TYPESCRIPT

SOMMAIRE

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

LOGISTIQUE

- Horaires
- Déjeuner & pauses
- Autres questions ?

INTRODUCTION

PLAN

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

HISTORIQUE DU LANGAGE

- Le JavaScript est un langage de script orienté **prototype**
- Créé par Brendan Eich pour Sun/Netscape en 1995
- Soumis à l'ECMA pour standardisation

- L'ECMA est un organisme privé européen de standardisation
- Il n'est pas spécialisé dans l'IT (plutôt l'électronique)
- Le comité en charge de la spécification s'appelle le TC39
- Depuis 2015, le processus de normalisation a été réorganisé
- On peut notamment suivre tous les travaux sur GitHub https://github.com/tc39

ECMASCRIPT

- Les versions du standard **ECMAScript**

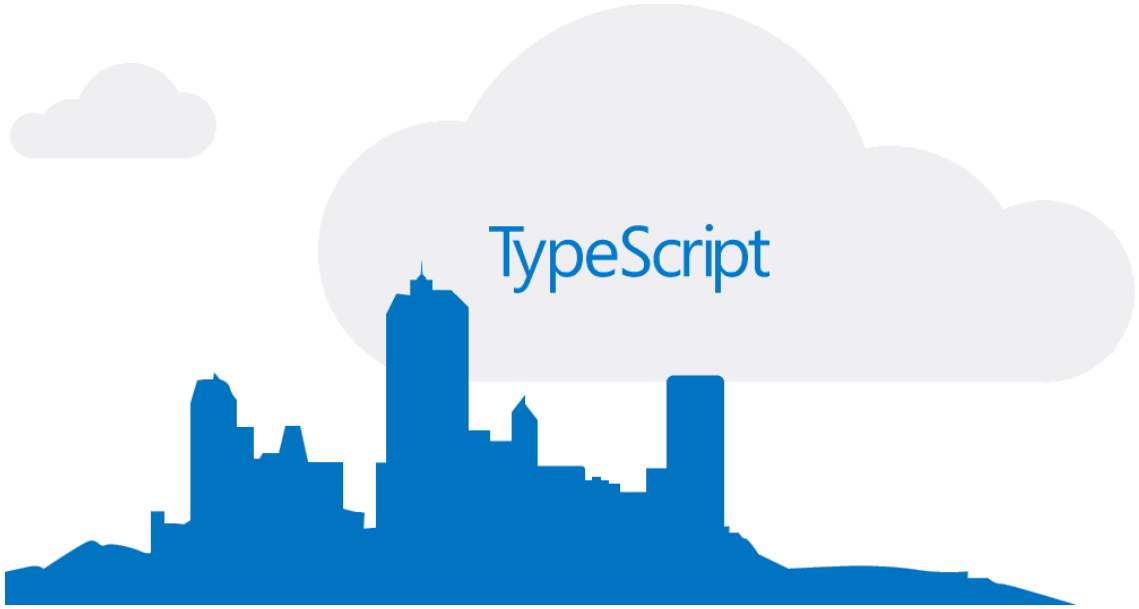
Ver	Date	Évolution
1	Juin 1997	Adoption de l'ECMAScript 1
2	Juin 1998	Réécriture de la norme, première version du JavaScript comme on le connaît
3	Décembre 1999	RegExp, Try/Catch, Erreur, ...
4	Abandonnée	
5	Décembre 2009	- Clarifie beaucoup d'ambiguïtés de la V3 - Version la plus répandue dans les navigateurs modernes (IE9+)
5.1	Juin 2011	Alignement norme ISO 16262
6	Juin 2015	Module, classe, destructuration, constante, arrow function, promise, generator, etc.

