

TypeScript



Sommaire

- Rappels
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales



Logistique



- Horaires
- Déjeuner & pauses
- Autres questions ?



Rappels

Plan



- *Rappels*
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales



Historique du langage

Le JavaScript est un langage de script orienté **prototype**

Date	Éditeur	Évènement
Déc. 1995	Sun/Netscape	Annonce de JavaScript (anciennement LiveScript)
Mars 1996	Netscape	JavaScript dans Netscape 2.0
Août 1996	Microsoft	Sortie de JScript dans Internet Explorer 3.0
Nov. 1996	Netscape	Standardisation de JavaScript à l'ECMA
Juin 1997	ECMA	Adoption de l'ECMAScript
1998	Adobe	ActionScript



- L'**ECMA** est un organisme privé européen de standardisation
- Il n'est pas spécialisé dans l'IT (plutôt l'électronique)

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, ... Version la plus répandue
4	Abandonnée	
5	Décembre 2009	Clarifie beaucoup d'ambiguïtés de la V3 Version actuellement dans Node.js
6	Juin 2015	Classes, modules, itérateurs (boucles for-of, générateurs), fonctions fléchées, tableaux binaires, collections à références fortes et faibles, proxies
7	Juin 2016	Augmentation de la terminologie du langage : Exponentiation (**) Array.prototype.includes

Instructions et points-virgules



- La structure des instructions est héritée du C
- Les instructions sont délimitées par des points-virgules ;
- Les points virgules sont ***optionnels***
 - Il y a un système d'insertion automatique
 - <http://bclary.com/log/2004/11/07/#a-7.9>
 - Schématiquement, s'il n'y a pas d'ambiguïté et un retour chariot, un point-virgule est ajouté automatiquement
- Peut amener à des erreurs si on comprend mal ce principe
- La plupart des règles de bonne conduite demandent de toujours mettre le point-virgule



Types de données

ECMAScript manipule différents types de données :

- Booléen : `true` ou `false`
- Nombre
 - Entier décimal : `0`, `5`, `-50`, `77`
 - Entier octale ou hexa. : `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

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

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

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

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

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

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

Math.PI; Math.random();

const add = function fnName(a, b) {return a + b};
```



Structuration de code : if

- Le **if**

```
if /* test */ {  
/* liste d'instructions */  
} else {  
/* autre instructions */  
}
```

- Opérateur ternaire

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

- **for in object**

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

- **Attention au for in array**

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

- **for of array (ES2015)**

```
for (let val of values) { console.log('Value', val); }
```



Les variables



- Il n'y a pas de contrainte sur la longueur de l'identifiant d'une variable
- Un identifiant JavaScript doit commencer par une lettre, **\$** ou **_**
- Les caractères qui suivent peuvent être des chiffres
- De nouveaux mots clés sont apparus pour déclarer des variables : **let** et **const**
- Il ne doit pas correspondre à un mot clef réservé

```
const myVariable1; // -> SyntaxError: Missing initializer in const declaration
const myVariable2 = 'maValeur';
const myVariable3, myVariable4 = 'maValeur'; // -> SyntaxError: Missing
initializer in const declaration
```



Les variables

- Les variables ont un typage dynamique

```
let a = 5;  
a = 'essai';  
a = {clef: 'valeur' };
```

- Utiliser une variable qui n'a pas été définie avec **var**
 - En lecture, on obtient **undefined**
 - En écriture, on définit une **variable globale**
 - L'utilisation des variables globale est à éviter

```
console.log(b); //-> undefined  
b = 5; // création de la variable globale b  
console.log(b); //-> 5
```





Les 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 */  
}
```

Les 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

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

Les fonctions : exemples



- Déclarer une fonction nommée crée également une référence du même nom

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

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

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

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

// Noms des fonctions
namedFunction.name //-> 'namedFunction'
variableExistingFunction.name //-> 'namedFunction'
variableNewFunction.name //-> 'newNamedFunction'
variableAnonymousFunction.name //-> ''
```

Les 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"]
```

Les 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 forEach(array, action) {  
    for(index in array){  
        action(array[index]);  
    }  
}  
forEach([1,2,3,5,8], console.log);
```

- **⚠ La fonction est passée sans parenthèse !**
- Il s'agit de la référence sur la fonction qui est passée



Visibilité des variables et des fonctions

- La limite dans laquelle une référence est visible est un **scope**
- 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**
- **let** et **const** en ES2015 reprend le comportement classique avec une visibilité par block {}



Visibilité des variables et des fonctions

- Visibilité des symboles au sein d'un scope
 - **Fonctions nommées** : utilisable partout dans le scope même avant la déclaration (**forward-reference**)
 - **Variables locale** : le symbole existe partout dans le scope sa valeur est `undefined` jusqu'à l'initialisation
 - **Variables globale** : le symbole devient une propriété du scope par défaut (window dans un navigateur)



Visibilité des variables et des fonctions

- Etude de la visibilité des variables JavaScript

```
function scope() {  
    // Forward reference  
    const forwardReferenceCall = namedFunction();  
    function namedFunction() { return 42; }  
    const classicReferenceCall = namedFunction();  
  
    // Création des variables  
    console.log('before declaration', fruit); //-> undefined  
    const fruit = 'kiwi';  
    console.log('after declaration', fruit); //-> kiwi  
  
    // Manipulation des scopes  
    if (true) { var animal = 'monkey'; }  
    console.log(animal);  
}
```





L'isolation et les fonctions anonymes

- Les variables globales posent de nombreux problèmes
- Surtout dans les navigateurs où il n'y a pas de modularisation
- Solution : **la fonction anonyme auto-appelante**

```
var globalVariable = 'monkey';

(function(globalVariableRedefined) { // maVariableGlobale devient locale
    var localVariable = 'banana';
}(globalVariable)); // maVariableGlobale est passé en paramètre
```

Les 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

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

Les 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

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

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

//-> Après 1s, on obtient : d d d
```



