 , x*y=30 et x*x=25
```

template-string multi-lignes :

```
/*
//ES5
let htmlPart=
"<select> \
  <option>1</option> \
  <option>2</option> \
</select>";
*/
```

```
//template multi-lignes ES2015:
let htmlPart=
`<select>
  <option>1</option>
  <option>2</option>
</select> `;
console.log(htmlPart);
```

### 3.2. Map , Set

```
// Sets (ensembles sans doublon)
var s = new Set();
s.add("hello").add("goodbye").add("hello");
if(s.size === 2)
  console.log("s comporte 2 elements");
if(s.has("hello"))
  console.log("s comporte hello");
```

// List ==> Array ordinaire (déjà en es5 , à remplir via .push() ).

```
// Maps (table d'association (clef,valeur))
var m = new Map();
m.set("hiver", "froid , neige");
m.set("printemps", "fleur , vert");
m.set("ete", "soleil , plage");
m.set("ete", "chaud , plage"); //la nouvelle valeur remplace l'ancienne .
m.set("automne", "feuilles mortes");
let carateristique_ete = m.get("ete");
console.log("carateristique_ete="+carateristique_ete); //chaud , plage
if(m.has("ete"))
    console.log("Map m comporte une valeur associée à ete");
for(saison of m.keys()){
    console.log("saison "+ saison + " - " + m.get(saison));
}
//m.values() permettrait d'effectuer une boucle sur les valeurs (peu importe les clefs)
for([k,v] of m.entries()){
    console.log("saison "+ k + " -- " + v);
}
m.forEach((val,key)=> console.log("saison "+ key + " --- " + val));
m.clear();
if(m.size===0)
    console.log("map m is empty");

//Bien que ce code soit lisible et explicite, un vieil objet javascript en faisait autant :
var objectMap = {
    hiver : "froid , neige",
    printemps : "fleur , vert",
};
objectMap["ete"]="chaud, plage" ;
console.log("carateristique_hiver="+ objectMap["hiver"]); // froid , neige

//Une des valeurs ajoutées par "Map" (es2015) est la possibilité d'avoir des clefs de n'importe
//quelle sorte possible (ex : window , document , element_arbre_DOM, ...).
```

**NB :** es2015 a également introduit les variantes "WeakMap" et "WeakSet" mais celles-ci ne sont utilisables et utiles que dans des cas très pointus (ex : programmation de "cache").

"WeakMap" et "WeakSet" sont exposés dans le chapitre (ou annexe) "Aspects divers et avancés" .

### 3.3. "Destructuring" (affectation multiple avec perte de structure)

**Destructuring objet** : extract object parts in several variables :

```
const p = { nom : 'Allemagne' , capitale : 'Berlin' , population : 83000000, superficie : 357386};
const { nom , capitale } = p;
console.log("nom="+nom+" capitale="+capitale);
//nom="?" ; interdit car nom et capitale sont considérées comme des variables "const"

//NB: les noms "population" et "superficie" doivent correspondre à des propriétés de l'objet
//dont il faut (partiellement) extraire certaines valeurs (sinon "undefined")
//l'ordre n'est pas important
const { superficie , population } = p;
console.log("population="+population+" superficie="+superficie);
==>
```

nom=Allemagne capitale=Berlin  
population=83000001 superficie=357386

**utilité concrète** (parmi d'autres) : *fonction avec paramètres nommés* :

```
function fxabc_with_named_param( { paramX=0 , a=0 , b=0 , c=0 } = {} ){
    //return ax^2+bx+c
    return a * Math.pow(paramX,2) + b * paramX + c;
}

let troisFois4 = fxabc_with_named_param( { paramX :4 , b : 3 } );
console.log("troisFois4="+troisFois4 );//12
let deuxFois4AuCarreplus6 = fxabc_with_named_param( { paramX :4 , a : 2 , c :6 } );
console.log("deuxFois4AuCarreplus6="+deuxFois4AuCarreplus6 );//38
```

**Destructuring iterable (array or ...)** :

```
const [ id , label ] = [ 123 , "abc" ];
console.log("id="+id+" label="+label);

//const arrayIterable = [ 123 , "abc" ];
//var iterable1 = arrayIterable;
const stringIterable = "XYZ";
var iterable1 = stringIterable;
const [ partie1 , partie2 ] = iterable1;
console.log("partie1="+partie1+" partie2="+partie2);
```

==>

id=123 label=abc  
partie1=X partie2=Y



Autre exemple plus artistique (Picasso) :

### 3.4. for (..of ..) with destructuring on Array , Map, ...

```
const dayArray = ['lundi', 'mardi', 'mercredi'];
for (const entry of dayArray.entries()) {
  console.log(entry);
}
// [ 0, 'lundi' ]
// [ 1, 'mardi' ]
// [ 2, 'mercredi' ]

for (const [index, element] of dayArray.entries()) {
  console.log(` ${index}. ${element} `);
}
// 0. lundi
// 1. mardi
// 2. mardi
```

```
const mapBoolNoYes = new Map([
  [false, 'no'],
  [true, 'yes'],
]);
for (const [key, value] of mapBoolNoYes) {
  console.log(` ${key} => ${value} `);
}
// false => no
// true => yes
```

## VIII - Annexe – Objets es2015 (class , extends, ...)

### 1. Prog. orientée objet "es2015" (class, extends, ...)

#### 1.1. Préambule ("poo via prototype es5" --> "poo es2015")

La programmation orientée objet existait déjà en "javascript/es5" mais avec une syntaxe très complexe , verbeuse et peu lisible (prototypes avancés).

Dès l'époque "es5" , le mot clef constructor existait et l'on pouvait même définir une relation d'héritage avec une syntaxe complexe et rebutante.

La nouvelle version "es6/es2015" du langage "javascript/ecmascript" a enfin apporté une nouvelle syntaxe "orientée objet" beaucoup plus claire et lisible donnant envie de programmer de nouvelles classes d'objet en javascript moderne.

#### 1.2. Classe et instances

```
class Compte{
  constructor(numero,label,solde){
    this.numero = numero;
    this.label=label;
    this.solde=solde;
  }

  debiter(montant) {
    this.solde -= montant; // this.solde = this.solde - montant;
  }

  crediter(montant) {
    this.solde += montant; // this.solde = this.solde + montant;
  }
}
```

```
let c1 = new Compte(); //instance (exemplaire) 1
console.log("numero et label de c1: " + c1.numero + " " + c1.label); // undefined undefined
console.log("solde de c1: " + c1.solde); // undefined

let c2 = new Compte(); //instance (exemplaire) 2
c2.solde = 100.0;
c2.crediter(50.0);
console.log("solde de c2: " + c2.solde); //150.0

let c3 = new Compte(3,"compte3",300); //instance (exemplaire) 3
```



```
console.log("c3: " + JSON.stringify(c3)); //{ "numero":3,"label":"compte3","solde":300}
```

NB: Sans initialisation explicite (via constructeur ou autre) , les propriétés internes d'un objet sont par défaut à la valeur **"undefined"** .

### 1.3. "constructor" avec éventuelles valeurs par défaut

Un constructeur est une méthode qui sert à initialiser les valeurs internes d'une instance dès sa construction (dès l'appel à new) .

En langage javascript/es2015 le constructeur se programme comme la méthode spéciale **"constructor"** (mot clef du langage) :

```
class Compte{
  constructor(numero,label,solde){
    this.numero = numero; this.label=label; this.solde=solde;
  }
  //...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
c1.crediter(50.0); console.log("solde de c1: " + c1.solde);
```

NB: contrairement au langage java (où la surcharge y est permise) , il n'est pas possible d'écrire plusieurs versions du constructeur (ou d'une fonction de même nom) en javascript/es2015:

```
constructor(numero, libelle, soldeInitial){
  this.numero = numero;
  this.label = libelle;
  this.solde = soldeInitial;
}
```

```
constructor(){
  this.=0;
  this.label="?";
  this.=0.0;
}
```

Il faut donc quasi systématiquement utiliser la syntaxe = **valeur\_par\_defaut** sur les arguments d'un constructeur pour pouvoir créer une nouvelle instance en précisant plus ou moins d'informations lors de la construction :

```
class Compte{
  constructor(numero=0, libelle="?", soldeInitial=0.0){
    this.numero = numero;
    this.label = libelle;
    this.solde = soldeInitial;
  }
  //...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
var c2 = new Compte(2,"compte 2");
var c3 = new Compte(3); var c4 = new Compte();
```

## 1.4. propriétés (pseudo attributs - mots clefs "get" et "set" )

Bien que "es2015" ne prenne pas en charge les mots clefs "~~public~~", "~~private~~" et "~~protected~~" au niveau des membres d'une classe et que toutes les méthodes définies soient publiques , il est néanmoins possible d'utiliser les mots clefs **get** et **set** de façon à ce qu'un **couple de méthodes "get xy()" et "set xy(...)" soit vu de l'extérieur comme une propriété (pseudo-attribut "xy")** de la classe .

Attention : contrairement au langage java , il ne s'agit pas de convention de nom getXy() / setXy() mais de véritables mot clefs "get" et "set" à utiliser comme préfixe (avec un espace) .

```
class Compte{
  get decouvertAutorise(){
    return (this._decouvertAutorise!=undefined)?this._decouvertAutorise:0;
  }
  set decouvertAutorise(decouvertAutorise){
    this._decouvertAutorise = decouvertAutorise;
  }
  //... }
```

```
let c4=new Compte() ;
c4.decouvertAutorise = -300;
let decouvertAutorisePourC4 = c4.decouvertAutorise;
console.log("decouvertAutorisePourC4="+decouvertAutorisePourC4);
```

NB :

- il est possible de ne coder que le "get xy()" pour une propriété en lecture seule .
- il faut un nom différent (avec par exemple un "\_" en plus) au niveau du nom de l'attribut interne préfixé par "this." car sinon il y a confusion entre attribut et propriété et cela mène à des boucles infinies .
- un "getter" peut éventuellement être supprimé via le mot clef **delete** (ex : delete obj.dernier)
- **Object.defineProperty()** permet (dans le cas pointus) de définir un "getter" par "méta-programmation" (ex : dans le cadre d'un framework ou d'une api générique).

En javascript es2015, le mot clef **get** sert surtout à définir **une propriété dont la valeur est calculée dynamiquement** :

```
var obj = {
  get dernier() {
    if (this.arrayXy.length > 0) {
      return this.arrayXy[this.arrayXy.length - 1];
    }
    else {
      return null;
    }
  }
}
```

```

    },
    arrayXy: ["un", "deux", "trois"]
  }

```

```
console.log("dernier="+obj.dernier); // "trois"
```

## 1.5. mot clef "static" pour méthodes de classe

De la même façon que dans beaucoup d'autres langages orientés objets (c++, java, ...) , le mot clef **static** permet de définir des méthodes de classe (dont l'appel s'effectue avec le préfixe "NomDeClasse." plutôt que "instancePrecise." ).

ES2015 ne permet pas d'utiliser static avec un attribut mais on peut utiliser "static" sur une propriété (avec mot clef get et éventuellement set ). Dans ce cas la valeur de la propriété sera partagée par toutes les instances de la classe (et l'accès se fera via le préfixe "NomDeClasse." )

Exemple:

```

class CompteEpargne {
  //...
  static get tauxInteret(){
    return CompteEpargne.prototype._tauxInteret;
  }
  static set tauxInteret(tauxInteret){
    CompteEpargne.prototype._tauxInteret=tauxInteret;
  }

  static get plafond(){
    return CompteEpargne.prototype._plafond;
  }
  static set plafond(plafond){
    CompteEpargne.prototype._plafond=plafond;
  }

  static methodeStatiqueUtilitaire(message){
    console.log(">>>" + message + "<<<");
  }
}

```

```

CompteEpargne.prototype._tauxInteret = 1.5 ; //1.5% par défaut
CompteEpargne.prototype._plafond = 12000; //par défaut

```

```

let cEpargne897 = new CompteEpargne() ;
//cEpargne897.solde = 250.0; // instancePrecise.proprieteOrdinairePasStatique
console.log("taux interet courant=" + CompteEpargne.tauxInteret);//1.5
console.log("plafond initial=" + CompteEpargne.plafond);//12000
CompteEpargne.plafond = 10000;
let messagePlafond= "nouveau plafond=" + CompteEpargne.plafond;//10000
CompteEpargne.methodeStatiqueUtilitaire(messagePlafond);

```

NB :

- En cas d'héritage , une sous classe peut faire référence à une méthode statique de la classe parente via le mot préfixe `super`.
- Une méthode statique ne peut pas être invoquée avec le préfixe `this` .

## 1.6. héritage et valeurs par défaut pour arguments:

```

class Animal {
  constructor(theName="default animal name") {
    this.name= theName;
  }
  move(meters = 0) {
    console.log(this.name + " moved " + meters + "m.");
  }
}

```

```

class Snake extends Animal {
  constructor(name) { super(name); }
  move(meters = 5) {
    console.log("Slithering...");
    super.move(meters);
  }
}

```

```

class Horse extends Animal {
  constructor(name) { super(name); }
  move(meters = 45) {
    console.log("Galloping...");
    super.move(meters);
  }
}

```

```

var a = new Animal(); //var a = new Animal("animal");
var sam = new Snake("Sammy the Python"); //var sam = new Snake();
var tom = new Horse("Tommy the Palomino");

a.move() ; // default animal name moved 0m.
sam.move(); // Slithering... Sammy the Python moved 5m.

tom.move(34); //avec polymorphisme (for Horse)
// Galloping... Tommy the Palomino moved 34m.

```

## 1.7. Object.assign()

**Object.assign(obj,otherObject)** ; permet (selon les cas) de :

- effectuer un clonage en mode "shallow copy" (copies des références vers propriétés)
- ajouter dynamiquement un complément
- ajouter le comportement d'une classe à un pur objet de données

Exemple d'objet original:

```
const subObj = { pa : "a1" , pb : "b1" };
const obj1 = { p1: 123 , p2 : 456 , p3 : "abc" , subObj : subObj };
```

Clonage imparfait (pas toujours en profondeur) avec Object.assign(clone,original)

```
var objCloneViaShallowCopy = {}
Object.assign(objCloneViaShallowCopy,obj1); //copy of property reference
console.log("clonage via assign / shallowCopy=" + JSON.stringify(objCloneViaShallowCopy));
// {"p1":123,"p2":456,"p3":"abc","subObj":{"pa":"a1","pb":"b1"}}
objCloneViaShallowCopy.subObj.pa="a2";
//modification à la fois sur objCloneViaShallowCopy.subObj et sur obj1.subObj
console.log("obj1" + JSON.stringify(obj1));
// {"p1":123,"p2":456,"p3":"abc","subObj":{"pa":"a2","pb":"b1"}}
objCloneViaShallowCopy.subObj.pa="a1"; //restituer ancienne valeur
```

Clonage en profondeur avec obj=JSON.parse(Json.stringify(original)) :