ECMASCRIPT SUITE

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

TYPESCRIPT



- Langage créé par Anders Hejlsberg en 2012
- Projet open-source maintenu par **Microsoft** (Version actuelle **3.9**)
- Propose une extension du JavaScript apportant le typage
- Influencé par **JavaScript**, **Java** et **C#**

TYPESCRIPT

- Phase de compilation nécessaire pour générer du **JavaScript**
- Ajout de nouvelles fonctionnalités au langage **JavaScript**
- Support d'ES3 / ES5 / ES2015 / ES2016 / ES2017 / ES2018 / ES2019 / ES2020 / ESNext
- Rétrocompatible: Tout programme **JavaScript** est un programme **TypeScript**
- Ajoute principalement les notions de typage et d'annotations

FONCTIONNALITÉS

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

INTÉGRATION AVEC ANGULAR

- L'équipe d'Angular voulait un langage typé pour la V2
- Ils se sont entendus avec les équipes de Microsoft pour adopter TypeScript
- Angular est aujourd'hui implémenté en TypeScript
- Angular incite les développeurs à également utiliser TypeScript

```
@Component({
  selector: 'my-app',
  template: '<h1>Hello {{ name }}</h1>'
})
class MyAppComponent {
  public name: string;

  constructor() {
    this.name = 'Alice';
  }
}
```

INTÉGRATION AVEC REACT

- Typescript s'intégre aussi avec React

```
class App extends React.Component<Object, Object>  {
    render() {
        return (
            <div className="app">
                <Board />
                <div>
                    <span className="description t1"> Player(X) </span>
                    <span className="description t2"> Computer(O) </span>
                </div>
                <RestartBtn />
                <GameStateBar />
            </div>
        )
    }
}

render(
    <App />, document.getElementById("content")
);
```

INTÉGRATION AVEC VUEJS

- Typescript s'intégre aussi avec VueJS

```
import Vue from 'vue'
import Component from 'vue-class-component'

@Component({
  template: '<button @click="onClick">Click!</button>'
})
export default class MyComponent extends Vue {
  message: string = 'Bonjour !'

  onClick (): void {
    window.alert(this.message)
  }
}
```

QUELQUES LIENS

- Site Officiel : http://www.typescriptlang.org/

  Documentation dans ***docs/handbook*** très complète

- Spécification : https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md

  Avant d'être un compilateur, c'est d'abord la spécification d'un langage

- 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 : http://blogs.msdn.com/b/typescript/

  Là ou Microsoft annonce les nouvelles versions

ECMASCRIPT 5

PLAN

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

TYPES DE DONNÉES

ECMAScript manipule différents types de données :

- Booléen : `true` ou `false`
- Nombre
  - Entier (base 10) : `0`, `5`, `-50`, `77`
  - Entier (base 8 ou 16) : `077` => **63**, `0xA` => **10**
  - Réel : `1000.566`, `.034`, `2.75e-2`
- Chaîne : `""`, `"Hello"`, `'World'`
- Tableau : `[]`, `[1, 2, 3]`, `[true, 'hello', 1, {}]`
- Map/Objet : `{}`, `{ clef: "valeur" }`,
  `{ "clef": {}, 'autre clef': [] }`

OBJETS STANDARDS

- Array, Boolean, Number, Date, String, RegExp, Math, Function
- Chaque objet standard dispose d'un certain nombre de méthodes
- Ces méthodes sont définies dans la spécification

```
var a = new Array(1, 3, 2); a.sort().reverse();

var b = new Boolean(); b.valueOf();

var n = new Number('5'); n.toLocaleString();

var d = new Date(); d.getTime();

var s = new String('Essai'); s.toUpperCase();

var r = new RegExp('^.{3}$'); /*ou*/ r = /^.{3}$/;
r.test('abcd'); r.exec('abc');

Math.PI; Math.random();

var add = new Function('a', 'b', 'return a + b');
```

STRUCTURATION DE CODE : IF

- Le `if`

```
if (/* test */) {
  /* liste d'instructions */
} else {
  /* autres instructions */
}
```

- Opérateur ternaire

```
var a = /* test */ ? /* true */ : /* false */;
```

- Attention, la valeur du test est **convertie** en booléen

STRUCTURATION DE CODE : SWITCH

- le `switch` / `case`

```
switch (/* valeur */) {
  case /* valeur 1 */:
    /* Liste d'instructions */
    /* Attention au fall through */
  case /* valeur 2 */:
    /* Liste d'instructions */
    break;
  case /* valeur x */:
    /* Liste d'instructions */
    break;
  default:
    /* Liste d'instructions */
}
```

