



TYPESCRIPT

SOMMAIRE



- Introduction
- ES2015+
- Outilage
- Types et inférence de types
- Classes
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés





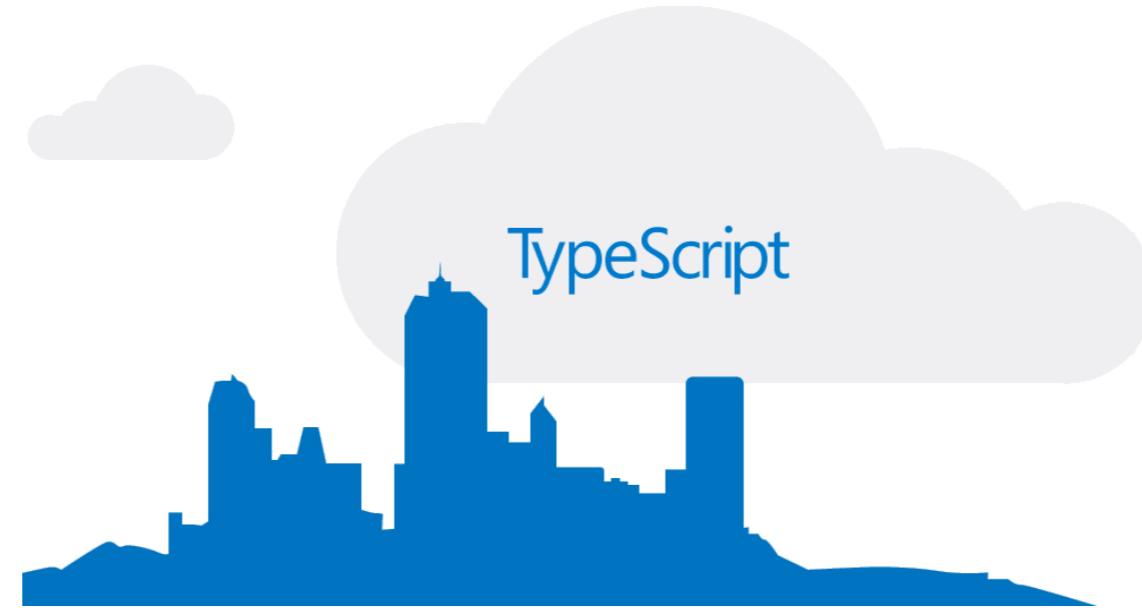
INTRODUCTION

PLAN



- *Introduction*
- ES2015+
- Outilage
- Types et inférence de types
- Classes
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés

TYPESCRIPT



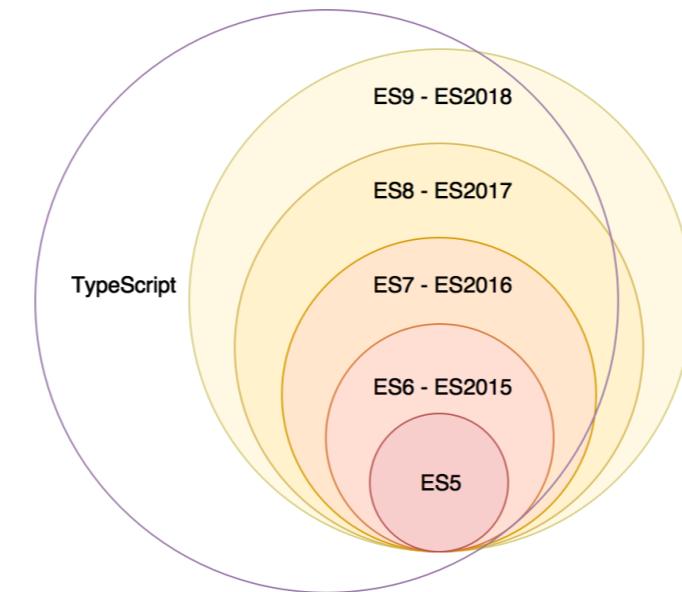
Langage créé par [Anders Hejlsberg](#) en 2012

Projet open-source maintenu par [Microsoft](#) qui propose une extension du JavaScript apportant le typage influencé par [Java](#) et [C#](#)

TYPESCRIPT



- Phase de compilation nécessaire pour générer du **JavaScript** : voir le [playground en ligne](#)
- Ajout de nouvelles fonctionnalités au langage **JavaScript** et supporte ESnext
- Rétrocompatible : tout programme **JavaScript** est un programme **TypeScript**
- Aujourd'hui, TS ajoute principalement les notions *de type* et d'annotations





POURQUOI TYPESCRIPT

Son utilisation permet de **valider votre code source en amont**, on parle de validation au **build time** qui permet d'éviter les erreurs au **runtime**.

A screenshot of a terminal window with a dark blue background. On the left, there are four small icons: a file, a magnifying glass, a gear, and a play button. The main area contains the following code:

```
const user = {
  firstName: "Angela",
  lastName: "Davis",
  role: "Professor"
}

console.log(user.name)
```

A red error message is displayed in a dark red box:

Property 'name' does not exist on type '{ firstName: string; lastName: string; role: string; }'.



POURQUOI TYPESCRIPT

Permet à votre IDE de fournir de l'**auto-completion**, même pour les librairies JS non typées.

A screenshot of the Visual Studio Code interface. On the left, there's a sidebar with four icons: a clipboard, a magnifying glass, a circular arrow, and a play button. The main editor area contains the following TypeScript code:

```
import express from "express"
const app = express()

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

app.listen(3001, () => {
    console.log("App is running on port 3001")
})
```

The cursor is at the end of the word 'send' in the first 'res.' call. A dropdown auto-completion menu is open, showing three suggestions: 'send', 'sendDate', and 'sendfile'. The 'send' option is highlighted.

Par exemple, l'auto-completion de VSCode sur des fichiers JS, c'est grâce aux fichiers de définitions et car il est entièrement écrit en Typescript.



POURQUOI TYPESCRIPT

Son utilisation permet de *décrire* vos objets et de faciliter la transmission et la maintenabilité de votre code.

A screenshot of a code editor interface, likely Visual Studio Code, showing a file with TypeScript code. The code defines an interface 'User' with properties id, firstName, lastName, and role, and a function 'updateUser' that takes an id and an update object (Partial<User>) to return a new user object. The code editor has a dark theme with icons on the left side.



FONCTIONNALITÉS

- Fonctionnalités **ES2015+**
- Types, inférence de type et type générique
- Enum, Tuples...
- Classes / Interfaces / Héritage / Décorateurs
- Développement modulaire
- Les fichiers de définitions

Notes :

HISTORIQUE DU LANGAGE JAVASCRIPT

Le JavaScript est un langage de script orienté **prototype** créé par **Brendan Eich** pour Sun/Netscape en 1995 et soumis plus tard à l'ECMA pour standardisation.

- 1 - 7 • L'**ECMA** est un organisme privé européen de standardisation

LA NORME ECMASCIPT



Historique des versions du standard **ECMAScript** pré-ES2015

Ver	Date	Évolution
1	Juin 1997	Adoption de l'ECMAScript 1
2	Juin 1998	Réécriture de la norme, première version du JavaScript comme on le connaît
3	Décembre 1999	RegExp, Mécanisme Try/Catch, Erreur, ...
4	Abandonnée	
5	Décembre 2009	- Clarifie beaucoup d'ambiguïtés de la V3 - Version la plus répandue dans les navigateurs modernes (IE9+)
5.1	Juin 2011	Alignement norme ISO 16262

LA NORME ECMASCIPT



Depuis 2015, les versions **ECMAScript** seront maintenant tous les ans, toutes les nouvelles fonctionnalités qui arrivent à maturité sont embarquées : la version prend le nom de l'année de sortie.

