eBook Gratuit

APPRENEZ TypeScript

MCours.com

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#typescript

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: typescript

It is an unofficial and free TypeScript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official TypeScript.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec TypeScript

Remarques

TypeScript se veut un sur-ensemble de JavaScript qui se transforme en JavaScript. En générant un code conforme à ECMAScript, TypeScript peut introduire de nouvelles fonctionnalités de langage tout en conservant la compatibilité avec les moteurs JavaScript existants. ES3, ES5 et ES6 sont actuellement des cibles prises en charge.

Les types facultatifs sont une caractéristique principale. Les types permettent une vérification statique dans le but de détecter rapidement les erreurs et peuvent améliorer les outils avec des fonctionnalités telles que la refactorisation du code.

TypeScript est un langage de programmation open source et multi-plateforme développé par Microsoft. Le code source est disponible sur GitHub .

Versions

Version	Date de sortie
2.4.1	2017-06-27
2.3.2	2017-04-28
2.3.1	2017-04-25
2.3.0 beta	2017-04-04
2.2.2	2017-03-13
2.2	2017-02-17
2.1.6	2017-02-07
2.2 bêta	2017-02-02
2.1.5	2017-01-05
2.1.4	2016-12-05
2.0.8	2016-11-08
2.0.7	2016-11-03
2.0.6	2016-10-23
2.0.5	2016-09-22

Version	Date de sortie
2.0 bêta	2016-07-08
1.8.10	2016-04-09
1.8.9	2016-03-16
1.8.5	2016-03-02
1.8.2	2016-02-17
1.7.5	2015-12-14
1,7	2015-11-20
1.6	2015-09-11
1.5.4	2015-07-15
1,5	2015-07-15
1.4	2015-01-13
1.3	2014-10-28
1.1.0.1	2014-09-23

Examples

Installation et configuration

Contexte

TypeScript est un sur-ensemble typé de JavaScript qui compile directement en code JavaScript. Les fichiers TypeScript utilisent généralement l'extension `.ts` . De nombreux IDE prennent en charge TypeScript sans aucune autre configuration requise, mais TypeScript peut également être compilé avec le package TypeScript Node.JS à partir de la ligne de commande.

IDE

Visual Studio

- `Visual Studio 2015` inclut TypeScript.
- `Visual Studio 2013 Update 2` ou version ultérieure inclut TypeScript, ou vous pouvez télécharger TypeScript pour les versions antérieures .

Code Visual Studio

- Visual Studio Code (vscode) fournit une saisie semi-automatique contextuelle ainsi que des outils de refactoring et de débogage pour TypeScript. vscode est lui-même implémenté dans TypeScript. Disponible pour Mac OS X, Windows et Linux.

WebStorm

- `WebStorm 2016.2` est livré avec TypeScript et un compilateur intégré. [Webstorm n'est pas gratuit]

IntelliJ IDEA

- `IntelliJ IDEA 2016.2` prend en charge Typescript et un compilateur via un plug - in géré par l'équipe Jetbrains. [IntelliJ n'est pas gratuit]

Atome & Atom-Typograph

- `Atom` prend en charge TypeScript avec le package atom-typescript .

Texte sublime

- `Sublime Text` prend en charge TypeScript avec le package typecript .

Installation de l'interface de ligne de commande

Installez Node.js

Installez le paquet npm globalement

Vous pouvez installer TypeScript globalement pour y accéder depuis n'importe quel répertoire.

```
npm install -g typescript
```

ou

Installez le paquet npm localement

Vous pouvez installer TypeScript localement et enregistrer dans package.json pour restreindre à un répertoire.

```
npm install typescript --save-dev
```

Canaux d'installation

Vous pouvez installer à partir de:

- Canal stable: `npm install typescript`
- Canal bêta: `npm install typescript@beta`
- Canal de développement: `npm install typescript@next`

Compiler du code TypeScript

La commande de compilation `tsc` est fournie avec `typescript` , qui peut être utilisé pour compiler du code.

```
tsc my-code.ts
```

Cela crée un fichier `my-code.js` .

Compiler en utilisant tsconfig.json

Vous pouvez également fournir des options de compilation qui voyagent avec votre code via un fichier `tsconfig.json` . Pour démarrer un nouveau projet TypeScript, `cd` dans le répertoire racine de votre projet dans une fenêtre de terminal et lancez `tsc --init` . Cette commande génère un fichier `tsconfig.json` avec des options de configuration minimales, similaires à celles ci-dessous.

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "pretty": true
    },
    "exclude": [
        "node_modules"
    ]
}
```

Avec un fichier `tsconfig.json` placé à la racine de votre projet TypeScript, vous pouvez utiliser la commande `tsc` pour exécuter la compilation.

Bonjour le monde

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
```

```
    }
    greet(): string {
        return this.greeting;
    }
};

let greeter = new Greeter("Hello, world!");
console.log(greeter.greet());
```

Nous avons ici une classe, `Greeter` , qui a un `constructor` et une méthode de `greet` . Nous pouvons construire une instance de la classe en utilisant le `new` mot - clé et passer dans une chaîne que nous voulons que la `greet` méthode pour la sortie de la console. L'instance de notre classe `Greeter` est stockée dans la variable `greeter` que nous appelons ensuite la méthode `greet` .

Syntaxe de base

TypeScript est un sur-ensemble typé de JavaScript, ce qui signifie que tout le code JavaScript est un code TypeScript valide. TypeScript ajoute beaucoup de nouvelles fonctionnalités à cela.

TypeScript rend JavaScript plus proche d'un langage orienté objet fortement typé, à savoir C # et Java. Cela signifie que le code TypeScript est plus facile à utiliser pour les grands projets et que ce code est plus facile à comprendre et à gérer. Le typage fort signifie également que le langage peut (et est) précompilé et que des variables ne peuvent pas être affectées en dehors de leur plage déclarée. Par exemple, lorsqu'une variable TypeScript est déclarée en tant que nombre, vous ne pouvez pas lui affecter de valeur de texte.

Cette forte typage et cette orientation des objets facilitent le débogage et la maintenance de TypeScript, deux des points les plus faibles du langage JavaScript standard.

Déclarations de type

Vous pouvez ajouter des déclarations de type aux variables, aux paramètres de fonction et aux types de retour de fonction. Le type est écrit après un deux-points après le nom de la variable, comme ceci: `var num: number = 5;` Le compilateur vérifiera alors les types (si possible) pendant la compilation et les erreurs de type de rapport.

```
var num: number = 5;
num = "this is a string";  // error: Type 'string' is not assignable to type 'number'.
```

Les types de base sont:

- `number` (entiers et nombres à virgule flottante)
- `string`
- `boolean`
- `Array` . Vous pouvez spécifier les types d'éléments d'un tableau. Il existe deux manières équivalentes de définir les types de tableau: `Array<T>` et `T[]` . Par exemple:
    - `number[]` - tableau de nombres
    - `Array<string>` - tableau de chaînes
- Tuples Les tuples ont un nombre fixe d'éléments avec des types spécifiques.

    - `[boolean, string]` - tuple où le premier élément est un booléen et le second une chaîne.
    - `[number, number, number]` - tuple de trois nombres.
- `{}` - objet, vous pouvez définir ses propriétés ou indexeur
    - `{name: string, age: number}` - objet avec attribut name et age
    - `{[key: string]: number}` - un dictionnaire de nombres indexés par chaîne
- `enum` - `{ Red = 0, Blue, Green }` - énumération mappée sur des nombres
- Fonction. Vous spécifiez les types pour les paramètres et la valeur de retour:
    - `(param: number) => string` - fonction prenant un paramètre de nombre renvoyant une chaîne
    - `() => number` - fonction sans paramètres renvoyant un numéro.
    - `(a: string, b?: boolean) => void` - fonction prenant une chaîne et éventuellement un booléen sans valeur de retour.
- `any` - Permet tout type. Les expressions impliquant `any` qui ne sont pas vérifiées.
- `void` - représente "rien", peut être utilisé comme valeur de retour de fonction. Seules les `null` et `undefined` font partie du type `void` .
- `never`
    - `let foo: never;` -Comme le type de variables sous les gardes de type qui ne sont jamais vraies.
    - `function error(message: string): never { throw new Error(message); }` - Comme type de retour des fonctions qui ne reviennent jamais.

- `null` - type pour la valeur `null` . `null` est implicitement partie de chaque type, sauf si les vérifications NULL strictes sont activées.

Fonderie

Vous pouvez effectuer un transtypage explicite entre crochets, par exemple:

```
var derived: MyInterface;
(<ImplementingClass>derived).someSpecificMethod();
```

Cet exemple montre une classe `derived` qui est traitée par le compilateur en tant que `MyInterface` . Sans le casting sur la deuxième ligne, le compilateur émettrait une exception car il ne comprend pas `someSpecificMethod()` , mais le passage via `<ImplementingClass>derived` suggère au compilateur ce qu'il doit faire.

Une autre façon de coulée dactylographiée utilise le `as` mot clé:

```
var derived: MyInterface;
(derived as ImplementingClass).someSpecificMethod();
```

Depuis Typecript 1.6, le mot-clé `as` est utilisé par défaut, car l'utilisation de `<>` est ambiguë dans les fichiers **.jsx** . Ceci est mentionné dans la documentation officielle de Typescript .

Des classes

Les classes peuvent être définies et utilisées dans le code TypeScript. Pour en savoir plus sur les classes, consultez la page de documentation Classes .