```
var obj = JSON.parse(JSON.stringify(obj1)); //clonage en profondeur
console.log("clonage en profondeur=" + JSON.stringify(obj));
obj.subObj.pa="a2"; //modification que sur obj.subObj
console.log("obj1" + JSON.stringify(obj1)); //obj1 inchangé ( "a1")
```

Ajout de données via Object.assign(obj, complément) :

```
Object.assign(obj, { p4: true , p5: "def" });
console.log("après assign complement=" + JSON.stringify(obj));
// {"p1":123,"p2":456,"p3":"abc","subObj":{"pa":"a2","pb":"b1"},"p4":true,"p5":"def"}
```

Ajout comportemental via obj=Object.assign(new MyClass(), obj):

```
class Pp {
    constructor(p1=0,p2=0,p3=0){
        this.p1=p1; this.p2=p2; this.p3=p3;
    }

    sumOfP1P2P3(){
        return this.p1+this.p2+this.p3;
    }
}

const subObj = { pa : "a1" , pb : "b1" };
const obj1 = { p1: 123 , p2 : 456 , p3 : "abc" , subObj : subObj };
obj = JSON.parse(JSON.stringify(obj1));//réinitialisation du clone "obj"
if(!(obj instanceof Pp))
    console.log("obj is not instance of Pp , no sumOfP1P2P3() method");
//console.log("obj.sumOfP1P2P3()="+obj.sumOfP1P2P3()); not working
obj = Object.assign(new Pp(),obj);
if((obj instanceof Pp)) console.log("obj is instance of Pp , with sumOfP1P2P3() method");
console.log("obj.sumOfP1P2P3()="+obj.sumOfP1P2P3()); //ok : 579abc
```

Mixin set of additional methods with Object.assign(C1.prototype , mixinXyz):

exemple :

```
class C1 {
    constructor(id=null,label="?"){
        this.id=id; this.label=label;
    }

    displayId(){
        console.log(`id=${this.id}`);
    }
}
```

```

let myMixin = {
  //mixin object = set of additional methods (without real inheritance):

  labelToUpperCase(){
    this.label = this.label.toUpperCase();
  },
  displayLabel(){
    console.log(`label=${this.label}`);
  }
}

let objet = new C1(1,"abc");
//objet.displayLabel();//not a method of C1
Object.assign(C1.prototype,myMixin); //ajouter méthodes de myMixin à C1
objet.displayId();
objet.labelToUpperCase();
objet.displayLabel(); //ABC
let objetBis = new C1(2,"def");
objetBis.displayId();
objetBis.displayLabel();//def

```

## 1.8. Notions "orientée objet" qui ne sont pas gérées pas es2015

Les éléments "orientés objets" suivants ne sont pas pris en charge par un moteur javascript/es2015 mais sont pris en charge par le langage "typescript" (".ts" à traduire en ".js" via babel ou "tsc" ) :

- classes et méthodes abstraites (mot clef "abstract")
- visibilités "public" , "private" , "protected"
- interfaces (mots clefs "interface" et "implements" )
- "public" , "private" ou "protected" au niveau des paramètres d'un constructeur pour définir automatiquement certaines variables d'instances (attributs)
- ...

# IX - Annexe – Promise es2015 , async/await es2017

## 1. Promise (es2015)

### 1.1. L'enfer des "callback" (sans promesses) :

Beaucoup d'api javascript ont été conçues pour fonctionner en mode asynchrone (sans blocage). Par exemple , pour séquentiellement saisir x, saisir y et calculer x+y , le code nécessaire qui serait très simple et très lisible en C/C++ ou java est assez complexe dans l'environnement node-js :

```
var stdin = process.stdin;
var stdout = process.stdout;

function ask(question, callback) {
    stdin.resume();
    stdout.write(question + ": ");
    stdin.once('data', function(data) {
        data = data.toString().trim();
        callback(data);
    });
}

//utilisation chaînée avec callbacks imbriquées:
ask("x", function(valX){
    var x=Number(valX);
    ask("y", function(valY){
        var y=Number(valY);
        var res=x+y ;
        console.log("res = (x+y)=" +res);
        process.exit();
    });
});
```

### 1.2. Principe de fonctionnement des promesses (Promise)

Lorsque l'on déclenche via un appel de fonction un traitement asynchrone dont le résultat ne sera prêt/connu que dans le futur , on peut retourner immédiatement un objet de type "Promise" qui encapsule l'attente d'une réponse promise.

Le résultat promis qui sera récupéré en différé dans le temps est soit une réponse positive (promesse tenue) soit une réponse négative (erreur / promesse rompue) .

A l'intérieur de la fonction asynchrone appelée, on crée et retourne une promise via l'instruction

```
return new Promise((resolve,reject) => { if(...) resolve(...) else reject(...) ; } ) ;
```

//où *resolve* et *reject* sont des noms logiques de callbacks appelées dans le futur

//pour transmettre l'issue positive ou négatif du traitement asynchrone .

A l'extérieur , l'appel s'effectue via la syntaxe



```
.then((resolvedValue)=>{ .... } , (rejectedValue) => { ... } ) ;
```

ou bien

```
.then((resolvedValue)=>{ ....})  
.catch((rejectedValue) => { ... } ) ;
```

**NB:** Si à l'intérieur d'un `.then(()=>{...})` on appelle et retourne une fonction asynchrone retournant à son tour une autre "Promise" , on peut alors enchaîner d'une manière lisible une séquence d'appel à d'autres `.then()` qui seront alors exécutés les uns après les autres au fur et à mesure de la résolution des promesses asynchrones :

```
appelAsynchrone1RetournantPromesse1(...)  
.then((resPromesse1)=>{ .... ; return appelAsynchrone2RetournantPromesse2(...);})  
.then((resPromesse2)=>{ .... ; return appelAsynchrone2RetournantPromesse3(...);})  
.then((resPromesse3)=>{ .... ; })  
.catch((premiereErreurPromesse1ou2ou3)=>{...});
```

**NB :** avant d'exister en version normalisée "es2015" , les "Promises" avaient été prises en charge via l'ancienne bibliothèque "q" (*var deferred = Q.defer(); .... deferred.resolve(data); ... return deferred.promise; ) avec même utilisation .then(...).then(...).catch(...)* .

Les "Promises" étaient déjà beaucoup utilisées à l'époque de es5/angular-js (avant 2015 et es6/es2015) .

Exemple (avec Promise/es2015) :

```
var stdin = process.stdin;  
var stdout = process.stdout;  
  
function ask_ (question) {  
    return new Promise ((resolve,reject)=> {  
        stdin.resume();  
        stdout.write(question + ": ");  
        stdin.once('data', function(data) {  
            data = data.toString().trim();  
            if(data=="fin")  
                reject("end/reject");  
            else  
                resolve(data);  
        });  
    });  
}  
  
var x,y,z;  
//calcul (x+y)*z après enchaînement lisible (proche séquentiel) de "saisir x" , "saisir y" , "saisir z":  
ask_ ("x")  
.then((valX)=>{ x=Number(valX); return ask_ ("y");})  
.then((valY)=> { y=Number(valY); let res=x+y ;  
                    console.log("(x+y)=" +res);  
                    return ask_ ("z");  
                })  
.then((valZ)=> { z=Number(valZ); let res=(x+y)*z ;  
                    console.log("(x+y)*z=" +res);  
                    process.exit();
```

```

    })
    .catch((err)=>{console.log(err);process.exit();});

```

### 1.3. Autre exemple simple (sans et avec "Promise"):

```

function strDateTime() {
    return (new Date()).toLocaleString();
}

const affDiffere = () => {
    setTimeout (()=> {console.log("after 2000 ms " + strDateTime());} , 2000);
};

```

Version sans "Promise" avec callbacks imbriquées :

*//NB : via .setTimeout(...., delay) et return { responseJsData } ; on simule ici une récupération de //données via un appel asynchrone vers une base de données ou un WS REST.*

```

const getUserInCb =
  (cbWithName) => {
    setTimeout (()=> { cbWithName({ name : "toto" });} , 2000);
  };

const getAddressFromNameInCb =
  (name , cbWithAddress) => {
    setTimeout (()=> { cbWithAddress({ adr : "75000 Paris for name="+name });}
      , 1500);
  };

console.log("debut :" + strDateTime() );
affDiffere();

getUserInCb(
  (user) => {
    console.log("username=" + user.name);
    getAddressFromNameInCb(user.name,
      (address) => { console.log("address=" + address.adr ); }
    );
  }
);

console.log("suite :" + strDateTime() );

```

résultats:

```

debut :2019-4-23 16:46:24
suite :2019-4-23 16:46:24

```

after 2000 ms 2019-4-23 16:46:26  
username=toto  
address=75000 Paris for name=toto

Même exemple avec "Promise es6/es2015" :

```
function getUserFromIdAsPromise(id){
  return new Promise (
    (resolveCbWithName, rejectCb) => {
      setTimeout (()=> { if(id) resolveCbWithName({ name : "toto" });
                        else rejectCb("id should not be null!");
                        }, 2000);
    });
}

function getAddressFromNameAsPromise(name){
  return new Promise (
    (resolveCbWithAddress) => {
      setTimeout (()=> { resolveCbWithAddress({ adr : "75000 Paris for name="
                                                +name });}, 1500);
    });
}

console.log("debut :" + strDateTime() ); affDiffere();

getUserFromIdAsPromise(1)
//getUserFromIdAsPromise(null)
  .then( (user) => { console.log("username=" + user.name);
                    //returning new Promise for next then() :
                    return getAddressFromNameAsPromise(user.name);
                  })
  .then( (address) => { console.log("address=" + address.adr ); } )
  .catch(error => { console.log("error:" + error); } );

console.log("suite :" + strDateTime() );
```

Résultats avec id=1 :

debut :2019-4-23 17:18:44  
suite :2019-4-23 17:18:44  
after 2000 ms 2019-4-23 17:18:46  
username=toto  
address=75000 Paris for name=toto

Résultats avec id=null :

debut :2019-4-23 17:33:17  
suite :2019-4-23 17:33:17  
after 2000 ms 2019-4-23 17:33:19  
error:id should not be null!

L'exemple ci-dessus montre que :

- l'on peut nommer comme on le souhaite les callbacks "resolve" et "reject" . Ce qui permet quelquefois de rendre le code plus intelligible
- la callback "reject" est facultative

## 1.4. Propriétés des "Promises"

```
fairePremiereChose()  
.then(result1 => faireSecondeChose(result1))  
.then(result2 => faireTroisiemeChose(result2))  
.then(finalResult3 => {  
  console.log('Résultat final : ' + finalResult3);  
})  
.catch(failureCallback);
```

où

```
(resultatAppelPrecedent) => faireNouvelleChose(resultatAppelPrecedent)
```

est synonyme de

```
(resultatAppelPrecedent) => { return faireNouvelleChose(resultatAppelPrecedent) ; }.
```

Si aucun catch , il est éventuellement possible de traiter l'événement "**unhandledrejection**"

```
window_or_worker.addEventListener("unhandledrejection", event => {  
  // Examiner la ou les promesse(s) qui posent problème en debug  
  // Nettoyer ce qui doit l'être quand ça se produit en réel  
}, false);
```

Dans des cas "ultra simples" ou "triviaux" , on pourra éventuellement créer et retourner **une promesse à résolution ou rejet immédiat** avec une syntaxe de ce type :

```
argValue => Promise.resolve(argValue);  
//version abrégée de argValue => new Promise((resolve)=>resolve(argValue))  
errMsg => Promise.reject(errMsg);
```

## 1.5. Compositions (all , race, ...)

On peut déclencher des traitements asynchrones en parallèle et attendre que tout soit fini pour analyser globalement les résultats :

```
Promise.all([func1(), func2(), func3()])
  .then(([resultat1, resultat2, resultat3]) => { /* utilisation de resultat1/2/3 */ });
```

La variante ci-après permet de déclencher des traitements asynchrones en parallèle et attendre que le premier résultat (retourné par la fonction asynchrone la plus rapide) :

```
Promise.race([func1(), func2()])
  .then( (firstReturnedValue) => { console.log(firstReturnedValue); });
```

Exemple :

```
function getUppercaseDataAfterDelay(data, delay){
  return new Promise (
    (resolve) => {
      setTimeout (()=> { resolve(data.toUpperCase());}, delay);
    }
  );
}

Promise.all( [ getUppercaseDataAfterDelay("abc",2000) ,
  getUppercaseDataAfterDelay("def",1500) ] )
  .then ( ([ res1 , res2 ]) => { console.log(">>" + res1 + "--" + res2 + "<<"); } );

Promise.race( [ getUppercaseDataAfterDelay("abc",2000) ,
  getUppercaseDataAfterDelay("def",1500) ] )
  .then ( (firstResult) => { console.log(">>>" + firstResult + "<<<"); } );
```

>>>DEF<<<

>>ABC--DEF<<

## 1.6. Appel ajax via api fetch et Promise

L'api "**fetch**" supportée par certains navigateurs modernes utilise en interne l'api "Promise" de façon à déclencher des appels HTTP/ajax en mode GET ou POST ou autres.

Exemple en mode GET :

```
function myGenericJsGetFetchData(url){
  return new Promise((resolveWithJsData,reject)=>{
    fetch(url)
      .then( (response) => {
        if (response.status !== 200) {
          var errString = 'Problem. Status Code: ' + response.status;
          console.log(errString); reject(errString); return;
        }
        // Examine the text in the response :
        response.json().then(function(data) {
          resolveWithJsData(data);
        })
      })
      .catch((err) =>{ console.log('Fetch Error :-S', err); reject(err); });
  });
}
```

```
myGenericJsGetFetchData("/rest/produit/" + numProd)
  .then( (data) => { console.log(data);
    var jsonString = JSON.stringify(data);
    document.querySelector("#resProd").innerHTML = jsonString;
  })
  .catch((err) => { console.log(err); });
```

Exemple partiel en mode POST :