Par exemple **ES6** devient **ES2015**

Version	Date	Évolution
6 / ES2015	Juin 2015	Module, classe, destructuration, constante, arrow function, promise, generator, etc.
7 / ES2016	Juin 2016	Isolation de code, opérateur exponentiel '**', Array.prototype.includes
8 / ES2017	Juin 2017	Gestion de la concurrence, async/await
9 / ES2018	Juin 2018	Amélioration RegExp, Promises finally, Itération asynchrone
10 / ES2019	Juin 2019	Amélioration Tableau, String, try ... catch
11 / ES2020	Juin 2020	BigInt, Optional Chaining, Nullish Coalescing, Promise.allSettled

POUR ALLER PLUS LOIN



- Site Officiel : <http://www.typescriptlang.org/>
Documentation dans *docs/handbook* très complète
- Testez TypeScript en ligne : <http://www.typescriptlang.org/play/>
Le playground permet de voir le code JavaScript généré en temps réel
- Repository Github : <https://github.com/Microsoft/TypeScript>
- Blog : <https://devblogs.microsoft.com/typescript/>
Là où Microsoft annonce les nouvelles versions





ECMASCRIPT 2015+

PLAN



- Introduction
- *ES2015+*
- Outilage
- Types et inférence de types
- Classes
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés



VARIABLES : LET & CONST

Remplacent avantageusement **var** et répondent à un scope traditionnel.

- **let**: référence modifiable avec un scope au bloc
- **const**: référence non modifiable avec un scope au bloc

```
let a = 1;
if (true) {
  let b = 2;
}
console.log(a, b); //-> 1 undefined

const c = { d: 4 };
c = 5; //-> error
c.d = 5; //-> ok
```

- Attention, avec **const** la référence est fixe, mais pas son contenu (exemple : un **array**)
- Utiliser **const** en priorité, **let** sinon

STRING INTERPOLATION



Ajout d'une troisième syntaxe pour définir les strings.

- ' et " existent toujours et ne sont pas modifiés
- ` est ajouté pour les strings et ajoute des possibilités
 - Support des string multi-lignes
 - Support de l'interpolation de variables
 - Ajout d'un mécanisme de templatisation

```
const a = `Je suis
une string multi-lignes`;

const b = `avec interpolation de ${a} au milieu`;

const tag = function (strings, value1, value2) {
  console.log(strings); //-> ['string ', ' template '];
  console.log(value1, value2); //-> 1 2
};
const c = tag`string ${1} template ${2}`;
```

ARROW FUNCTIONS



Nouvelle syntaxe pour définir les fonctions, impacte beaucoup la physionomie du code !

```
// "ancienne" version
let double = function (arg) {
    return arg * 2;
};

// arrow function !
double = (arg) => {
    return arg * 2;
};

// sans block = return automatique
double = (arg) => arg * 2;

// un seul argument = parenthèses optionnelles
double = arg => arg * 2;
```

ARROW FUNCTIONS



⚠️ Attention les arrows functions ont un fonctionnement légèrement différent

- Une arrow function ne crée pas de scope
- Très pratique pour ne pas avoir les problèmes de **this**
- Mais le manque du **this** peut poser problème dans certains cas

```
const a = {
  b: 3,
  c: function () {
    [1, 2].forEach((value) => console.log(value + this.b)); //-> 4 5 \o/ !
    [1, 2].forEach(function (value) {
      console.log(value + this.b); //-> NaN NaN : this.b is undefined !
    });
  },
  d: () => console.log(this.b), //-> undefined
};
```

CLASSES



- Mot clé et la notion de classe qui fonctionne en interne sur le modèle des prototypes
- Propose une partie seulement du modèle habituel (Java, C#)
 - OK : héritage, constructeur, méthodes, méthodes statiques
 - NOK : champs, champs statiques, interfaces, classes abstraites

```
class Chien extends Animal {  
    constructor() {  
        super();  
    }  
  
    uneMethode() {  
        return 42;  
    }  
  
    static uneMethodeStatic() {  
        return 43;  
    }  
}
```

MODULE ES2015



Chaque fichier JavaScript représente un module et :

- est isolé par défaut des autres modules au niveau du scope
- peut publier une API (**export**)
- peut utiliser l'API d'autres modules (**import**)

```
// Module A
export const name = "zenika";
export default 2;

// Module B
import two, { name } from "moduleA";
console.log(two, name); //-> 2 'zenika'
```

Deux imports identiques du même module rendent la même référence



DESTRUCTURING

- Permet d'affecter des variables rapidement
- Fonctionne en reproduisant la forme de la donnée d'origine
 - Sur les tableaux en fonction de leur position
 - Sur les objets en fonction de leur clé

```
const source = [1, 2];
const [a, b] = source;
console.log(a, b); //-> 1 2

const [d, e, ...f] = [1, 2, 3, 4, 5];
console.log(d, e, f); //-> 1 2 [3, 4, 5]

const { g, h } = { g: 1, h: 2 };
console.log(g, h); //-> 1 2

const func = () => [1, 2, 3];
const [i, ,j] = func();
console.log(`i = ${i}, j = ${j}`); //-> i = 1, j = 3
```

REST & SPREAD



Opérateur **...** permet d'accumuler ou distribuer les valeurs d'un tableau

```
// Rest arguments
const func = (a, b, ...c) => console.log(a, b, c);
func(1, 2, 3, 4); //-> 1 2 [3, 4]

// Spread arguments
const tab = [1, 2, 3, 4];
func(...tab); //-> 1 2 [3, 4]

// Spread array
const newTab = [1, 2, ...tab];
console.log(newTab); //-> [1, 2, 1, 2, 3, 4]

// Spread object (ES2017)
const obj = { a: 1, b: 2, c: 3 };
const newObj = { a: 4, ...obj, b: 5 };
console.log(newObj); //-> {a: 1, b: 5, c: 3}
```



FOR OF

- Nouveau `for /* ... */ of` pour compléter le `for /* ... */ in`
- Permet d'itérer sur les *valeurs* comme on s'y attend (contrairement au `in`)

```
let iterable = [10, 20, 30];

for (let value of iterable) {
  console.log(value);
} // output: 10 20 30

for (let value in iterable) {
  console.log(value)
} // output: 0 1 2
```

Fonctionne sur une abstraction d'objets `iterable` (Array, Map, Set, String répondent à cette abstraction).

Il est possible de rendre son objet itérable

PROMISES



JavaScript s'appuie énormément sur un fonctionnement asynchrone

Historiquement, ce fonctionnement asynchrone était implémenté avec des callbacks

```
somethingWhichTakesTime(arg1, arg2, function callback() {  
  console.log("done !");  
});
```

- Les callbacks posent d'important problème de lisibilité
- ES2015 normalise une nouvelle approche : les promesses avec la class **Promise** qui fait maintenant partie des types de base.
- Une promesse est un objet représentant l'état d'un traitement asynchrone

```
somethingWhichTakesTime(arg1, arg2).then(() => {  
  console.log("done");  
});
```

PROMISES



Les promesses améliorent nettement les codes asynchrones avec de nombreux avantages :
Chainage, erreurs, traitements parallèles...

```
somethingWhichTakesTime()
  .then((data) => data.userId) // On peut retourner un type simple
  .then((userId) => {
    // On peut retourner une autre promesse
    return somethingElseWhichTakesTime(userId);
  })
  .then((secondData) => {
    // Sera exécuté à la résolution de la seconde promesse
    console.log(secondData);
  })
  .catch((err) => {
    // Gestion centralisée des erreurs de toute la chaîne
    console.error(err);
  })
  .finally(() => {
    // On peut gérer des tâches à la fin de toutes les promesses, qu'elles aient échoué ou non
  });
}
```

ASYNC / AWAIT



Permet l'écriture de code asynchrone de façon synchrone

```
async function ping() {
  for (let i = 0; i < 10; i++) {
    await delay(300);
    console.log("ping");
  }
}

function delay(ms: number) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}

ping();

// ping
// ping
// ping
// ...
```