TypeScript REPL dans Node.js

Pour utiliser TypeScript REPL dans Node.js, vous pouvez utiliser le package tsun

Installez-le globalement avec

```
npm install -g tsun
```

et exécutez dans votre terminal ou l'invite de commande avec la commande `tsun`

Exemple d'utilisation:

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl
$ function multiply(x, y) {
..return x * y;
..}
undefined
$ multiply(3, 4)
12
```

Exécution de TypeScript en utilisant ts-node

ts-node est un paquet npm qui permet à l'utilisateur d'exécuter directement des fichiers texte, sans avoir besoin de précompiler avec `tsc` . Il fournit également REPL .

Installer ts-node en utilisant globalement

```
npm install -g ts-node
```

ts-node ne regroupe pas le compilateur de typecript, vous devrez donc peut-être l'installer.

```
npm install -g typescript
```

Exécution du script

Pour exécuter un script nommé *main.ts* , exécutez

```
ts-node main.ts
```

```
// main.ts
console.log("Hello world");
```

Exemple d'utilisation

```
$ ts-node main.ts
Hello world
```

Exécution de REPL

Pour exécuter la commande d'exécution REPL `ts-node`

Exemple d'utilisation

```
$ ts-node
> const sum = (a, b): number => a + b;
undefined
> sum(2, 2)
4
> .exit
```

Pour quitter REPL, utilisez la commande `.exit` ou appuyez deux fois sur `CTRL+C`

Lire Démarrer avec TypeScript en ligne: https://riptutorial.com/fr/typescript/topic/764/demarrer-avec-typescript

Chapitre 2: Comment utiliser une bibliothèque javascript sans fichier de définition de type

Introduction

Alors que certaines bibliothèques JavaScript existantes ont des fichiers de définition de type , beaucoup ne le font pas.

TypeScript propose quelques modèles pour gérer les déclarations manquantes.

Examples

Déclarer un tout global

Il est parfois plus simple de simplement déclarer un global de type `any` , en particulier dans les projets simples.

Si jQuery n'avait pas de déclaration de type (c'est le cas), vous pouvez mettre

```
declare var $: any;
```

Maintenant , toute utilisation de $ sera tapé `any` .

Faire un module qui exporte une valeur par défaut

Pour les projets plus compliqués ou dans les cas où vous avez l'intention de taper progressivement une dépendance, il peut être plus propre de créer un module.

En utilisant JQuery (bien que des typages soient disponibles) comme exemple:

```
// place in jquery.d.ts
declare let $: any;
export default $;
```

Et puis, dans n'importe quel fichier de votre projet, vous pouvez importer cette définition avec:

```
// some other .ts file
import $ from "jquery";
```

Après cette importation, `$` sera saisi comme `any` .

Si la bibliothèque contient plusieurs variables de niveau supérieur, exportez-les et importez-les par nom à la place:

```
// place in jquery.d.ts
declare module "jquery" {
   let $: any;
   let jQuery: any;

   export { $ };
   export { jQuery };
}
```

Vous pouvez ensuite importer et utiliser les deux noms:

```
// some other .ts file
import {$, jQuery} from "jquery";

$.doThing();
jQuery.doOtherThing();
```

Utiliser un module d'ambiance

Si vous voulez simplement indiquer l' *intention* d'une importation (afin de ne pas vouloir déclarer un global) mais que vous ne souhaitez pas avoir de définitions explicites, vous pouvez importer un module ambient.

```
// in a declarations file (like declarations.d.ts)
declare module "jquery";    // note that there are no defined exports
```

Vous pouvez ensuite importer depuis le module ambient.

```
// some other .ts file
import {$, jQuery} from "jquery";
```

Tout ce qui est importé du module déclaré (comme `$` et `jQuery`) ci-dessus sera de type `any`

Lire Comment utiliser une bibliothèque javascript sans fichier de définition de type en ligne: https://riptutorial.com/fr/typescript/topic/8249/comment-utiliser-une-bibliotheque-javascript-sans-fichier-de-definition-de-type

Chapitre 3: Configurez le projet typecript pour compiler tous les fichiers en texte dactylographié.

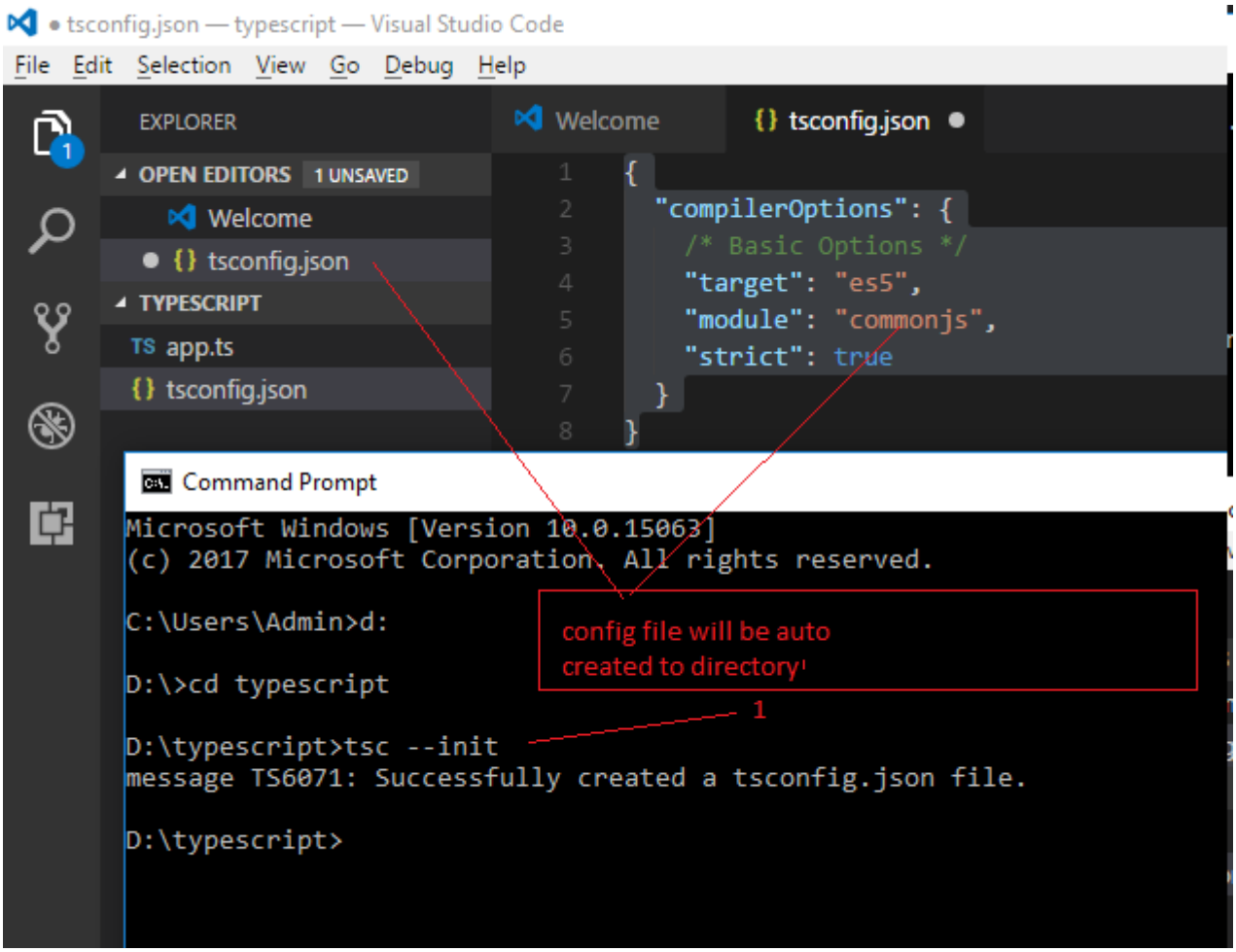
Introduction

créer votre premier fichier de configuration .tsconfig qui indiquera au compilateur TypeScript comment traiter vos fichiers .ts

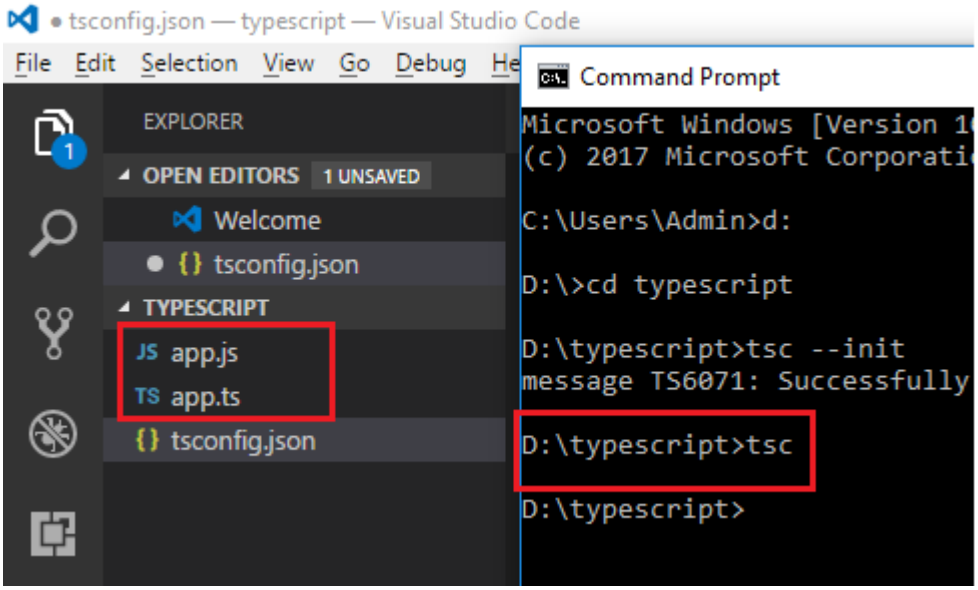
Examples

Configuration du fichier de configuration typographique

- Entrez la commande " **tsc --init** " et appuyez sur Entrée.
- Avant cela, nous devons compiler le fichier ts avec la commande " **tsc app.ts** " maintenant tout est défini dans le fichier de configuration ci-dessous automatiquement.



- Maintenant, vous pouvez compiler tous les manuscrits par commande " **tsc** ". Il créera automatiquement le fichier ".js" de votre fichier dactylographié.



- Si vous voulez créer un autre texte et appuyez sur la commande "tsc" dans l'invite de commande ou le fichier javascript du terminal sera automatiquement créé pour le fichier dactylographié.

Je vous remercie,

Lire Configurez le projet typecript pour compiler tous les fichiers en texte dactylographié. en ligne: https://riptutorial.com/fr/typescript/topic/10537/configurez-le-projet-typecript-pour-compiler-tous-les-fichiers-en-texte-dactylographie-

Chapitre 4: Contrôles Nuls Strict

Examples

Strict null vérifie en action

Par défaut, tous les types de TypeScript autorisent `null` :

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // TypeScript does not complain, but this is a runtime error.
```

