

Langage typescript

Table des matières

I - Typescript (utilisation, bases ,)	3
1. Typescript : présentation , utilisation	3
1.1. Rappel: Version de javascript (ES5 ou ES6/es2015)	3
1.2. Présentation de TypeScript / ts	3
1.3. Environnement de développement (typescript)	5
1.4. utilisation indirecte de tsc via npm	6
1.5. utilisation directe de tsc (paramétré par tsconfig.json)	7
1.6. Source map (.map)	7
1.7. Quelques utilisations du code traduit en javascript	8
1.8. Configuration de Visual Studio code pour un debug pas à pas de code ".ts" traduit en ".js" et s'exécutant dans un navigateur	8
2. Bases syntaxiques du langage typescript (ts)	11
2.1. précision des types de données	11
2.2. Tableaux (construction et parcours)	12
2.3. Portées (var , let) et constantes (const)	13
II - Typescript objet	14
1. Programmation objet avec typescript (ts)	14
1.1. Classe et instances	14
1.2. constructor	15

1.3. Propriété "private".....	16
1.4. Accesseurs automatiques get xxx() /set xxx().....	16
1.5. Mixage "structure & constructor" avec public ou private.....	17
1.6. mot clef "static".....	18
1.7. héritage et valeurs par défaut pour arguments:.....	18
1.8. Classes abstraites (avec opérations abstraites).....	19
1.9. Interfaces.....	19

III - Lambda , Generics , ... / typescript.....24

1. Programmation fonctionnelle (lambda, ...)	24
2. Generics de typescript (ts)	26
2.1. Fonctions génériques :	26
2.2. Classes génériques :	26

IV - Modules et aspects avancés (ts).....27

1. Modules typescript (ts)	27
1.1. Modules internes et externes.....	27
1.2. Modules internes (sémantique de "namespace").....	27
1.3. Modules externes et organisation en fichiers.....	28
1.4. Types de modules (cjs , amd , es2015 , umd , ...)	29
1.5. Technologies de "packaging" (webpack , rollup , ...) et autres.....	30
1.6. Import de modules au format es6/es2015 (code source .ts).....	30
1.7. Packaging web via rollup / es2015 et npm.....	31
2. Aspects divers et avancés de typescript (ts)	33
2.1. Surcharge très limitée.....	33
2.2. Ambient ts.....	34
2.3. Options du compilateur "tsc" et tsconfig.json.....	34
2.4. Mixins.....	35

V - Annexe – Références , TP.....37

1. Références "typescript"	37
1.1. Sites de référence sur la technologie "typescript".....	37
1.2. Technologies annexes (pour typescript).....	37
2. TP "typescript"	37
2.1. Installation de l'environnement de développement.....	37
2.2. Traditionnel "Hello world" à écrire , transformer et exécuter.....	38
2.3. Autres tests en mode texte via atom , tsc et node.....	38
2.4. Tp sur bases de la programmation orientée objet.....	38
2.5. Tp "dessin vectoriel (html5/canvas) en typescript objet".....	38
2.6. Tp sur modules (export / import).....	43
2.7. Autres Tp.....	44

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) .