Notes :



OPTIONAL CHAINING

La nouvelle écriture

```
// Before
if (foo && foo.bar && foo.bar.baz) {
    // ...
}

// After-ish
if (foo?.bar?.baz) {
    // ...
}
```



OPTIONAL CHAINING

- Appel optionnel à des fonctions
- Appelle la fonction si celle-ci est définie (non null ou undefined)

```
async function makeRequest(url, log) {
  log?.(`Request started at ${new Date().toISOString()}`);
  // roughly equivalent to
  // if (log != null) {
  //   log(`Request started at ${new Date().toISOString()}`);
  // }

  const result = (await fetch(url)).json();

  log?.(`Request finished at ${new Date().toISOString()}`);

  return result;
}
```



NULLISH COALESING

Utilisable grâce à l'opérateur `??`

- Si la première expression n'est pas `null` ou `undefined`, alors cela retourne cette expression, sinon cela retourne la seconde
- Cette fonctionnalité existe aussi dans JavaScript depuis [ES2020](#).

```
let x = foo ?? bar();

// Equivalent to
let x = foo !== null && foo !== undefined ? foo : bar();
```



NULLISH COALESCING

À préférer à `||`, qui convertie en boolean l'opérande de gauche puis si elle est fausse, évalue l'opérande de droite

```
function initializeAudio() {  
  let volume = localStorage.volume || 0.5;  
  
  // ...  
}
```

Quand `localStorage.volume` est à 0, la page va setter le volume à 0,5 ce qui n'est pas attendu.

`??` évite les mauvaises interpretations de `0`, `NaN` et `""` comme des **falsy values**.





OUTILLAGE

PLAN



- Introduction
- ES2015+
- *Outillage*
- Types et inférence de types
- Classes
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés

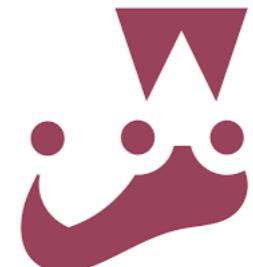


TOOLING

- Node.js
- npm
- TypeScript
- Eslint
- Jest



TypeScript



NODE.JS



Node.js est une plateforme basée sur un moteur JavaScript

La très grande majorité des outils de développement Web est actuellement réalisée avec Node.js

Fonctionne avec le moteur JavaScript V8 de Google Chrome

Étend le JavaScript avec une API système lui permettant de communiquer avec le système d'exploitation



npm est le gestionnaire de paquets de Node.js. Paquets disponibles sur <https://www.npmjs.com/>.

La commande npm est installée en même temps que Node.js

Pour installer un paquet dans le répertoire courant

```
npm install my-package
```

Pour installer un paquet globalement au niveau du système

```
npm install --global my-package
```



NPM - PACKAGE.JSON

- Fichier qui permet de définir les dépendances d'un projet
- Similaire au **pom.xml** de Maven

```
{  
  "name": "project-name",  
  "version": "0.0.0",  
  "description": "project's description",  
  "author": "zenika",  
  
  "dependencies": {  
    "@angular/core": "*"  
  },  
  "devDependencies": {  
    "typescript": "^0.7.0"  
  },  
  
  "private": true  
}
```



NPM - DEPENDENCIES

Deux types de dépendances :

- **dependencies** : nécessaires au projet lui-même
- **devDependencies** : utiles uniquement pour le développement

Pour installer un paquet et le sauvegarder dans la liste des dépendances :

- **dependencies** :

```
npm install my-package
```

- **devDependencies** :

```
npm install --save-dev my-package
```



TypeScript

TypeScript se présente sous la forme d'un paquet npm et contient une CLI qui permet d'utiliser le compilateur.

- Pour l'installer `npm install --save-dev typescript`
- Après l'installation, vous avez accès à la commande `tsc`
 - `tsc --help` pour avoir la liste de toutes les commandes
 - `tsc file.ts` compile le fichier `file.ts`
 - `tsc --sourceMap file.ts` génère le source map

TYPESCRIPT



Toutes les options de compilation sont accessibles via la CLI, mais le plus souvent, on utilisera un fichier de configuration **tsconfig.json**.

Pour l'initialiser :

```
tsc --init
```

Utilisation du fichier

- Si à la racine : **tsc**
- Si chemin spécifique : **tsc --config path/tsconfig.json**



TYPESCRIPT - TSCONFIG.JSON

Exemple de fichier de configuration

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "ES2019",
    "noImplicitAny": false,
    "sourceMap": false,
    "strict": true, // Less permissive
    "outDir": "./dist"
  },
  "include": ["src/**/*"]
}
```



TYPESCRIPT

tsc file.ts

```
// file.ts
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return `Hello, ${this.greeting}`;
  }
}
let greeter = new Greeter("world");
```



TYPESCRIPT

==> résultat de la transpilation du fichier `file.ts` avec `target = ES5`

```
"use strict";
var Greeter = /** @class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, ".concat(this.greeting);
    };
    return Greeter;
}());
var greeter = new Greeter("world");
```



TSConfig.json

Spécification des options de compilation et des fichiers à inclure / exclure

```
{  
  "compilerOptions": {  
    "target": "ES2019",  
    "module": "commonjs",  
    "removeComments": true,  
    "outDir": "./dist",  
    "sourceMap": true  
  },  
  "files": ["file1.ts", "file2.ts"],  
  "excludes": ["node_modules", "file3.ts"]  
}
```

Schema JSON : <http://json.schemastore.org/tsconfig>



- Linter (analyse statique du code) pour JavaScript et TypeScript
- Remplace **TSLint**, déprécié depuis décembre 2019
- Gère les règles spécifiques au langage TypeScript

```
npm i -D eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin
```

- Après l'installation, vous avez accès à la commande **eslint**
 - L'argument **--fix** permet de corriger directement la plupart des erreurs.
 - L'argument **--cache** permet de linter uniquement les fichiers modifiés. Sur un gros projet, cela améliore bien le temps d'exécution.



.ESLINTRC

Le fichier `.eslintrc` permet de paramétrer ESLint.

Exemple :

```
{  
  "parser": "@typescript-eslint/parser",  
  "extends": ["plugin:@typescript-eslint/recommended"],  
  "parserOptions": {  
    "ecmaVersion": 2019,  
    "sourceType": "module"  
  }  
}
```



TESTS

Pour les tests TypeScript, on utilise les frameworks JavaScript

Il existe un grand nombre de frameworks et librairies

- Frameworks de Tests : [Jasmine](#), [Mocha](#), [qUnit](#), [AVA](#)
- Tests Runners : [Karma](#), [Jest](#), [Vitest](#)
- Librairies utilitaires : [Chai](#), [Sinon](#)

JEST



Framework de tests et test runner : <https://jestjs.io/>. Permet l'exécution des tests en parallèle, peu de configuration

```
npm install --save-dev jest
```

Association de la commande **jest** à un script npm

```
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watchAll"
  }
}
```



JEST & TYPESCRIPT

Possibilité d'écrire les tests en TypeScript avec le preprocessor *ts-jest*

```
npm install --save-dev ts-jest
```

Rajouter le preprocessor à la configuration jest

```
npx ts-jest config:init
```

Lancer les tests

```
jest <regexForTestFiles>
```



STRUCTURE D'UN TEST JEST

- Fonction `describe` et `it` ou `test` pour définir un test
- Système d'assertions: `toBeTruthy`, `toBeFalsy`, `toBe`, `toContain`, `toThrow`, ...