- Il est possible de faire un `switch` sur les `String`

STRUCTURATION DE CODE : WHILE

- `while`

```
while (/* test */) {
  /* liste d'instructions */
}
```

- `do while`

```
do {
  /* liste d'instructions */
} while (/* test */);
```

- Toujours faire attention à la conversion de la valeur du test en booléen

STRUCTURATION DE CODE : FOR

- for

```
for (var i = 0; i < 5; i++) {
    console.log('The number is ', i );
}
```

- for in object

```
var object = { prop1: 1, prop2: 2 }
for (var prop in object) {
    console.log('Property', prop, '=', object[prop]);
}
```

- **Attention au for in array**

```
var values = [0, 1, 2, 3, 5, 8, 13, 21, 34, 55];
for (var i in values) {
    console.log('Index', i, ', Value', values[i]);
}
```

FONCTIONS : DÉFINITION

- Une fonction est un ensemble d'instructions
  - Une fonction ***peut*** être appelée avec des arguments
  - Une fonction retourne une valeur
- Définition d'une fonction
  - le mot-clé `function`
  - un nom optionnel
  - une liste de paramètres entre parenthèses (peut être vide)
  - un corps entre accolades (peut être vide)

```
function myFunction(parametre1, parametre2) {
  /* instructions */
}
```

FONCTIONS : DÉFINITION

- Nom de la fonction
  - Les noms de fonctions fonctionnent comme les noms de variables
  - Une fonction sans nom est dite anonyme
  - La propriété `name` d'une fonction donne son nom
- Les fonctions sont des objets
  - Il est possible de les affecter à des variables

```
var myVariable = function myFunction(parametre) {
  /* instructions */
};
```

FONCTIONS : EXEMPLES

Déclarer une fonction nommée crée également une référence du même nom

```
// Fonction nommée "namedFunction"
function namedFunction() { /**/ }

// Variable pointant sur une fonction existante
var variableExistingFunction = namedFunction;

// Variable pointant sur une nouvelle fonction nommée
var variableNewFunction = function newNamedFunction() { /**/ }

// Variable pointant sur une nouvelle fonction anonyme
var variableAnonymousFunction = function() { /**/ }

// Noms des fonctions
namedFunction.name //-> 'namedFunction'
variableExistingFunction.name //-> 'namedFunction'
variableNewFunction.name //-> 'newNamedFunction'
variableAnonymousFunction.name //-> 'variableAnonymousFunction'
(function(){}).name //-> ''
```

FONCTIONS : ARGUMENTS

- Pour appeler une fonction il faut disposer d'une référence
- L'appel de la fonction se fait avec les parenthèses
- Il est possible de lui passer autant d'arguments que souhaité

```
function display(arg1, arg2) {
  console.log(arg1, arg2);
}

display(); //-> undefined, undefined
display('a', 42) //-> a 42
display('a', 'b', 'c', 'd') //-> a b
```

```
// arguments est un mot-clé retournant tous les arguments
function display() { console.log(arguments); }

display('a', 'b', 'c', 'd') //-> ["a", "b", "c", "d"]
```

FONCTIONS : PROGRAMMATION FONCTIONNELLE

- Il est possible de passer une variable contenant une fonction en argument à une autre fonction
- Cela permet de passer un comportement en paramètre
- C'est le point de départ de la **programmation fonctionnelle**

```
function log() {
  console.log(arguments);
}

function forEach(array, action) {
  for(index in array){
    action(array[index]);
  }
}

forEach([1,2,3,5,8], log);
```

SCOPES

- La limite dans laquelle une référence est visible est un **scope**
- En JavaScript **les scopes sont délimités par les corps des fonctions**
    - Les blocs `if` / `for` / `while` ne créent pas de scope
    - **⚠ Différent de la plupart des langages**

```
function scope() {
  if (true) {
    var animal = 'monkey';
  }

  console.log(animal); //-> 'monkey'
}
```