TypeScript 2.0 prend en charge les contrôles NULL stricts. Si vous définissez `--strictNullChecks` lors de l'exécution de `tsc` (ou définissez cet indicateur dans votre `tsconfig.json`), alors les types n'autorisent plus `null` :

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // error: Argument of type 'null' is not assignable to parameter of type
'Element'.
```

Vous devez autoriser explicitement les valeurs `null` :

```
function getId(x: Element|null) {
  return x.id;  // error TS2531: Object is possibly 'null'.
}
getId(null);
```

Avec un garde approprié, le type de code vérifie et fonctionne correctement:

```
function getId(x: Element|null) {
  if (x) {
    return x.id;  // In this branch, x's type is Element
  } else {
    return null;  // In this branch, x's type is null.
  }
}
getId(null);
```

Assertions non nulles

L'opérateur d'assertion non nulle, `!` , vous permet d'affirmer qu'une expression n'est pas `null` ou `undefined` lorsque le compilateur TypeScript ne peut en déduire automatiquement:

```
type ListNode = { data: number; next?: ListNode; };

function addNext(node: ListNode) {
```

```
    if (node.next === undefined) {
        node.next = {data: 0};
    }
}

function setNextValue(node: ListNode, value: number) {
    addNext(node);

    // Even though we know `node.next` is defined because we just called `addNext`,
    // TypeScript isn't able to infer this in the line of code below:
    // node.next.data = value;

    // So, we can use the non-null assertion operator, !,
    // to assert that node.next isn't undefined and silence the compiler warning
    node.next!.data = value;
}
```

Lire Contrôles Nuls Strict en ligne: https://riptutorial.com/fr/typescript/topic/1727/controles-nuls-strict

Chapitre 5: Décorateur de classe

Paramètres

Paramètre	Détails
cible	La classe en cours de décoration

Examples

Décorateur de classe de base

Un décorateur de classe est juste une fonction qui prend la classe comme seul argument et la retourne après avoir fait quelque chose avec:

```
function log<T>(target: T) {

    // Do something with target
    console.log(target);

    // Return target
    return target;

}
```

Nous pouvons ensuite appliquer le décorateur de classe à une classe:

```
@log
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}
```

Générer des métadonnées en utilisant un décorateur de classe

Cette fois, nous allons déclarer un décorateur de classe qui ajoutera des métadonnées à une classe lorsque nous lui avons appliqué:

```
function addMetadata(target: any) {

    // Add some metadata
    target.__customMetadata = {
        someKey: "someValue"
    };
```

```
    // Return target
    return target;

}
```

Nous pouvons alors appliquer le décorateur de classe:

```
@addMetadata
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}

function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromClass(Person));
```

Le décorateur est appliqué lorsque la classe n'est pas déclarée lorsque nous créons des instances de la classe. Cela signifie que les métadonnées sont partagées entre toutes les instances d'une classe:

```
function getMetadataFromInstance(target: any) {
    return target.constructor.__customMetadata;
}

let person1 = new Person("John");
let person2 = new Person("Lisa");

console.log(getMetadataFromInstance(person1));
console.log(getMetadataFromInstance(person2));
```

Passer des arguments à un décorateur de classe

Nous pouvons envelopper un décorateur de classe avec une autre fonction pour permettre la personnalisation:

```
function addMetadata(metadata: any) {
    return function log(target: any) {

        // Add metadata
        target.__customMetadata = metadata;

        // Return target
        return target;

    }
}
```

`addMetadata` prend quelques arguments utilisés comme configuration, puis retourne une fonction sans nom qui est le décorateur réel. Dans le décorateur, nous pouvons accéder aux arguments car il y a une fermeture en place.

On peut alors invoquer le décorateur en passant des valeurs de configuration:

```
@addMetadata({ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" })
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}
```

Nous pouvons utiliser la fonction suivante pour accéder aux métadonnées générées:

```
function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromInstance(Person));
```

Si tout s'est bien passé, la console devrait afficher:

```
{ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }
```

Lire Décorateur de classe en ligne: https://riptutorial.com/fr/typescript/topic/4592/decorateur-de-classe

Chapitre 6: Des classes

Introduction

TypeScript, comme ECMA Script 6, prend en charge la programmation orientée objet à l'aide de classes. Cela contraste avec les anciennes versions de JavaScript, qui ne supportaient que la chaîne d'héritage basée sur des prototypes.

Le support de classe dans TypeScript est similaire à celui de langages comme Java et C #, dans la mesure où les classes peuvent hériter d'autres classes, tandis que les objets sont instanciés en tant qu'occurrences de classe.

Aussi similaires à ces langages, les classes TypeScript peuvent implémenter des interfaces ou utiliser des génériques.

Examples

Classe simple

```
class Car {
    public position: number = 0;
    private speed: number = 42;

    move() {
        this.position += this.speed;
    }
}
```

Dans cet exemple, nous déclarons une classe simple `Car` . La classe a trois membres: une `speed` propriété privée, une `position` propriété publique et un `move` méthode publique. Notez que chaque membre est public par défaut. C'est pourquoi `move()` est public, même si nous n'avons pas utilisé le mot-clé `public` .

```
var car = new Car();        // create an instance of Car
car.move();                 // call a method
console.log(car.position);  // access a public property
```

Héritage de base

```
class Car {
    public position: number = 0;
    protected speed: number = 42;

    move() {
        this.position += this.speed;
    }
}
```

```
class SelfDrivingCar extends Car {

    move() {
        // start moving around :-)
        super.move();
        super.move();
    }
}
```

Cet exemple montre comment créer une sous-classe très simple de la classe `Car` utilisant le mot `extends` clé extend. La classe `SelfDrivingCar` remplace la méthode `move()` et utilise l'implémentation de la classe de base en utilisant `super` .

Constructeurs

Dans cet exemple, nous utilisons le `constructor` pour déclarer une `position` propriété publique et une `speed` propriété protégée dans la classe de base. Ces propriétés sont appelées *propriétés de paramètre* . Ils nous permettent de déclarer un paramètre constructeur et un membre au même endroit.

L'un des meilleurs avantages de TypeScript est l'affectation automatique des paramètres du constructeur à la propriété concernée.

```
class Car {
    public position: number;
    protected speed: number;

    constructor(position: number, speed: number) {
        this.position = position;
        this.speed = speed;
    }

    move() {
        this.position += this.speed;
    }
}
```

Tout ce code peut être repris dans un seul constructeur:

```
class Car {
    constructor(public position: number, protected speed: number) {}

    move() {
        this.position += this.speed;
    }
}
```

Et les deux seront transférés de TypeScript (temps de conception et de compilation) en JavaScript avec le même résultat, mais en écrivant beaucoup moins de code:

```
var Car = (function () {
    function Car(position, speed) {
        this.position = position;
```

```
        this.speed = speed;
    }
    Car.prototype.move = function () {
        this.position += this.speed;
    };
    return Car;
}());
```

Les constructeurs de classes dérivées doivent appeler le constructeur de classe de base avec `super()` .

```
class SelfDrivingCar extends Car {
    constructor(startAutoPilot: boolean) {
        super(0, 42);
        if (startAutoPilot) {
            this.move();
        }
    }
}

let car = new SelfDrivingCar(true);
console.log(car.position);  // access the public property position
```

Les accesseurs

Dans cet exemple, nous modifions l'exemple "Simple class" pour autoriser l'accès à la propriété `speed` . Les accesseurs de typecript nous permettent d’ajouter du code supplémentaire dans les getters ou les setters.