```
describe("My tests", () => {
  it("should return true", () => {
    let flag = true;
    expect(flag).toBeTruthy();
  });
});
```

```
test("if it returns true", () => {
  let flag = true;
  expect(flag).toBeTruthy();
});
```



STRUCTURE D'UN TEST JEST

- Méthodes **before, after, beforeEach, afterEach**
- Exécution d'une fonction avant ou après, tous ou chaque test

```
describe("Mes tests", () => {
  let flag;

  beforeEach(() => {
    flag = true;
  });

  it("should return true", () => {
    expect(flag).toBeTruthy();
  });
});
```

- Les tests peuvent être asynchrones
- Support natif des **callback, Promise, Observable** ou **async/await**



MOCK ET SPIES

- Il existe deux façons de mocker des fonctions :
 - soit en créant une fonction de mock à utiliser dans le test
 - soit en mockant automatique une dépendance de module
- La fonction de mock : `const mock = jest.fn()`
- Le mock de module : `jest.mock('../foo');`
- Vérifier l'exécution de la méthode *espionnée*

```
test("should call console.log with args", (t) => {
  const spy = jest.spyOn(console, "log");
  console.log("Hello Zenika");
  expect(spy).toHaveBeenCalled();
  expect(spy).toHaveBeenCalledWith("Hello Zenika");
});
```





Lab 1



TYPES ET INFÉRENCE DE TYPES

PLAN



- Introduction
- ES2015+
- Outilage
- *Types et inférence de types*
- Classes
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés



VARIABLES

Mêmes types de variables que JavaScript : **var**, **let** et **const**

TypeScript ajoute le type avec la syntaxe : **type**

```
var variableName: variableType = value;  
  
let variableName2: variableType = value;  
  
const variableName3: variableType = value;
```



TYPES PRIMITIFS

TypeScript propose un certain nombre de types primitifs

- Données : `boolean`, `number`, `string`, `bigint`
- Structure : `Array`, `Tuple`, `Enum`
- Spécifique : `unknown`, `any`, `void`, `null`, `undefined`, `never`

```
const isDone: boolean = false;
const height: number = 6;
const phone_number: number = 555_734_223;
const million: number = 1_000_000;
const name: string = "Carl";
const names: string[] = ["Carl", "Laurent"];
const notSure: unknown = 4;
const readable = 5; //→ implicit number
```



TYPES PRIMITIFS

Une variable typée permet d'utiliser les prototypes des objets primitifs

```
const name: string = "Carl";
name.indexOf("C"); // 0

const num: number = 6.01;
num.toFixed(1); // 6.0

const arr: string[] = ["carl", "laurent"];
arr.filter((element) => element === "carl"); // ['carl']
```



FONCTIONS

TypeScript permet toutes les syntaxes d'ES2015+ pour les fonctions

On y ajoute le typage des arguments et les valeurs de retour

```
function namedFunction(arg1: string, arg2: number): string {  
    return `${arg1} - ${arg2}`;  
}
```

- On peut typer les variables qui contiendront une fonction
- Pour cela il faut décrire les paramètres et le type de retour

```
let myFunc: (arg1: string, arg2: number) => string;
```



PARAMÈTRES OPTIONNELS

Un paramètre peut être optionnel

- Utilisation du caractère `?` avant le type
- L'ordre de définition est très important
- Aucune implication dans le code JavaScript généré
- S'il n'est pas défini, le paramètre aura la valeur `undefined`

```
function log(name: string, surname?: string): void {  
    console.log(name, surname);  
}  
  
log(); //→ error  
log("carl"); //→ 'carl' undefined  
log("carl", "boss"); //→ 'carl' 'boss'
```



PARAMÈTRES PAR DÉFAUT

Existe en ES2015. Possibilité de définir une valeur par défaut pour chaque paramètre

- Directement dans la signature de la méthode
- Utilisation du signe `=` après le type
- Ajoutera une condition `if` dans le JavaScript généré si `target < ES2015`

```
function log(name: string = "carl"): void {
  console.log(name);
}

log(); //-> 'carl'
log("laurent"); //-> 'laurent'
```



REST PARAMETERS

Existe en ES2015. Permet de manipuler un ensemble de paramètres en tant que groupe

- Utilisation de la syntaxe `...`
- Doit correspondre au dernier paramètre de la fonction
- Est un tableau d'objets

```
function company(nom: string = "zenika", ...agencies: string[]) {  
    console.log(nom, agencies);  
}  
  
company("zenika", "spotify", 4); //-> error  
  
company("zenika", "spotify", "blablacar"); //-> 'zenika' ['spotify', 'blablacar']  
  
company(); //-> 'zenika' []
```



UNION TYPES

Une fonction peut retourner un type différent en fonction des paramètres.

- On la type avec un **Union**.

```
function returnParam(param: string | number): string | number {  
    return param  
}
```



ARRAYS

Permet de manipuler un tableau d'objets. 2 syntaxes pour créer des tableaux

- Syntaxe Littérale

```
const list: number[] = [1, 2, 3];
```

- Syntaxe utilisant le constructeur **Array**

```
const list: Array<number> = [1, 2, 3];
```

Ces 2 syntaxes aboutiront au même code JavaScript



TUPLES

TypeScript apporte la notion de **tuple** car JavaScript n'en possède pas.

C'est pourtant une notion couramment utilisée par d'autres langages

- La version compilée fonctionne avec des tableaux
- Possibilité d'utiliser le destructuring avec les tuples

```
const tuple: [string, number] = ["Zenika", 10];  
  
const [name, age] = tuple;
```



TUPLES

- Possibilité de rendre optionnel un élément dans un tuple

```
let t: [number, string?, boolean?];
t = [42, "hello", true];
t = [42, "hello"];
t = [42];
```

- Intégration avec les Rest parameters et les Spread expressions

```
// Rest parameters
declare function foo(...args: [number, string, boolean]): void;
declare function foo(args_0: number, args_1: string, args_2: boolean): void;

// Spread expressions
const args: [number, string, boolean] = [42, "hello", true];
foo(42, "hello", true);
foo(args[0], args[1], args[2]);
foo(...args);
```

ENUM



Ajout de la notion d'**Enum** pour les listes finies de valeurs

```
enum Music {  
    Rock,  
    Jazz,  
    Blues  
}  
  
const c: Music = Music.Jazz;
```

TypeScript associe automatiquement à une valeur numérique **zero-based**

INFÉRENCE DE TYPES



TypeScript va tenter de définir le type des variables non typées explicitement. Il va se baser sur :

- Les types de données à l'initialisation des variables
- Les valeurs par défaut des arguments de fonctions
- Le type de données retourné dans une fonction

```
const maVariableNumber = 3; // type number

function log(name = "carl") {
    // type string
    return name; // type string
}
```

S'il ne trouve pas, il utilisera **any**.

INFÉRENCE DE TYPES



Cela lui permet de lever des erreurs à la compilation, même lorsque le type des variables n'avait pas été fixé

```
function func(name: string): void {
  console.log(name.trim());
}

const a = 42; // inference de type number

func("  toto  "); //-> 'toto'
func(a); // error (Argument of type '42' is not assignable to parameter of type 'string')
```







CLASSES

PLAN



- Introduction
- ES2015+
- Outilage
- Types et inférence de types
- *Classes*
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés



CLASSES

Système de **classes** et **interfaces** similaire à la programmation orientée objet.

- Le code javascript généré utilisera le système de **prototype**
- Possibilité de définir un constructeur, des méthodes et des propriétés
- Propriétés/méthodes accessibles via l'objet **this**
- Attention, comme en JavaScript, le **this** est **toujours** explicite
- Reprend la syntaxe des classes ES2015 et y ajoute des possibilités

```
class Person {  
    constructor() {}  
}  
  
const person: Person = new Person();
```



CONSTRUCTEURS

Comme en JavaScript :

- Le constructeur est défini avec le mot clé **constructor**
- Il ne peut y avoir qu'un seul constructeur et il est optionnel
- Le constructeur peut avoir des arguments typés ou non
- Il est possible d'utiliser toutes les fonctionnalités pour les arguments
- Valeur par défaut, argument optionnel, rest