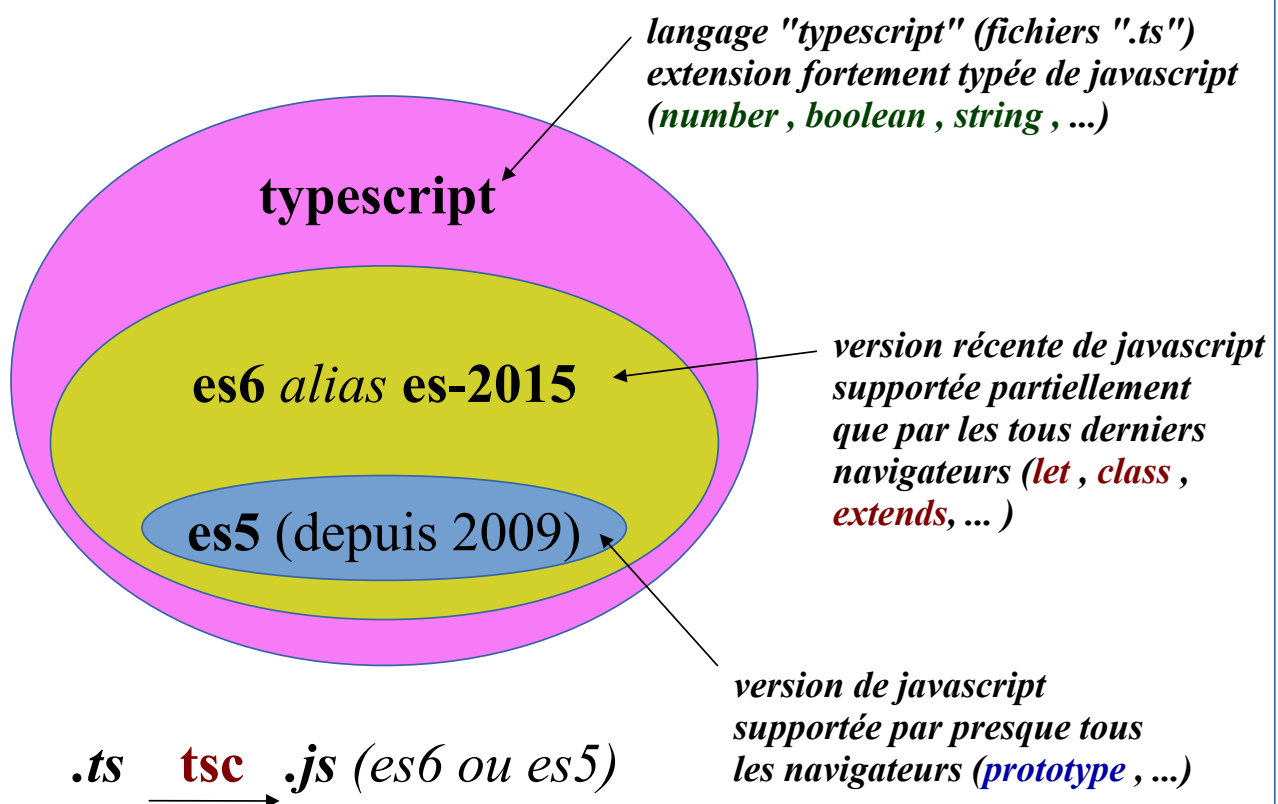
Les versions modernes sont :

- **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** .

En 2016, 2017, 2018, ... , 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 .

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 environnement et notamment par **Angular2+ 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. 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) .

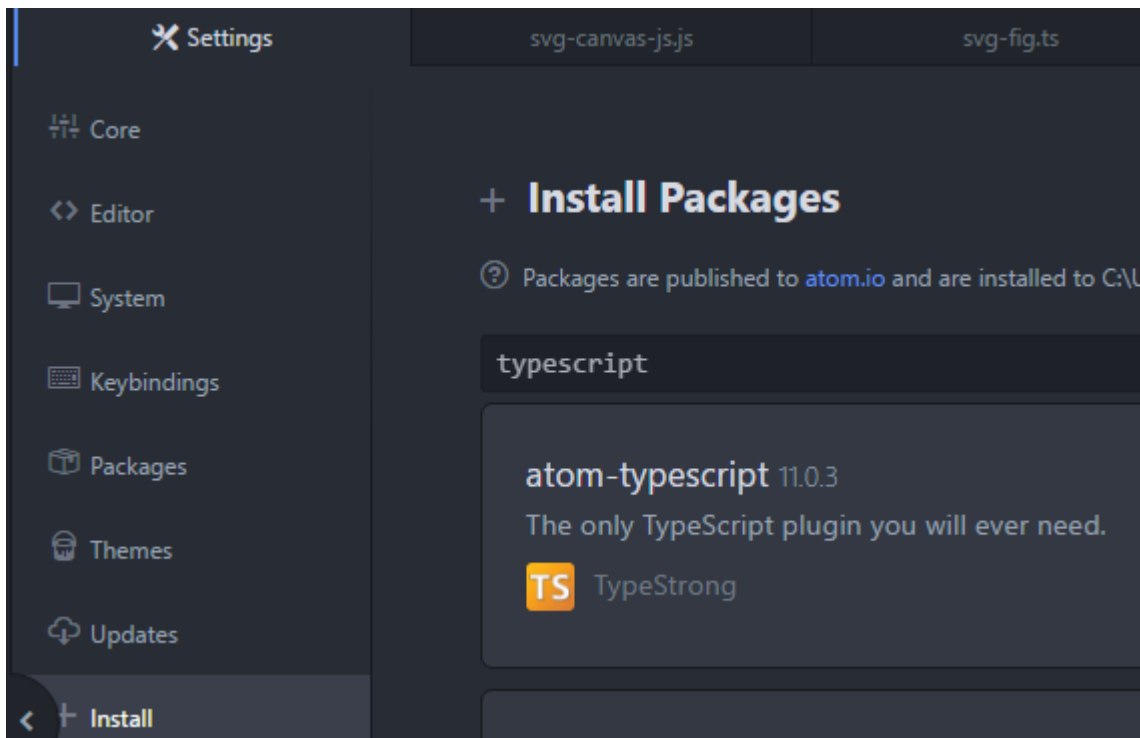
D'autres environnement comportent des "plugins" pour travailler sur des fichiers ".ts" (en langage typescript) :