CLOSURES

- Closure signifie fermeture
- Le principe est de capturer un scope
- Et le rendre disponible pour une autre fonction
- Ce principe s'applique à la déclaration d'une fonction
- Le scope courant est alors capturé

```
var capturedVariable = 'picture';

function capturingFunction() {
  //capturedVariable a été capturée et est visible ici par closure
  console.log(capturedVariable) //-> picture
}
```

CLOSURES : PIÈGES

- Les closures peuvent sembler très naturelles
- Elles sont très utilisées en JavaScript
- Mais attention aux pièges
- Il s'agit de la **référence** (pointeur) qui est capturée
- Les données ne sont pas copiées dans la nouvelle fonction

```
var alphabet = ['a', 'b', 'c', 'd', 'e'];

for(var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(alphabet[i]);
  }, 1000);
}

//-> Après 1s, on obtient : d d d
```

OBJETS

- JavaScript est un langage **orienté objet** à **prototype**
- http://fr.wikipedia.org/wiki/Programmation_orientée_prototype
- Il n'y a **pas de notion de classe**, seulement d'objet
- Par contre, chaque objet possède un prototype
- Un prototype est **aussi** un objet (possibilité de chaînage)
- Un objet accède de façon transparente à son prototype
- Les objets ont des **propriétés** de n'importe quel type
- Les **propriétés** qui ont comme valeur une fonction sont généralement appelées **méthodes**
- Le nom d'une propriété est appelé **clé**

OBJETS : MODIFICATION DES PROPRIÉTÉS

- Il est possible d'assigner des valeurs aux propriétés
- Mais aussi d'en ajouter et d'en supprimer

```
var animal = new Object();
animal.noise = function() {
  console.log('whines');
};

var fruit = {
  color: 'green'
};
fruit.name = 'kiwi';
delete fruit.color;

console.log(animal); //-> { noise: function }
console.log(fruit); //-> { name: 'kiwi' }
```

OBJETS : CONSTRUCTEURS

Pour créer un objet :

- On peut utiliser la syntaxe **littérale** `{ property: value }`
- Utiliser un **constructeur**
  - Une fonction comme constructeur avec le mot-clé `new`
  - Dans le constructeur, `this` représente alors l'objet créé

```
function Contact(firstname, lastname) {
  this.firstname = firstname;
  this.lastname = lastname;
  this.toString = function() {
    return this.firstname + ' ' +this.lastname;
  }
}
var contact = new Contact('Bruce', 'Wayne');
contact.toString(); //-> Bruce Wayne
```

OBJETS : PROTOTYPE

- Tout objet a forcément un prototype
  - Le prototype est lui-même un objet
  - `fn.prototype` permet de définir le prototype des objets construits avec cette fonction comme constructeur
  - Accéder au prototype d'un objet
    - ***Deprecated*** `objet.__proto__`
    - `Object.getPrototypeOf(objet)`
- Lorsqu'on accède à une propriété de l'objet
  - La propriété est recherchée dans l'objet
  - Puis dans son prototype et récursivement

OBJETS : PROTOTYPE

Exemple d'utilisation du prototype pour réaliser une sorte d'héritage

```
function Animal() {
  this.eat = function () {
    return this.food;
  }
}

function Monkey() {
  this.food = 'banana';
}

Monkey.prototype = new Animal();

var monkey = new Monkey();

monkey.eat(); //-> banana
```

ECMASCRIPT 2015+

PLAN

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

LET & CONST

- Remplacent avantageusement `var`
- 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
```

STRING INTERPOLLATION

- Ajout d'une troisième syntaxe pour les définir les string
- ' et " existent toujours et ne sont pas modifiés
- ` est ajouté pour les string mais 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 = parentèses 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`
- Manque du this pouvant 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

- Ajout du mot clé et de la notion de classe
- 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;
  }
}
```