```
class Person {  
  constructor(  
    firstName: string,  
    lastName: string = "Dupont",  
    age?: number,  
    ...hobbies: string[]  
  ) {}  
}
```

MÉTHODES



- Les méthodes sont ajoutées dans le corps de la classe
- Elles sont définies comme des fonctions (mais sans le mot clé **function**)
- Lors de la compilation, les méthodes sont ajoutées au **prototype** de l'objet

```
class Person {  
  sayHello(message: string): void {  
    console.log(`Hello ${message}`);  
  }  
}  
  
const person: Person = new Person();  
person.sayHello("World"); //-> 'Hello World'
```

PROPRIÉTÉS



- Elles sont définies également dans le corps de la classe sans mettre `let` ou `const`
- Se comporte comme un `let`, la référence est modifiable
- Pour obtenir une propriété non modifiable, utiliser le flag `readonly`
- Possibilité d'initialiser la propriété, sinon elle est `undefined`

```
class Person {  
    firstName: string = 'Marc';  
    lastName: string = "Dupont";  
    readonly age: number = 20;  
    hobbies: string[] = [];  
}  
  
const person: Person = new Person();  
console.log(person.lastName); //-> 'Dupont'  
person.age = 30 // Error  
// Cannot assign to 'age' because it is a read-only property.
```



SCOPES

- Méthodes et propriétés ont forcément un scope
- 3 scopes disponibles : **public, private, protected**
- Lorsqu'il n'est pas défini, c'est automatiquement **public**
- Il existe une règle ESLint pour forcer l'écriture explicite

```
class Person {  
    private message: string = "World";  
    public sayHello(): void {  
        console.log(`Hello ${this.message}`);  
    }  
}  
  
const person: Person = new Person();  
person.sayHello(); //-> 'Hello World'  
console.log(person.message); // compilation error  
// Property 'message' is private and only accessible within class 'Person'.
```



SCOPES

Le scope **private** n'a aucun effet sur le code Javascript généré. Le caractère privé d'une variable marquée comme **private** n'est donc pas garantie.

Si votre cible est \geq ES2015 (cf. propriété **target** de votre config Typescript), utiliser plutôt la syntaxe **Private class features** pour obtenir une réelle encapsulation, en préfixant votre propriété par **#**:

```
class Box {  
    #secretField: string = "my secret"; // encapsulation garantie, même en JS  
    private notSoSecretField: string = "not so secret" // not so secret when compiling in JS  
    myPublicProp: string = "my public prop"  
  
    get secret(): string {  
        return this.#secretField  
    }  
}  
  
const box = new Box();  
console.log(box.myPublicProp);  
console.log(box.secret) // my secret stuff  
console.log(box.#secretField) // Error
```

STATIC



- Possibilité de définir des propriétés et des méthodes statiques avec **static**
- Automatiquement public
- Les propriétés et méthodes statiques ne sont pas accessibles via le **this**

```
class Person {  
    static message: string = "World";  
  
    static sayHello(): void {  
        console.log(`Hello ${Person.message}`);  
    }  
}  
  
Person.sayHello(); //-> 'Hello World'  
console.log(Person.message); // -> 'World'  
  
const person: Person = new Person();  
person.sayHello(); // Error
```

PROPRIÉTÉS DANS LE CONSTRUCTEUR



- TypeScript emprunte au C# un raccourci pour déclarer et initialiser les propriétés en une fois
- Il suffit d'ajouter le scope aux paramètres du constructeur

```
class Person1 {
  constructor(public message: string, private age: number) {}
}

class Person2 {
  public message: string;
  private age: number;

  constructor(message: string, age: number) {
    this.message = message;
    this.age = age;
  }
}
```

ACCESSEURS GETTERS / SETTERS



Possibilité de définir des accesseurs pour accéder à une propriété, ou un dérivé de cette propriété.

Utiliser les mots clés `get` et `set`

```
class Person {  
    #secret: string; // Le préfixe "#" permet de créer une propriété privée, même après  
    compilation en JS  
  
    get secret(): string {  
        return this.#secret.toLowerCase();  
    }  
    set secret(value: string) {  
        this.#secret = value;  
    }  
}  
  
const person = new Person();  
person.secret = "ABC";  
console.log(person.secret); //-> 'abc'
```

HÉRITAGE



Fonctionne comme en ES2015 avec l'ajout de la gestion des scopes

- L'héritage entre classes utilise le mot-clé **extends**
- Si constructeur non défini, exécute celui de la classe parente
- Si défini, il doit appeler celui de la classe parente via **super**
- Accès aux propriétés de la classe parente si **public** ou **protected**

```
class Person {  
    constructor() {}  
    speak() {}  
}  
  
class Child extends Person {  
    constructor() {  
        super();  
    }  
    speak() { super.speak(); }  
}
```

INTERFACES



Utilisées par le compilateur pour vérifier la cohérence des différents objets

- Aucun impact sur le JavaScript généré
- Système d'héritage entre interfaces
- Les interfaces ne servent pas seulement à vérifier les classes
- Plusieurs utilisations possibles
 - Vérification des paramètres d'une fonction
 - Vérification de la signature d'une fonction
 - Vérification de l'implémentation d'une classe

INTERFACES - PARAMÈTRES D'UNE FONCTION



Vérification des paramètres d'une fonction

```
interface Message {  
  message: string;  
  title?: string;  
}  
  
function sayHello(options: Message) {  
  console.log(`Hello ${options.message}`);  
}  
  
const message: Message = { message: "World" };  
  
sayHello({ message: "World", title: "Zenika" }); //-> 'Hello World'  
sayHello({ message: "World" }); //-> 'Hello World'  
sayHello(message); //-> 'Hello World'  
sayHello(); // compilation error  
// Supplied parameters do not match any signature of call target.
```

INTERFACES - SIGNATURE D'UNE FONCTION



Vérification de la signature d'une fonction

```
interface SayHello {
  (message: string): string;
}

let sayHello: SayHello;

sayHello = function (source: string): string {
  return source.toLowerCase();
};

sayHello = function (age: number): boolean {
  return age > 18;
}; // compilation error
// Type '(age: number) => boolean' is not assignable to type 'SayHello'.
// Types of parameters 'age' and 'message' are incompatible.
// Type 'string' is not assignable to type 'number'.
```

INTERFACES - IMPLÉMENTATION D'UNE CLASSE



Vérification de l'implémentation d'une classe. Erreur de compilation tant que la classe ne respecte pas le contrat

```
interface Person {
    sayHello(message: string): void;
}

class Adult implements Person {
    sayHello(message: string): void {
        console.log(`Hello ${message}`);
    }
}

class Duck implements Person {
    quack(): void {
        console.log("Quack");
    }
} // compilation error
// Class 'Duck' incorrectly implements interface 'Person'.
// Property 'sayHello' is missing in type 'Duck'.
```

DUCK TYPING



- TypeScript propose une fonctionnalité appelée le Duck Typing
- Rend valide un type qui possède des propriétés communes avec un autre type
- **Attention, peut mener à des erreurs au runtime**

Si je vois un oiseau qui vole comme un canard, cancane comme un canard, et nage comme un canard, alors j'appelle cet oiseau un canard

DUCK TYPING



Par exemple :

```
interface Animal {
  color: string;
  eat(food: string): void;
}

class Duck {
  color = 'brown';
  eat (food: string): void {
    console.log(food)
  };
}

class Stuff {
  color: string = 'black';
}

const saveAnimal = (animal: Animal): void => {};

saveAnimal(new Duck()); // OK
saveAnimal(new Stuff()); // Error
```

CLASSES - CLASSES ABSTRAITES



- Ajout d'un mot-clé **abstract**
- Classes qui ne peuvent pas être instanciées directement
- Peut contenir des méthodes implémentées ou **abstract**