Les 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**

Les objets : modification des propriétés



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

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

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

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

Les objets : constructeurs



- Pour définir un objet
 - Utiliser la syntaxe littérale `{ property: value }`
 - Une fonction comme constructeur avec le mot-clé `new`
 - Dans le constructeur, `this` représente alors l'objet
 - Toute fonction peut être utilisée comme constructeur

```
function Contact() {  
    this.firstname = '';  
    this.lastname = '';  
    this.toString = function() {  
        return this.firstname + ' ' +this.lastname;  
    }  
}  
const contact = new Contact();
```



Les objets : constructeurs



- Il est possible de simuler des variables privées
- C'est une astuce qui s'appuie sur les closures
- Les variables privées sont définies dans le scope du constructeur

```
function Contact() {  
    const firstname = '';  
    const lastname = '';  
  
    const getFullName = function() {  
        return firstname + ' ' + lastname;  
    }  
  
    this.toString = function (){  
        return getFullName();  
    }  
}
```



Les 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



Les objets : prototype

- L'utilisation des prototypes permet de reproduire
 - Une notion d'**héritage** : en faisant une chaîne de prototype
 - Une notion de **classe** : le prototype contient les méthodes

Les objets : prototype



- Notion d'**héritage**

```
function Animal() {  
    this.eat = function () {  
        return this.food;  
    }  
}  
  
function Monkey() {  
    Animal.apply(this, arguments);  
    this.food = 'banana';  
}  
Monkey.prototype = Object.create(Animal.prototype);  
  
var monkey = new Monkey();  
console.log(monkey.eat()); // -> banana
```

Les objets : prototype



- Une notion de **classe**

```
const Monkey = (function () {  
  
    const secretFavoriteActress;  
  
    function Monkey(actress) {  
        secretFavoriteActress = actress;  
    }  
  
    Monkey.prototype.showFavoriteActress = function () {  
        console.log(secretFavoriteActress);  
    };  
  
    return Monkey;  
}());
```



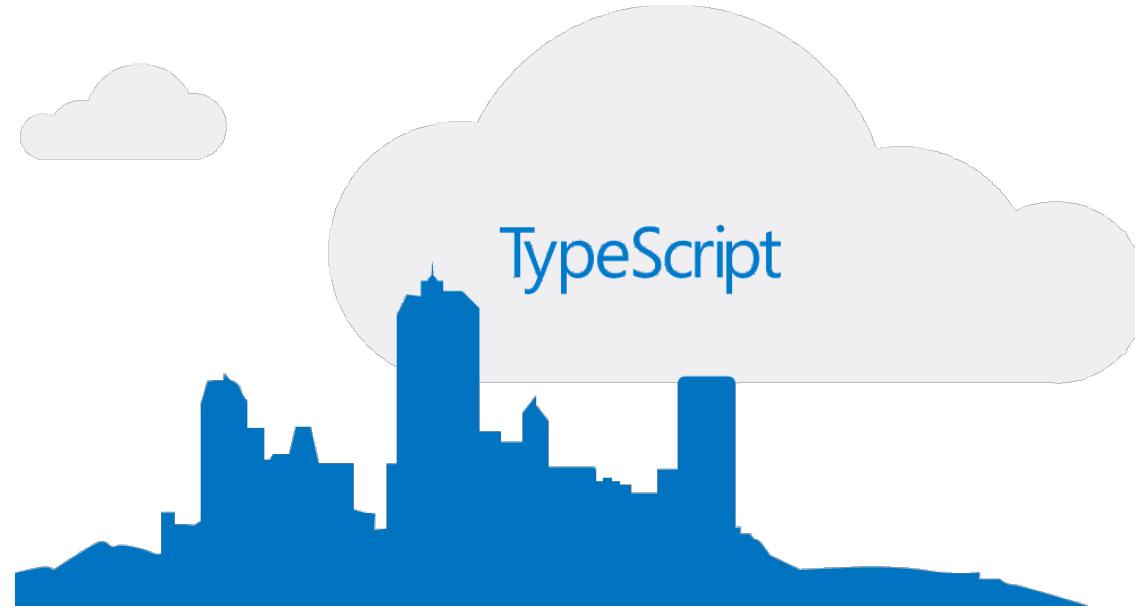
Introduction

Plan



- Rappels
- *Introduction*
- Outillage
- Tests
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales

Introduction

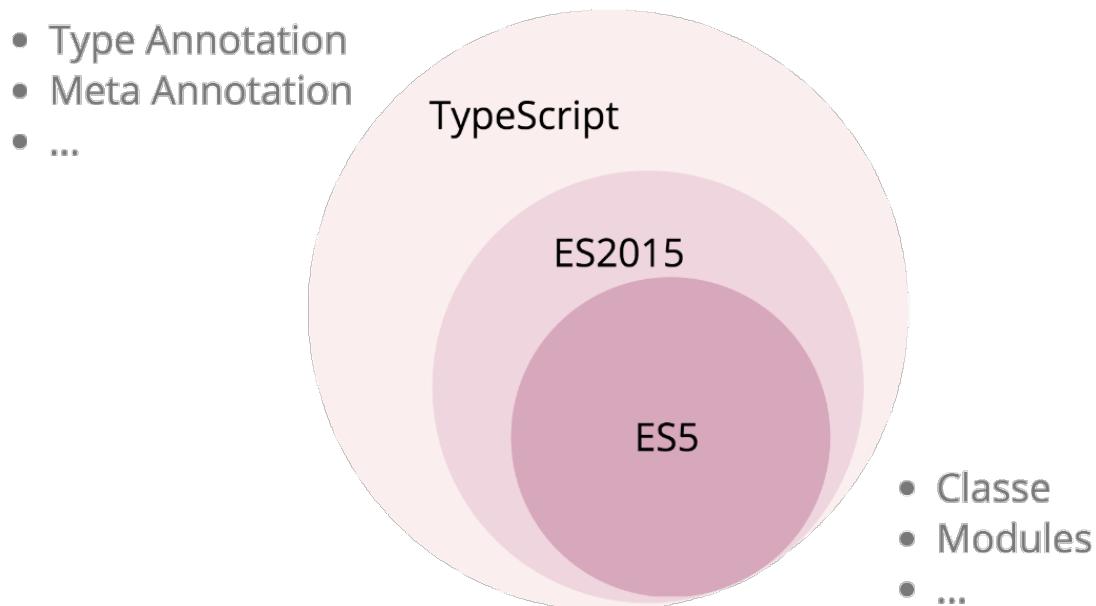


- Langage créé par **Anders Hejlsberg** en 2012
- Projet open-source maintenu par **Microsoft** (Version actuelle **2.1**)
- Influencé par **JavaScript**, **Java** et **C#**
- Alternatives : CoffeeScript, Dart, Haxe ou Flow

Introduction



- Phase de compilation nécessaire pour générer du **JavaScript**
- Ajout de nouvelles fonctionnalités au langage **JavaScript**
- Support d'ES3 / ES5 / ES2015 / ES2016
- Certaines fonctionnalités n'ont aucun impact sur le JavaScript généré
- Tout programme **JavaScript** est un programme **TypeScript**



Introduction - Fonctionnalités



- Typage
- Génériques
- Classes / Interfaces / Héritage
- Développement modulaire
- Les fichiers de définitions
- Mixins
- Décorateurs
- Fonctionnalités **ES2015**



Introduction - Intégration avec Angular

- Langage choisi pour implémenter la version 2 du framework Angular

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

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



Introduction - Quelques Liens

- Site Officiel: <http://www.typescriptlang.org/>
- Spécification:
<https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>
- Testez TypeScript en ligne: <http://www.typescriptlang.org/play/>
- Repository Github: <https://github.com/Microsoft/TypeScript>
- Blog: <http://blogs.msdn.com/b/typescript/>



Outillage

Plan



- Rappels
- Introduction
- *Outillage*
- Tests
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales



Tooling

- Node
- NPM
- CLI TypeScript
- TSLint
- Gulp

Node



- Node.js est une plateforme basée sur un moteur JavaScript
- La très grande majorité des outils de développement Web est actuellement réalisée avec Node



- Fonctionne avec le moteur JavaScript V8 de Google Chrome
- Étend le JavaScript avec une API système lui permettant par exemple d'intervenir sur les fichiers



- NPM est un gestionnaire de paquets pour Node
- L'annuaire des paquets est disponible sur <https://www.npmjs.com/>
- La commande `npm` est installée en même temps que Node
- Pour installer un paquet :
`npm install 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 --save my-package`
 - devDependencies :
`npm install --save-dev my-package`

CLI TypeScript



- Le CLI TypeScript permet de compiler le code en JavaScript
- Présent dans les dépôts NPM
- Pour l'installer : `npm install -g typescript`
- Pour créer le fichier de configuration (`tsconfig.json`) : `tsc --init`
- Compiler un fichier : `tsc name_file.ts`
- Compiler avec un sourceMap : `tsc --sourceMap app.ts`
- Pour utiliser le fichier de configuration :
 - Si à la racine : `tsc`
 - Si chemin spécifique : `tsc --config path/tsconfig.json`
- Pour plus d'informations : `tsc -h`



tsconfig.json

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

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "removeComments": true,  
    "outDir": "./build",  
    "sourceMap": true  
  },  
  "files": [  
    "file1.ts",  
    "file2.ts"  
  ],  
  "excludes": [  
    "node_modules",  
    "file3.ts"  
  ]  
}
```

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



TypeScript Linter

- Sert à valider la syntaxe du code
- Équivalent à ESLint pour le JavaScript
- Utilise un fichier de configuration au format JSON :

```
{  
  "rules": {  
    "no-eval": true,  
    "semicolon": true  
  }  
}
```

- Commande pour valider un fichier :

```
tslint -c tslint.json filesToTest.ts
```

Gulp



- Outil open-source permettant d'automatiser des tâches
- Basé sur un système de ***stream***
- Utilisation limitée de fichiers
- De nombreux plugins disponibles
<https://www.npmjs.com/browse/keyword/gulpplugin>
- Tâches définies dans un fichier ***gulpfile.js***

Gulp



- Exemple de tâche **Gulp** :

```
//Cette tâche peut être appelée via la ligne de commande
gulp.task('hello', function() {
  console.log('hello world !')
});
```

Gulp - Streams



- Création d'un **stream** d'objets représentant les fichiers à manipuler
- Définir le répertoire source : **gulp.src**
- Définir le répertoire de destination : **gulp.dest**
- L'utilisation de la méthode **pipe** permet de chaîner les différentes transformations

Gulp - Streams - Exemple



- Compilation et minification des fichiers TypeScript :

```
const gulp = require('gulp'),
typescript = require('gulp-typescript'),
uglify = require('gulp-uglify'),
tsProject = typescript.createProject('tsconfig.json')

gulp.task('typescript', function() {
  gulp.src('src/**/*.{ts}')
    .pipe(tsProject())
    .pipe(uglify())
    .pipe(gulp.dest('dist/'));
});
```

Gulp



- Une action peut agréger d'autres actions
- Ces actions seront exécutées de façon asynchrone

```
gulp.task('build', ['css', 'ts', 'images']);
```

- Définir qu'une action doit être exécutée avant une autre :

```
gulp.task('typescript', ['copy'], function () {  
});
```

- La commande **gulp** correspond à la tâche **default**