```
class Car {
    public position: number = 0;
    private _speed: number = 42;
    private _MAX_SPEED = 100

    move() {
        this.position += this._speed;
    }

    get speed(): number {
        return this._speed;
    }

    set speed(value: number) {
        this._speed = Math.min(value, this._MAX_SPEED);
    }
}

let car = new Car();
car.speed = 120;
console.log(car.speed);  // 100
```

Classes abstraites

```
abstract class Machine {
    constructor(public manufacturer: string) {
```

```
    }

    // An abstract class can define methods of it's own, or...
    summary(): string {
        return `${this.manufacturer} makes this machine.`;
    }

    // Require inheriting classes to implement methods
    abstract moreInfo(): string;
}

class Car extends Machine {
    constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

    move() {
        this.position += this.speed;
    }

    moreInfo() {
        return `This is a car located at ${this.position} and going ${this.speed}mph!`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // position is now 80
console.log(myCar.summary()); // prints "Konda makes this machine."
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"
```

Les classes abstraites sont des classes de base à partir desquelles d'autres classes peuvent s'étendre. Ils ne peuvent pas être instanciés eux-mêmes (c.-à-d. Que vous **ne pouvez pas** faire de `new Machine("Konda")`).

Les deux caractéristiques clés d'une classe abstraite dans Typescript sont:

1. Ils peuvent mettre en œuvre leurs propres méthodes.
2. Ils peuvent définir des méthodes que les classes héritées **doivent** implémenter.

Pour cette raison, les classes abstraites peuvent être considérées comme une **combinaison d'une interface et d'une classe** .

Singe patch une fonction dans une classe existante

Parfois, il est utile de pouvoir étendre une classe avec de nouvelles fonctions. Par exemple, supposons qu'une chaîne soit convertie en une chaîne de casse camel. Nous devons donc dire à TypeScript, que `String` contient une fonction appelée `toCamelCase` , qui renvoie une `string` .

```
interface String {
    toCamelCase(): string;
}
```

Maintenant, nous pouvons corriger cette fonction dans l'implémentation `String` .

```
String.prototype.toCamelCase = function() : string {
    return this.replace(/[^a-z ]/ig, '')
        .replace(/(?:^\w|[A-Z]|\b\w|\s+)/g, (match: any, index: number) => {
            return +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();
        });
}
```

Si cette extension de `String` est chargée, elle est utilisable comme ceci:

```
"This is an example".toCamelCase();    // => "thisIsAnExample"
```

Transpilation

Étant donné une classe `SomeClass` , voyons comment le TypeScript est transposé en JavaScript.

TypeScript source

```
class SomeClass {

    public static SomeStaticValue: string = "hello";
    public someMemberValue: number = 15;
    private somePrivateValue: boolean = false;

    constructor () {
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }

    public static getGoodbye(): string {
        return "goodbye!";
    }

    public getFortyTwo(): number {
        return 42;
    }

    private getTrue(): boolean {
        return true;
    }

}
```

Source JavaScript

Lorsque transpiled en utilisant TypeScript `v2.2.2` , la sortie est comme `v2.2.2` :

```
var SomeClass = (function () {
    function SomeClass() {
        this.someMemberValue = 15;
        this.somePrivateValue = false;
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
```

```
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }
    SomeClass.getGoodbye = function () {
        return "goodbye!";
    };
    SomeClass.prototype.getFortyTwo = function () {
        return 42;
    };
    SomeClass.prototype.getTrue = function () {
        return true;
    };
    return SomeClass;
}());
SomeClass.SomeStaticValue = "hello";
```

Observations

- La modification du prototype de la classe est enveloppée dans un IIFE .
- Les variables membres sont définies dans la `function` classe principale.
- Les propriétés statiques sont ajoutées directement à l'objet de classe, tandis que les propriétés d'instance sont ajoutées au prototype.

Lire Des classes en ligne: https://riptutorial.com/fr/typescript/topic/1560/des-classes

Chapitre 7: Enums

Examples

Comment obtenir toutes les valeurs énumérées

```
enum SomeEnum { A, B }

let enumValues:Array<string>= [];

for(let value in SomeEnum) {
    if(typeof SomeEnum[value] === 'number') {
        enumValues.push(value);
    }
}

enumValues.forEach(v=> console.log(v))
//A
//B
```

Enums avec des valeurs explicites

Par défaut, toutes les valeurs `enum` sont résolues en nombres. Disons si vous avez quelque chose comme

```
enum MimeType {
  JPEG,
  PNG,
  PDF
}
```

la valeur réelle derrière, par exemple, `MimeType.PDF` sera `2` .

Mais parfois, il est important que l'énumération se résout à un autre type. Par exemple, vous recevez la valeur de backend / frontend / un autre système qui est définitivement une chaîne. Cela pourrait être une douleur, mais heureusement il y a cette méthode:

```
enum MimeType {
  JPEG = <any>'image/jpeg',
  PNG = <any>'image/png',
  PDF = <any>'application/pdf'
}
```

Cela résout le `MimeType.PDF` à l' `application/pdf` .

Depuis TypeScript 2.4, il est possible de déclarer des énumérations de chaînes :

```
enum MimeType {
  JPEG = 'image/jpeg',
  PNG = 'image/png',
```

```
  PDF = 'application/pdf',
}
```

Vous pouvez explicitement fournir des valeurs numériques en utilisant la même méthode

```
enum MyType {
   Value = 3,
   ValueEx = 30,
   ValueEx2 = 300
}
```

Les types plus fantaisistes fonctionnent également, puisque les énumérations non-const sont des objets réels à l'exécution, par exemple

```
enum FancyType {
   OneArr = <any>[1],
   TwoArr = <any>[2, 2],
   ThreeArr = <any>[3, 3, 3]
}
```

devient

```
var FancyType;
(function (FancyType) {
    FancyType[FancyType["OneArr"] = [1]] = "OneArr";
    FancyType[FancyType["TwoArr"] = [2, 2]] = "TwoArr";
    FancyType[FancyType["ThreeArr"] = [3, 3, 3]] = "ThreeArr";
})(FancyType || (FancyType = {}));
```

Mise en œuvre personnalisée: étend pour les énumérations

Parfois, il est nécessaire d'implémenter Enum seul. Par exemple, il n'y a pas de moyen clair d'étendre d'autres énumérations. L'implémentation personnalisée permet ceci:

```
class Enum {
  constructor(protected value: string) {}

  public toString() {
    return String(this.value);
  }

  public is(value: Enum | string) {
    return this.value = value.toString();
  }
}

class SourceEnum extends Enum {
  public static value1 = new SourceEnum('value1');
  public static value2 = new SourceEnum('value2');
}

class TestEnum extends SourceEnum {
  public static value3 = new TestEnum('value3');
  public static value4 = new TestEnum('value4');
```

```
}

function check(test: TestEnum) {
  return test === TestEnum.value2;
}

let value1 = TestEnum.value1;

console.log(value1 + 'hello');
console.log(value1.toString() === 'value1');
console.log(value1.is('value1'));
console.log(!TestEnum.value3.is(TestEnum.value3));
console.log(check(TestEnum.value2));
// this works but perhaps your TSLint would complain
// attention! does not work with ===
// use .is() instead
console.log(TestEnum.value1 == <any>'value1');
```

Extension des énumérations sans implémentation d'énumération personnalisée

```
enum SourceEnum {
  value1 = <any>'value1',
  value2 = <any>'value2'
}

enum AdditionToSourceEnum {
  value3 = <any>'value3',
  value4 = <any>'value4'
}

// we need this type for TypeScript to resolve the types correctly
type TestEnumType = SourceEnum | AdditionToSourceEnum;
// and we need this value "instance" to use values
let TestEnum = Object.assign({}, SourceEnum, AdditionToSourceEnum);
// also works fine the TypeScript 2 feature
// let TestEnum = { ...SourceEnum, ...AdditionToSourceEnum };

function check(test: TestEnumType) {
  return test === TestEnum.value2;
}

console.log(TestEnum.value1);
console.log(TestEnum.value2 === <any>'value2');
console.log(check(TestEnum.value2));
console.log(check(TestEnum.value3));
```

Lire Enums en ligne: https://riptutorial.com/fr/typescript/topic/4954/enums

Chapitre 8: Exemples de base de texte

Remarques

Ceci est un exemple de base qui étend une classe de voiture générique et définit une méthode de description de voiture.

Trouvez plus d'exemples TypeScript ici - Exemples TypeScript GitRepo

Examples

1 exemple d'héritage de classe de base utilisant extend et super keyword

Une classe de voiture générique a des propriétés de voiture et une méthode de description

```
class Car{
    name:string;
    engineCapacity:string;

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

    describeCar(){
        console.log(`${this.name} car comes with ${this.engineCapacity} displacement`);
    }
}

new Car("maruti ciaz","1500cc").describeCar();
```

HondaCar étend la classe de voiture générique existante et ajoute une nouvelle propriété.