```
fetch(url,{ method: 'POST' ,
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body : JSON.stringify(jsObj)
})
  .then( (response) => {
    if (response.status !== 200) {
      var errString = 'Problem. Status Code: ' + response.status;
      console.log(errString); return;
    }
    response.json().then(function(data) {console.log(JSON.stringify(data));})
  })
  .catch((err) =>{ console.log('Fetch Error :-S', err); });
```

## 2. async/await (es2017)

**async** et **await** sont de nouveaux mots clefs de **es2017** (inspiré de typescript et de l'univers .net/Microsoft) .

Ces nouveaux mots clefs permettent de simplifier les enchaînement de fonctions asynchrones en générant et attendant automatiquement des promesses ("Promise de es2015") .

### 2.1. Principes async/await :

- **return resultat** dans une fonction **async**  
est (depuis es2017) équivalent à **return new Promise.resolve(resultat);**
- **throw new Error('erreur')** dans une fonction **async**  
est (depuis es2017) équivalent à **return new Promise.reject(new Error('erreur'));**
- **await** permet d'attendre la résolution d'une promesse (liée à un sous appel asynchrone) et de récupérer la valeur dans une variable (équivalent de `.then(...)` automatique) .  
**NB : *await ne peut être utilisé qu'au sein d'une fonction préfixée par async* .**
- au sein d'une fonction préfixée par **async**, un bloc **try { ... } catch { ... }** ordinaire permet de récupérer aussi bien certaines exceptions synchrones que certains échecs liés à des promesses non tenues par des sous appels asynchrones déclenchés via **await** .  
Autrement dit : **try { await appel\_async1(...) ;  
                    await appel\_async2(...) } catch { ... }**  
peut remplacer `appel_async1().then( () =>... ; return appel_async2(...) ; )  
                    .then(()=>...)  
                    .catch((e)=>... ) ;`

### 2.2. Exemple 1 (async/await):

*Preliminaire (avec "Promise" ordinaire) :*

```
function strDateTime() {
    return (new Date()).toLocaleString();
}

function myGenericTimeoutPromise(cbToDelay,delay){
    return new Promise (
        (resolve) => {
            setTimeout (()=> { resolve(cbToDelay());}
                        , delay);
        });
}
```

```
const affDiffere = () => {
    myGenericTimeoutPromise()=> {return("after 2000 ms " + strDateTime());} , 2000)
    .then((message)=> {console.log("ok - " + message); });
};

console.log("debut :" + strDateTime() ); //debut :2019-4-30 15:50:04
affDiffere(); //ok - after 2000 ms 2019-4-30 15:50:06
```

```
async function getUserFromIdAsAutomaticPromise(id){
    const user = await myGenericTimeoutPromise(
        ()=> { return { name : "toto" };}, 2000);
    if(id) return user; //as Promise.resolve(user)
    else throw new Error("id should not be null!");//as Promise.reject(...)
}
```

```
async function getAddressFromNameAsAutomaticPromise(name){
    const address = await myGenericTimeoutPromise(
        ()=> { return { adr : "75000 Paris for name="+name }; } , 1500);
    return address; //as Promise.resolve(address)
}
```

```
function appelsClassiquesPromises(){
getUserFromIdAsAutomaticPromise(1)
//getUserFromIdAsAutomaticPromise(null)
    .then( (user) => { console.log("username=" + user.name);
        //returning new Promise for next then() :
        return getAddressFromNameAsAutomaticPromise(user.name);
    })
    .then( (address) => { console.log("address=" + address.adr ); } )
    .catch( (err) => { console.log("my error:" + err.message); } );
}

appelsClassiquesPromises();
```

```
async function appelsViaAwait(){
    try{
        const user = await getUserFromIdAsAutomaticPromise(1);
        //const user = await getUserFromIdAsAutomaticPromise(null);
        console.log("username=" + user.name);
        const address = await getAddressFromNameAsAutomaticPromise(user.name);
        console.log("address=" + address.adr );
    }
    catch(err){
        console.log("my error:" + err);
    }
}

appelsViaAwait()
//.catch( (err) => { console.log("my error:" + err); } );
```

==>

*username=toto (après 2s)*

*address=75000 Paris for name=toto (encore après 1.5s)*



## 2.3. Exemple 2 (async/await) :

```

var stdin = process.stdin;
var stdout = process.stdout;

function ask_(question) {
    return new Promise ((resolve,reject)=> {
        stdin.resume();
        stdout.write(question + ": ");
        stdin.once('data', function(data) {
            data = data.toString().trim();
            if(data=="fin")
                reject("end/reject");
            else
                resolve(data);
        });
    });
}

async function ask_and_compute_x_plus_y(){
    try{
        let x,y;
        const valX = await ask_("x"); x=Number(valX);
        const valY = await ask_("y"); y=Number(valY);
        let xPlusY=x+y ;console.log("(x+y)=" +xPlusY);
        return xPlusY;
    }
    catch(e){
        console.log(e);
        throw new Error("xPlusY-error:"+e);
    }
}

async function x_plus_y_mult_z(){
    try{ /*
        const valX = await ask_("x"); let x=Number(valX);
        const valY = await ask_("y"); let y=Number(valY);
        let xPlusY=x+y ;console.log("(x+y)=" +xPlusY);
        */
        const xPlusY = await ask_and_compute_x_plus_y();
        const valZ = await ask_("z"); const z=Number(valZ);
        let res=xPlusY * z ;console.log("(x+y)*z=" +res);
    }
    catch(e){
        console.log(e);
    }
    process.exit();
}

x_plus_y_mult_z();

```

```

==>
x: 5
y: 6
(x+y)=11
z: 3
(x+y)*z=33

```

Equivalent sans async/await avec .then().then.catch() de es6/es2015 :

```

var x,y,z; //(x+y)*z
ask_("x")
.then((valX)=>{ x=Number(valX); return ask_("y");})
.then((valY)=> { y=Number(valY); let res=x+y ;
                  console.log("(x+y)=" +res);
                  return ask_("z");
                })
.then((valZ)=> { z=Number(valZ); let res=(x+y)*z ;
                  console.log("(x+y)*z=" +res);
                  process.exit();
                })
.catch((err)=>{console.log(err);process.exit();});

```

## 2.4. Combinaison de async/await avec Promise.all et Promise.race

```

function getUppercaseDataAfterDelay(data, delay){
  return new Promise (
    (resolve)=> {
      setTimeout (()=> { resolve(data.toUpperCase());}, delay);
    });
}

async function test_await_Promise_all_and_race(){

  //attendre les résultats de 2 traitements asynchrones lancés en parallèle :
  const [ res1 , res2 ] = await Promise.all( [ getUppercaseDataAfterDelay("abc",2000) ,
                                             getUppercaseDataAfterDelay("def",1500) ] );
  console.log(">>" +res1+"--"+res2+"<<");

  //attendre le premier résultat (le plus rapidement retourné)
  // de 2 traitements asynchrones lancés en parallèle :
  const firstResult = await Promise.race( [ getUppercaseDataAfterDelay("abc",2000) ,
                                             getUppercaseDataAfterDelay("def",1500) ] );
  console.log(">>>" +firstResult+"<<<");
}
test_await_Promise_all_and_race();

```

```

==>
>>ABC—DEF<< (2 s après)
>>>DEF<<< (1.s après)

```

...

# X - Annexe – Modules es2015

## 1. Modules (es2015)

### 1.1. Types de modules (cjs , amd , es2015 , umd , ...)

Beaucoup de technologies javascript modernes s'exécutent dans un environnement prenant en charge des modules (bien délimités) de code (avec import/export) . Le développement d'une application "Angular2+" s'effectue à fond dans ce contexte.

Les principales technologies de "modules javascript" sont les suivantes :

- **CommonJS (cjs)** – modules "synchrones" , **syntaxe** "var xyz = **requires**('xyz')"  
*NB* : node (nodeJs) utilise partiellement les idées et syntaxes de CommonJS .
- **AMD** (Asynchronous Module Definition) avec chargements asynchrones
- **ES2015 Modules** : syntaxiquement standardisé , mots clef "import {...} from '...'" et export pour la gestion dynamique des modules(possibilité de générer des bundles (es5 ou es6) regroupant plusieurs modules statiquement assemblés ensemble ) .
- **SystemJS** (très récent et pas encore complètement stabilisé) supporte en théorie les 3 technologies de modules précédentes (cjs , amd, es2015) . SystemJS nécessite certains paramétrages (quelquefois complexes) et s'utilise assez souvent avec gulp .  
--> Attention : SystemJS s'est révélé assez instable et n'est pas toujours ce qui y a de mieux .

Il existe aussi les **formats de modules suivants** :

- **umd** (universal module definition) – *fichiers xyz.umd.js*
- **iife** (immediately-invoked function expression) – *fonctions anonymes auto-exécutées*

### 1.2. Modules "es6/es2015" et organisation en fichiers

Les modules ES6 :

- ont une syntaxe simple et sont basés sur le découpage en fichiers (un module = un fichier),
- sont automatiquement en mode « strict » (rigoureux),
- offrent un support pour un chargement asynchrone et permet de générer des bundles "statiques" via rollup ou webpack .

Les modules doivent exposer leurs variables et méthodes de façon explicite. On dispose donc des deux mots clés :

- **export** : pour exporter tout ce qui doit être accessible en dehors du module,
- **import** : pour importer tout ce qui doit être utilisé dans le module (et qui est donc exporté par un autre module).

### 1.3. Exemple avec chargement direct depuis navigateur récent:

#### *math-util.js*

```
export function additionner(x , y) {
  return x + y;
}

export function multiplier(x, y) {
  return x * y;
}
```

#### *ou bien*

```
function additionner(x , y) {
  return x + y;
}

function mult(x, y) {
  return x * y;
}

export { additionner, mult as multiplier };
```

#### *dom-util.js*

```
export class DomUtil {
  static displayInDiv(divId,message){
    document.querySelector('#'+divId).innerHTML = message;
  }

  static multilineMessage(...args){
    //NB: la syntaxe ... permet de récupérer tous (ou bien les derniers) arguments (en nombre variable)
    //sous forme de tableau . Cette syntaxe est permise en mode "strict" alors que la
    //syntaxe Dom.multilineMessage.arguments est interdite en mode "strict" (dans module es6)
    let nb_arg=args.length;
    let messages=null;
    if(nb_arg>=1) messages=args[0];
    for(let i=1;i<nb_arg;i++){
      messages+="<br/>" + args[i];
    }
    return messages;
  }
}
```

}

**main.js**

```

import { additionner as add, multiplier } from "./math-util.js";
import { DomUtil } from "./dom-util.js";

function carre(x){
  return multiplier(x,x) ;
}
/*
var msg1 = "Le carre de 5 est " + carre(5); console.log(msg1);
var msg2 = "4 * 3 vaut " + multiplier(4, 3); console.log(msg2);
var msg3 = "5 + 6 vaut " + add(5, 6); console.log(msg3);
document.querySelector('#divA').innerHTML = msg1 + "<br/>" + msg2 + "<br/>" + msg3;
*/
DomUtil.displayInDiv('divA',
  DomUtil.multilineMessage(
    "Le carre de 5 est " + carre(5),
    "4 * 3 vaut " + multiplier(4, 3),
    "5 + 6 vaut " + add(5, 6)
  ));

```

```

<html>
  <body>
    <script type="module" src="main.js"></script>
    <div id="divA"></div>
  </body>
</html>

```

**Attention :**

- **type="module"** est **indispensable** mais **n'est supporté que par les navigateurs récents**
- la page html et les modules javascripts doivent être téléchargés via http (par exemple via `http://localhost:3000/` et `lite-server`) .

## 1.4. default export (one per module)

**xy.js**

```
export function mult(x, y) {
  return x * y;
}

//export default function_or_object_or_class (ONE PER MODULE)
export default {
  name : "xy",
  features : { x : 1 , y: 3 }
}
```

**main.js**

```
import xy , { mult } from "./xy.js";
...
let msg = xy.name + "--" + JSON.stringify(xy.features) + "--" + mult(3,4);
```

## 1.5. Agrégation de modules

**mod1.js**

```
export function f1(msg) { return "*" + msg }
export function f2(msg) { return "*" + msg; }
export function f2bis(msg) { return "*" + msg; }
```

**mod2.js**

```
export function f3(msg) { return "#" + msg }
export function f4(msg) { return "##" + msg; }
```

**mod1-2.js**

```
//agrégation de modules : mod1-2 = mod1 + mod2
//importer certains éléments du module "mod2" et les ré-exporter:
export { f1, f2 } from "./mod1.js"
//importer tous les éléments du module "mod2" et les ré-exporter tous :
export * from "./mod2.js"
```

*main.js*

```
import { f1 , f2 , f3 , f4 } from "./mod1-2.js";
let f_msg=f1('abc')+'-'+f2('abc')+'-'+f3('abc')+'-'+f4('abc');
```

ou bien

```
import * as f from "./mod1-2.js";
let f_msg=f.f1('abc')+'-'+f.f2('abc')+'-'+f.f3('abc')+'-'+f.f4('abc');
```

## 1.6. Technologies de "packaging" (webpack , rollup, ...) et autres

De façon à éviter le téléchargement d'une multitude de petits fichiers , il est possible de créer des gros paquets appelés "**bundles**" .

Les principales technologies de packaging "javascript" sont les suivantes :

- **webpack** (mature et supportant les modules "csj" , "amd" , ...)
- **rollup** (récent et pour modules "es2015" ) . Rollup est une fusion intelligente de n fichiers en 1 (remplacement des imports/exports par sous contenu ajustés , prise en compte de la chaîne des dépendances en partant par exemple de main.js )
- **SystemJs-builder** (technologie assez récente et un peu moins mature )

### Autres technologies annexes (proches) :

- **browserify** : technologie déjà assez ancienne permettant de faire fonctionner un module "nodeJs" dans un navigateur après transformation.
- **babel** : transformation (par exemple es2015 vers es5)
- **uglify** : minification (enlever tous les espaces et commentaires inutiles, simplifier noms des variables, ...) → code beaucoup plus compact (xyz.min.js) .
- **gzip** : compression .Les fichiers bundlexy.min.js.gz sont automatiquement traités par quasiment tous les serveurs HTTP et les navigateurs : décompression automatique après transfert réseau).

## 1.7. Packaging web via rollup / es2015 et npm

L'un des principaux atouts de la structure des "modules es2015" tient dans les imports statiques et précis qui peuvent ainsi être analysés pour une génération optimisée des bundles à déployer en production.

Au lieu d'écrire `var/const xyModule = requires('xyModule')` comme en "cjs", la syntaxe plus précise de es2015 permet d'écrire **import** { **Composant1** , ... , **ComposantN** } **from** 'xyModule' .

Ainsi l'optimisation dite "**Tree-Shaking**" permet d'exclure tous les composants jamais utilisés de certaines librairies et la taille des bundles générés est plus petite.

La technologie de packaging "rollup" qui est spécialisée "es2015" peut ainsi exploiter cette optimisation via la chaîne de transformation suivante :

```
main[.es2015].js
  with import          → rollup → myBundle.es2015.js → myBundle.es5.js
subModuleXx[.es2015].js
subModuleYy[.es2015].js      (*)
```

(\*) **es2015-to-es5** via **babel** (*presets* : *es2015* ou *[env]*) ou autres permet d'obtenir un bundle interprétable par quasiment tous les navigateurs.