```
gulp.task('default', ['lint', 'typescript', 'deploy']);
```







Tests

Plan



- Rappels
- Introduction
- Outilage
- *Tests*
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales

Tests



- Possibilité d'écrire des tests avec différentes librairies/frameworks
 - Frameworks de Tests : **Jasmine**, **Mocha**, **qUnit**, **AVA**
 - Librairies d'assertions : **Chai**
 - Librairies de mocks : **Sinon**
 - Tests Runners : **Karma**, **Testem**, **chutzpah**, **AVA**, **Jest**, **Jasmine**



Tests - AVA





Tests - AVA

- Framework de Tests : <https://github.com/avajs/ava>
- Exécution des tests en parallèle
- Plus rapide par rapport à d'autres solutions
- Fournit un outil en ligne de commande
- Nécessite peu de configuration

```
npm install -g ava  
ava --init
```

- Association de la commande **ava** à un script NPM

```
{  
  "scripts": { "test": "ava" }  
}
```



Tests - AVA

- Possibilité d'écrire les tests en TypeScript
 - Ajout de la librairie **es2015** dans la configuration du compilateur

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "target": "es5",  
    "noImplicitAny": false,  
    "sourceMap": false,  
    "lib": [  
      "es2015"  
    ]  
  }  
}
```

- Compilation avant l'exécution des tests

```
{  
  "scripts": { "test": "tsc && ava" }  
}
```



Tests - Structure d'un test AVA

- Méthodes **test** pour décrire la suite de tests
- Système de **assertions** : **truthy**, **falsy**, **is**, **deepEqual**, ...
- Possibilité d'utiliser les librairies **Chai** ou **SinonJS**

```
import test from 'ava';

test('will return true', t => {
  let flag = true;
  t.true(flag);
});

test(t => {
  tthrows(functionThrowingAnError());
});
```



Tests - Structure d'un test AVA

- Méthodes `before`, `after`, `beforeEach`, `afterEach`
- Exécution d'une fonction avant ou après chaque test
- La fonction peut être asynchrone : utilisation des `Promise`, `Observable` ou `async/await`

```
import test from 'ava';

let flag;

test.beforeEach(t => {
  flag = true;
});

test('will return true', t => {
  t.true(flag);
});
```





Tests - Structure d'un test AVA

- Possibilité de définir des **Spies** grâce au module **Sinon.JS**
- Vérifier l'exécution de la méthode **espionnée**
 - `called`, `calledWith`
 - `neverCalledWith`, `calledOnce`, `calledTwice...`
 - `callCount`

```
import test from 'ava';
import * as sinon from 'sinon';

const spy = sinon.spy(console, 'log');

test('will return true', t => {
  console.log('Hello Zenika');
  t.true(spy.called);
  t.is(spy.calledWith('Hello Zenika'));
});
```





Types et Inférence de types

Plan



- Rappels
- Introduction
- Outilage
- Tests
- *Types et Inférence de types*
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales



Types primitifs

- Pour déclarer une variable :

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

- boolean : const isDone: boolean = false;
- number : const height: number = 6;
- string : const name: string = 'Carl';
- array : const names: string[] = ['Carl', 'Laurent'];
- any : const notSure: any = 4;



Types primitifs

- Accès aux prototypes des objets primitifs du langage JavaScript

```
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



- Comme en JavaScript, possibilité de créer des fonctions nommées ou anonymes

```
//Fonction nommée
function namedFunction():void { }

//Fonction anonyme
const variableAnonymousFunction = function(): void {
}
```

- Peut retourner une valeur grâce au mot-clé **return**
- Accès aux variables définies en dehors du scope de la fonction

```
const externalScope:number = 10;

function add(localArg: number): number {
    return localArg + externalScope;
}
```



Fonctions - Paramètres (Optionnels)

- Une fonction peut prendre des paramètres

```
function fn(name: string, forename: string) { }
```

- Un paramètre peut être optionnel
 - utilisation du caractère ?
 - ordre de définition très important
 - aucune implication dans le code JavaScript généré
 - si pas défini, le paramètre aura la valeur **undefined**

```
function fn(name: string, surname?: string) { }
```



Fonctions - Paramètres par défaut

- Possibilité de définir une valeur par défaut pour chaque paramètre
 - directement dans la signature de la méthode
 - utilisation du signe =
 - ajoutera une condition `if` dans le javascript généré

```
function fn(name: string = 'carl') {  
}
```



Fonctions - Paramètres par défaut

- Version TypeScript

```
function fn(name: string = 'carl') {  
}
```

- Version JavaScript

```
function fn(name) {  
    if (name === void 0) { name = 'carl'; }  
}
```

- **void 0** retourne l'objet **undefined**
 - Evite les problèmes de redéfinition de la variable globale **undefined**



Fonctions - Rest Parameters

- 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
 - se base sur l'objet JavaScript **arguments**

```
function company(nom: string = 'zenika', ... tribus: string[]) {  
}  
  
company('zenika', 'web', 'java'); // nom => 'zenika', tribus => ['web', 'java']  
  
company('zenika', 'web'); // nom => 'zenika', tribus => ['web']
```



Fonctions - Rest Parameters

- Version TypeScript

```
function company(name: string, ...tribes: string[]) {  
}
```

- Version JavaScript

```
function company(name) {  
    var tribes = [];  
    for (var _i = 1; _i < arguments.length; _i++) {  
        tribes[_i - 1] = arguments[_i];  
    }  
}
```



Fonctions - Surcharge

- Une fonction peut retourner un type différent en fonction des paramètres
- Permet d'indiquer au compilateur TypeScript quel est le type retourné en fonction des paramètres passés

```
function fn(param: string): number;
function fn(param: number): string;
function fn(param: any): any {
  if (typeof param === "number") {

  } else {
    }

}
}
```



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

- JavaScript ne propose pas de syntaxe pour la gestion des **tuples**
 - se base sur des tableaux
- TypeScript propose une syntaxe pour créer des tuples