```
class HondaCar extends Car{
    seatingCapacity:number;

    constructor(name:string,engineCapacity:string,seatingCapacity:number){
        super(name,engineCapacity);
        this.seatingCapacity=seatingCapacity;
    }

    describeHondaCar(){
        super.describeCar();
        console.log(`this cars comes with seating capacity of ${this.seatingCapacity}`);
    }
}
new HondaCar("honda jazz","1200cc",4).describeHondaCar();
```

2 exemple de variable de classe statique - nombre de fois que la méthode est appelée

ici countInstance est une variable de classe statique

```
class StaticTest{
    static countInstance : number= 0;
    constructor(){
        StaticTest.countInstance++;
    }
}

new StaticTest();
new StaticTest();
console.log(StaticTest.countInstance);
```

Lire Exemples de base de texte en ligne: https://riptutorial.com/fr/typescript/topic/7721/exemples-de-base-de-texte

Chapitre 9: Gardes de type définis par l'utilisateur

Syntaxe

- typeof x === "nom du type"
- x instanceof TypeName
- function (foo: any): foo est TypeName {/ * code retournant booléen * /}

Remarques

L'utilisation d'annotations de type dans TypeScript contraint les types possibles avec lesquels votre code devra traiter, mais il est toujours nécessaire de prendre des chemins de code différents en fonction du type d'exécution d'une variable.

Les gardes de type vous permettent d'écrire du code discriminant en fonction du type d'exécution d'une variable, tout en restant fortement typé et en évitant les conversions (également appelées assertions de type).

Examples

Utiliser instanceof

`instanceof` nécessite que la variable soit de type `any` .

Ce code (essayez-le):

```
class Pet { }
class Dog extends Pet {
    bark() {
        console.log("woof");
    }
}
class Cat extends Pet {
    purr() {
        console.log("meow");
    }
}

function example(foo: any) {
    if (foo instanceof Dog) {
        // foo is type Dog in this block
        foo.bark();
    }

    if (foo instanceof Cat) {
        // foo is type Cat in this block
        foo.purr();
```

```
    }
}

example(new Dog());
example(new Cat());
```

estampes

```
woof
meom
```

à la console.

En utilisant typeof

`typeof` est utilisé lorsque vous devez distinguer les types `number` , `string` , `boolean` et `symbol` . Les autres constantes de chaîne ne seront pas erronées, mais ne seront pas non plus utilisées pour restreindre les types.

Contrairement à `instanceof` , `typeof` fonctionnera avec une variable de tout type. Dans l'exemple ci-dessous, `foo` peut être saisi en tant que `number | string` sans issue.

Ce code (essayez-le):

```
function example(foo: any) {
    if (typeof foo === "number") {
        // foo is type number in this block
        console.log(foo + 100);
    }

    if (typeof foo === "string") {
        // fooi is type string in this block
        console.log("not a number: " + foo);
    }
}

example(23);
example("foo");
```

estampes

```
123
not a number: foo
```

Fonctions de protection de type

Vous pouvez déclarer des fonctions qui servent de gardes de type en utilisant la logique de votre choix.

Ils prennent la forme:

```
function functionName(variableName: any): variableName is DesiredType {
```

```
    // body that returns boolean
}
```

Si la fonction renvoie true, TypeScript restreindra le type à `DesiredType` dans tout bloc protégé par un appel à la fonction.

Par exemple (essayez-le):

```
function isString(test: any): test is string {
    return typeof test === "string";
}

function example(foo: any) {
    if (isString(foo)) {
        // foo is type as a string in this block
        console.log("it's a string: " + foo);
    } else {
        // foo is type any in this block
        console.log("don't know what this is! [" + foo + "]");
    }
}

example("hello world");          // prints "it's a string: hello world"
example({ something: "else" });  // prints "don't know what this is! [[object Object]]"
```

Le prédicat de type de fonction d'un garde (le `foo is Bar` dans la position du type de retour de fonction) est utilisé au moment de la compilation pour restreindre les types, le corps de la fonction est utilisé lors de l'exécution. Le prédicat de type et la fonction doivent être compatibles ou votre code ne fonctionnera pas.

Les fonctions de type guard n'ont pas à utiliser `typeof` ou `instanceof` , elles peuvent utiliser une logique plus compliquée.

Par exemple, ce code détermine si vous avez un objet jQuery en vérifiant sa chaîne de version.

```
function isJQuery(foo): foo is JQuery {
    // test for jQuery's version string
    return foo.jquery !== undefined;
}

function example(foo) {
    if (isJQuery(foo)) {
        // foo is typed JQuery here
        foo.eq(0);
    }
}
```

Lire Gardes de type définis par l'utilisateur en ligne:
https://riptutorial.com/fr/typescript/topic/8034/gardes-de-type-definis-par-l-utilisateur

Chapitre 10: Génériques

Syntaxe

- Les types génériques déclarés dans les crochets triangulaires: `<T>`
- La contrainte des types génériques se fait avec le mot-clé extend: `<T extends Car>`

Remarques

Les paramètres génériques ne sont pas disponibles à l'exécution, ils ne sont que pour la compilation. Cela signifie que vous ne pouvez pas faire quelque chose comme ça:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (T instanceof Executable1) {    // Compilation error
            ...
        } else if (U instanceof Executable2){    // Compilation error
            ...
        }
    }
}
```

Cependant, les informations sur les classes sont toujours conservées. Vous pouvez donc toujours tester le type d'une variable, car vous avez toujours été capable de:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (executable instanceof Executable1) {
            ...
        } else if (executable instanceof Executable2){
            ...
        } // But in this method, since there is no parameter of type `U` it is non-sensical to
ask about U's "type"
    }
}
```

Examples

Interfaces Génériques

Déclarer une interface générique

```
interface IResult<T> {
    wasSuccessfull: boolean;
    error: T;
}
```

```
var result: IResult<string> = ....
var error: string = result.error;
```

Interface générique avec plusieurs paramètres de type

```
interface IRunnable<T, U> {
    run(input: T): U;
}

var runnable: IRunnable<string, number> = ...
var input: string;
var result: number = runnable.run(input);
```

Implémenter une interface générique

```
interface IResult<T>{
    wasSuccessfull: boolean;
    error: T;

    clone(): IResult<T>;
}
```

Implémentez-le avec la classe générique:

```
class Result<T> implements IResult<T> {
    constructor(public result: boolean, public error: T) {
    }

    public clone(): IResult<T> {
        return new Result<T>(this.result, this.error);
    }
}
```

Implémentez-le avec une classe non générique:

```
class StringResult implements IResult<string> {
    constructor(public result: boolean, public error: string) {
    }

    public clone(): IResult<string> {
        return new StringResult(this.result, this.error);
    }
}
```

Classe générique

```
class Result<T> {
```

```
    constructor(public wasSuccessful: boolean, public error: T) {
    }

    public clone(): Result<T> {
       ...
    }
}

let r1 = new Result(false, 'error: 42');  // Compiler infers T to string
let r2 = new Result(false, 42);           // Compiler infers T to number
let r3 = new Result<string>(true, null);  // Explicitly set T to string
let r4 = new Result<string>(true, 4);     // Compilation error because 4 is not a string
```

Contraintes génériques

Contrainte simple:

```
interface IRunnable {
    run(): void;
}

interface IRunner<T extends IRunnable> {
    runSafe(runnable: T): void;
}
```

Contrainte plus complexe:

```
interface IRunnble<U> {
    run(): U;
}

interface IRunner<T extends IRunnable<U>, U> {
    runSafe(runnable: T): U;
}
```

Encore plus complexe:

```
interface IRunnble<V> {
    run(parameter: U): V;
}

interface IRunner<T extends IRunnable<U, V>, U, V> {
    runSafe(runnable: T, parameter: U): V;
}
```

Contraintes de type en ligne:

```
interface IRunnable<T extends { run(): void }> {
    runSafe(runnable: T): void;
}
```

Fonctions génériques

Dans les interfaces:

```
interface IRunner {
    runSafe<T extends IRunnable>(runnable: T): void;
}
```

En cours:

```
class Runner implements IRunner {

    public runSafe<T extends IRunnable>(runnable: T): void {
        try {
            runnable.run();
        } catch(e) {
        }
    }

}
```

Fonctions simples:

```
function runSafe<T extends IRunnable>(runnable: T): void {
    try {
        runnable.run();
    } catch(e) {
    }
}
```

Utilisation de classes et de fonctions génériques:

Créez une instance de classe générique:

```
var stringRunnable = new Runnable<string>();
```

Exécuter la fonction générique:

```
function runSafe<T extends Runnable<U>, U>(runnable: T);

// Specify the generic types:
runSafe<Runnable<string>, string>(stringRunnable);

// Let typescript figure the generic types by himself:
runSafe(stringRunnable);
```

Tapez les paramètres en tant que contraintes

Avec TypeScript 1.8, il devient possible pour une contrainte de paramètre de type de faire référence à des paramètres de type de la même liste de paramètres de type. Auparavant, c'était une erreur.

```
 function assign<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = source[id];
    }
```

```
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 });  // Error
```

Lire Génériques en ligne: https://riptutorial.com/fr/typescript/topic/2132/generiques

Chapitre 11: Importer des bibliothèques externes

Syntaxe

- `import {component} from 'libName'; // Will import the class "component"`
- `import {component as c} from 'libName'; // Will import the class "component" into a "c" object`
- `import component from 'libname'; // Will import the default export from libName`
- `import * as lib from 'libName'; // Will import everything from libName into a "lib" object`
- `import lib = require('libName'); // Will import everything from libName into a "lib" object`
- `const lib: any = require('libName'); // Will import everything from libName into a "lib" object`
- `import 'libName'; // Will import libName module for its side effects only`

Remarques

Il peut sembler que la syntaxe

```
import * as lib from 'libName';
```

et

```
import lib = require('libName');
```

sont la même chose, mais ils ne sont pas!

Considérons que nous voulons importer une classe **Personne** exportée avec `export =` syntax spécifique à TypeScript:

```
class Person {
...
}
export = Person;
```

Dans ce cas, il n'est pas possible de l'importer avec la syntaxe es6 (nous aurions une erreur à la compilation), vous devez utiliser la syntaxe `import =` spécifique à TypeScript.

```
import * as Person from 'Person';  //compile error
import Person = require('Person');  //OK
```

L'inverse est vrai: les modules classiques peuvent être importés avec la deuxième syntaxe, de sorte que, d'une certaine manière, la dernière syntaxe est plus puissante puisqu'elle est capable d'importer toutes les exportations.

Pour plus d'informations, voir la documentation officielle .

Examples

Importer un module à partir de npm

Si vous avez un fichier de définition de type (d.ts) pour le module, vous pouvez utiliser une instruction d' `import` .

```
import _ = require('lodash');
```

Si vous ne disposez pas d'un fichier de définition pour le module, TypeScript générera une erreur lors de la compilation car il ne peut pas trouver le module que vous essayez d'importer.

Dans ce cas, vous pouvez importer le module avec la durée normale `require` fonction. Cela le retourne comme `any` type, cependant.