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

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 l'une des versions les plus récentes de **typescript** (mi 2017) .

1.4. 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) :

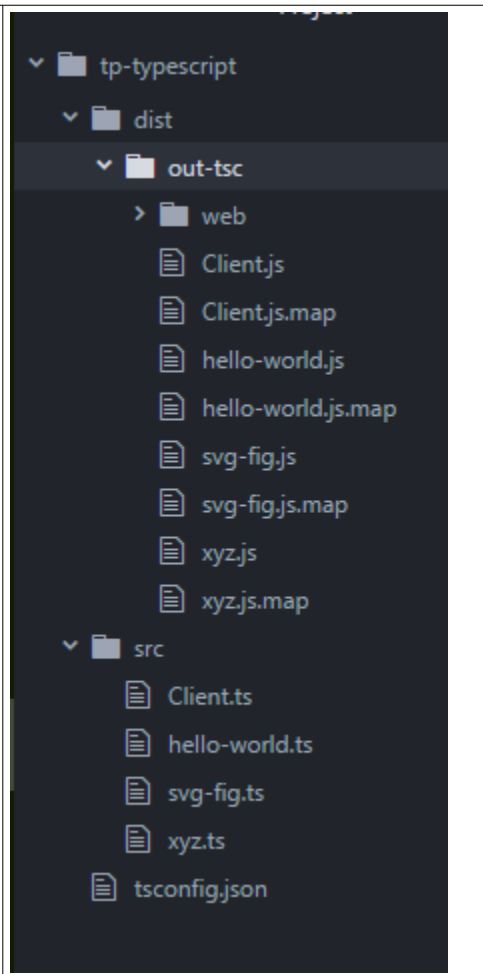
compile ts to js.bat

```
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 :
- ...

1.5. 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>tsconfig.json</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 : Visual Studio + plugin, 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.

1.6. Source map (.map)

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

1.7. 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

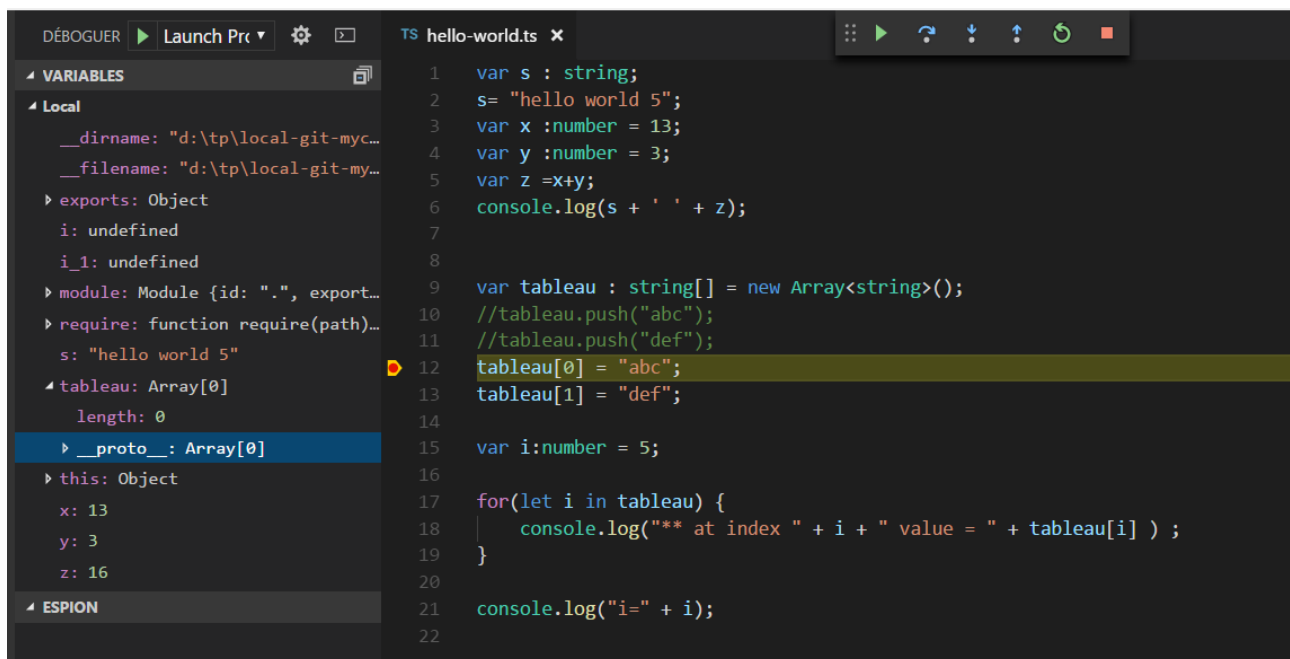
→ Une application "sérieuse" aura un cadre/contexte au cas par cas (ex : Angular 2+ et projet basé sur @angular/cli) .

→ 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" .


1.8. Configuration de Visual Studio code pour un debug pas à pas de code ".ts" traduit en ".js" et 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 .

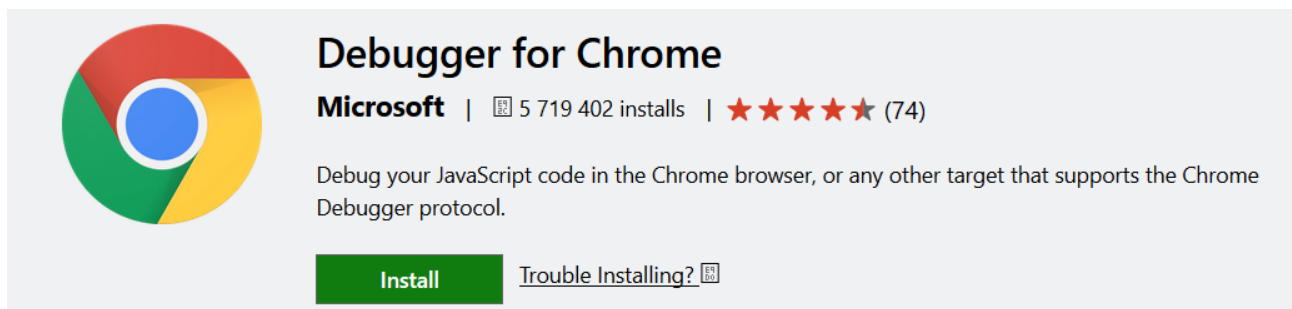
Pour bien paramétrer les chemins menant au code de l'application à exécuter et déboguer on peut ajuster le fichier suivant (quelquefois créé lors du premier lancement via l'icône ).

.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 Visual Studio Code) , 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

boolean	<code>var isDone: boolean = false;</code>
number	<code>var height: number = 6;</code> <code>var size : number = 1.83 ;</code>
string	<code>var name: string = "bob";</code> <code>name = 'smith';</code>
array	<code>var list1 : number[] = [1, 2, 3];</code> <code>var list2 : Array<number> = [1, 2, 3];</code>
enum	<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>
any	<code>var notSure: any = 4;</code> <code>notSure = "maybe a string instead";</code> <code>notSure = false;</code>
void	<code>function warnUser(): void {</code> <code> alert("This is my warning message");</code> <code>}</code>
object	(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. Tableaux (construction et parcours)

```
var tableau : string[] = new Array<string>();
```

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

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

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

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

Au moins 3 parcours possibles:

```
var n : number = 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( let s of tableau){
    console.log("## val = " + s) ;
}
```

2.3. Portées (var , let) et constantes (const)

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) . Ceci est maintenant interdit en "typescript".

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 : string[] = new Array<string>();
tableau[0] = "abc";
tableau[1] = "def";

var i : number = 5;
var j : number = 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
```

II - Typescript objet

1. Programmation objet avec typescript (ts)

1.1. Classe et instances

```
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" .

1.2. 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.

1.3. 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) !!!

1.4. 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

```

1.5. Mixage "structure & constructeur" 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) , certains 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) .