```
const tuple: [string, number] = ['Zenika', 10];
```

- Possibilité d'utiliser les tuples en combinaison du système d'extraction de données

```
const tuple: [string, number] = ['Zenika', 10];
```

```
const [name, age] = tuple
```



Tuples

- Version TypeScript

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

- Version JavaScript

```
var tuple;  
tuple = ['Zenika', 10];  
var name = tuple[0], age = tuple[1];
```



Type Enum

- Possibilité de définir un type pour expliciter un ensemble de données numériques

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

- 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[4]; //Jazz
```





Type Enum

- Version TypeScript

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

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

- Version JavaScript

```
var Music;
(function (Music) {
    Music[Music["Rock"] = 2] = "Rock";
    Music[Music["Jazz"] = 4] = "Jazz";
    Music[Music["Blues"] = 8] = "Blues";
})(Music || (Music = {}));
;
var c = Music.Jazz;
```



Type Enum

- 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
- Version TypeScript

```
const enum Music { Jazz, Blues }

const m = Music.Jazz;
```

- Version JavaScript

```
var m = 0; // Jazz
```

Inférence de types



- Utilisée pour définir le type d'une variable quand aucun type n'est défini
 - initialisation des variables
 - définition des valeurs par défaut
 - détermination de la valeur renvoyée par une fonction

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

function fn(name = 'carl') { // type string
  return name; // type string
}
```

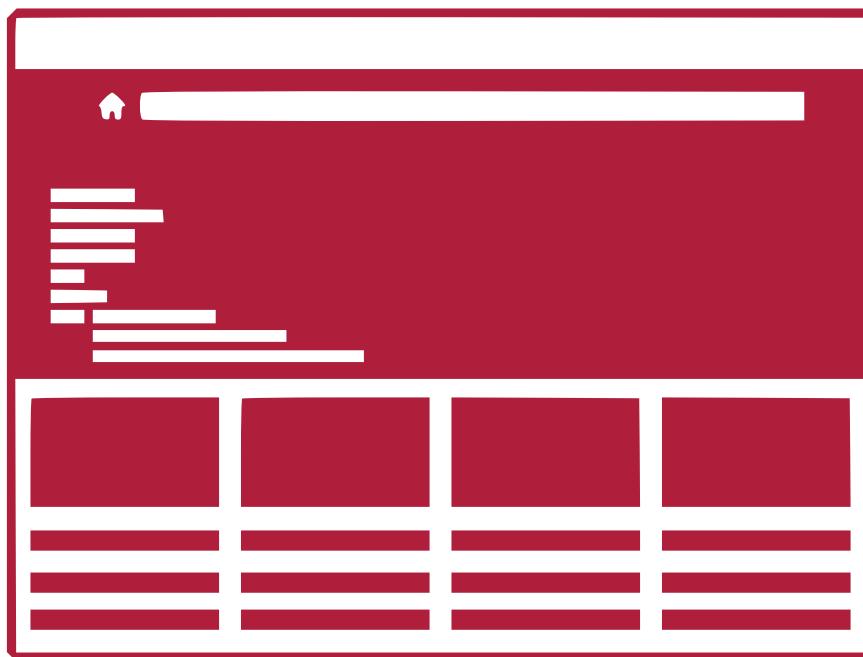


Inférence de types

- Quand le type doit être défini à partir de plusieurs expression, recherche du type le plus compatible

```
const x = [0, 1, "null"]; // array<string | number>
```





Objects

Plan



- Rappels
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- *Objects*
- Modules
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales

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**

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

Classes



- Version TypeScript :

```
class Person {  
    constructor() {}  
}
```

- Version JavaScript :

```
var Person = (function () {  
    function Person() {}  
    return Person;  
})();
```

Classes - Méthodes



- Méthodes ajoutées au **prototype** de l'objet
- Version TypeScript

```
class Person {  
    constructor() {}  
  
    sayHello(message: string) { }  
}
```

- Version JavaScript

```
var Person = (function () {  
    function Person() { }  
    Person.prototype.sayHello = function (message) { };  
    return Person;  
})();
```

Classes - Propriétés



- Trois scopes disponibles : **public**, **private** et **protected**
- Utilise le scope **public** par défaut
- Scope **protected** apparu en TypeScript 1.3
- Propriétés ajoutées sur l'objet en cours d'instanciation (**this**)
- Possibilité de définir des propriétés statiques (**static**)
 - Tous les types supportés : types primitifs, fonctions, ...
 - Propriété ajoutée au constructeur de l'objet



Classes - Propriétés

- Version TypeScript

```
class Person {  
    firstName: string;  
    constructor(firstName: string){  
        this.firstName = firstName;  
    }  
}
```

- Version JavaScript

```
var Person = (function () {  
    function Person(firstName) {  
        this.firstName = firstName;  
    }  
    return Person;  
})();
```



Classes - Propriétés

- Seconde version pour initialiser des propriétés
- Version TypeScript

```
class Person {  
    constructor(public firstName: string) {}  
}
```

- Version JavaScript

```
var Person = (function () {  
    function Person(firstName) {  
        this.firstName = firstName;  
    }  
    return Person;  
})();
```

Classes - Accesseurs



- Possibilité de définir des accesseurs pour accéder à une propriété
- Utiliser les mots clé **get** et **set**
- Attention à l'espacement apres les mots clé
- Nécessité de générer du code JavaScript compatible ES5
- Le code JavaScript généré utilisera **Object.defineProperty**

```
class Person {  
    private _secret: string;  
    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'
```



Classes - Accesseurs

- Version JavaScript

```
var Person = (function () {  
    function Person() { }  
  
    Object.defineProperty(Person.prototype, "secret", {  
        get: function () { return this._secret; },  
        set: function (value) { this._secret = value; },  
        enumerable: true,  
        configurable: true  
    });  
  
    return Person;  
}());
```

Classes - Héritage



- Système d'héritage entre classes via 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(); }  
}
```