```
npm install -g rollup
npm init
npm install --save-dev babel-cli
npm install --save-dev babel-preset-env
```

### *package.json*

```
{
  "name": "with-modules-and-rollup",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "es6bundle-to-es5": "babel dist/build-es2015 -d dist/build-es5",
    "build" : "rollup --config rollup.config.js && babel dist/build-es2015 -d dist/build-es5"
  },
  "devDependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-env": "^1.7.0"
  }
}
```

```
rollup --config rollup.config.js
```

ou bien

```
npm run build
```



**rollup.config.js**

```
export default {
  input: 'src/main.js',
  output : {
    file: 'dist/build-es2015/main-bundle.js',
    format: 'iife'
  }
};
```

**.babelrc**

```
{
  "presets": ["env"]
}
```

Dans la configuration *rollup.config.js*,

input :.../main.js correspond au point d'entrée (.js) autrement dit la racine d'un arbre import/export entre différents fichiers (es6) qui seront analysés et gérés par rollup.

Il peut quelquefois y avoir plusieurs input/output dans le fichier rollup.config.js.

le **format** peut être "cjs" pour une future interprétation via node/nodeJs

ou "iife" pour une future interprétation via html/js (navigateur)

## 1.8. Packaging web via webpack

--> voir chapitre ou annexe suivante ...

## 1.9. Pattern module et IIFE (compatible es5)

IIFE (i.e. *Immediately-Invoked Function Expression* ou Expression de fonction invoquée immédiatement) permet de mettre en oeuvre le pattern "module" dans une syntaxe comprise par n'importe quel navigateur (de niveau es5) :

Une **IIFE** s'écrit de cette façon:

```
var ModuleXy = (function() {
  // Bloc de code à exécuter
})();
```

Pour utiliser une IIFE pour définir un module, on peut écrire par exemple:

```
var ModuleXy = (function() {
  var self = {};
  function privateFunction() {
    // ...
  };

  self.publicFunc = function() {
    privateFunction();
  };
});
```

```
    return self;  
  })();
```

Cette écriture permet de définir une variable appelée `ModuleXy` qui va contenir des membres et des fonctions, ce qui correspond à la notion de module:

- `privateFunction()` est une fonction privée.
- `publicFunc()` est une fonction publique.

Pour utiliser ce type de module, on peut écrire

```
moduleXy.publicFunc();
```

# XI - Annexe – aspects divers et avancés es2015

## 1. Aspects divers et avancés (es2015)

Attention :

Les éléments exposés dans ce chapitre sont complexes et ne sont utiles que dans certains cas très pointus . C'est pour les développeurs déjà bien expérimentés en javascript (ayant envie de programmer une librairie de code réutilisable du genre "mini framework xyz" par exemple).

### 1.1. WeakMap et WeakSet

Rappel : Map et Set sont de nouvelles structures de données introduites par la version es2015.

Par rapport à un simple objet javascript (déjà en interne géré comme une map entre noms et valeurs de propriétés) , une Map (de es2015) peut éventuellement comporter des clefs de types quelconques (pas obligatoirement de type string) .

Les variantes "**WeakMap**" et "**WeakSet**" (de es2015) apportent :

- beaucoup de restrictions (pas d'itération possible , moins de méthodes disponibles)
- une gestion différente de la mémoire (weak-reference permettant d'éviter quelquefois des fuites de mémoire)

```
//non primitive key for WeakMap and non primitiveValue for WeakSet
class MyObjectClass {
    constructor(v){ this.v = v; this.V = v.toUpperCase(); }
}
```

Une "WeakMap" ne peut comporter que des clefs de type "référence sur objet" (les types "primitifs sont interdits) .

Si après avoir ajouter par exemple 10 entrées de type

[ copieDeRéférenceSurObjetJouantRoleDeClef , valeur ] ,

du code externe à la "WeakMap" supprime (par exemple via un **delete**) une référence (copiée) sur un des objets jouant de rôle de Clef , alors cet objet ne sera référencé que par une référence faible (weak) interne à la map .

Le ramasse-miettes (garbage collector) du moteur javascript va alors considérer que cet objet n'a plus de référence forte/ordinaire pointant vers lui et va alors supprimer l'objet clef et va indirectement supprimer l'entrée associée dans la "WeakMap" qui comportera alors un élément de moins.

```
// Weak Maps
// Weak Maps (!!! with no .size , no .forEach)
//NB: les éléments stockés dans une weakMap (dont les clefs sont obligatoirement des références
//sur des objets) ne seront conservés que si il existe encore une autre référence (externe)
//sur la même valeur "objet" d'une clef.
//Autrement dit la référence constituée par la clef d'une entrée d'une WeakMap est considérée
//comme faible et ne compte pas dans la logique de fonctionnement du "garbage collector" .
var wm = new WeakMap();
console.objKey1 = new MyObjectClass("key1");
console.objKey2 = new MyObjectClass("key2");
wm.set(console.objKey1,"val1");
wm.set(console.objKey2,"val2");
console.log("in weakMap , for console.objKey1 , value is " + wm.get(console.objKey1));
console.log("in weakMap , for console.objKey2 , value is " + wm.get(console.objKey2));
delete console.objKey2;
console.log("after delete console.objKey2 , in weakMap , for console.objKey1 , value is " +
wm.get(console.objKey1));
console.log("after delete console.objKey2 , in weakMap , for console.objKey2 , value is " +
wm.get(console.objKey2));
if(wm.has(console.objKey1))
    console.log("weakMap wm comporte encore une valeur associée à console.objKey1");
if(!wm.has(console.objKey2))
    console.log("weakMap wm ne comporte plus de valeur associée à console.objKey2");
```

```
// Weak Sets
// Weak Sets is not iterable and not very useful !!!!
var ws = new WeakSet();
console.obj1=new MyObjectClass("obj1");
console.obj2=new MyObjectClass("obj2");
ws.add(console.obj1);
ws.add(console.obj2);
delete console.obj2;
if(ws.has(console.obj1))
    console.log("weakSet ws comporte encore console.obj1");
if(!ws.has(console.obj2))
    console.log("weakSet ws ne comporte plus console.obj2");
```

## 1.2. nouvelles méthodes es6 sur Array, String, Number, Math

```
Number.EPSILON
Number.isInteger(Infinity) // false
```

```
Math.hypot(3, 4) // 5
```

```
"abcde".includes("cd") // true
"abc".repeat(3) // "abccabccabc"
```

```
Array.from(document.querySelectorAll("*")) // Returns a real Array
Array.of(1, 2, 3) // Similar to new Array(...), but without special one-arg behavior
[0, 0, 0].fill(7, 1) // [0,7,7]
[1,2,3].findIndex(x => x === 2) // 1
["a", "b", "c"].entries() // iterator [0, "a"], [1, "b"], [2, "c"]
["a", "b", "c"].keys() // iterator 0, 1, 2
["a", "b", "c"].values() // iterator "a", "b", "c"
```

```
Object.assign(Point, { origin: new Point(0,0) })
```

## 1.3. Symbols (clefs uniques)

```
const symbol1 = Symbol();
const symbol2 = Symbol(42);
const symbol3 = Symbol('foo');

console.log(typeof symbol1);
// expected output: "symbol"

console.log(symbol3.toString());
// expected output: "Symbol(foo)"

console.log(Symbol('foo') === Symbol('foo'));
// expected output: false
```

### Utilisation classique n° 1 : clef pour propriété (privée ou ...)

```
let ageKey=Symbol('age'); //as speudo private key/property
let sizeKey=Symbol('size'); //as speudo private key/property
class Person {
  constructor(nom=null, age = 0 , size=0){
    this.nom=nom;
    this[ageKey]=age;
    this[sizeKey]=size;
  }

  get age(){
```

```

        return this[ageKey];
    }

    set age(newAge){
        if(newAge>=0)
            this[ageKey]=newAge;
    }

    get size(){
        return this[sizeKey];
    }

    set size(newSize){
        if(newSize>=0)
            this[sizeKey]=newSize;
    }

    logSymbolProperties(){
        for(let key of Object.getOwnPropertySymbols(this)){
            console.log(key.toString()+"_"+this[key]);
        }
    }
}

let p1 = new Person("toto",30);
console.log(p1, JSON.stringify(p1) , p1.age);
p1.nom="Toto"; p1.age=40; p1.size=160;
console.log(p1, JSON.stringify(p1) , p1.age);
p1.age=-5; //no effect , newAge invalid
p1.size=-23;
console.log(p1, JSON.stringify(p1) , p1.age);
p1.logSymbolProperties();

```

==>

```

Person { nom: 'toto', [Symbol(age)]: 30, [Symbol(size)]: 0 } '{"nom":"toto"}' 30
Person { nom: 'Toto', [Symbol(age)]: 40, [Symbol(size)]: 160 } '{"nom":"Toto"}' 40
Person { nom: 'Toto', [Symbol(age)]: 40, [Symbol(size)]: 160 } '{"nom":"Toto"}' 40
Symbol(age)_40
Symbol(size)_160

```

### Utilisation classique n° 2 : constantes liées à certains concepts

Exemples : (couleurs , ...)

```
const COLOR_RED = Symbol('Red');
```

## 1.4. itérateurs et générateurs

Un itérateur est un objet technique de bas niveau ayant les principales caractéristiques suivantes :

- méthode **next()** pour itérer
- comporte une propriété **.value** (valeur quelconque du i ème élément)

- comporte une propriété booléenne **.done** (true si fin de parcours/itération)
- prédéfini sur beaucoup de structure de données (String , Set, Map , Array , ...)
- utilisé en interne de façon transparente par la nouvelle boucle **for(let e of collection)** de es6/es2015 .
- utilisé en interne de façon transparente par la syntaxe **...itérateur** au sein d'un dernier paramètre d'un appel de fonction (*rest parameters ...*) ou bien d'un paquets d'éléments d'un tableau (*spread operator ...*)
- correspond à la fonction prototype **[Symbol.iterator]** d'une structure de données

Exemple : itérateur prédéfini sur "String" :

```
var strAbc = "abc";
console.log(typeof strAbc[Symbol.iterator]); // "function"

let itStrAbc = strAbc[Symbol.iterator]();
console.log(itStrAbc.next()); // { value: "a", done: false }
console.log(itStrAbc.next()); // { value: "b", done: false }
console.log(itStrAbc.next()); // { value: "c", done: false }
console.log(itStrAbc.next()); // { value: undefined, done: true }

let itAbc = strAbc[Symbol.iterator]();
var tabAbc = [ 'a', 'b', 'c' ];
//let itAbc = tabAbc[Symbol.iterator]();
/*
let loopItem = null;
while((loopItem=itAbc.next()) && !loopItem.done) {
    console.log(loopItem.value);
}*/
for(let eltOfAbc of itAbc){
    console.log(">" + eltOfAbc);
} // >a >b >c
```

Construction d'un nouvel itérable élémentaire avec "function\*" et "yield" :

```
var monIterable = {};
monIterable[Symbol.iterator] = function* () {
    yield 'e1';
    yield 'e2'; //yield signifie "rendre , produire , donner , générer , ..."
    yield 'e3';
};
//NB: String, Array, TypedArray, Map et Set sont des itérables natifs
//car les prototypes de chacun ont tous une méthode Symbol.iterator.

for(let elt of monIterable){
    console.log(">>" + elt);
} //>>e1 >>e2 >>e3

var myArray1 = [ 'e0' , ...monIterable , 'e4' , 'e5' ];
var myArray2 = [ ...monIterable ];
console.log(myArray1); // [ 'e0', 'e1', 'e2', 'e3', 'e4', 'e5' ]
console.log(myArray2); // [ 'e1', 'e2', 'e3' ]
```

//Fonction génératrice élémentaire avec syntaxe "function\*" et "yield" :

```
function* idMaker(){
    var index = 0;
```

```

while(index<10)
  yield index++; //yield retourne la valeur et se met en pause (attente du futur appel)
}
//NB: le fait que la fonction génératrice (function*) soit prévue pour être appelée
//plusieurs fois via un itérateur et que yield établisse automatiquement une attente
//du prochain appel correspond à une fonctionnalité très spéciale du langage es6/es2015
//appelée PROTOCOLE d'itération .

//itérable1 basé sur générateur:
var genIt1 = idMaker();
console.log(genIt1.next().value); // 0
console.log(genIt1.next().value); // 1
console.log(genIt1.next().value); // 2
console.log("-----");
//itérable2 basé sur même générateur:
var genIt2 = idMaker();
console.log(genIt2.next().value); // 0
console.log(genIt2.next().value); // 1
console.log("-----");
//itérable3 basé sur même générateur:
var genIt3 = idMaker();
var myArray3 = [ ...genIt3 ];
console.log(myArray3);//[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

```

**NB:** Un générateur peut éventuellement être codé comme une méthode spéciale d'un objet littéral ou bien d'une classe :

```

class MyClass {
  //or const obj = {

    * generatorMethod() { ...
  }
}

```

Exemple :

```

class MyBasicFifo {
  constructor() { this.internalArray = [];
    this[Symbol.iterator]=this.fifoIteratorGenerator;
  }

  pop() { if(this.internalArray.length>0)
    return this.internalArray.pop();
  }

  push(elt) { this.internalArray.push(elt);
    let taille=this.internalArray.length;
    for(let i=taille-1;i>0;i--){
      this.internalArray[i]=this.internalArray[i-1];
    }
    this.internalArray[0]=elt;
  }

  * fifoIteratorGenerator() {
    var index = this.internalArray.length -1;

```



```
while(index>=0)
  yield this.internalArray[index--];
}
```

```
let fifo1 = new MyBasicFifo();
fifo1.push("a"); fifo1.push("b"); fifo1.push("c");
console.log(fifo1.pop());
console.log(fifo1.pop());
console.log(fifo1.pop());

fifo1.push("aa"); fifo1.push("bb"); fifo1.push("cc");
let fifo1It = fifo1.fifoIteratorGenerator();
let arr1 = [ ...fifo1It ]
console.log(arr1);

for(let e of fifo1){
  console.log(`>>${e}`);
}
```

==>

```
a
b
c
[ 'aa', 'bb', 'cc' ]
>>aa
>>bb
>>cc
```

liens à suivre pour approfondir le sujet :

- [https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/iterateurs\\_et\\_generateurs](https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/iterateurs_et_generateurs)

## 1.5. Proxies

Proxy permet de coder des **intercepteurs** .

Exemple simple :

```
const targetObject = {
  prenom: 'Jean',
  nom: 'Bon',
  taille: 1.75
};

const interceptorUpperCaseHandler = {
  get(target, propKey, receiver) {
    let val = target[propKey];
    // NO let val = receiver[propKey]; ==> STACK-OVERFLOW / get interceptor calling get , ...
    console.log('intercept get ' + propKey);
    if(typeof val==='string' && val!=null)
      val=val.toUpperCase();
    return val;
  }
};

const validateNumberHandler = {
  set(target, key, value) {
    if (key === 'age' || key === 'taille') {
      if (typeof value !== 'number' || Number.isNaN(value)) {
        throw new TypeError(key + ' must be a number')
      }
      if (value <= 0) {
        throw new TypeError(key + ' must be a positive number')
      }
    }
    target[key]=value; //default behavior
    return true; //indicate success
  }
};

var mixedHandler = Object.assign(interceptorUpperCaseHandler,validateNumberHandler);

const proxyObject = new Proxy(targetObject, interceptorUpperCaseHandler);
const proxyValidObject = new Proxy(targetObject, /*validateNumberHandler*/ mixedHandler);
//proxyValidObject.taille="abc"; --> TypeError : taille must be a number
//proxyValidObject.taille=-56; --> TypeError: taille must be a positive number
proxyValidObject.taille=1.80;
console.log("nouvelle taille=" + proxyValidObject.taille);

console.log(JSON.stringify(proxyObject));
```

==>

intercept get taille  
nouvelle taille=1.8  
intercept get toJSON