1.6. 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());
```

1.7. 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 aux attributs/propriétés "**protected**".

1.8. 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

```

1.9. 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("cancer", "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. Programmation fonctionnelle (lambda, ...)

Rappels (2 syntaxes "javascript" ordinaires) valables en "typescript" :

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

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

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" :

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 et aspects avancés (ts)

1. Modules typescript (ts)

1.1. Modules internes et externes

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

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

1.2. Modules internes (sémantique de "namespace")

Module (avec utilisation locale dans même fichier):

```

module 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 valideur :
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

```

--> limité à un seul fichier , les modules internes sont finalement assez peu intéressants et très peu utilisés.

1.3. Modules externes et organisation en fichiers

Exemple :

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);
  }
}

```

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

```
npm run tsc:m validateurs.ts test-validateurs.ts
```

test-validateurs.ts

```
import validateurs = require('./validateurs');

// Some samples to try
var strings = ['Hello', '98052', '101'];
// Validators to use
var validators: { [s: string]: validateurs.StringValidator; } = {};
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);
  }
});
```

1.4. 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 "synchrone", **syntaxe** "var xyz = **requires**('xyz')"
NB : node (nodeJs) utilise partiellement les idées et syntaxes de CommonJS .
- **AMD** (Asynchronous **M**odule **D**efinition) avec chargements asynchrones
- **ES2015 Modules** : syntaxiquement standardisé , mots clef "import {...} from '...'" et export pour la gestion statique des modules et System.import("....") possible pour liaisons dynamiques.
- **SystemJS** (très récent et pas encore complètement stabilisé) supporte en théorie les 3 technologies de modules précédentes (cjs , amd, es2015) et c'est pour cette raison que le tutorial "Angular 2 / Tour of Heroes" s'appuyait dessus dans la version "bêta" pour charger le code en mémoire au niveau du navigateur et de la page d'accueil index.html.
SystemJS est à priori capable de gérer une compilation/transpilation "typescript → javascript" au dernier moment (juste à temps) mais l'opération est lente et donc rarement retenue. --> 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 **m**odule **d**efinition) – *fichiers xyz.umd.js*
- **iife** (immediately-invoked **f**unction **e**xpression) – *fonctions anonymes auto-exécutées*

1.5. 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 très récente par encore 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 bundlexyz.min.js.gz sont automatiquement traités par quasiment tous les serveurs HTTP et les navigateurs : décompression automatique après transfert réseau).

1.6. Import de modules au format es6/es2015 (code source .ts)

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

et placer "**module**": "**commonjs**" ou bien "**module**": "**es6**" dans tsconfig.json
selon la plate-forme cible.

Par exemple "nodeJs" supporte des modules "commonjs" mais pas encore des modules "es6/es2015".

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 :

```
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 meilleure 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()`**;