Classes - Héritage

- Version JavaScript

```
var Person = (function () {
    function Person() { }
    Person.prototype.speak = function () { };
    return Person;
})();

var Child = (function (_super) {
    __extends(Child, _super); // reprise du prototype parent

    function Child() {
        _super.call(this);
    }
    Child.prototype.speak = function () {
        _super.prototype.speak.call(this);
    };
    return Child;
})(Person);
```

Interfaces



- Utilisée par le compilateur pour vérifier la cohérence des différents objets
- Aucun impact sur le JavaScript généré
- Système d'héritage entre interfaces
- Plusieurs cas d'utilisation possible
 - Vérification des paramètres d'une fonction
 - Vérification de la signature d'une fonction
 - Vérification de l'implémentation d'une classe

Interfaces



- Typescript réalise une compilation en Duck Typing
- 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

```
class Speaker {  
    constructor(public message: string) {}  
}  
  
function quack(params: { message: string }) {  
    console.log(params.message)  
}  
  
quack(); // Incorrect  
quack({ message: 'hello' }); // Correct  
quack(new Speaker('hello')); // Correct
```

```
interface Message {  
    message:string  
}  
  
function quack(params: Message) { }  
  
quack(); // Incorrect  
quack({ message: 'hello' }); // Correct
```



Interfaces - Propriétés optionnelles

- Possibilité de définir des propriétés optionnelles

```
interface Message {  
    message:string  
    title?:string  
}  
  
function quack(params: Message) { }  
  
quack(); // Incorrect  
quack({ message: 'hello' }); // Correct  
quack({ message: 'hello', title: 'Titre' }); // Correct
```



Interfaces - Signature d'une fonction

- Vérification de la signature d'une fonction

```
interface transform {  
    (source: string): string;  
}  
  
let transformFn: transform;  
  
transformFn = function(source: string) {  
    return source.toLowerCase();  
} // Correct  
  
transformFn = function(age: number) {  
    return age > 18;  
} // Incorrect
```

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 défini par l'interface

```
interface Musician {  
    play(): void;  
}  
  
class TrumpetPlay implements Musician {  
    play() {}  
}
```



Classes - Classes abstraites

- Ajout d'un mot-clé **abstract**
- Possibilité de créer des classes qui ne peuvent pas être instanciées directement
- Avant ce mot-clé (v1.6), deux possibilités :
 - Créer une interface, mais impossible d'implémenter les méthodes
 - Créer une classe, mais avec la possibilité de l'instancier

```
abstract class Person { }

class Adult extends Person { }

new Person(); // Incorrect
new Adult(); // Correct
```



Génériques

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

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
identity(5).toFixed(2); // Correct  
  
identity('hello').toFixed(2); // Incorrect  
  
identity(true);
```





Génériques

- Possibilité de définir une classe générique
- Définition d'une liste de paramètres de types de manière globale

```
class Log<T> {  
    log(value: T) {  
        console.log(value);  
    }  
}  
  
const numericLog = new Log<number>();  
  
numericLog.log(5); // Correct  
numericLog.log('hello'); // Incorrect
```

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(); // Correct
numberTransform.transform(3).toFixed(2); // Incorrect
```

Génériques



- Possibilité de définir des contraintes pour restreindre les types supportés
- Utilisation du mot-clé **extends**
- Par défaut, le compilateur vérifie que la méthode/propriété est disponible pour tous les types

```
class JazzPlayer { play() {} }
class RockStar { play() {} }

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

Génériques



- Création d'une interface pour décrire la contrainte

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

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

playAll([
    new JazzPlayer(), // Correct
    new RockStar() // Incorrect
]);
```





Modules

Plan



- Rappels
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- Objects
- *Modules*
- Type Definitions
- Concepts Avancés
- Roadmap et fonctions expérimentales

Modules



- Eviter de polluer le namespace global
- Permet d'organiser votre code TypeScript
- Par défaut, les variables, functions, classes... ne sont pas visibles de l'extérieur
- Deux types de modules : Interne et Externe
- L'import d'un module réalisé via `import` (syntaxe **EcmaScript 2015**)
- Pour exporter des objets, utilisation de `export`

```
import { environment, Logger } from './utils';

if (environment === 'DEV') {
  Logger.debug('You are running in dev mode');
}
```



Modules Internes - Namespaces

- Création d'un ou plusieurs Namespaces
- Définition du namespace via le mot-clé `namespace`
- Possibilité de définir des modules imbriqués



Modules Internes - Namespaces

- Définition de deux namespaces : `Utils` et `Utils.String`

```
namespace Utils {  
    export class Logger { [ ... ] }  
  
    export namespace String {  
        export class Formatter { [ ... ] }  
  
        function privateFormatFunction(){ [ ... ] }  
    }  
}  
  
new Utils.Logger();  
new Utils.String.Formatter();
```



Modules Internes - Namespaces

- Fonctionnalité non disponible en **ES2015**
- Nécessité d'utiliser des fonctions **IIFE**

```
var MyModule;  
(function (MyModule) {  
    MyModule.test = function() { alert('ok'); };  
})(MyModule || (MyModule = {}));
```

Modules Externes



- Support de plusieurs mécanismes de chargement : **CommonJS**, **AMD**, **UMD**, **System** et **ECMAScript 2015**
- Tous fichiers commençant par un `import` ou `export` considéré comme un module
- L'utilisation de `namespace` n'est plus nécessaire
- Mécanisme de chargement configurable via la propriété `module` lors de la compilation.
- Syntaxe identique à celle d'**ES2015**