```
abstract class Person {  
    abstract sayHello(message: string): void;  
}  
  
class Adult extends Person {  
    sayHello(message: string): void {  
        console.log(`Hello ${message}`);  
    }  
}  
  
const adult = new Adult();  
const person = new Person(); // compilation error  
// Cannot create an instance of the abstract class 'Person'.
```



PROPRIÉTÉS ABSTRACT

- Possibilité d'indiquer qu'une propriété est abstraite (dans une classe abstraite)
- Force la classe **fille** à implémenter la propriété

```
abstract class Foo {  
    abstract bar: string;  
}
```

```
class Bar extends Foo {  
    bar: string = 'foo'  
    // sinon erreur !  
}
```



PROPRIÉTÉS OPTIONNELLES

- Possibilité d'indiquer qu'une propriété est optionnelle dans une classe
- Même fonctionnement que pour une interface

```
class Foo {  
    foo: boolean;  
    bar: string;  
    baz?: string;  
}  
  
function doSomethingWithFoo(foo: Foo) { ... }  
  
doSomethingWithFoo({ foo: true, bar: 'hello' }); // Ok
```

GÉNÉRIQUES



- Fonctionnalité permettant de variabiliser un type
- Inspiration des génériques disponibles en Java ou C#
- Nécessité de définir un (ou plusieurs) paramètre(s) de type sur :
fonction / variable / classe / interface générique

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

identity(5).toFixed(2); //-> '5.00'

identity(true); //-> true

identity("hello").toFixed(2); // compilation error
// Property 'toFixed' does not exist on type '"hello"'.
```

GÉNÉRIQUES



- Possibilité de définir une classe générique
- Définition d'une liste de paramètres de types de manière globale
- Un type générique peut avoir une valeur par défaut

```
class List<T, Index=number> {
    push(value: T) {}
    add(value: T) {}
    splice(value: T, index: Index) {}
}

const numericArray = new List<number>();
numericArray.add(5);
numericArray.splice(5, 2);

numericArray.add('hello'); // compilation error
// Argument of type '"hello"' is not assignable to parameter of type 'number'.

const bigArray = new List<string, bigint>()
```

GÉNÉRIQUES



- Implémentation d'une interface générique

```
interface transform<T, U> {
    transform: (value: T) => U;
}

class NumberToStringTransform implements transform<number, string> {
    transform(value: number): string {
        return value.toString();
    }
}

const numberTransform = new NumberToStringTransform();

numberTransform.transform(3).toLowerCase(); //-> '3'

numberTransform.transform(3).toFixed(2); // compilation error
// Property 'toFixed' does not exist on type 'string'.
```

GÉNÉRIQUES



Possibilité d'utiliser le mot-clé **extends** sur le type paramétré

```
interface Musician {  
    play: () => void;  
}  
class JazzPlayer implements Musician {  
    play() {}  
}  
class PopSinger {  
    play() {}  
}  
class RockStar {  
    shout() {}  
}
```

GÉNÉRIQUES



Et son usage

```
function playAll<T extends Musician>(...musicians: T[]): void {
  musicians.forEach((musician) => {
    musician.play();
  });
}

playAll(
  new JazzPlayer(), // OK
  new PopSinger(), // OK
  new RockStar() // compilation error
);
// The type argument for type parameter 'T' cannot be inferred from the usage. Consider
// specifying the type arguments explicitly.
// Type argument candidate 'JazzPlayer' is not a valid type argument because it is not a
// supertype of candidate 'RockStar'.
// Property 'play' is missing in type 'RockStar'.
```





Lab 3



MODULES



PLAN

- Introduction
- ES2015+
- Outilage
- Types et inférence de types
- Classes
- *Modules*
- Type Definitions
- Décorateurs
- Concepts Avancés



MODULES

- Attention, la notion et les terminologies ont changé pour TypeScript 1.5
- TypeScript avait son propre système de modules
- Lors de la nouvelle version, ils se sont alignés sur la spécification ES2015
- *Internal modules* sont à présent des *namespaces*
- *External modules* sont à présent des *modules*

En TypeScript, comme en ES2015, n'importe quel fichier contenant un **import** ou un **export** au premier niveau est considéré être un module.



MODULES

- Éviter de polluer le namespace global (window dans les navigateurs)
- Permet d'organiser votre code TypeScript
- Par défaut, les variables, fonctions, classes... ne sont pas visibles de l'extérieur
- Syntaxe identique à celle d'**ES2015**
- L'import d'un module réalisé via **import**
- Pour exporter des objets, utilisation de **export**
- TypeScript peut compiler vers plusieurs mécanismes de chargement : *CommonJS, AMD, UMD, System, ES2015, ES2020 et ESNext*
- Configurable via la propriété **module** lors de la compilation.



MODULES

- Définition d'un module dans un fichier *utils.ts*

```
export function createLogger(): Logger { ... }

export class Logger {
  ...
}

export function getDate() { ... }
```

- Utilisation du module *utils*

```
import { createLogger, Logger, getDate } from "./utils";

const logger: Logger = createLogger();

logger.log("Hello World", getDate());
```



MODULES - DEFAULT

- Tout module a un et un seul export par default
- Utilisation du mot clé **default** lors de l'export
- Pas d'accolades dans la syntaxe de l'import
- L'objet pourra être importé en utilisant n'importe quel libellé
- Possibilité d'avoir un module avec un **default** et des exports nommés

```
export default class MyClass { ... }
```

```
export class MyOtherClass { ... }
```

```
import MyClassAlias, { MyOtherClass } from "./MyClass";
```

```
const a: MyClassAlias = new MyClassAlias();
```

```
const b: MyOtherClass = new MyOtherClass();
```



MODULES - AUTRES SYNTAXES

- Possibilité de renommer un import
- Permet de gérer des conflits de nommage

```
import { createLogger as newLogger, Logger } from "./utils";
let logger: Logger = newLogger();
```

- Import d'un module entier
- Crée un objet contenant chaque export dans la propriété du même nom
- Très utilisé pour consommer les bibliothèques Node.js historiques

```
import * as Utils from "./utils";
let logger = Utils.createLogger();
```



MODULES

- Il est possible de définir des alias via le fichier `tsconfig.json`
- Cela permet :
 - d'éviter les chemins relatifs
 - de faciliter l'extraction vers un nouveau module NPM

```
{  
  "compilerOptions": {  
    "baseUrl": "./src",  
    "paths": {  
      "@datorama/utils/*": ["app/utils/*"],  
      "@datorama/pipes/*": ["app/pipes/*"]  
    }  
  }  
}
```

```
import { forIn } from "@datorama/utils/array";
```





Lab 4



TYPE DEFINITIONS

PLAN



- Introduction
- Rappel ES5
- ES2015+
- Outilage
- Types et inférence de types
- Classes
- Modules
- *Type Definitions*
- Décorateurs
- Concepts Avancés



TYPE DEFINITIONS

- Fichier permettant de décrire une librairie JavaScript
- Extension **.d.ts**
- Depuis TypeScript 2, système de chargement automatique
 - Fichiers publiés dans l'organisation **@types** par le projet **DefinitelyTyped**
 - Fichiers définis directement dans une dépendance de votre projet
- Génération des types pour votre projet via **tsc --declaration**
- Permet de bénéficier
 - de **l'autocompletion**
 - du **type checking**



@TYPES

- Fichiers disponibles sur le repository Github
<https://github.com/DefinitelyTyped/DefinitelyTyped>
- Possibilité d'envoyer des Pull Requests avec les fichiers de définition de vos librairies
- <http://definitelytyped.org/>
- Dépendance uniquement nécessaire au développement

```
npm install --save-dev @types/jquery
(ou)
yarn add --dev @types/jquery
```



TSConfig.json

- Le système de détection automatique des types est configurable
- Tout est dans le *tsconfig.json* avec des valeurs par défaut
- **typeRoots**: répertoire dans lequel chercher pour les types.
(default: `node_modules/@types` et tous les `node_modules` "au dessus")
- **types**: liste des types à charger.
Attention, si définie, il n'y a plus de chargement automatique

```
{  
  "compilerOptions": {  
    "typeRoots": ["./typings", "./node_modules/@types"],  
    "types": ["node", "lodash", "express"]  
  }  
}
```