```
// The _ variable is of type any, so TypeScript will not perform any type checking.
const _: any = require('lodash');
```

À partir de TypeScript 2.0, vous pouvez également utiliser une *déclaration abrégée de module d'ambiance* afin d'indiquer à TypeScript qu'un module existe lorsque vous n'avez pas de fichier de définition de type pour le module. TypeScript ne sera cependant pas en mesure de fournir une vérification de type significative dans ce cas.

```
declare module "lodash";

// you can now import from lodash in any way you wish:
import { flatten } from "lodash";
import * as _ from "lodash";
```

A partir de TypeScript 2.1, les règles ont été encore assouplies. Maintenant, tant qu'un module existe dans votre répertoire `node_modules` , TypeScript vous permettra de l'importer, même sans déclaration de module. (Notez que si vous utilisez l'option de compilation `--noImplicitAny` le `--noImplicitAny` ci-dessous générera toujours un avertissement.)

```
// Will work if `node_modules/someModule/index.js` exists, or if
`node_modules/someModule/package.json` has a valid "main" entry point
import { foo } from "someModule";
```

Recherche de fichiers de définition

pour dactylographier 2.x:

les définitions de DefinitelyTyped sont disponibles via le package @types npm

```
npm i --save lodash
npm i --save-dev @types/lodash
```

mais dans le cas où vous souhaitez utiliser des types d'autres repos, vous pouvez utiliser

l'ancienne méthode:

pour typecript 1.x:

Typings est un package npm qui peut installer automatiquement des fichiers de définition de type dans un projet local. Je vous recommande de lire le quickstart .

```
npm install -global typings
```

Maintenant, nous avons accès aux types de cli.

1. La première étape consiste à rechercher le package utilisé par le projet.

```
typings search lodash
NAME              SOURCE HOMEPAGE                                          DESCRIPTION
VERSIONS UPDATED
lodash            dt     http://lodash.com/                                          2
2016-07-20T00:13:09.000Z
lodash            global                                                             1
2016-07-01T20:51:07.000Z
lodash            npm    https://www.npmjs.com/package/lodash                        1
2016-07-01T20:51:07.000Z
```

2. Ensuite, décidez de la source à installer. J'utilise dt qui signifie DefinitelyTyped un repo GitHub où la communauté peut éditer les typages, c'est aussi normalement le plus récemment mis à jour.

3. Installer les fichiers de saisie

```
typings install dt~lodash --global --save
```

Décomposons la dernière commande. Nous installons la version DefinitelyTyped de lodash en tant que fichier de typage global dans notre projet et l'enregistrons en tant que dépendance dans le `typings.json` . Maintenant, où que nous importions lodash, typecript chargera le fichier typage lodash.

4. Si nous voulons installer des typages qui ne seront utilisés que pour l'environnement de développement, nous pouvons fournir le drapeau `--save-dev` :

```
typings install chai --save-dev
```

Utiliser des bibliothèques externes globales sans typage

Bien que les modules soient idéaux, si la bibliothèque que vous utilisez est référencée par une variable globale (telle que $ ou _), car elle est chargée par une balise de `script` , vous pouvez créer une déclaration ambiante pour y faire référence:

```
declare const _: any;
```

Recherche de fichiers de définition avec les typescript 2.x

Avec les versions 2.x de typescript, les types sont maintenant disponibles dans le dépôt npm @types . Celles-ci sont automatiquement résolues par le compilateur typecript et sont beaucoup plus simples à utiliser.

Pour installer une définition de type, installez-la simplement en tant que dépendance de dev dans vos projets package.json

par exemple

```
npm i -S lodash
npm i -D @types/lodash
```

après l'installation, vous utilisez simplement le module comme avant

```
import * as _ from 'lodash'
```

Lire Importer des bibliothèques externes en ligne:
https://riptutorial.com/fr/typescript/topic/1542/importer-des-bibliotheques-externes

Chapitre 12: Intégration avec les outils de construction

Remarques

Pour plus d'informations, vous pouvez aller sur la page Web officielle dactylographiée intégrant des outils de construction

Examples

Installer et configurer webpack + loaders

Installation

```
npm install -D webpack typescript ts-loader
```

webpack.config.js

```
module.exports = {
  entry: {
    app: ['./src/'],
  },
  output: {
    path: __dirname,
    filename: './dist/[name].js',
  },
  resolve: {
    extensions: ['', '.js', '.ts'],
  },
  module: {
    loaders: [{
      test: /\.ts(x)$/, loaders: ['ts-loader'], exclude: /node_modules/
    }],
  }
};
```

Naviguer

Installer

```
npm install tsify
```

Utilisation de l'interface de ligne de

commande

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Utiliser l'API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
  .add("main.ts")
  .plugin("tsify", { noImplicitAny: true })
  .bundle()
  .pipe(process.stdout);
```

Plus de détails: smrq / tsify

Grognement

Installer

```
npm install grunt-ts
```

Basic Gruntfile.js

```
module.exports = function(grunt) {
    grunt.initConfig({
        ts: {
            default : {
                src: ["**/*.ts", "!node_modules/**/*.ts"]
            }
        }
    });
    grunt.loadNpmTasks("grunt-ts");
    grunt.registerTask("default", ["ts"]);
};
```

Plus de détails: TypeStrong / grunt-ts

Gorgée

Installer

```
npm install gulp-typescript
```

Gulpfile.js de base

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
    var tsResult = gulp.src("src/*.ts")
        .pipe(ts({
              noImplicitAny: true,
              out: "output.js"
        }));
    return tsResult.js.pipe(gulp.dest("built/local"));
});
```

gulpfile.js utilisant un tsconfig.json existant

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

var tsProject = ts.createProject('tsconfig.json', {
    noImplicitAny: true // You can add and overwrite parameters here
});

gulp.task("default", function () {
    var tsResult = tsProject.src()
        .pipe(tsProject());
    return tsResult.js.pipe(gulp.dest('release'));
});
```

Plus de détails: ivogabe / gulp-typescript

Webpack

Installer

```
npm install ts-loader --save-dev
```

Webpack.config.js de base

webpack 2.x, 3.x

```
module.exports = {
    resolve: {
```

```
        extensions: ['.ts', '.tsx', '.js']
    },
    module: {
        rules: [
            {
                // Set up ts-loader for .ts/.tsx files and exclude any imports from
node_modules.
                test: /\.tsx?$/,
                loaders: ['ts-loader'],
                exclude: /node_modules/
            }
        ]
    },
    entry: [
        // Set index.tsx as application entry point.
        './index.tsx'
    ],
    output: {
      filename: "bundle.js"
    }
};
```

webpack 1.x

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
        filename: "bundle.js"
    },
    resolve: {
        // Add '.ts' and '.tsx' as a resolvable extension.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },
    module: {
        loaders: [
            // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
            { test: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
        ]
    }
}
```

Voir plus de détails sur ts-loader ici

Alternatives:

- chargeur-dactylographié génial

MSBuild

Mettez à jour le fichier de projet pour inclure les fichiers `Microsoft.TypeScript.Default.props` (en haut) et `Microsoft.TypeScript.targets` (en bas) installés localement:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

```
  <!-- Include default props at the bottom -->
  <Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft


Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript
/>

  <!-- TypeScript configurations go here -->
  <PropertyGroup Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Include default targets at the bottom -->
  <Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft


Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript
/>
</Project>
```

Plus de détails sur la définition des options du compilateur MSBuild: Définition des options du compilateur dans les projets MSBuild

NuGet

- Clic droit -> Gérer les paquets NuGet
- Rechercher `Microsoft.TypeScript.MSBuild`
- Hit `Install`
- Une fois l'installation terminée, reconstruisez!

Plus de détails peuvent être trouvés sur la boîte de dialogue Gestionnaire de paquets et l' utilisation de builds de nuit avec NuGet

Lire Intégration avec les outils de construction en ligne: https://riptutorial.com/fr/typescript/topic/2860/integration-avec-les-outils-de-construction

Chapitre 13: Interfaces

Introduction

Une interface spécifie une liste de champs et de fonctions pouvant être attendus sur toute classe implémentant l'interface. À l'inverse, une classe ne peut implémenter une interface que si elle possède tous les champs et toutes les fonctions spécifiés sur l'interface.

Le principal avantage de l'utilisation des interfaces est qu'il permet d'utiliser des objets de différents types de manière polymorphe. C'est parce que toute classe implémentant l'interface a au moins ces champs et fonctions.

Syntaxe

- interface NomInterface {
- parameterName: parameterType;
- optionalParameterName ?: parameterType;
- }

Remarques

Interfaces vs types d'alias

Les interfaces permettent de spécifier la forme d'un objet, par exemple pour un objet personne que vous pouvez spécifier

```
interface person {
    id?: number;
    name: string;
    age: number;
}
```

Cependant, que se passe-t-il si vous souhaitez, par exemple, représenter la manière dont une personne est stockée dans une base de données SQL? Étant donné que chaque entrée de base de données consiste en une ligne de forme `[string, string, number]` (donc un tableau de chaînes ou de nombres), vous ne pouvez pas représenter cela comme une forme d'objet, car la ligne n'a aucune *propriété* en tant que tel, c'est juste un tableau.

C'est une occasion où les types sont utiles. Au lieu de spécifier dans chaque fonction qui accepte une `function processRow(row: [string, string, number])` paramètre de ligne `function processRow(row: [string, string, number])` , vous pouvez créer un alias de type distinct pour une ligne, puis l'utiliser dans chaque fonction:

```
type Row = [string, string, number];
```

```
function processRow(row: Row)
```

Documentation de l'interface officielle

https://www.typescriptlang.org/docs/handbook/interfaces.html

Examples

Ajouter des fonctions ou des propriétés à une interface existante

Supposons que nous ayons une référence à la définition de type `JQuery` et que nous souhaitons l'étendre pour avoir des fonctions supplémentaires à partir d'un plug-in inclus et qui n'a pas de définition de type officielle. Nous pouvons facilement l'étendre en déclarant les fonctions ajoutées par le plug-in dans une déclaration d'interface distincte avec le même nom `JQuery` :

```
interface JQuery {
  pluginFunctionThatDoesNothing(): void;

  // create chainable function
  manipulateDOM(HTMLElement): JQuery;
}
```

Le compilateur fusionnera toutes les déclarations du même nom en un seul - voir la fusion des déclarations pour plus de détails.

Interface de classe

Déclarez `public` variables `public` et les méthodes de type dans l'interface pour définir comment un autre code dactylographié peut interagir avec lui.