2. Aspects divers et avancés de typescript (ts)

2.1. 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");

```

2.2. Ambient ts

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

Ceci permet d'appeler des fonctions "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/...

2.3. 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>

2.4. Mixins

<https://www.typescriptlang.org/docs/handbook/mixins.html>

ANNEXES

V - 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

1. Installer NodeJs et npm
2. installer "tsc" via nodeJs : **npm install -g typescript**
3. Installer l'éditeur Atom
4. Installer le plugin "typescript" au sein de l'éditeur Atom
5. créer un répertoire tp-typescript et y placer le fichier de paramétrage "tsconfig.json" selon le modèle (exemple) du support de cours
6. lancer tsc
7.

2.2. Traditionnel "Hello world" à écrire , transformer et exécuter

- Ecrire le contenu de hello-world.ts avec l'éditeur "Atom"
- lancer si nécessaire la transpilation (".ts" → ".js") via tsc
- lancer l'exécution via node

2.3. Autres tests en mode texte via atom , tsc et node

Effectuer tous un tas d'autres essais élémentaires (tableaux , ...) au sein du même environnement.

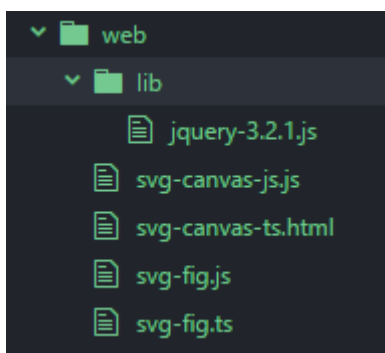
Tenir compte des indications du formateur
ou bien suivre un tutorial en ligne

2.4. Tp sur bases de la programmation orientée objet

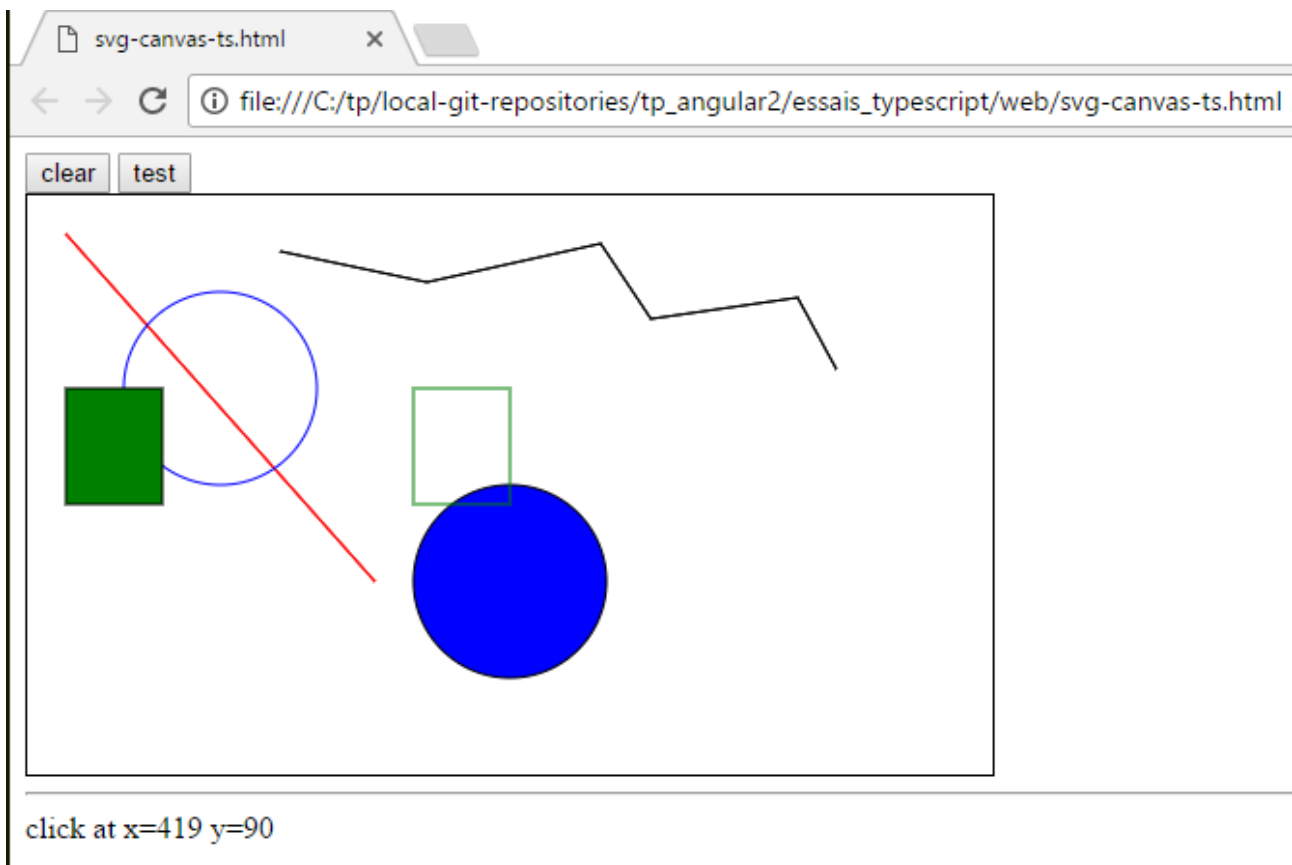
Classe "**Client**" avec numero , nom et age.
L'age (_age privé) doit toujours être positifs (via get/set).

....

2.5. Tp "dessin vectoriel (html5/canvas) en typescript objet"



(structure à un peu adapter si src et dist/tsc-out séparés)



dist/tsc-out/web/svg-canvas-ts.html