```

intercept get prenom
intercept get nom
intercept get taille
{"prenom": "JEAN", "nom": "BON", "taille": 1.8}

```

Autre exemple :

```

// Proxying a normal object :
var targetObj = {
  id : 1 ,
  label : "o1"
};
var objectDefaultValueHandler = {
  get (target, propertyName /*, receiver */) {
    /* let val = target[propertyName];
    if(val==undefined)
      return `interceptor default value for property ${propertyName}`;
    else
      return val; */
    return (propertyName in target) ? target[propertyName]
      : `interceptor default value for property ${propertyName}`;
  }
};

var p = new Proxy(targetObj, objectDefaultValueHandler);
console.log("p.id="+p.id); // 1
console.log("p.label="+p.label); // o1
console.log("p.p3="+p.p3); // interceptor default value for property p3
console.log("p.p4="+p.p4); // interceptor default value for property p4

```

Listes des "traps" / "intercepteurs" possibles :

- defineProperty(target, propKey, propDesc) : boolean
  - Object.defineProperty(proxy, propKey, propDesc)
- deleteProperty(target, propKey) : boolean
  - delete proxy[propKey]
  - delete proxy.foo // propKey = 'foo'
- get(target, propKey, receiver) : any
  - receiver[propKey]
  - receiver.foo // propKey = 'foo'
- set(target, propKey, value, receiver) : boolean
  - receiver[propKey] = value
  - receiver.foo = value // propKey = 'foo'
- getOwnPropertyDescriptor(target, propKey) : PropDesc|Undefined
  - Object.getOwnPropertyDescriptor(proxy, propKey)

- `getPrototypeOf(target) : Object|Null`
  - `Object.getPrototypeOf(proxy)`
- `setPrototypeOf(target, proto) : boolean`
  - `Object.setPrototypeOf(proxy, proto)`
- `has(target, propKey) : boolean`
  - `propKey in proxy`
- `isExtensible(target) : boolean`
  - `Object.isExtensible(proxy)`
- `ownKeys(target) : Array<PropertyKey>`
  - `Object.getOwnPropertyNames(proxy)` (only uses string keys)
  - `Object.getOwnPropertySymbols(proxy)` (only uses symbol keys)
  - `Object.keys(proxy)` (only uses enumerable string keys; enumerability is checked via `Object.getOwnPropertyDescriptor`)
- `preventExtensions(target) : boolean`
  - `Object.preventExtensions(proxy)`

Liste des "traps" / "intercepteurs" possibles pour fonctions :

- `apply(target, thisArgument, argumentsList) : any`
  - `proxy.apply(thisArgument, argumentsList)`
  - `proxy.call(thisArgument, ...argumentsList)`
  - `proxy(...argumentsList)`
- `construct(target, argumentsList, newTarget) : Object`
  - `new proxy(..argumentsList)`

Exemple (proxy fonction et methode) :

```
// Proxying a function object
var targetRepeatWordFct = function (n , word) {
  return word.repeat(n); //NB: String.repeat(n) est une nouvelle méthode de es2015
};

var fctHandler = {
  apply(targetFct, thisArg, argsList) {
    let res=targetFct.apply(thisArg, argsList); //appel de la fonction d'origine
    return res.toUpperCase(); //retourne le résultat transformé en majuscules
  }
};

var proxyFct = new Proxy(targetRepeatWordFct, fctHandler);
console.log( targetRepeatWordFct(3,"ha")); //hahaha
console.log( proxyFct(3,"ha")); //HAHAHA
```

Etant donné que les mécanismes internes de javascript traite un appel de méthode en 2 étapes :

1) get function from methodName as propertyName

2) function call

l'interception d'une méthode peut donc se faire de la façon suivante :

```
// Proxying a object with method
var targetObj = {
  id : 1 ,
  label : "obj1",
  idAndLabel(){ return ""+this.id+", "+this.label; }
};

var objectHandler = {
  get (target, propertyName /* , receiver */) {
    if((typeof target[propertyName])=== "function"){
      return function(...args){
        /* let origMethod = target[propertyName];
        let result = origMethod.apply(this, args);
        if(typeof result === 'string')
          result=result.toUpperCase();
        return result;
        */
        var proxyMethodFct = new Proxy(target[propertyName], fctHandler);
        //application (réutilisation) de fctHandler (de l'exemple précédent)
        return proxyMethodFct.apply(this, args);
      }
    }
    else { //normal attribute , not a function/method :
      let val = target[propertyName];
      if(typeof val==='string' && val!==null)
        val=val.toUpperCase();

      return val;
    }
  } //end of get()
};

var proxyObjWithMethod=new Proxy(targetObj,objectHandler);
console.log(proxyObjWithMethod.idAndLabel()); //1,OBJ1
```

## 1.6. Reflect Api

Etant donné qu'un objet javascript est codé en interne comme une map entre noms et valeurs de propriétés/méthodes, on pouvait déjà en es5 découvrir au runtime la structure d'un objet javascript quelconque et ajuster du code dynamique en conséquence.

La version "es6/es2015" a cependant apporté une API de "reflection" se voulant plus explicite et rigoureuse. Le coeur de cette Api est l'objet "**Reflect**".

Pas de ~~new Reflect(....)~~ mais des appels "static" (ex : Reflect.get(obj,"propertyName"))

Exemples :

```
var obj={
  id: 1,
  label : "obj1"
}

let valueOfIdProperty=Reflect.get(obj,"id"); //equivalent à obj["id"]
console.log("value of Property id = "+valueOfIdProperty); //value of Property id = 1
console.log("value of Property label = "+Reflect.get(obj,"label"));
//value of Property label = obj1

Reflect.set(obj, "label", "labelXy");
console.log("modified obj.label = "+obj.label); //modified obj.label = labelXy

console.log("obj="+JSON.stringify(obj)); //obj={"id":1,"label":"labelXy"}

let boolRes= Reflect.defineProperty(obj, "name", {value: 'nomQuiVaBien' ,
  writable : true, enumerable : true, configurable : true});
//propertyDescriptor with enumerable=true to see property in loop like for (.. in) or JSON.stringify
// configurable=true to enable changing attribute property (delete it , ...)
console.log("obj.name="+obj.name); //nomQuiVaBien
console.log("obj="+JSON.stringify(obj));
//obj={"id":1,"label":"labelXy","name":"nomQuiVaBien"}
Reflect.deleteProperty(obj, "name");
console.log("obj.name="+obj.name); //undefined
console.log("obj="+JSON.stringify(obj)); obj={"id":1,"label":"labelXy"}

console.log("obj has label property=" + Reflect.has(obj, "label")); //true
console.log("obj has name property=" + Reflect.has(obj, "name")); //false

var labelPropertyDescriptor = Reflect.getOwnPropertyDescriptor(obj, "label");
console.log("labelPropertyDescriptor="+JSON.stringify(labelPropertyDescriptor));
//labelPropertyDescriptor={"value":"labelXy","writable":true,"enumerable":true,"configurable":true}

let arrayOfPropKeys = Reflect.ownKeys(obj); //may ignoring inheritance in old version
console.log("arrayOfPropKeys="+arrayOfPropKeys); //arrayOfPropKeys=id,label

class Person {
  constructor(id=0,name='?') { this.id=id; this.name=name; }
}
class Employee extends Person {
  constructor(id=0, name='emp?' , salary=0) { super(id,name); this.salary=salary; }
}
var emp1 = new Employee(1,"employee1",1000);
console.log("properties of Employee="+Reflect.ownKeys(emp1)); //id,name,salary
//var allPropsIterator = Reflect.enumerate(emp1); is now obsolete : not use it !!!!
```

```

function addFct(x, y){
    return x + y;
}
console.log("10+20="+Reflect.apply(addFct, null /*thisArg*/, [10, 20]));//10+20=30

function functionForObject(x, y){
    return this.num + x + y;
}
let computeObj={
    num:30,
    methXy:functionForObject
}
var value = Reflect.apply(functionForObject, computeObj /*thisArg*/, [10, 20]);
console.log(value + " is equals to " + computeObj.methXy(10,20)); //60 is equals to 60

Reflect.preventExtensions(obj);//cannot add new property
console.log("obj is extensible after preventExtensions:"+Reflect.isExtensible(obj)); //false
obj.newAttr="newValue";//no effect
console.log("obj.newAttr="+obj.newAttr);//undefined

function constructorAB(a, b)
{
    this.a = a;
    this.b = b;
    this.fctAdd = function(){
        return this.a + this.b;
    }
}

var builtObj = Reflect.construct(constructorAB, [10, 20]);

console.log(builtObj.fctAdd()); //30

```

## 1.7. Typed Arrays are an ES6 API for handling binary data.

Exemple:

```

const typedArray = new Uint8Array([0,1,2]);
console.log(typedArray.length); // 3
typedArray[0] = 5;
const normalArray = [...typedArray]; // [5,1,2]

// The elements are stored in typedArray.buffer.
// Get a different view on the same data:
const dataView = new DataView(typedArray.buffer);
console.log(dataView.getUint8(0)); // 5

```

Instances of `ArrayBuffer` store the binary data to be processed. Two kinds of *views* are used to access the data:

- Typed Arrays (`Uint8Array`, `Int16Array`, `Float32Array`, etc.) interpret the

ArrayBuffer as an indexed sequence of elements of a single type.

- Instances of `DataView` let you access data as elements of several types (`Uint8`, `Int16`, `Float32`, etc.), at any byte offset inside an `ArrayBuffer`.

The following browser APIs support Typed Arrays :

- File API
- XMLHttpRequest
- Fetch API
- Canvas
- WebSockets
- ...



## XII - Annexe – webPack

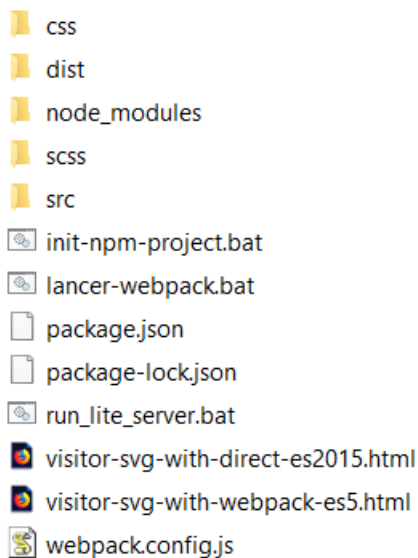
### 1. Webpack

**webpack** est une technologie javascript permettant de générer des fichiers "xyz-bundles" à partir d'un tas de petits fichiers complémentaires .

- Tout comme "rollup", webpack sait tenir compte des inter-relations entre fichiers/modules "es2015".
- webpack peut également être configuré pour déclencher des conversions "es2015 --> es5" via "**babel**".
- D'autre part , webpack peut également prendre en charge des bundles css en déclenchant si besoin un pré-processeur **saas** pour transformer ".scss" en ".css"

Au sein de cette présentation, c'est la version 4 (assez récente) de webpack qui sera utilisée .

#### 1.1. Structure possible d'un projet "npm" intégrant webpack



Le répertoire **src** comportera les fichiers ".js" à assembler .

Le répertoire **dist** comportera les "bundles" générés par webpack

#### 1.2. Configuration webpack pour des modules es2015

*en mode global :*

**npm install -g webpack webpack-cli**

*dans projet "xyz" initialisé via npm init :*

**npm install --save-dev webpack webpack-cli**

NB :

- *install -g pour lancement commande webpack*  
*et install --save-dev pour accès api webpack depuis webpack.config.js*
- *ajuster le fichier webpack.config.js avant de lancer webpack*

### Configuration de webpack pour modules "es2015" :

#### **webpack.config.js**

```
const webpack = require("webpack");
const path = require("path");

//entry: "./src/index.js" ou entry: "./src/main.js" ou ...
let config = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "./main-bundle.js"
  }
}

module.exports = config;
```

### Lancement de webpack :

#### **lancer-webpack.bat**

*REM ajuster le fichier webpack.config.js avant de lancer webpack*

**webpack** > **statut-webpack.txt**

*REM NB: webpack --config ./webpack-xyz.config.js si autre config que webpack.config.js*

ou bien

**npm run webpack-watch**

si **package.json** comporte (après ajout) :

```
... ,
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "webpack-watch": "webpack --watch"
}, ...
```

NB : l'option **--watch** permet de surveiller les fichiers sources et de relancer automatiquement webpack dès qu'un fichier a changé .

### Exemple de fichier ".html" utilisant un bundle javascript construit par webpack :

```
<html> <head> <title>visitor-svg</title>
  <!-- <link rel="stylesheet" href="dist/styles-bundle.css"> -->
  <script src="dist/main-bundle.js"></script> </head>
<body> <h3> visitor-svg (with es2015 modules packed in es5 bundle) </h3>
```

```
<canvas width="500" height="400" id="myCanvas"></canvas>
</body> </html>
```

**Exemple de modules es2015 (ici pour dessiner des figures géométriques) :**

*main.js*

```
import { Fig2D , Line, Circle, Rectangle } from './fig2d-core.js' ;
import { CanvasVisitor } from './canvas-visitor-ext.js' ;

//import "../scss/styles.scss";
//import "../css/styles-ext.css";

function my_test(){
  var tabFig = new Array();
  tabFig.push(new Line(20,20,180,200,"red"));
  tabFig.push(new Circle(100,100,50,"blue", 2,"orange"));
  tabFig.push(new Circle(250,200,50,"black",1,"blue"));
  tabFig.push(new Rectangle(200,100,50,60,"green",4));
  tabFig.push(new Rectangle(20,100,50,60,"black",1,"green"));
  var visitor = new CanvasVisitor('myCanvas');
  for( let f of tabFig){
    f.performVisit(visitor);
  }
}

window.addEventListener('load', function() {
  my_test();
});
```

*fig2d-core.js*

```
export class Fig2D {
  constructor(lineColor = "black",  lineWidth = 1,  fillColor = null){
    this.lineColor = lineColor;  this.lineWidth = lineWidth;  this.fillColor = fillColor;
  }
  performVisit(visitor){}
```