```
interface ISampleClassInterface {
  sampleVariable: string;

  sampleMethod(): void;

  optionalVariable?: string;
}
```

Ici, nous créons une classe qui implémente l'interface.

```
class SampleClass implements ISampleClassInterface {
  public sampleVariable: string;
  private answerToLifeTheUniverseAndEverything: number;

  constructor() {
    this.sampleVariable = 'string value';
    this.answerToLifeTheUniverseAndEverything = 42;
  }
```

```
  public sampleMethod(): void {
    // do nothing
  }
  private answer(q: any): number {
    return this.answerToLifeTheUniverseAndEverything;
  }
}
```

L'exemple montre comment créer une interface `ISampleClassInterface` et une classe `SampleClass` qui `implements` l'interface.

Interface d'extension

Supposons que nous ayons une interface:

```
interface IPerson {
    name: string;
    age: number;

    breath(): void;
}
```

Et nous voulons créer une interface plus spécifique qui possède les mêmes propriétés que la personne, nous pouvons le faire en utilisant le mot `extends` clé extend:

```
interface IManager extends IPerson {
    managerId: number;

    managePeople(people: IPerson[]): void;
}
```

En outre, il est possible d'étendre plusieurs interfaces.

Utilisation d'interfaces pour appliquer des types

L'un des principaux avantages de Typescript est qu'il applique les types de données que vous transmettez à votre code pour éviter les erreurs.

Disons que vous faites une application de rencontre pour animaux de compagnie.

Vous avez cette fonction simple qui vérifie si deux animaux sont compatibles les uns avec les autres ...

```
checkCompatible(petOne, petTwo) {
  if (petOne.species === petTwo.species &&
      Math.abs(petOne.age - petTwo.age) <= 5) {
    return true;
  }
}
```

C'est du code complètement fonctionnel, mais il serait trop facile pour quelqu'un, surtout pour d'autres personnes travaillant sur cette application qui n'ont pas écrit cette fonction, de ne pas

savoir qu'ils sont censés lui transmettre des objets avec 'species' et 'age' Propriétés. Ils peuvent essayer par erreur `checkCompatible(petOne.species, petTwo.species)` , puis laisser les erreurs se produire lorsque la fonction essaie d'accéder à petOne.species.species ou petOne.species.age!

Une des manières d'empêcher que cela se produise est de spécifier les propriétés que nous voulons sur les paramètres de familier:

```
checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age:
number}) {
    //...
}
```

Dans ce cas, Typescript s'assurera que tout ce qui est passé à la fonction possède des propriétés 'species' et 'age' (si elles ont des propriétés supplémentaires), mais c'est une solution assez lourde, même avec seulement deux propriétés spécifiées. Avec les interfaces, il y a un meilleur moyen!

Tout d'abord, nous définissons notre interface:

```
interface Pet {
  species: string;
  age: number;
  //We can add more properties if we choose.
}
```

Il ne nous reste plus qu'à spécifier le type de nos paramètres en tant que nouvelle interface, comme ça ...

```
checkCompatible(petOne: Pet, petTwo: Pet) {
  //...
}
```

... et Typescript s'assurera que les paramètres transmis à notre fonction contiennent les propriétés spécifiées dans l'interface Pet!

Interfaces Génériques

Comme les classes, les interfaces peuvent également recevoir des paramètres polymorphes (aka Generics).

Déclaration de paramètres génériques sur les interfaces

```
interface IStatus<U> {
    code: U;
}

interface IEvents<T> {
    list: T[];
    emit(event: T): void;
    getAll(): T[];
```

```
}
```

Ici, vous pouvez voir que nos deux interfaces prennent des paramètres génériques, **T** et **U.**

Implémentation d'interfaces génériques

Nous allons créer une classe simple pour implémenter l'interface **IEvents** .

```
class State<T> implements IEvents<T> {

    list: T[];

    constructor() {
        this.list = [];
    }

    emit(event: T): void {
        this.list.push(event);
    }

    getAll(): T[] {
        return this.list;
    }

}
```

Créons des instances de notre classe **State** .

Dans notre exemple, la classe `State` gère un statut générique en utilisant `IStatus<T>` . De cette manière, l'interface `IEvent<T>` gérera également un `IStatus<T>` .

```
const s = new State<IStatus<number>>();

// The 'code' property is expected to be a number, so:
s.emit({ code: 200 }); // works
s.emit({ code: '500' }); // type error

s.getAll().forEach(event => console.log(event.code));
```

Ici, notre classe d' `State` est typée `ISatus<number>` .

```
const s2 = new State<IStatus<Code>>();

//We are able to emit code as the type Code
s2.emit({ code: { message: 'OK', status: 200 } });

s2.getAll().map(event => event.code).forEach(event => {
    console.log(event.message);
    console.log(event.status);
});
```

Notre classe d' `State` est typée `IStatus<Code>` . De cette façon, nous pouvons transmettre un type plus complexe à notre méthode d’émission.

Comme vous pouvez le voir, les interfaces génériques peuvent être un outil très utile pour le code statique.

Utilisation d'interfaces pour le polymorphisme

La principale raison d'utiliser des interfaces pour obtenir un polymorphisme et fournir aux développeurs la possibilité d'implémenter à leur manière à l'avenir en implémentant les méthodes de l'interface.

Supposons que nous ayons une interface et trois classes:

```
interface Connector{
    doConnect(): boolean;
}
```

C'est l'interface du connecteur. Maintenant, nous allons implémenter cela pour la communication Wifi.

```
export class WifiConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via wifi");
        console.log("Get password");
        console.log("Lease an IP for 24 hours");
        console.log("Connected");
        return true
    }

}
```

Ici, nous avons développé notre classe concrète nommée `WifiConnector` qui a sa propre implémentation. Ceci est maintenant tapez `Connector` .

Nous créons maintenant notre `System` doté d'un composant `Connector` . Cela s'appelle l'injection de dépendance.

```
export class System {
    constructor(private connector: Connector){ #inject Connector type
        connector.doConnect()
    }
}
```

`constructor(private connector: Connector)` cette ligne est très importante ici. `Connector` est une interface et doit avoir `doConnect()` . Comme `Connector` est une interface, cette classe `System` a beaucoup plus de flexibilité. Nous pouvons transmettre n'importe quel type qui a implémenté une interface de `Connector` . Dans le futur, le développeur obtient plus de flexibilité. Par exemple, le développeur veut maintenant ajouter le module de connexion Bluetooth:

```
export class BluetoothConnector implements Connector{

    public doConnect(): boolean{
```

```
        console.log("Connecting via Bluetooth");
        console.log("Pair with PIN");
        console.log("Connected");
        return true
    }

}
```

Voir que Wifi et Bluetooth ont sa propre implémentation. Il existe différentes manières de se connecter. Cependant, les deux ont donc implémenté Type `Connector` le sont maintenant Type `Connector` . Pour que nous puissions transmettre n'importe lequel de ceux-ci à la classe `System` tant que paramètre constructeur. C'est ce qu'on appelle le polymorphisme. La classe `System` ne sait plus si c'est Bluetooth / Wifi, même si nous pouvons ajouter un autre module de communication comme Inferade, Bluetooth5 et tout simplement en implémentant l'interface de `Connector` .

Cela s'appelle Duck typing . `Connector` type de `Connector` est maintenant dynamique car `doConnect()` est juste un espace réservé et le développeur l'implémente comme le sien.

si au `constructor(private connector: WifiConnector)` où `WifiConnector` est une classe concrète, que se passera-t-il? Ensuite `System` classe `System` ne couplera étroitement avec rien avec WifiConnector. Ici, l'interface a résolu notre problème par polymorphisme.

Implémentation implicite et forme d'objet

TypeScript supporte les interfaces, mais le compilateur génère du JavaScript, ce qui n'est pas le cas. Par conséquent, les interfaces sont effectivement perdues lors de l'étape de compilation. C'est pourquoi la vérification de type sur les interfaces repose sur la *forme* de l'objet - c'est-à-dire si l'objet prend en charge les champs et les fonctions de l'interface - et non sur le fait que l'interface soit réellement implémentée ou non.