```
<html><head>
  <script src="lib/jquery-3.2.1.js"></script>
  <script src="svg-canvas-js.js"></script>
  <!-- <script src="svg-fig.js"></script> -->
  <script src="../svg-fig.js"></script> <!--svg-fig.ts transpile svg-fig.js -->
  <style type="text/css"> </style>
<body>
  <div id="divDessin">
    <input type='button' value="clear" id="btnClear" />
    <input type='button' value="test" id="btnTest" /> <br/>
    <canvas id="myCanvas" width="500" height="300"
      style="border:1px solid #000000;">
    </canvas>

    <hr/>
    <span id="spanMsg"></span> <br/>
  </div>
</body></html>
```

dist/tsc-out/web/svg-canvas-js.js

```
var lastXc = null;
var lastYc = null;
```

```
$(function() {

    $('#btnClear').on('click',function(){
        console.log("clear");
        clear_canvas();
    });

    $('#btnTest').on('click',function(){
        console.log("test");
        my_ts_test();
    });

    $('#myCanvas').on('click',function(evt){
        log_coords(evt);
    });
});
```

```
function clear_canvas(){
    var canvasElement = document.getElementById("myCanvas");
    var ctx = canvasElement.getContext("2d");
    ctx.clearRect ( 0 , 0 , canvasElement.width, canvasElement.height );
    lastXc=null; lastYc=null;//reset last coord for next line
}
```

```
function log_coords(evt){
    var canvasElement = document.getElementById("myCanvas");
    var xC = evt.pageX - canvasElement.offsetLeft; //xC = x relative to canvas
    var yC = evt.pageY - canvasElement.offsetTop; //yC = relative to canvas
    var msg="click at x=" + xC + " y=" + yC;
    console.log(msg);
    $("#spanMsg").html(msg);

    var ctx = canvasElement.getContext("2d");
    if(lastXc == null && lastYc == null){
        lastXc=xC; lastYc=yC;
    }
    ctx.beginPath();
    ctx.moveTo(lastXc,lastYc);//from last x,y
    ctx.lineTo(xC,yC);//to new xC,yC
    ctx.closePath();

    lastXc=xC; lastYc=yC;//store last coord for next line
    ctx.stroke();
};
```


src/svg-fig.ts → dist/tsc-out/svg-fig.js

```
//...
function my_ts_test(){
//code typescript (qui sera transformé en javascript) et qui sera appelé/déclenché via jQuery
// en appuyant sur le bouton "test"
}
```

But du TP:

Mettre en place une hiérarchie de classes pour gérer les coordonnées des figures géométriques à dessiner:

Fig2D (lineColor , lineWidth , fillColor) ,

Line (x1,y1,x2,y2) héritant de Fig2D ,

Circle (xc,yc,r) héritant de Fig2D ,

Rectangle (x1,y1,width,height) héritant de Fig2D

NB : les instructions de dessin seront déclenchées en s'appuyant sur le design pattern "viseur" :

```
interface FigVisitor {
  doActionForCircle( c : Circle) : void;
  doActionForLine( l : Line) : void;
  doActionForRectangle(r : Rectangle) : void;
}
```

```
class SvgInCanvasVisitor implements FigVisitor{
  private _canvasElement : any = null;
  private _ctx : any = null; //2d (svg) context in html5 canvas
  constructor(public canvasId : string){
    this._canvasElement = document.getElementById(canvasId);
    this._ctx = this._canvasElement.getContext("2d");
  }
  doActionForCircle( c : Circle) : void {
    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.closePath();
    this._ctx.stroke();
}
doActionForLine( l : Line) : void {
    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.closePath();
    this._ctx.stroke();
}

doActionForRectangle( r : Rectangle) : void {
    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.closePath();  this._ctx.stroke();
}
}

```

```

function my_ts_test(){ //.....
    tabFig.push(new Line(20,20,180,200,"red"));
    tabFig.push(new Circle(100,100,50,"blue"));
    tabFig.push(new Circle(250,200,50,"black",1,"blue"));
    tabFig.push(new Rectangle(200,100,50,60,"green"));
    tabFig.push(new Rectangle(20,100,50,60,"black",1,"green"));
    var svgVisitor = new SvgInCanvasVisitor("myCanvas");
}

```