MODULES

- ES5 ne propose pas de syntaxe officielle pour définir un module
- Principaux systèmes de modules :
  - Asynchronous Module Definition (AMD)
  - CommonJS (`require()`)
  - Universal Module Definition (UMD)
  - System.register

MODULE ES2015

- Chaque fichier JavaScript représente un module
- Chaque module
    - est isolé par défaut des autres modules
    - 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 toto from 'moduleA'
import {name} from 'moduleA'
// Ou import toto, {name} from 'moduleA'
console.log(toto, 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

- Ajout de l'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) {
  value += 1;
  console.log(value);
}

//-> 11 21 31
```

- Fonctionne sur une abstraction d'objets `iterable`
- Les objets Array, Map, Set, String répondent à cette abstraction
- (Avancé) Il est possible de rendre son objet itérable

PROMISES

- Le JavaScript s'appuie énormément sur un fonctionnement asynchrone
- Historiquement, ce fonctionnement asynchrone était implémenté avec des callback

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

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

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

PROMISES

- Les promesses améliorent nettement les codes asynchrones
- Elles apportent de nombreux avantages
    - Chaînage, on parlera plutôt d'encapsulation
    - Gestion des erreurs
    - Traitements parallèles

```
somethingWichTakesTime()
  .then(data => data.userId) // On peut retourner un type simple
  .then(userId => { // On peut retourner une autre promesse
    return somethingElseWichTakesTime(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
```

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
// ...
```

OUTILLAGE

PLAN

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

TOOLING

- Node.js
- npm
- TypeScript
- ESLint
- Jest

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

- npm est le gestionnaire de paquets de Node.js

- L'annuaire des paquets est disponible 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
```

- Global sert pour les commandes utilisées pour la console, **pas pour les dépendances**

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
- Le paquet contient un CLI qui permet d'utiliser le compilateur
- Pour l'installer `npm install --global typescript`
- Après l'installation, vous avez accès à la commande `tsc`
- La commande `tsc` propose de nombreuses fonctionnalités
  - `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 le CLI
- Le plus souvent, on utilisera un fichier de configuration `tsconfig.json`
- `tsc --init` pour initialiser un fichier `tsconfig.json`
- 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": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "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");
```

```
// file.js
var Greeter = (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return "Hello, " + this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("world");
```

ESLINT

- Linter (analyse statique du code) pour JavaScript et TypeScript
- Pour utiliser ESLint avec TypeScript, il faut installer plusieurs dépendances :

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

- Exemple de configuration `.eslintrc.js` avec TypeScript :

```
module.exports = {
  root: true,
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint'],
  extends: ['eslint:recommended', 'plugin:@typescript-
eslint/recommended'],
};
```

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, tape
  - Tests Runners : Karma, Jest, Testem, chutzpah
  - Librairies utilitaires : Chai, Sinon

JEST

- Framework de Tests : https://jestjs.io/
- Exécution des tests en parallèle, peu de configuration

  `npm install --global jest`
- Association de la commande `jest` à un script npm

```
{
  "scripts": {
    "test": "jest"
  }
}
```

JEST & TYPESCRIPT

- Possibilité d'écrire les tests en TypeScript
- Ajouter 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('will return true', () => {
        let flag = true;
        expect(flag).toBeTruthy();
    });

});
```

```
test('will return 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('will return true', () => {
        expect(flag).toBeTruthy();
    });
});
```

- Les tests peuvent être asynchrone
- 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
- ES5
- ES2015+
- Outillage
- *Types et inférence de types*
- Classes
- Modules
- Type Definitions
- Décorateurs
- Concepts Avancés

VARIABLES

- Mêmes types de variables que JavaScript : `var`, `let` et `const`
- Mêmes spécifications et comportement
- TypeScript ajoute le typage 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`
- Structure: `Array`, `Tuple`, `Enum`
- Spécifique: `unknown` (v3.0), `any`, `void`, `null`, `undefined`, `never`