```

}

export class Line extends Fig2D{
  constructor(x1= 0 , y1= 0 , x2= 0 , y2= 0, lineColor = "black", lineWidth = 1){
    super(lineColor,lineWidth);
    this.x1=x1; this.y1=y1; this.x2=x2; this.y2=y2;
  }
  performVisit(visitor){
    visitor.doActionForLine(this);
  }
}

export class Circle extends Fig2D{
  constructor(xC = 0 , yC = 0 , r = 0, lineColor = "black", lineWidth = 1, fillColor = null){
    super(lineColor,lineWidth,fillColor);
    this.xC = xC; this.yC=yC; this.r=r;
  }
  performVisit(visitor) {
    visitor.doActionForCircle(this);
  }
}

export class Rectangle extends Fig2D{
  constructor(x1=0, y1 =0,width=0, height =0, lineColor = "black", lineWidth =1,fillColor =null){
    super(lineColor,lineWidth,fillColor);
    this.x1=x1; this.y1=y1; this.width=width; this.height=height;
  }
  performVisit(visitor) {
    visitor.doActionForRectangle(this);
  }
}

export class FigVisitor {
  doActionForCircle(c){}
  doActionForLine(l){}
  doActionForRectangle(r){}
}

```

}

*canvas-visitor-ext.js*

```

import { Fig2D , Line, Circle, Rectangle , FigVisitor} from './fig2d-core.js' ;

export class CanvasVisitor extends FigVisitor{
  constructor(canvasId){ super();
  this._canvasElement = document.getElementById(canvasId);
  this._ctx = this._canvasElement.getContext("2d");
  }

  doActionForCircle( c ) {
    this._ctx.beginPath();
    this._ctx.arc(c.xC, c.yC, c.r, 0, 2 * Math.PI, false);
    if(c.fillColor != null){
      this._ctx.fillStyle = c.fillColor;  this._ctx.fill();
    }
    this._ctx.lineWidth = c.lineWidth;
    this._ctx.strokeStyle = c.lineColor; //'#003300';
    this._ctx.stroke();
  }

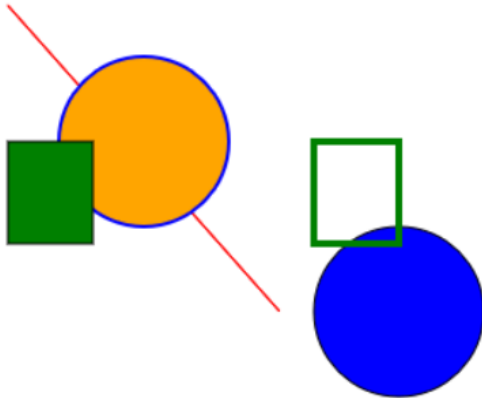
  doActionForLine( l ) {
    this._ctx.beginPath();
    this._ctx.moveTo(l.x1,l.y1);  this._ctx.lineTo(l.x2,l.y2);
    this._ctx.strokeStyle = l.lineColor;  this._ctx.lineWidth = l.lineWidth;
    this._ctx.stroke();
  }

  doActionForRectangle( r ) {
    this._ctx.beginPath();
    this._ctx.rect(r.x1,r.y1,r.width,r.height);
    if(r.fillColor != null){
      this._ctx.fillStyle = r.fillColor;  this._ctx.fill();
    }
    this._ctx.strokeStyle = r.lineColor;  this._ctx.lineWidth = r.lineWidth;
    this._ctx.stroke();
  }

```

```
}
}
```

visitor-svg (with es2015 modules packed)



### 1.3. Intégrer babel dans webpack pour générer des bundles "es5"

```
npm install --save-dev babel-loader @babel/core
npm install --save-dev @babel/preset-env @babel/register
```

**webpack.config.js**

```
const webpack = require("webpack");
const path = require("path");

let config = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "main-bundle.js"
  },
  module: {
    rules: [{
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      use: [{
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }]
    }]
  }
};

module.exports = config;
```

Grâce à cette configuration le fichier généré (ici *dist/main-bundle.js*) est plus volumineux car il ne

contient plus les mots clefs "class" , "extends" , ... compréhensible que par les navigateurs récents supportant "es6/es2015" mais une traduction d'un équivalent en ancien javascript "es5" .

## 1.4. Gestion des css et scss dans webpack

**npm install --save-dev mini-css-extract-plugin**

**npm install --save-dev style-loader css-loader**

**npm install --save-dev sass-loader node-sass**

NB.:

- les modules **mini-css-extract-plugin** , **style-loader** et **css-loader** servent à générer des **bundles css** depuis **webpack** .
- les modules **sass-loader** et **node-sass** servent à gérer les fichier **".scss"** .
- autre ajout possible : **postcss-loader** pour générer des styles css spécifiques aux navigateurs

Exemple de fichier **package.json** :

```
{
  "name": "xyz",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "webpack-watch": "webpack --watch"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/core": "^7.4.0",
    "@babel/preset-env": "^7.4.2",
    "@babel/register": "^7.4.0",
    "babel-loader": "^8.0.5",
    "css-loader": "^2.1.1",
    "mini-css-extract-plugin": "^0.5.0",
    "node-sass": "^4.11.0",
    "sass-loader": "^7.1.0",
    "style-loader": "^0.23.1",
    "webpack": "^4.29.6",
    "webpack-cli": "^3.3.0"
  }
}
```

Exemple :

dans **main.js** :

```
//...
```

```
import "../scss/styles.scss";
import "../css/styles-ext.css";
//...
```

*css/styles-ext.css*

```
h3 { color: red; }
```

*scss/styles.scss*

```
$midnight-blue : #2c3e50;
body { background-color: $midnight-blue ; }
```

*webpack.config.js*

```
const webpack = require("webpack");
const path = require("path");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");//for webpack4

let config = {
  entry: "./src/main.js",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "./main-bundle.js"
  },
  module: {
    rules: [{
      test: /\.js$/,    exclude: /(node_modules|bower_components)/,
      use: [{ loader: 'babel-loader', options: { presets: ['@babel/preset-env'] } } ]
    }, {
      test: /\.css$/,
      use: ['style-loader' , MiniCssExtractPlugin.loader, 'css-loader' ]
    }, {
      test: /\.scss$/,
      use: ['style-loader' , MiniCssExtractPlugin.loader, 'css-loader' , 'sass-loader']
    } ] },
  plugins: [    new MiniCssExtractPlugin({ filename: 'styles-bundle.css'}) ]
}

module.exports = config;
```



==> ceci permet de générer le fichier *dist/**styles-bundle.css*** (référéncé depuis une page html)

```
body { background-color: #2c3e50; }  
h3 { color: red; }
```

## XIII - Annexe – RxJs

### 1. introduction à RxJs

#### 1.1. Principes de la programmation réactive

La programmation réactive consiste essentiellement à programmer un enchaînement de traitements asynchrones pour réagir à un flux de données entrantes .

Un des intérêts de la programmation réactive réside dans la souplesse et la flexibilité des traitements fonctionnels mis en place :

- En entrée, on pourra faire librement varier la source des données ( jeux de données statiques , réponses http , input "web-socket" , événement DOM/js , ....)
- En sortie, on pourra éventuellement enregistrer plusieurs observateurs/consommateurs (si besoin de traitement en parallèles . par exemple : affichages multiples synchronisés ) .

#### 1.2. RxJs (présentation / évolution)

**RxJs** est une bibliothèque **javascript** (assimilable à un mini framework) spécialisée dans la programmation réactive .

Il existe des variantes dans d'autres langages de programmation (ex : RxJava , ... ) .

RxJs s'est largement inspiré de certains éléments de programmation fonctionnelle issus du langage "**scala**" (map , flatMap , filter , reduce , ...) et a été à son tour une source d'inspiration pour "**Reactor**" utilisable au sein de Spring 5 .

**RxJs** a été dès 2015/2016 mis en avant par le framework Angular qui a fait le choix d'utiliser "Observable" de RxJs plutôt que "Promise" dès la version 2.0 (Angular 2) .

Depuis, le framework "Angular" a continuer d'exploiter à fond la bibliothèque RxJs .  
Cependant , les 2 frameworks ont beaucoup évolué depuis 2015/2016 .

La version **4.3** de **Angular** a apporter de grandes simplifications dans les appels de WS-REST via le service **HttpClient** (rendant *obsolète* l'ancien service *Http* ) .

**La version 6 de Angular** a de son coté été **restructurée** pour intégrer les gros changements de **RxJs 6** . Heureusement, pas de bouleversement en V7 (tranquille continuité).

La version 6 de RxJs s'est restructurée en profondeur sur les points suivants :

- changement des éléments à importer (nouvelles syntaxes pour les import { } from "" )
- changements au niveau des opérateurs à enchaîner (plus de préfixe , pipe() , ... ) .

### 1.3. Principales fonctionnalités d'un "Observable"

Observable est la structure de données principale de l'api "RxJs" .

- Par certains cotés , un "Observable" ressemble beaucoup à un objet "Promise" et permet d'enregistrer élégamment une suite de traitements à effectuer de manière asynchrone .
- En plus de cela , un "Observable" peut facilement être manipulé / transformé via tout un tas d'opérateurs fonctionnels prédéfinis (ex : map , filter , sort , ...)
- En outre , comme son nom l'indique , un "Observable" correspond à une mise en oeuvre possible du design pattern "observateur" (différents observateurs synchronisés autour d'un même sujet observable).

## 2. Fonctionnement (sources et consommations)

**Source configurée et initialisée**

```
.pipe(
  callback_fonctionnelle_1 ,
  callback_fonctionnelle_2 ,
  ...
) .subscribe(callback_success , callback_error , callback_terminate)
```

NB : les paramètres callback\_error et callback\_terminate de .subscribe() sont facultatifs et peuvent donc être omis s'ils ne sont pas utiles en fonction du contexte.

NB : il faut (en règle général , sauf cas particulier/indication contraire) appeler .subscribe() pour que la chaîne de traitement puisse commencer à s'exécuter .

## 3. Réorganisation de RxJs (avant et après v5,v6)

La version 6 de RxJs a été beaucoup restructurée :

- plus de préfixe "Observable." devant of() et autres fonctions
- pipe() nécessaire pour enchaîner une série d'opérateurs (ex : map , filter , ...)
- réorganisation des éléments à importer

Quelques correspondances "avant/après" pour une éventuelle migration :

anciennes versions de RxJs (ex : v4)	versions récentes de RxJs (ex : v6)
Observable.of(data) ;	of(data) ;
import { } from "" ;	import { Observable, of } from "rxjs";  import { map , flatMap ,toArray ,filter} from 'rxjs/operators';

### 3.1. Imports dans projet Angular / typescrit

```
import { Observable, of } from 'rxjs';
import { map, flatMap, toArray, filter } from 'rxjs/operators';
```

exemple d'utilisation (dans classe de Service) :

```
public rechercherProduitSimu$(prixMaxi : number) : Observable<Produit[]> {
  let tabProduit = [
    { numero : 1, label : "produit 1", prix : 50 },
    { numero : 2, label : "produit 2", prix : 30 },
    { numero : 3, label : "produit 3", prix : 80 },
    { numero : 4, label : "produit 4", prix : 500 }
  ]
  return of(tabProduit)
    .pipe(
      flatMap(pInTab=>pInTab),
      map((p : Produit)>=>{p.label = p.label.toUpperCase(); return p;}),
      filter((p) => p.prix <= prixMaxi),
      toArray()
    );
}
```

### 3.2. Imports "umd" dans fichier js (navigateur récent)

*EssaiRxjs.html*

```
...
<body>
  Essai Rxjs (Observable, pipe, subscribe)
  <hr/>
  <p>ouvrir la console web du navigateur</p>
  <script src="lib/rxjs.umd.min.js"></script>
  <script src="js/essaiRxjs.js"></script>
</body>
...
```

avec `rxjs.umd.min.js` récupéré via <https://unpkg.com/rxjs/bundles/rxjs.umd.min.js> ou bien via une URL externe directe (CDN) `<script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>`

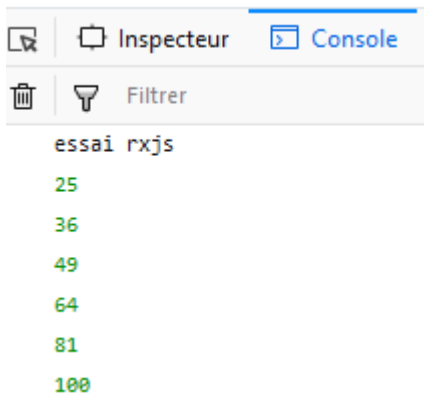
NB : "The global namespace for rxjs is rxjs"

*essaiRxJs.js*

```
console.log('essai rxjs');

const { range } = rxjs;
const { map, filter } = rxjs.operators;

range(1, 10).pipe(
  filter(x => x >= 5),
  map(x => x * x)
).subscribe(x => console.log(x));
```



```

essai rxjs
25
36
49
64
81
100

```

## 4. Sources classiques générant des "Observables"

### 4.1. Données statiques (tests temporaires , cas très simples)

```

let jsObject = { p1 : "val1" , p2 : "val2" } ;
of(jsObject)...subscribe(...) ;

let tabObj = [ { ...} , { ... } ] ;
of(tabObj)...subscribe(...) ;

of(value1 , value2 , ... , valueN)...subscribe(...) ;

```

### 4.2. Données numériques : range(startValue,endValue)

```

range(1, 10).subscribe(x => console.log(x)) ;
1
2
...
10

```

### 4.3. source périodique en tant que compteur d'occurrence

```

const obsvI1 = interval(1000 /*ms*/);
//subscriptionObjsvI1 is the result of .subscribe() call
const subscriptionObjsvI1 = obsvI1.subscribe(n =>
{ console.log(` the number of interval  occurrence (starting at 0) is ${n} `);
  if(n>=5) {
    subscriptionObjsvI1.unsubscribe(); //stop if n>=5
  }
});

```

## 4.4. source en tant qu'événement js/DOM

```
import { fromEvent } from 'rxjs';

const e1 = document.getElementById('my-element');

// Create an Observable that will publish mouse movements
const mouseMoves = fromEvent(e1, 'mousemove');

// Subscribe to start listening for mouse-move events
const subscription = mouseMoves.subscribe((evt /*: MouseEvent */) => {
  // Log coords of mouse movements
  console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);

  // When the mouse is over the upper-left of the screen,
  // unsubscribe to stop listening for mouse movements
  if (evt.clientX < 40 && evt.clientY < 40) {
    subscription.unsubscribe();
  }
});
```

## 4.5. source en tant que réponse http (sans angular HttpClient)

```
import { ajax } from 'rxjs/ajax';

// Create an Observable that will create an AJAX request
const apiData = ajax('/api/data');
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

## 4.6. source en tant que données reçues sur canal web-socket

...