Pour faire son propre fichier de typage : un fichier `index.d.ts` doit être ajouté dans un sous-dossier de `typings`



PACKAGE.JSON

- Le fichier de définition peut être également packagé avec le module associé
- Ajout d'une propriété **types** dans le fichier **package.json** du module

```
{  
  "name": "typescript",  
  "author": "Zenika",  
  "version": "1.0.0",  
  "main": "./lib/main.js",  
  "types": "./lib/main.d.ts"  
}
```

- Ou par défaut : un fichier **index.d.ts** situé à la racine du module



SYNTAXE

- L'écriture d'un fichier de définitions dépend de la structure de la librairie
 - Librairie globale (disponible via l'objet **window**)
 - Librairie modulaire (utilisation des patterns **CommonJS, AMD, ...**)
 - Librairie globale et modulaire (pattern **UMD**)
- Différents templates disponibles sur le site TypeScript
- Utilisation du mot-clé **declare**

```
declare module angular {
  export interface IAngularStatic {
    bootstrap(
      element: string | Element,
      modules?: Array<string | Function | any[]>,
      config?: IAngularBootstrapConfig
    ): auto.IInjectorService;
  }
}
```



ACCESSEURS (GETTERS / SETTERS)

- Depuis la version 3.6, il est possible d'ajouter les accesseurs dans les fichiers de définitions de types

```
declare class Foo {  
    get x(): number;  
    set x(val: number);  
}
```





Lab 5



DÉCORATEURS



PLAN

- Introduction
- ES2015+
- Outilage
- Types et inférence de types
- Classes
- Modules
- Type Definitions
- *Décorateurs*
- Concepts Avancés

DÉCORATEURS



- Standard proposé par Yehuda Katz pour ECMAScript
- Dans le processus de normalisation (stage 2 pour Javascript)
- Annoter / Modifier des classes, méthodes, variables ou paramètres
- Un décorateur est une fonction ayant accès à l'objet à modifier
- Utilisation du caractère @ pour déterminer les décorateurs à utiliser
- Le compilateur retournera une erreur si le décorateur n'est pas défini

```
function decorator(target) {
  target.prototype.isDecorated = function () {
    console.log("decorated");
  };
}

@decorator
class DecoratedClass {}

new DecoratedClass().isDecorated(); //-> 'decorated'
```

DÉCORATEURS



- Les décorateurs sont implémentés dans TypeScript
- Mais ils ne sont pas activés par défaut :
 - le paramètre **--experimentalDecorators** en ligne de commande
 - dans le fichier **tsconfig.json**

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": false,  
    "experimentalDecorators": true,  
    "sourceMap": false  
  }  
}
```

DÉCORATEURS - LES DIFFÉRENTS TYPES



- ClassDecorator

```
@log  
class Person {}
```

- MethodDecorator

```
class Person {  
    @log  
    foo() {}  
}
```

DÉCORATEURS - LES DIFFÉRENTS TYPES



- PropertyDecorator

```
class Person {  
  @log  
  public name: string;  
}
```

- ParameterDecorator

```
class Person {  
  foo(@log param: string) {}  
}
```

DÉCORATEURS - LES DIFFÉRENTS TYPES



- La signature de la méthode est différente en fonction du décorateur

```
function classDecorator(target: Function) {}

function methodDecorator(
  target: any,
  key: string,
  description: PropertyDescriptor
) {}

function propertyDecorator(target: any, key: string) {}

function parameterDecorator(target: any, key: string, index: number) {}
```



CLASS DECORATORS

Reçoit le constructeur de la classe en argument et doit rendre le constructeur ou un nouveau

```
type ConstructorType = { new (...args: any[]): {} };

function log<T extends ConstructorType>(constructor: T): T {
  return class extends constructor {
    protected additionalProperty: string = "Hello";

    constructor() {
      super();
      console.log(`Person constructor called with ${this.additionalProperty}`);
    }
  };
}

@log
class Person {
  constructor(public name: string) {}
}

new Person("world");
// => New person overridden created with new property
```



METHOD DECORATORS

Reçoit en premier paramètre le prototype contenant la méthode ou la fonction Constructeur de la class dans le cas d'une méthode **static**, puis en deuxième le nom de la méthode décorée, et un descripteur. Doit rendre ce descripteur en l'état ou modifié

```
function log(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;
  if (originalMethod) {
    descriptor.value = (...args: any[]) => {
      const result = originalMethod.apply(target, args);
      console.log(`Method ${propertyKey} called with result ${result}`);
      return result;
    };
  }
  return descriptor;
}

class Person {
  @log
  sayHello() {
    return "Hello World";
  }
}
```

PROPERTY DECORATORS



- Reçoit le prototype de l'objet et le nom de la propriété
- La valeur de retour n'étant pas utilisée, son usage est assez limité. Il ne permet pas de récupérer la valeur de la propriété, ni de la modifier.
 - Il faut utiliser une dépendance ([reflect-metadata](#)), pour pouvoir en avoir un usage plus poussé

```
function log(target: any, propertyKey: string): void {
  console.log(target);
  console.log(`Valeur de la propriété décorée : ${target[propertyKey]}`);
}

class Person {
  @log
  private message: string = "Hello";
}

const JackSparrow = new Person();
// {} -> Le prototype est ici un objet vide
// Valeur de la propriété décorée : undefined
```



PARAMETER DECORATORS

- Reçoit le prototype de l'objet, le nom de la propriété de la méthode et le numéro du paramètre
- La valeur de retour n'est pas utilisée
- Comme le **Property Decorator**, son usage est très limité sans l'utilisation de reflect-metadata

```
function log(target: any, propertyKey: string, parameterIndex): void {
  console.log(`Method ${propertyKey} parameter ${parameterIndex} decorated`);
}

class Person {
  public sayHello(@log message: string): void {
    console.log(`Hello ${message}`);
  }
}
```

DECORATOR FACTORY



Il est possible pour un décorateur d'avoir des paramètres

```
function logDecoratorWithParam(prefix: string) {
  return function (target: any, key: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;
    if (originalMethod) {
      descriptor.value = (...args: any[]) => {
        const result = originalMethod.apply(target, args);
        console.log(` ${prefix} Method ${key} called with result ${result}`);
        return result;
      };
    }
    return descriptor;
  };
}

class Person {
  @logDecoratorWithParam("=> ")
  sayHello() { return "Hello World"; }
}

new Person().sayHello(); // => Method sayHello called with result Hello World
```



DECORATOR MIXTE

- Créer un décorateur avec une signature très générique
- En fonction des paramètres, utiliser la bonne implémentation

```
function log(...args: any[]) {
  switch (args.length) {
    case 1:
      return logClass.apply(this, args);
    case 2:
      return logProperty.apply(this, args);
    case 3:
      if (typeof args[2] === "number") {
        return logParameter.apply(this, args);
      }
      return logMethod.apply(this, args);
  }
}
```





Lab 6



CONCEPTS AVANCÉS



PLAN

- Introduction
- ES2015+
- Outilage
- Types et inférence de types
- Classes
- Modules
- Type Definitions
- Décorateurs
- *Concepts Avancés*



ALIAS

- Crédation d'alias à partir des types primitifs et des classes TypeScript créées
- Utilisation du mot-clé **type**
- Aucun impact sur le code JavaScript généré
- Disponible depuis TypeScript 1.4
- Ne peut être déclaré deux fois avec le même nom dans le même module
- Plus simple que les *interfaces* pour manier des compositions

```
type MyNumber = number;  
  
const num: MyNumber = 2;
```



TYPE UNION

- Permet de définir des variables pouvant être de différents types