```
const isDone: boolean = false;

const height: number = 6;

const phone_number: number = 555_734_2231;

const milion: number = 1_000_000;

const name: string = 'Carl';

const names: string[] = ['Carl', 'Laurent'];

const notSure: any = 4;

const alsoNotSure = 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
- La syntaxe est `(arguments) => typeRetour`

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

FONCTIONS 3.0

```
type Function<A extends any[], R> = (...args: A) => R;
```

- ici on a un "safe typings" des paramètres

PARAMÈTRES OPTIONNELS

- Un paramètre peut être optionnel
  - Utilisation du caractère `?` avant le type
  - L'ordre de définition 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(); //-> compilation 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

```
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', 'aya', 4); //-> compilation error

company('zenika', 'aya', 'unkipple'); //-> 'zenika' ['aya', 'unkipple']

company(); //-> 'zenika' []
```

SURCHARGE DE FONCTIONS

- Une fonction peut retourner un type différent en fonction des paramètres
- Permet d'indiquer quel est le type retourné en fonction des paramètres
- Nécessite tout de même d'implémenter la version générique avec les `if`

```
function log(param: string): number;
function log(param: number): string;
function log(param: string|number): string|number {
  if (typeof param === "number") {
    return "string";
  } else {
    return 42;
  }
}
```

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 du **tuple**
- JavaScript ne propose pas de syntaxe pour la gestion des **tuples**
- 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 3.0

- Possibilité de rendre optionel un élément dans un tuple
- 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);

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

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 une valeur numérique
- La valeur numérique commence par défaut à `0`
- Possibilité de surcharger les valeurs numériques

```
enum Music { Rock = 2, Jazz = 4, Blues = 8 };

const c: Music = Music.Jazz;
```

- Récupération de la chaîne de caractères associée à la valeur numérique

```
const style: string = Music[Music.Jazz]; //Jazz
```

ENUM INLINE

- Depuis TypeScript 1.4, ajout de la syntaxe `const enum`
- ***Inline*** les valeurs de **l'enum** lors de la compilation
- Améliore la performance, la taille du code source
- Perd la possibilité de faire référence dynamiquement à la valeur d'un enum

ENUM

- Version TypeScript

```
enum Music { Rock, Jazz, Blues };
const enum MusicInline { Rock, Jazz, Blues };

const a: Music = Music.Jazz;
const b: MusicInline = MusicInline.Rock;
const c: MusicInline = MusicInline['Bl' + 'ues']
```

- Version JavaScript

```
var Music;
(function (Music) {
    Music[Music["Rock"] = 0] = "Rock";
    Music[Music["Jazz"] = 1] = "Jazz";
    Music[Music["Blues"] = 2] = "Blues";
})(Music || (Music = {}));

var a = Music.Jazz;
var b = 0 /* Rock */;
// compilation error
// (A const enum member can only be accessed using a string literal.)
```

INFÉRENCE DE TYPES

- TypeScript va tenter de définir le type des variables non typées explicitement
- S'il ne trouve pas, il utilisera any
- 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
}
```

INFÉRENCE DE TYPES

- Cela lui permet de lever des erreurs à la compilation
- Même lorsque le type des variables n'avaient pas été fixé

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

const a = 42; // inference de type number

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

TYPE OPTIONNEL CHAINABLE (V3.7)

- Il est possible de chainer la vérification des types optionnels grâce à l'opérateur `?.`
- Permet d'utiliser le système de "short-circuiting"

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

// Same as
let x = (foo === null || foo === undefined) ?
    undefined :
    foo.bar.baz();
```

```
let result = foo?.bar / someComputation()

// Same as
let temp = (foo === null || foo === undefined) ?
    undefined :
    foo.bar;

let result = temp / someComputation();
```

TYPE OPTIONNEL CHAINABLE : FALSY VALUES (V3.7)

- La nouvelle écriture

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

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

TYPE OPTIONNEL CHAINABLE : OPTIONAL CALL (V3.7)

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

```
function makeRequest(url: string, log?: (msg: string) => void) {
    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 at ${new Date().toISOString()}`);

    return result;
}
```

NULLISH COALESCING OPERATOR (V3.7)

- Utilisable grace à l'opérateur ??
- Si première opérande n'est pas null retourne cette derniere sinon retourne la seconde

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

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

- A preférer à ||

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

    // ...
}
```

Lab 2

CLASSES

PLAN

- Introduction
- ES5
- ES2015+
- Outillage
- 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 ES2105 et y ajoute des possibilités
- Une fois définie une classe sert de constructeur et de type !