## 5. Principaux opérateurs (à enchaîner via pipe)

Rappel (syntaxe générale des enchaînements) :

```
Source_configurée_et_initialisée
.pipe(
  callback_fonctionnelle_1 ,
  callback_fonctionnelle_2 ,
  ....
).subscribe(callback_success , callback_error , callback_terminate)
```

avec plein de variantes possibles

Exemple :

Principaux opérateurs :

<b>map</b>	Transformations quelconques (calculs , majuscules , tri , ....)
flatMap	
toArray	
<b>filter</b>	Filtrages (selon comparaison, ....)

### 5.1. map() : transformations

En sortie , résultat (retourné via return) d'une modification effectuée sur l'entrée .

Exemple 1:

```
const obsNums = of(1, 2, 3 ,4 ,5);
const squareValuesFunctionOnObs = map((val) => val * val);
const obsSquaredNums = squareValuesFunctionOnObs(obsNums);
obsSquaredNums.subscribe(x => console.log(x));
```

// affiche 1 4 9 16 25

Exemple 2:

```
const obsStrs = of("un" , "deux" , "trois");
obsStrs.pipe(
```

```

        map( s => s.toUpperCase() )
    )
    .subscribe(s => console.log(s));
// affiche UN  DEUX  TROIS

```

## 5.2. flatMap() et toArray()

De façon à itérer une séquence d'opérateurs sur chaque élément d'un tableau tout en évitant une imbrication complexe et peu lisible de ce type :

```

observableSurUnTableau
.pipe(
    map((tableau)=>{
        return tableau.map(
            (itemInTab)=>{itemInTab.label = itemInTab.label.toUpperCase();
            return itemInTab;}
        );
    });
);

```

on pourra placer une séquence d'opérateurs qui agiront sur chacun des éléments du tableau entre **flatMap()** et **toArray()** :

```

observableSurUnTableau
.pipe(
    flatMap(itemInTab=>itemInTab) ,
    map(( itemInTab )=>{ ... } ) ,
    filter((itemInTab) => itemInTab.prix <= 300 ) ,
    toArray()
).subscribe( (tableau) => { ... } );

```

## 5.3. filter()

```

const obsVals = of(12 , -15 , 30 , -8 , 40);
obsVals.pipe(
    filter( (v) => v >= 0 )
)
.subscribe(v => console.log(v));

```

//affiche 12 30 et 40

```

range(1,10).pipe(
    filter( (v) => v % 2 === 0 )
)
.subscribe(v => console.log(v + " est une valeur paire"));

```

## 5.4. map with sort on array

```

const obsTab = of([ {numero:1,label:'produit1',prix:40.0},
                    {numero:2,label:'produit2',prix:30.0},
                    {numero:3,label:'produit3',prix:35.0},

```



```

        {numero:4,label:'produit4',prix:15.0},
        {numero:5,label:'produit5',prix:35.0}
    ]);
obsTab.pipe(
    map( (tab) => tab.sort( (p1,p2) => (p1.prix > p2.prix) ) )
)
.subscribe(t => console.log( JSON.stringify(t) ));

```

## 6. Passerelles entre "Observable" et "Promise"

### 6.1. Source "Observable" initiée depuis une "Promise" :

```

import { from } from 'rxjs';

// Create an Observable out of a promise :
const observableResponse = from(fetch('/api/endpoint'));

observableResponse.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});

```

### 6.2. Convertir un "Observable" en "Promise"

```

observableXy.toPromise()
    .then(...)
    .catch(...)

```

## XIV - Annexe – Références , TP

### 1. Références "typescript"

#### 1.1. Sites de référence sur la technologie "typescript"

<https://github.com/Microsoft/TypeScript/wiki>

<https://www.typescriptlang.org/docs/home.html>

Autres exemples:

<http://devdocs.io/typescript/>

#### 1.2. Technologies annexes (pour typescript)

<https://nodejs.org>

<https://jquery.com/>

<https://atom.io/>

### 2. TP "typescript"

La série de TPs suivante n'est qu'une proposition .  
Il ne faut pas hésiter à tester des variantes.

#### 2.1. Installation de l'environnement de développement et préparation de l'arborescence des projets (tp)

1. **Installer** si nécessaire **NodeJs** (avec sa sous partie **npm**)
2. installer "tsc" via nodeJs : **npm install -g typescript**
3. Installer l'éditeur "*Visual Studio Code*" si Tp sur "pc windows" (ou bien éventuellement l'éditeur "*Atom*" + le plugin "*typescript*" au sein de l'éditeur *Atom* si Tp sous linux) .
4. créer quelque-part sur le poste de développement un répertoire **group-tp-typescript** (qui pourrait éventuellement correspondre à un référentiel git) et se placer dedans.

5. créer ensuite 4 sous-répertoires qui correspondront à des **projets** pour tps successifs :  
**tp-ts-node** (pour exécution coté serveur via node , sans navigateur)  
**tp-ts-web** (pour exécution coté client/web via un navigateur)  
**tp-ts-modules-node** (avec modules ts (cjs ou es2015) dans node)  
**tp-ts-modules-web** (avec modules ts/es2015 avec ou sans bundle es5 dans navigateur)
6. au sein du répertoire **tp-ts-node** créer la sous arborescence suivante :  
    **src**  
    **dist/out-tsc**
7. Au sein du répertoire **tp-ts-node**, générer un début de fichier "**tsconfig.json**" via la commande "**tsc --init**".
8. Ouvrir l'ensemble de projet/répertoire "**tp-ts-node**" via l'éditeur "*Visual Studio Code*"
9. adapter ensuite le fichier **tsconfig.json** selon un modèle (exemple) du support de cours rappelé ci-après :

```
{
"compilerOptions": {
  ...
  "outDir": "dist/out-tsc",
  "sourceMap": true,
  "target": "es5",
  "noEmitOnError" : false
},
"include": [
  "src/**/*.ts"
]
}
```

## 2.2. Traditionnel "Hello world" à écrire , transformer et exécuter

- Ouvrir l'ensemble de projet/répertoire "*tp-ts-node*" via l'éditeur "Visual Studio Code"
- créer un nouveau fichier "**hello-world.ts**" au sein du répertoire "**src**" .
- Ecrire le contenu de **hello-world.ts** avec l'éditeur "*Visual Studio Code*" ou "*Atom*"
- lancer la transpilation (".ts" → ".js") via tsc en ligne de commande, avec ou sans option -w
- lancer une ou plusieurs fois l'exécution via **node dist/out-tsc/hello-world.js** .
- Générer facultativement "*package.json*" via "*npm init*" . ajouter "*tsc:w*": "*tsc -w*" dans la partie "*scripts*" de "*package.json*" de façon à pouvoir lancer "*npm run tsc:w*"
- introduire volontairement des erreurs (ex : types incompatibles) dans le fichier **hello-world.ts** et observer le comportement de tsc (messages d'erreur, fichier ".js" généré ou pas , ....)
- corriger les erreurs dans **hello-world.ts** et relancer **node dist/out-tsc/hello-world.js** .
- ....
- Pour effectuer rapidement des autres/futurs tp , activer l'option "watch" (-w) par exemple via "*Ctrl-Shift-B* de *Visual Studio Code*" .

src/hello-world.ts

```
let message : string ;
//message = 5; //wrong type
message = "hello world";
console.log(message);
```

## 2.3. Autres tests en mode texte via tsc et node , auto-complétion

- Au sein d'un fichier **src/calculs.ts** écrire le code d'une fonction **addition(a:number , b:number):number { ...}** qui :
  - déclare une variable locale "res" de type "number" via le mot clef "let"
  - place le résultat de a+b dans la variable res
  - affiche la valeur de res via **console.log()** et une "*template string*" es2015 entre quotes inverses telle que ``pour a=${a} ,b=${b} res=(a+b)=${res}``
  - retourne la valeur du résultat "res"
- Appeler ensuite cette fonction pour vérifier l'auto-complétion apportée par l'éditeur intelligent "Visual Studio Code" ou "Atom" ou ...
- lancer l'exécution via **node dist/out-tsc/calculs.js** .
- Effectuer facultativement de façon libre tous un tas d'autres essais élémentaires (tableaux , ...) au sein du même environnement (tenir éventuellement compte des indications du formateur ou bien suivre un tutoriel en ligne) .

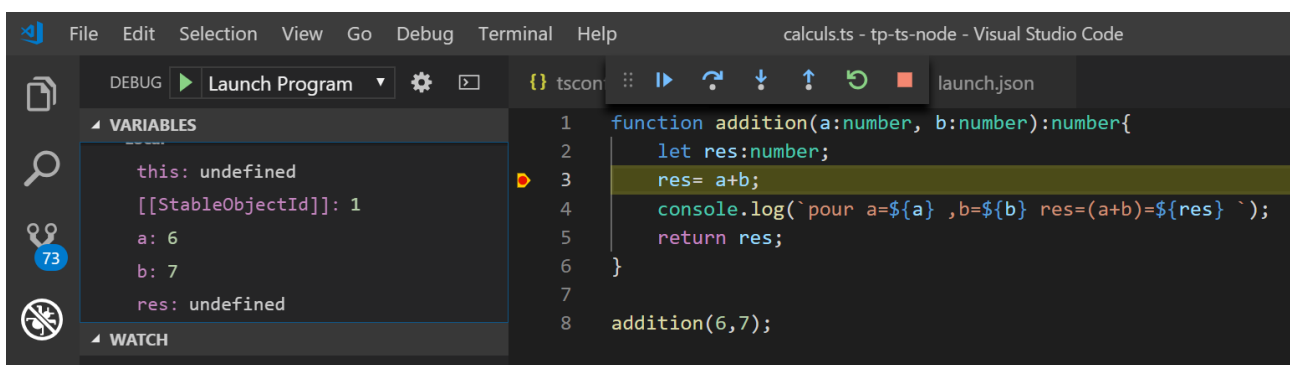
## 2.4. debug typescript / node avec sourceMap

- Activer l'option **"sourceMap": true** au sein du fichier **"tsconfig.json"** et déclencher si nécessaire **"Ctrl-Shift-B / watch"** de façon visualiser des fichiers **".map"** dans le répertoire **dist/out-tsc** .
- Au sein de l'éditeur **"Visual Studio code"** , lancer **"Debug / Start without Debugging"** et choisir **"node"** de manière à générer un début de fichier **tp-ts-node/.vscode/launch.json**
- **ajuster les chemins relatif menant aux fichiers ".js"** construits en ajustant le contenu du fichier **tp-ts-node/.vscode/launch.json**

### .vscode/launch.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceFolder}\\dist\\out-tsc\\calculs.js",
      "outFiles": [
        "${workspaceFolder}/dist/out-tsc/**/*.js"
      ]
    }
  ]
}
```

- Placer un point d'arrêt sur la ligne **res=a+b ;** de **calculs.ts** en cliquant dans la marge
- Lancer **Debug / StartDebugging** .
- Effectuer un debug "pas à pas" (*step-over*) en observant l'évolution du contenu de la variable **res** dans la fenêtre **"VARIABLES"**

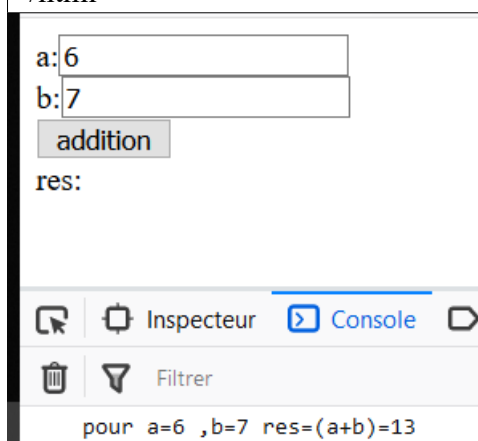


## 2.5. début du tp sur petit projet web

- au sein du répertoire **tp-ts-web** créer rapidement par copie du contenu de **tp-ts-node** la sous arborescence suivante :  
**src** (avec *calcul.ts*)  
**dist/out-tsc**
- placer dans le répertoire **tp-ts-web**, une copie du fichier "**tsconfig.json**" (de **tp-ts-node**) .
- ajouter le répertoire **dist/web** (qui contiendra les pages html référençant les .js construits dans *dist/out-tsc* )
- au sein de **dist/web** ajouter la page **calcul.html** ci-après .
- Ouvrir l'ensemble du projet **tp-ts-web** via l'éditeur "Visual Studio code" et lancer tsc via "Ctrl-Shif-B" et "watch..." . Ouvrir ensuite la page **calcul.html** avec un navigateur pour la faire fonctionner normalement (et éventuellement corriger les bugs) .

dist/web/calcul.html

```
<html>
<head>
<title>calcul</title>
<script src="../out-tsc/calculs.js"></script>
<script>
  window.addEventListener("load", function(){
    document.querySelector('#btnAdd').addEventListener('click',
      function(){
        var valA=Number(document.querySelector('#a').value);
        var valB=Number(document.querySelector('#b').value);
        var valRes=addition(valA,valB);
        document.querySelector('#res').innerHTML=valRes;
      });
  });
</script>
</head>
<body>
a:<input id="a" /> <br/>
b:<input id="b" /> <br/>
<input type="button" value="addition" id="btnAdd"/><br/>
res: <span id="res"></span> <br/>
</body>
</html>
```

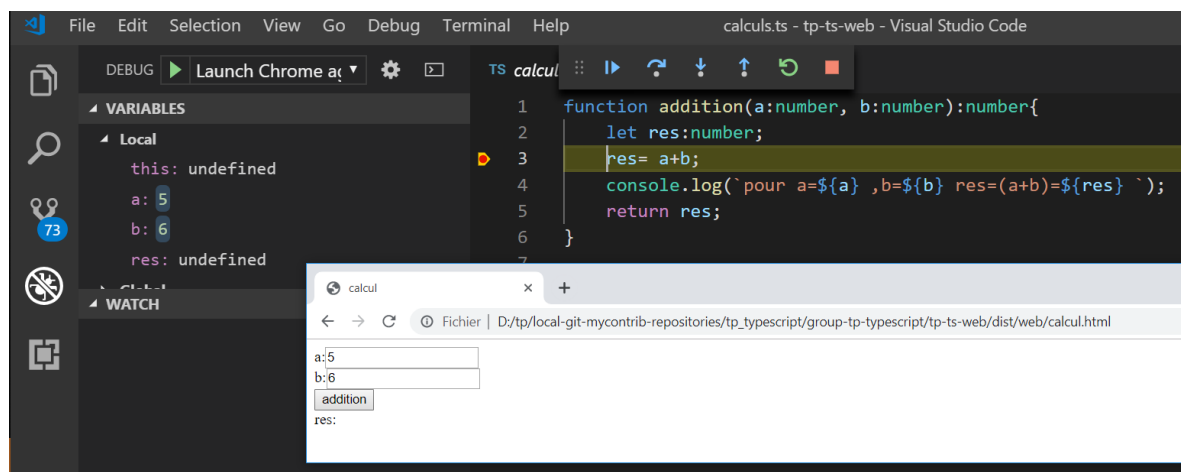


## 2.6. suite tp web avec debug navigateur (via .map)

- vérifier l'option **"sourceMap" : true** dans `tp-ts-web/tsconfig.json` et observer le contenu du fichier généré `tp-ts-web/dist/out-tsc/calcul.js.map`
- installer le plugin **"Debugger for Chrome"** au sein de **"Visual Studio Code"** via <https://marketplace.visualstudio.com/items?itemName=msjsdiag.debugger-for-chrome>
- Au sein de l'éditeur *Visual Studio code* , lancer **"Debug / Start without Debugging"** et choisir **"Chrome"** de manière à générer un début de fichier `tp-ts-web/.vscode/launch.json`
- Ajuster le fichier `tp-ts-web/.vscode/launch.json` selon les indications fournies au sein du support de cours ou bien selon cet exemple (à adapter si besoin) :

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "chrome",
      "request": "launch",
      "name": "Launch Chrome against localhost",
      "url": "http://localhost:8080",
      "url": "file:///D:/tp/local-git-mycontrib-repositories/tp_typescript/group-tp-typescript/tp-ts-web/dist/web/calcul.html",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

- placer un point d'arrêt sur la première ligne de la fonction **addition** de **calcul.ts** et effectuer un **"debug" pas à pas** depuis l'IDE **"Visual Studio Code"** en lançant **Debug / Start Debugging** .



## 2.7. Tp sur bases de la programmation orientée objet

Au sein du projet "*tp-ts-node*" (à ré-ouvrir depuis *Visual Studio Code*) coder un nouveau fichier **Personne.ts** qui comportera le code d'une classe "**Personne**" ainsi qu'un petit test :

Classe "**Personne**" avec numero , nom et age.

L'age ( `_age` privé ) doit toujours être positifs (via `get/set` ).