```
let stringAndNumber: string | number;  
  
stringAndNumber = 1; // OK  
stringAndNumber = "string"; // OK  
  
stringAndNumber = false; // KO
```

- Accès aux propriétés communes à tous les types

```
const stringOrStringArray: string[] | string;  
  
console.log(stringOrStringArray.length);  
//OK car la propriété length disponible pour les deux types
```



TYPE UNION ET ALIAS

- Il est possible de cumuler **type union et alias**

```
type ArrayOfPrimitives = Array<string | number | boolean>;  
  
const array: ArrayOfPrimitives = ["string", 1, false];
```

UNKNOWN



- `unknown` est la contrepartie de `any` en termes de typage
- Tout est assignable à `unknown`, mais `unknown` n'est assignable à rien d'autre qu'à lui-même et à `any`

```
function f21<T>(pAny: any, pNever: never, pT: T) {  
    let x: unknown;  
    x = 123;  
    x = new Error();  
    x = x;  
    x = pAny;  
    x = pNever;  
    x = pT;  
}  
  
function f22(x: unknown) {  
    let v1: any = x;  
    let v2: unknown = x;  
    let v3: {} | null | undefined = x;  
    let v4: object = x; // Error  
    let v5: number = x; // Error  
    let v6: {} = x; // Error  
}
```



UNKNOWN NO OPERATION

- Aucune opération n'est possible sur `unknown` (hormis les tests d'égalités)

```
function f10(x: unknown) {  
    x == 5;  
    x !== 10;  
    x >= 0; // Error  
    x + 1; // Error  
    x * 2; // Error  
    -x; // Error  
    +x; // Error  
}  
  
function f11(x: unknown) {  
    x.foo; // Error  
    x[5]; // Error  
    x(); // Error  
    new x(); // Error  
}
```



TYPE ASSERTIONS

Vous pouvez dire à Typescript : "Fais moi confiance".

C'est apparenté à un casting mais il n'y pas "d'opérations" effectuées au runtime.

as

```
let someValue: unknown = "this is a string";
let strLength: number = (someValue as string).length;
```

chevron

```
let someValue: unknown = "this is a string";
let strLength: number = <string>someValue.length;
```



TYPE GUARDS

- Permet de déterminer le type d'une expression
- Dans le scope d'une instruction `if`, le type est changé pour correspondre à la clause définie par `typeof` ou `instanceOf`
- Utilisation de `typeof` ou `instanceOf` similaire à JavaScript

```
const stringOrStringArray: string | string[];  
  
if (typeof stringOrStringArray === "string") {  
  console.log(stringOrStringArray.toLowerCase()); //OK  
}  
  
console.log(stringOrStringArray.toLowerCase()); //KO
```



TYPE GUARDS

- Possibilité d'écrire une fonction renvoyant une vérification de type
- Type de retour : **[param] is [type]**

```
function isFish(pet: Fish | Bird): pet is Fish {  
    return (pet as Fish).swim !== undefined;  
}  
  
if (isFish(pet)) {  
    pet.swim();  
} else {  
    pet.fly();  
}  
  
console.log(pet.swim()); //KO, TS ne connaît pas son sous type ici
```



TYPE GUARDS INFERRÉ PAR IN

- `[literal] in [var]` permet de vérifier que la variable à bien la propriété literal
- On peut utiliser directement l'appel à l'élément

```
interface A {
  a: number;
}
interface B {
  b: string;
}

function foo(x: A | B) {
  if ("a" in x) {
    return x.a;
  }
  return x.b;
}
```

OMIT



- Utilisation du mot-clé **Omit**
- Permet de créer un type en enlevant certaines propriétés
- Aucun impact sur le code JavaScript généré
- Disponible depuis TypeScript 3.5

```
type Person = {  
    name: string;  
    age: number;  
    location: string;  
};  
  
type QuantumPerson = Omit<Person, "location">;  
  
// equivalent to  
type QuantumPerson = {  
    name: string;  
    age: number;  
};
```



PARAMETERS

Retourne un tuple à partir des types des arguments d'une fonction.

```
declare function f1(arg: { a: number; b: string }): void;  
  
type T = Parameters<typeof f1>;  
//   ^ = type T = [arg: {  
//     a: number;  
//     b: string;  
//   }]
```

RETURNTYPE



Construit un type qui équivaut au type de retour d'une fonction

```
declare function f1(): { a: number; b: string };

type T = ReturnType<typeof f1>;
//   ^ = type T = {
//     a: number;
//     b: string;
//   }
```



LOOKUP TYPES

- Opérateur **keyof** permettant d'indiquer qu'une API s'attend à avoir le nom d'une propriété comme paramètre

```
class Person {  
    constructor(public name: string) {}  
}  
  
function orderPeople(property: keyof Person) {}  
  
orderPeople("firstName"); //KO  
orderPeople("name"); //OK
```

- Utilisé par le langage pour définir les classes **Partial**, **Readonly**, **Record** et **Pick**

```
type Readonly<T> = {  
    readonly [P in keyof T]: T[P];  
};  
let person: Readonly<Person> = new Person("Carl");  
person.name = "Laurent"; //KO
```



SUPPORT DES FICHIERS JSX

- **JSX** : extension du langage JavaScript (similaire au **XML**)
- Utilisé pour définir une structure d'arbre avec attributs
- Nécessité de créer des fichiers TSX et d'activer l'option **jsx** (v1.6)
- Intégration TypeScript permettant de bénéficier du **type checking**

```
const myDivElement = <div className="foo" />;
```

- Deux modes disponibles : **preserve** et **react**
- Compilation du TSX au format JS ou JSX



TYPES NULL ET UNDEFINED

- En mode *strict null checking* (`--strictNullChecks`)
 - `null` et `undefined` peuvent être utilisés comme types explicites
 - `null` et `undefined` ne font plus partie du domaine de chaque type

```
// Compilation avec l'option --strictNullChecks
let x: number;
let y: number | undefined;
let z: number | null | undefined;
x = 1; // Ok
y = 1; // Ok
z = 1; // Ok
x = undefined; // Erreur
y = undefined; // Ok
z = undefined; // Ok
x = null; // Erreur
y = null; // Erreur
z = null; // Ok
```



INITIALISATION INDIRECTE

- `!` utilisé lorsque l'analyse de Typescript n'arrive pas à détecter que cet élément est initialisé

```
let x!: number;
initialize();

// erreur sans le !
console.log(x + x);

function initialize() {
  x = 10;
}
```



TOP LEVEL AWAIT

TypeScript offre depuis la version 3.8 la possibilité d'utiliser l'opérateur **await** directement dans un script.

```
// JavaScript
async function getData() {
  const response = await fetch("url");
  return response.json();
}
// L'opérateur await est contenu dans une fonction

await getData(); // Uncaught SyntaxError: await is only valid in async function

getData().then(); // Depuis un script, le seul moyen est d'utiliser un then
```

```
// TypeScript

await getData(); // valide :)
```



LITERAL TYPES

Un sous type. Ensemble fini d'élément d'un type.

- String

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
```

- Numeric

```
interface MapConfig {  
  lng: number;  
  lat: number;  
  tileSize: 8 | 16 | 32;  
}
```

- Boolean



TEMPLATE LITERAL TYPES

Depuis **4.1**, possibilité de créer des alias **type** avec des strings en template literal.

```
type VerticalAlignment = "top" | "middle" | "bottom";
type HorizontalAlignment = "left" | "center" | "right";

// Takes
//   | "top-left"    | "top-center"    | "top-right"
//   | "middle-left" | "middle-center" | "middle-right"
//   | "bottom-left" | "bottom-center" | "bottom-right"
declare function setAlignment(
  value: `${VerticalAlignment}-${HorizontalAlignment}`
): void;

setAlignment("top-left"); // works!
setAlignment("top-middel"); // error!
setAlignment("top-pot"); // error! but good doughnuts if you're ever in Seattle
```