```
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 (pas de surcharge)
  - Le constructeur 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

- TypeScript ajoute le concept de propriété qu'il n'y a pas en JavaScript
- Elles sont définies également dans le corps de la classe
- Bien penser à ne **pas** mettre `let` ou `const` mais mettre un `;`
- Se comporte comme un `let`, la référence est modifiable
- Possibilité d'initialiser la propriété, sinon elle est `undefined`

```
class Person {
  firstName: string;
  lastName: string = "Dupont";
  age: number;
  hobbies: string[];
}

const person: Person = new Person();
console.log(person.lastName); //-> 'Dupont'
```

SCOPES

- Méthodes et propriétés ont forcément un scope
- Quatre scopes disponibles : `public`, `private`, `protected` et `readonly`
- 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'.
```

STATIC

- Possibilité de définir des propriétés et des méthodes statiques (`static`)
- Ne pas mettre de scope, c'est 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}`);
  }
}

const person: Person = new Person();
Person.sayHello(); //-> 'Hello World'
console.log(Person.message); // -> 'World'
person.sayHello(); // compilation error
// Property 'sayHello' does not exist on type 'Person'
```

PROPRIÉTÉS INITIALISÉES DANS LE CONSTRUCTEUR

- TypeScript emprunte au C# un raccourci pour initialiser les propriétés
- Raccourci permettant de déclarer une propriété et l'initialiser 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
- Utiliser les mots clés **get** et **set**
- S'appuie sur une fonctionnalité ES5, ne fonctionne pas avec la target ES3
- Il vous faudra gérer la ***vraie*** propriété dans l'objet par vous-même

```
class Person {
   private _secret: string; // Le préfixe "_" est une convention

   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
- Possibilité d'appeler l'implémentation 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 - DUCK TYPING

- TypeScript propose une fonctionnalité appelé le Duck Typing
- Permet de facilement typer un objet qui ne l'est pas encore

> *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*

```
interface Duck {
  color: string
  fly(height: number): void
  quack: (message: string) => void
}

const duck: Duck = {
  color: 'yellow',
  fly: height => {},
  quack: message => {}
}
```

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

- Cas d'utilisation le plus connu des interfaces
- 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
```

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 les `getter` / `setter` de la propriété héritée

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

class Bar extends Foo {
  get bar(): string { ... }
  set bar(value: string) { ... }
  // 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) {
    ...
  }
  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() { } }

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
- ES5
- ES2015+
- Outillage
- 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`
- Les modules ne sont pas encore supportés par les navigateurs
- TypeScript peut compiler vers plusieurs mécanismes de chargement
- Support des mécanismes : ***CommonJS***, ***AMD***, ***UMD***, ***SystemJS***, ***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 librairies Node.js historiques

```
import * as Utils from './utils';

let logger = Utils.createLogger();
```

MODULES - COMPILATION

- ***AMD*** principalement utilisé par ***RequireJS***
- Chargement asynchrone des modules

```
define('module1', 'module2', function(module1, module2) { ... });
```

- ***CommonJS*** principalement utilisé par ***Node.js***
- Chargement synchrone des modules

```
const module1 = require('module1');
const module1 = require('module2');
/* ... */
```

- ***ES2015*** principalement utilisé pour ***Webpack*** (2.2+)
- TypeScript ne transforme pas le code des `import` et `export`
- Webpack réalise alors un bundle
  (fichier unique qui recompose les liens entre les modules)

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'
```

NAMESPACES

- TypeScript possède une notation pour définir des modules internes avec `namespace`
- La clé `export` permet de publier des propriétés