```
class Personne {
  ...
}

var p1 : Personne = new Personne(1,"toto");
p1.age = 30; p1.age = -50;
console.log("age de p1 :" + p1.age);
console.log("nom de p1 :" + p1.nom);
console.log("numero de p1 :" + p1.numero);
```

Tester via **node dist/out-tsc/Personne.js**

==>

*age negatif interdit*

*age de p1 :30*

*nom de p1 :toto*

*numero de p1 :1*

Au sein du fichier **Personne.ts** , ajouter une classe **Employe** qui hérite de la classe **Personne** et qui possède un salaire en plus :

```
....

class Employe .....{
  ...
}

var e1 = new Employe(1,"toto",2500);
e1.salaire=3000;
console.log("Empoye e1 :" + JSON.stringify(e1));
//Empoye e1 :{"numero":1,"nom":"toto","_age":0,"salaire":3000}
```



Au sein du projet "*tp-ts-node*" coder un nouveau fichier *Serie.ts* qui comportera le code d'une classe "Serie" ainsi qu'un petit test :

Classe "Serie" avec label et values.

```
class Serie {  
  ...  
}  
  
let serie1 = new Serie("serie1",[2,6,8]);  
serie1.push(4);serie1.push(5);//[2,6,8 , 4, 5 ]  
console.log(JSON.stringify(serie1));
```

Tester via *node dist/out-tsc/Serie.js*

==>

```
{"label":"serie1","values":[2,6,8,4,5]}
```

---

Lorsque le *chapitre/paragraphe* "Generic" (avec <T>) aura été abordé en cours ,

On pourra améliorer le fichier *Serie.ts* avec une version générique de *Serie<T>*

pouvant s'utiliser de cette manière :

```
let serie1 = new Serie("serie1",[2,6,8]);  
serie1.push(4); serie1.push(5);  
console.log(JSON.stringify(serie1));  
  
let serie2 : Serie<string>;  
serie2 = new Serie("serie2",['abc', 'def']);  
serie2.push('xyz');  
console.log(JSON.stringify(serie2));
```

==>

```
{"label":"serie1","values":[2,6,8,4,5]}
```

```
{"label":"serie2","values":["abc","def","xyz"]}
```

---

Au sein du projet "*tp-ts-node*" coder un nouveau fichier *Stats.ts* qui comportera le code d'une classe "StatComputer" qui implémentera l'interface StatBuilder:

*Stats.ts* ( \*\*\*\*\* TP FACULTATIF \*\*\*\*\* )

```
interface Stat{  
  size:number;  
  average:number;  
  sum:number;  
  ecartType? :number;  
}
```

```
interface StatBuilder {
  buildStat(withEcartType:boolean):Stat;
}
```

```
interface WithStats extends StatBuilder {
  size() : number;
  average():number;
  sum():number;
  ecartType():number;
}
```

```
class StatComputer .... WithStats {

  constructor(.....values : Array<number>=.....){
  }
  ...
}
```

```
let myValues=[8, 2 ,6, 4 , 10];
let myStatComputer=new StatComputer(myValues);
console.log("size="+myStatComputer.size());
console.log("sum="+myStatComputer.sum());
console.log("average="+myStatComputer.average());
console.log("ecartType="+myStatComputer.ecartType());
console.log("myStatComputer.buildStat()="+JSON.stringify(myStatComputer.buildStat(true)));
```

```
==>
```

```
size=5
```

```
sum=30
```

```
average=6
```

```
ecartType=2.8284271247461903
```

```
myStatComputer.buildStat()={"size":5,"sum":30,"average":6,"ecartType":2.8284271247461903}
```

## 2.8. Tp "modules Typescripts" en environnement nodeJs

Soit *tp-ts-modules-node* une copie adaptée du projet *tp-ts-node* pour tester l'aspect modulaire de typescript selon la norme "es2015" .

Objectif du Tp :

**node** *dist/out-tsc/main.js* doit permettre de lancer la version transpilée de *main.ts*

**main.ts**

```
//importer Serie depuis Serie(.ts)
//importer Stat et buildStatFromValues depuis Stats(.ts)

function myMainFunction(){
    let serie1 = new Serie("serie1",[2,6,8]);
    serie1.push(4);serie1.push(10);
    console.log(JSON.stringify(serie1));

    let statForSerie1 : Stat;
    statForSerie1 = buildStatFromValues(serie1.values);
    console.log("moyenne =" +statForSerie1.average)
}

myMainFunction();
```

- adapter *Serie.ts* pour que la classe *Serie* y soit **exportée**
- adapter *Stats.ts* pour que l'interface *Stat* y soit **exportée**  
Stats.ts n'exportera pas la classe StatComputer.  
mais exportera la fonction suivante :

```
...
... function buildStatFromValues(values: Array<number>) : Stat{
    let internatStatComputer=new StatComputer(values);
    return internatStatComputer.buildStat();
}
```

- Coder les "import" dans main.ts
- Transpiler les ".ts" en ".js" et tester main.js via node

## 2.9. Tp "modules Typescripts" en environnement web (navigateur)

Objectif du Tp :

passer de

```
<script src="../out-tsc/Serie.js"></script>
<script src="../out-tsc/MyGraph.js"></script>
<script src="../out-tsc/htmlGraph.js"></script>
```

à

```
<script src="../build-es5/app-bundle.js"></script>
```

au sein de `dist/web/myGraph.html`

via des import/export et l'utilisation de **rollup** pour générer un bundle .

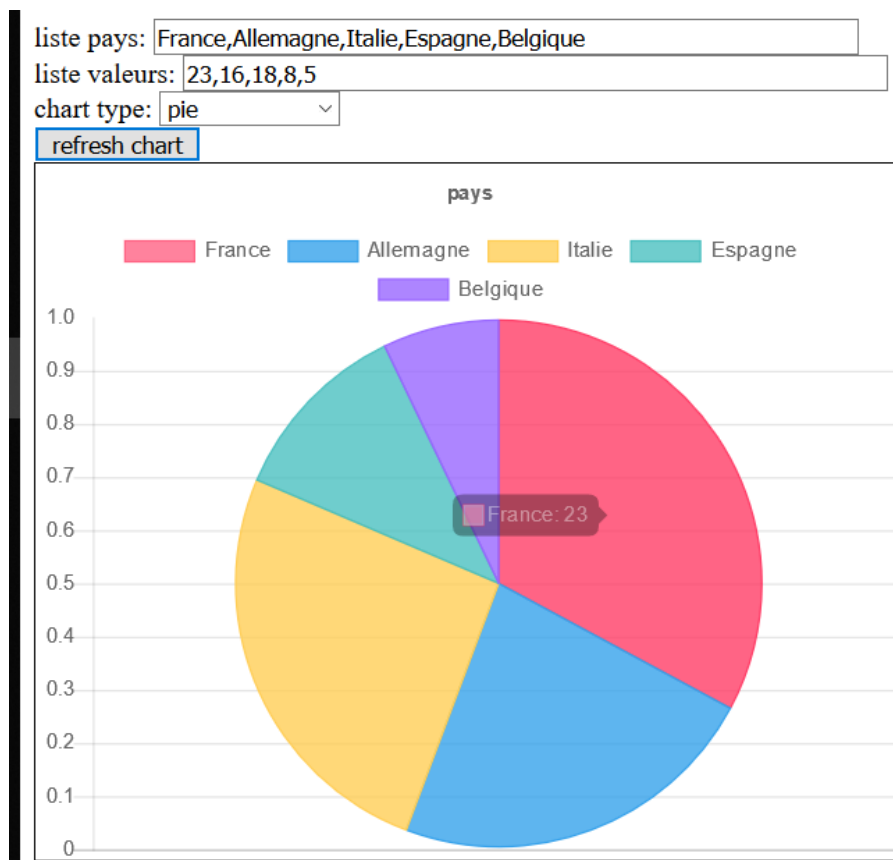
Installation de rollup :

```
npm install -g rollup
```

Code initial au début du tp :

Récupérer le code du début de Tp **fourni par le formateur** pour le projet ***tp-ts-modules-web***

Ouvrir le fichier `dist/web/myGraph.html` avec un navigateur et faire fonctionner l'exemple :



- Effectuer les modifications suivantes au sein de **tsconfig.json**

```
"target": "es5",  
"module": "commonjs",  
"target": "es2015",  
"module": "es2015",
```

- Repérer la ligne de script

```
"build" : "tsc && rollup -c rollup.config.js && tsc --out ./dist/build-es5/app-bundle.js  
--target es5 --allowJs dist/build-es2015/app-bundle.js"
```

au sein de **package.json** permettant de lancer **npm run build** .

- Adapter le fichier **Serie.ts** en ajoutant un **export**
- Adapter le fichier **MyGraph.ts** en exportant la classe **MySimpleGraph**
- Adapter le fichier **htmlGraph.ts** en important **Serie** et **MySimpleGraph**
- Modifier **dist/web/myGraph.html** selon l'objectif de l'énoncé de ce Tp (page précédente)
- Lancer **npm run build** (éventuellement via **build\_es5\_bundle.bat** )
- Tester le bon fonctionnement de **dist/web/myGraph.html**

## 2.10. Tp "mixin"

- Se replacer sur le projet **group-tp-typescript\tp-ts-modules-node**
- Ajouter **"lib" : [ "es2015" , "dom" ]** au sein de **tsconfig.json**
- Au sein d'un nouveau fichier **my-mixin.ts** ajouter un **mixin "WithSimpleStat"** basé sur l'ajout des fonctionnalités suivantes :

```
getValues() : Array<number>{  
    //essai d'accès à la propriété ".values" de l'objet courant :  
    return Reflect.get(this,"values"); //nécessite lib "es2015" dans tsconfig.json  
}  
  
size(): number {  
    return this.getValues().length;  
}  
  
average(): number {  
    return this.sum()/this.size();  
}  
  
sum(): number {  
    let s=0;  
    for(let i=0;i<this.getValues().length;i++){  
        s+=this.getValues()[i];  
    }  
}
```

```

    return s;
}

```

- ajouter un fichier **mixinSerieStat.ts** en complétant la ligne manquante :

```

import { Serie } from "../Serie";
import { WithSimpleStat } from "../my-mixins";

//nouvelle classe SerieWithStat basée sur classe Série et enrichie via Mixin WithSimpleStat :
.....

//test de SerieWithStat :
let serie3 = new SerieWithStat("serie3",[2,6,8]);
serie3.push(4);serie3.push(10);
console.log("size of serie3="+serie3.size());
console.log("sum of serie3="+serie3.sum());
console.log("average of serie3="+serie3.average());

```

construire et lancer **node** dist/out-tsc/mixinSerieStat.js

==>

size of serie3=5  
 sum of serie3=30  
 average of serie3=6s

## 2.11. Tp libre autour d'un "module de définition"

- Se replacer sur le projet group-tp-typescript\tp-ts-web
- créer les répertoires tp-ts-web/lib/js et tp-ts-web/lib/d.ts
- Ajouter **quelques fonctions "javascript"** dans un fichier lib\js\my-js-lib.js
- Ajouter les définitions "typescript" correspondantes dans un fichier lib\d.ts\my-js-lib.d.ts
- Ajouter un fichier tp-ts-web/src/test-my-js-lib.ts qui :
  - sera écrit en ".ts"
  - comportera `/// <reference path = "../lib/d.ts/my-js-lib.d.ts" />`
  - invoquera quelques fonctions codées dans lib\js\my-js-lib.js
- Bien visualiser l'**auto-complétion** normalement apportée par l'IDE "Visual Studio Code"
- Ecrire dans tp-ts-web/dist/web une page **test-def.html** regroupant les fichiers ../../lib/js/my-js-lib.js et ../out-tsc/test-my-js-lib.js et déclenchant l'appel d'une fonction du fichier test-my-js-lib.js d'une manière ou d'une autre.

Idee de Tp :

dans **my-js-lib.js** :

```

function Circle(cx,cy,r){
    this.cx=cx;
    this.cy=cy;
    this.r=r;
}

```

```

    this.perimetre = function(){
        return Math.PI * 2 * this.r;
    }
    this.translate=function(dx,dy){
        this.cx = this.cx+ dx;
        this.cy = this.cy + dy;
    }
}

function calculerSurfaceCercle(c){
    return Math.PI * c.r * c.r;
}

```

Dans **test-my-js-lib.ts**

```

function log_essai_Circle() {
    var c1,c2 : Circle;
    c1 = new Circle();
    c1.cx=12.5;
    c1.cy=45;
    c1.r=20;
    console.log("c1.perimetre()" + c1.perimetre());
    c2=new Circle(25,30,10);
    console.log("c2.perimetre()" + c2.perimetre());
    c2.translate(10,10);
    console.log("c2=" + JSON.stringify(c2));
    let s : number;
    s=calculerSurfaceCercle(c2);
    console.log("surface de c2=" + s);
}

```

=> **A faire en tp : Déterminer le contenu de *my-js-lib.d.ts***

=> résultats attendus à l'exécution :

*c1.perimetre()*=125.66370614359172

*c2.perimetre()*=62.83185307179586

*c2*={"cx":35,"cy":40,"r":10}

*surface de c2*=314.1592653589793