Modules Externes - Exemple

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

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

export class Logger { ... }

export function getDate() { ... }

export class String {
  ...
}
```

- Utilisation du module ***utils***

```
import { createLogger, Logger, String } from './utils';
```





Modules Externes - Exemple

- Utilisation de `default` pour exporter un seul objet dans un module
- Un seul `default` par fichier
- L'objet pourra être importé en utilisant n'importe quel libellé
- Les imports nommés sont préférables, car ils facilitent le refactoring

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

```
import MyClassAlias from './MyClass';
```





Modules Externes - Exemple

- Possibilité de renommer un import

```
import { createLogger as newLogger } from './utils';
let logger = newLogger();
```

- Import d'un module entier

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



Module Externes - AMD

- Principalement utilisé par **RequireJS**
- Définition d'un module via **define** et import via **require**
- Chargement asynchrone des modules

```
require('module1', 'module2', function(module1, module2) { ... })
```

- Génération des modules compatibles **AMD**

```
tsc --module amd
```





Modules Externes - CommonJS

- Principalement utilisé par **Node.JS**
- Définition des objets à exporter via **export** et import via **require**
- Chargement synchrone des modules

```
const module1 = require('module1');
const module1 = require('module2');

[ ... ]
```

- Génération des modules compatibles **CommonJS**

```
tsc --module commonjs
```



Testez vos modules

- Nécessité d'utiliser le pattern **AMD** pour vos tests
- Utilisation du module **karma-requirejs**

```
npm install --save-dev karma-requirejs
```

- Configuration **Karma** pour
 - Ajouter le plugin **requirejs**
 - Importer **require.js**
 - Utiliser le pattern **AMD**
 - Définir un fichier pour la configuration **requirejs**



Testez vos modules

- Exemple de configuration **Karma**

```
frameworks: ['jasmine', 'requirejs'],  
  
files: [  
  { pattern: 'src/**/*.ts', included: false },  
  { pattern: 'spec/**/*.ts', included: false },  
  'spec/test-main.js'  
],  
  
 preprocessors: { '+(spec|src)/**/*.ts': ['typescript'] },  
  
typescriptPreprocessor: {  

```





Type Definitions

Plan



- Rappels
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- Objects
- Modules
- *Type Definitions*
- Concepts Avancés
- Roadmap et fonctions expérimentales



Type Definitions

- Fichier permettant de décrire une librairie JavaScript
- Extension `.d.ts`
- Fichiers automatiquement détectés par TypeScript
 - Fichiers publiés dans l'organisation `@types` par le projet ***DefinitelyTyped***
 - Fichiers définis directement dans un dépendance de votre projet
- Eviter d'utiliser les outils `tsd` et `typings`
- Génération via le paramètre `tsc --declaration`
- Permet de bénéficier
 - de **l'autocompletion**
 - du **type checking**



Type Definitions - @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/>
- Dépendances uniquement nécessaire au développement

```
npm install --save-dev @types/jquery
```



Type Definitions - Package NPM

- Le fichier de définition peut être également packagé avec le module associé
- Deux solutions :
 - Lecture 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"  
}
```

- Chargement d'un fichier **index.d.ts** situé à la racine du module



Type Definitions - Ecriture

- 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;
    }
}
```





Concepts Avancés

Plan



- Rappels
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- *Concepts Avancés*
- Roadmap et fonctions expérimentales

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;
```



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
```



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 permet de bénéficier du **type checking**

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

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

Décorateurs



- Standard proposé par Yehuda Katz pour **ECMAScript 7**
- 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

```
@decorator class DecoratedClass { }
function decorator(target) {
  target.prototype.isDecorated = function() {
    console.log('decorated');
  }
}
console.log(new DecoratedClass().isDecorated());
```

Décorateurs



- Fonctionnalité à activer via :
 - 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

- **MethodDecorator**

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

- **PropertyDecorator**

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



Décorateurs - Les différents types

- **ClassDecorator**

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

- **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: any) { }

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

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

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



Décorateurs - MethodDecorator

- Permet de décorer les méthodes d'une classe

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

- **target** : Prototype contenant la méthode décorée
- **key**: Le nom de la méthode qui est décorée
- **description**: Le résultat retournée par la méthode
`Object.getOwnPropertyDescriptor`



Décorateurs - MethodDecorator

- Version TypeScript :

```
class Obj {  
  
    @validate  
    public setName(name : string) { }  
}  
  
function validate(target: any, key: string, descriptor: any) { }
```



Décorateurs - MethodDecorator

- Version JavaScript :

```
var Obj = (function () {
    function Obj() { }
    Obj.prototype.setName = function (name) {
    };

    Object.defineProperty(Obj.prototype, "setName",
        __decorate([ validate ],
            Obj.prototype, "setName",
            Object.getOwnPropertyDescriptor(Obj.prototype, "setName")));
    return Obj;
})();
function validate(target, key, descriptor) { }
```

Décorateurs - MethodDecorator



- Exemple de **MethodDecorator** :

```
class Obj {  
    @validate  
    public setName(name: string) { }  
}  
  
function validate(target: any, key: string, descriptor: any) {  
    const originalMethod = descriptor.value;  
    descriptor.value = function(...args: any[]) {  
        if (typeof args[0] === 'string') {  
            if (args[0].length < 5) {  
                throw new Error('too small');  
            }  
        }  
        return originalMethod.apply(this, args);  
    }  
    return descriptor;  
}
```





Décorateurs - PropertyDecorator

- Permet de décorer les propriétés d'une classe

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

- **target** : Objet dans lequel la propriété décorée est définie
- **key**: Le nom de la propriété qui est décorée



Décorateurs - PropertyDecorator

- Version TypeScript

```
class Obj {  
  @toLowerCase public name:string;  
  constructor(name : string) { this.name = name; }  
}  
  
function toLowerCase(target: any, key: string) { }
```

- Version JavaScript

```
var Obj = (function () {  
  function Obj(name) { this.name = name; }  
  __decorate([toLowerCase], Obj.prototype, "name");  
  return Obj;  
})();
```



Décorateurs - PropertyDecorator

- Exemple de **PropertyDecorator** : `toLowerCase`