```
namespace Utils {
  export class Logger { [ ... ] }

  export namespace String {
    export class Formatter { [ ... ] }

    function privateFormatFunction(){ [ ... ] }
  }
}

new Utils.Logger();
new Utils.String.Formatter();
```

Attention : cette fonctionnalité a été dépréciée par ESLint. La bonne pratique consiste à ne plus utiliser de `namespace`, mais directement les imports ES2015. (règle : `no-namespace`)

Lab 4

TYPE DEFINITIONS

PLAN

- Introduction
- ES5
- ES2015+
- Outillage
- 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 un dépendance de votre projet
- Les outils `tsd` et `typings` sont considérés obsolètes
- 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 définitions 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"]
  }
}
```

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 var angular: angular.IAngularStatic;
export as namespace angular;

declare namespace angular {
  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): void;
}
```

Lab 5

DÉCORATEURS

PLAN

- Introduction
- ES5
- ES2015+
- Outillage
- 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)
- 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, descriptor:
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
- Doit rendre le constructeur ou un nouveau

```
function Log<T extends {new(...args:any[]):{}}>(constructor:T) {
    return class extends constructor {
        newProperty = "new property";
        name = "overridden";

      constructor(...args:any[]) {
        super()
          console.log(`New person ${this.name} created with
${this.newProperty}`);
        }
    }
}

@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 Contructeur 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'est pas utilisée
- Son usage est assez limité puisqu'il ne permet pas de récupérer simplement 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(`Person with default message ${target[propertyKey]}`);
}

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

const JackSparrow = new Person();
// => Person with default message 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';
    }
}
```

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

GÉNÉRATEURS

- Fonction qui peut être stoppée avec le mot clef `yield` et reprise par la suite.
- Un générateur est déclaré à l'aide d'une `*` après `function`
- Pour récupérer les valeurs intermédiaires, il faut utiliser la fonction `next()` sur le résultat du générateur.

```
function* idMaker(){
  let index = 0;
  while (index < 3) {
    yield index++;
  }
}

const gen = idMaker();

console.log(gen.next()); // { value: 0, done: false }
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: undefined, done: true }
// ...
```

ALIAS

- Création 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

```
type myNumber = number;

const num: myNumber = 2;
```

OMIT

- Utilisation du mot-clé `Omit`
- Permet de créer un type en enlevant certaines propriétées
- 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;
};
```

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];
```

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 isString(x: any): x is string {
  return x instanceof string;
}

var stringOrStringArray: string|string[];

if (isString(stringOrStringArray)) {
  console.log(stringOrStringArray.toLowerCase()); //OK
}

console.log(stringOrStringArray.toLowerCase()); //KO
```

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;
}
```

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
```

- Utiliser 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
```

UNKNOWN (V3.0)

- unknown est la contrepartie de any en terme 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: object = x;  // Error
    let v4: number = x;  // Error
    let v6: {} = x;  // Error
    let v7: {} | null | undefined = x;  // Error
}
```

UNKNOWN NO OPERATION (V3.0)

- 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
}
```

UNKNOWN UNION ET INTERSECTION (V3.0)

```
// In an intersection everything absorbs unknown
type T00 = unknown & null;  // null
type T01 = unknown & undefined;  // undefined
type T02 = unknown & null & undefined;  // null & undefined (which becomes
never)
type T03 = unknown & string;  // string
type T04 = unknown & string[];  // string[]
type T05 = unknown & unknown;  // unknown
type T06 = unknown & any;  // any

// In a union an unknown absorbs everything
type T10 = unknown | null;  // unknown
type T11 = unknown | undefined;  // unknown
type T12 = unknown | null | undefined;  // unknown
type T13 = unknown | string;  // unknown
type T14 = unknown | string[];  // unknown
type T15 = unknown | unknown;  // unknown
type T16 = unknown | any;  // any

// Type variable and unknown in union and intersection
type T20<T> = T & {};  // T & {}
type T21<T> = T | {};  // T | {}
```

INITIALISATION INDIRECTE

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

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

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

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