```
//dans boucle sur f en tant qu'élément de tabFig
    //f.performVisit(svgVisitor);
}
```

2.6. Tp sur modules (export / import)

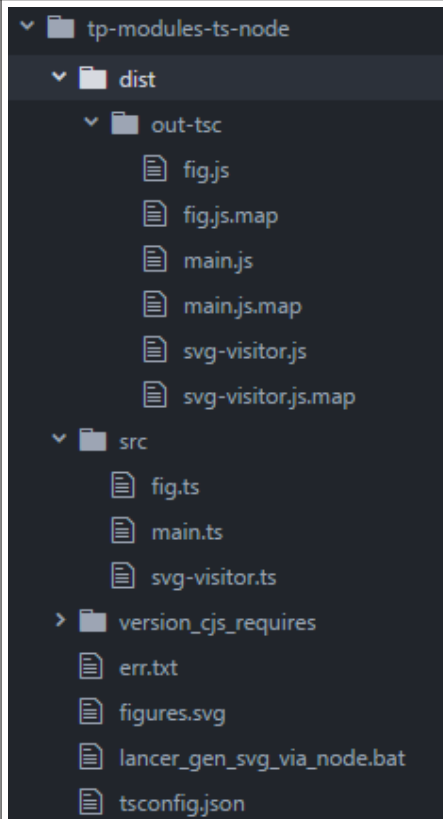
Dans un nouveau projet "**tp-modules-ts-node**", reprendre et restructurer le code de l'exemple précédent en plusieurs modules complémentaires :

fig.ts (module avec export de Fig2D , Line , Circle , Rectangle , SvgVisitor)

svg-visitor.ts (module avec import nécessaires + export de SvgGenVisitor)

main.ts (import nécessaires + fonction my_svg_gen_in_console() et appel)

NB : choisir "**module**": "**commonjs**" dans tsconfig.json et utiliser la syntaxe **import { Xxxx } from './xxxx'** ;



main.ts

```
//....

function my_svg_gen_in_console(){
  var tabFig : Fig2D[] = new Array<Fig2D>();
  tabFig.push(new Line(20,20,180,200,"red"));
  tabFig.push(new Circle(100,100,50,"blue"));
  tabFig.push(new Circle(250,200,50,"black",1,"blue"));
  tabFig.push(new Rectangle(200,100,50,60,"green"));
  tabFig.push(new Rectangle(20,100,50,60,"black",1,"green"));
  var svgVisitor = new SvgGenVisitor();
  //for(let index in tabFig) { ... }
  for( let f of tabFig){
    f.performVisit(svgVisitor);
  }
  // avec redirection externe de type
  // node dist/out-tsc/main.js 2>err.txt > figures.svg
  // pour générer un fichier ".svg" pret à être afficher
  // par un navigateur
```

```

    console.log(svgVisitor.getAllSvgFileContent());
}

my_svg_gen_in_console();

```

Visiteur en version "génération de fichier svg" :

```

class SvgGenVisitor implements FigVisitor{
    private _svgHeader : string;
    private _svgContent : string;
    constructor(){
        this._svgHeader =
        '<svg width="500" height="400" '
        +' xmlns="http://www.w3.org/2000/svg">';
        this._svgContent = "";
    }

    public getAllSvgFileContent(): string {
        return this._svgHeader + this._svgContent
            + "</svg>";
    }

    doActionForCircle( c : Circle) : void {
        this._svgContent += ' <circle cx="'+c.xC+'" cy="'+c.yC
            +'" r="'+c.r+'" stroke="'+c.lineColor
            +'" stroke-width="'+c.lineWidth
            +'" fill="'+c.fillColor+'" />';
    }

    doActionForLine( l : Line) : void {
        this._svgContent += ' <line x1="'+l.x1
            +'" y1="'+l.y1
            +'" x2="'+l.x2
            +'" y2="'+l.y2+
            '" style="stroke:'+l.lineColor
            +';stroke-width:'+l.lineWidth+'" />';
    }

    doActionForRectangle( r : Rectangle) : void {
        this._svgContent += ' <rect x="'+r.x1+'" y="'+r.y1+
            '" width="'+r.width+
            '" height="'+r.height+
            '" style="fill:'+r.fillColor+
            ';stroke-width:'+r.lineWidth+
            ';stroke:'+r.lineColor+'" />';
    }
}

```

2.7. Autres Tp

....