```
function toLowerCase(target: any, key: string){  
    const _val = this[key];  
  
    const getter = function () { return _val; };  
  
    const setter = function (newVal) {  
        _val = newVal.toLowerCase();  
    };  
  
    Object.defineProperty(target, key, {  
        get: getter,  
        set: setter,  
        enumerable: true,  
        configurable: true  
    });  
}
```



Décorateurs - ClassDecorator

- Permet de décorer une classe
- En TypeScript :

```
@log class Person { }
function log(target: any) { }
```

- En JavaScript :

```
var Person = (function () {
    function Person() { }
    Person = __decorate([log], Person);
    return Person;
})();
function log(target) { }
```



Décorateurs - ClassDecorator

- Exemple de **ClassDecorator** :

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

    constructor() {
        this.name = 'emmanuel';
    }
}

function log(target: any) {
    const constructor: any = function (...args) {
        target.apply(this, args);
        console.log("Constructeur exécuté");
    }
    constructor.prototype = Object.create(target.prototype);
    return <any>constructor;
}
```





Décorateurs - ParameterDecorator

- Permet de décorer les paramètres d'une méthode
- Ne peut pas modifier l'implémentation d'un constructeur/méthode
- Utilisé pour ajouter des metadonnées via la [Metadata API](#)
- API disponible via un polyfill : [reflect-metadata](#)

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

- **target** : Prototype de la classe décorée
- **key** : Le nom de la méthode dont le paramètre est décoré
- **index** : L'index du paramètre décoré



Décorateurs - ParameterDecorator

- Version TypeScript :

```
class Form {  
  
    @validate  
    public saveData(@required name: string) {  
  
    }  
}  
function required(target: any, key: string, index: number) { }
```

- Version JavaScript :

```
__decorate([  
    validate,  
    __param(0, required)  
], Form.prototype, "saveData", null);  
function required(target, key, index) { }
```



Décorateurs - ParameterDecorator



- Exemple de **ParameterDecorator** :

```
import "reflect-metadata";

class Form {

    @validate
    public saveData(@required name: string) {

    }
}

const key = Symbol("required");

function required(target: any, key: string, index: number) {

    let required: number[] = Reflect.getOwnMetadata(key, target, key) || [];
    required.push(parameterIndex);

    Reflect.defineMetadata(key, required, target, propertyKey);
}
```

Décorateurs - ParameterDecorator



- Récupération des metadonnées dans le décorateur `validate`

```
function validate(target: any, property: string,
                  descriptor: TypedPropertyDescriptor<Function>) {

    let method = descriptor.value;

    descriptor.value = function () {
        let required: number[] = Reflect.getOwnMetadata(key, target, property);

        for (let parameterIndex of required) {
            if (arguments[parameterIndex] === undefined) {
                throw new Error("Missing required argument.");
            }
        }

        return method.apply(this, arguments);
    }
}
```



Décorateurs - Factory



- Implémenter un décorateur pour tous les types présentés précédemment

```
@log
class Person {

    @log
    public name: string;

    constructor(name: string, surname: string) {
        this.name = name;
    }

    @log
    public hello(@log message: string): string { }
}
```



Décorateurs - Factory

- Créer un nouveau décorateur
 - en charge d'exécuter la bonne implémentation
 - en fonction des paramètres définis

```
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);  
    }  
}
```



Décorateurs - Configuration

- Possibilité de paramétriser un décorateur
- Version TypeScript :

```
@log({ prefix : '>>' })
class Person {
  constructor() { }
}
```

- Version JavaScript

```
var Person = (function () {
  function Person() {}
  Person = __decorate([log({ prefix: '>>' })], Person);
  return Person;
})();
```

Décorateurs - Configuration



- La fonction doit retourner une fonction implémentant le décorateur
- Les paramètres du décorateur sont accessibles via les paramètres de la fonction

```
function log(option) {  
    return (target: any) => {  
        const constructor : any = function (...args) {  
            target.apply(this, args);  
            console.log(option.prefix, "Constructeur exécuté");  
        }  
        constructor.prototype = Object.create(target.prototype);  
        return <any>constructor;  
    }  
}
```







Roadmap et fonctions expérimentales

Plan



- Rappels
- Introduction
- Outilage
- Tests
- Types et Inférence de types
- Objects
- Modules
- Type Definitions
- Concepts Avancés
- *Roadmap et fonctions expérimentales*

Roadmap



- Roadmap vers TypeScript 2.0
 - Support des types `Null` et `Undefined`
 - Support des propriétés `readonly`
 - Support des propriétés `abstract`
 - Propriétés optionnelles dans les classes
 - Support des générateurs ES2015 pour ES5 / ES3
 - Support `async` / `await` pour ES5 / ES3



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
```





Propriétés readonly

- Marqueur d'immutabilité
- Peut s'appliquer sur une propriété de classe ou une signature d'index

```
class Foo {  
    readonly bar = 'bar'; // propriété  
    readonly baz: string; // propriété  
    constructor() {  
        this.baz = 'baz'; // Assignation autorisée dans le constructeur  
    }  
}  
  
interface Foo {  
    readonly [key: number]: string; // signature d'index  
}  
const foo: Foo = { 1: 'hello', 2: 'world' }; // immuable
```





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
```

Support des générateurs ES2015 pour ES5 / ES3



- Possibilité de compiler des générateurs ES2015 en ES5 / ES3
- Un générateur est une fonction qui peut être stoppée et reprise

```
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 }  
// ...
```



Support async / await pour ES5 / ES3

- Permet l'écriture de code asynchrone de façon synchrone
- Utilisation des générateurs ES2015

```
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
// ...
```