```
interface IKickable {
  kick(distance: number): void;
}
class Ball {
  kick(distance: number): void {
    console.log("Kicked", distance, "meters!");
  }
}
let kickable: IKickable = new Ball();
kickable.kick(40);
```

Ainsi, même si `Ball IKickable` pas explicitement `IKickable` , une instance de `Ball` peut être affectée à (et manipulée comme) un `IKickable` , même si le type est spécifié.

Lire Interfaces en ligne: https://riptutorial.com/fr/typescript/topic/2023/interfaces

Chapitre 14: Le débogage

Introduction

Il existe deux manières d'exécuter et de déboguer TypeScript:

Transpile en JavaScript , exécute dans le noeud et utilise les mappages pour **créer un** lien vers les fichiers source TypeScript

ou

Exécuter directement TypeScript en utilisant ts-node

Cet article décrit les deux façons d'utiliser Visual Studio Code et WebStorm . Tous les exemples supposent que votre fichier principal est *index.ts* .

Examples

JavaScript avec SourceMaps dans le code Visual Studio

Dans l'ensemble `tsconfig.json`

```
"sourceMap": true,
```

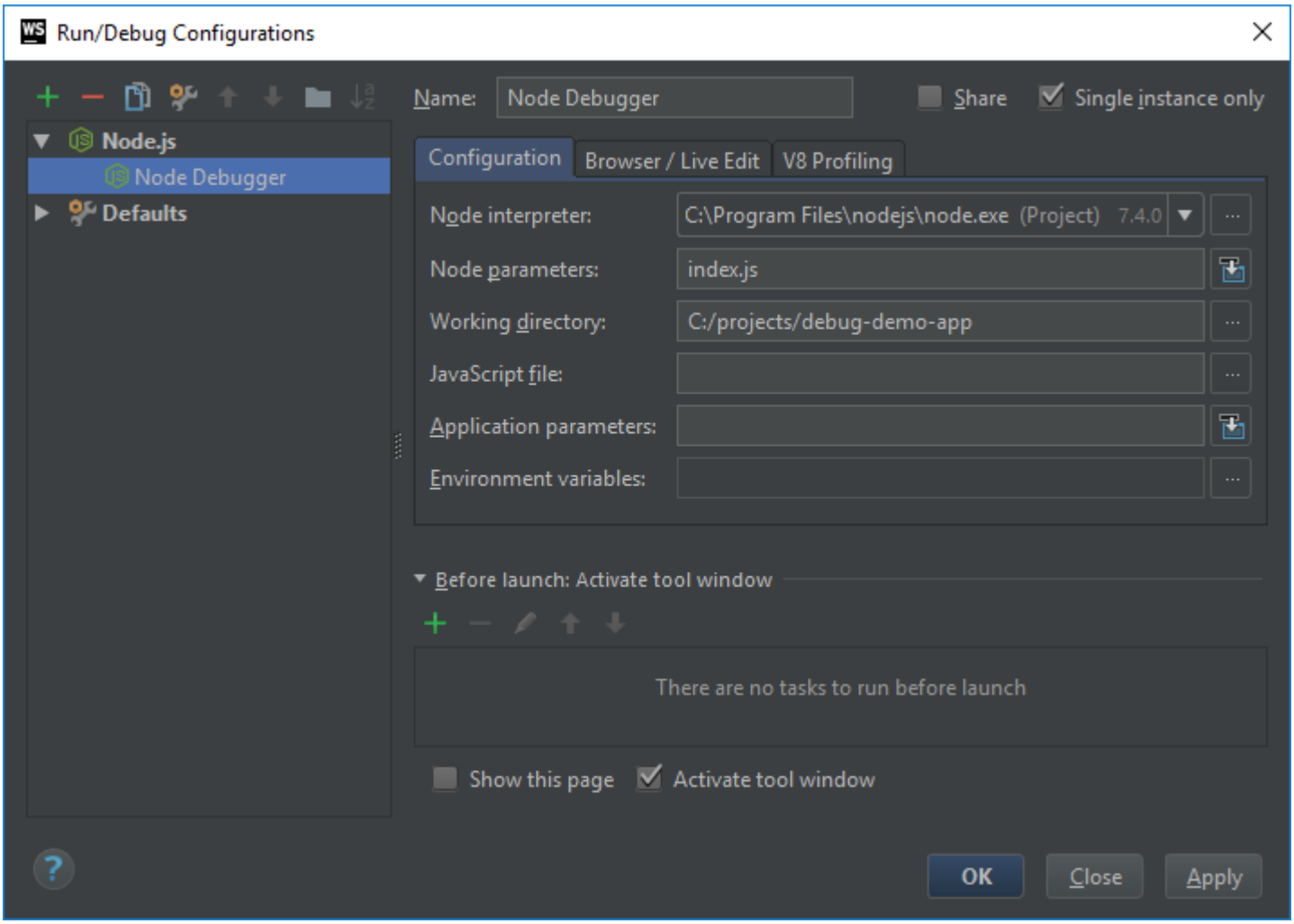
générer des mappages avec js-files à partir des sources TypeScript à l'aide de la commande `tsc` . Le fichier launch.json :

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${workspaceRoot}\\index.js",
            "cwd": "${workspaceRoot}",
            "outFiles": [],
            "sourceMaps": true
        }
    ]
}
```

Cela démarre le noeud avec le fichier index.js généré (si votre fichier principal est index.ts) et le débogueur dans Visual Studio Code qui s'arrête sur les points d'arrêt et résout les valeurs de variables dans votre code TypeScript.

JavaScript avec SourceMaps dans WebStorm

Créez une configuration de débogage *Node.js* et utilisez `index.js` comme *paramètre Node* .



TypeScript avec ts-node dans Visual Studio Code

Ajoutez ts-node à votre projet TypeScript:

```
npm i ts-node
```

Ajoutez un script à votre `package.json` :

```
"start:debug": "ts-node --inspect=5858 --debug-brk --ignore false index.ts"
```

Le `launch.json` doit être configuré pour utiliser le type *node2* et démarrer npm en exécutant le script `start:debug` :

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node2",
            "request": "launch",
            "name": "Launch Program",
            "runtimeExecutable": "npm",
            "windows": {
```

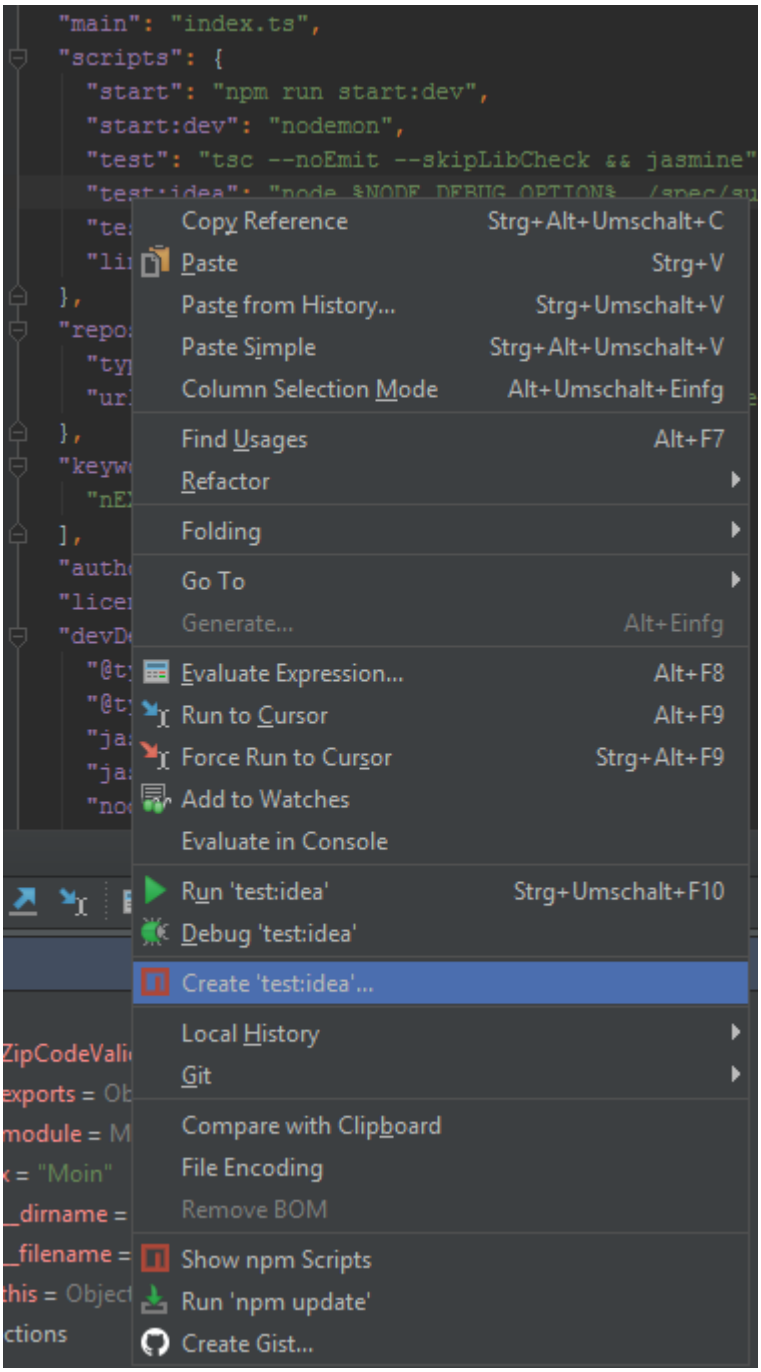
```
                "runtimeExecutable": "npm.cmd"
            },
            "runtimeArgs": [
                "run-script",
                "start:debug"
            ],
            "cwd": "${workspaceRoot}/server",
            "outFiles": [],
            "port": 5858,
            "sourceMaps": true
        }
    ]
}
```

TypeScript avec ts-node dans WebStorm

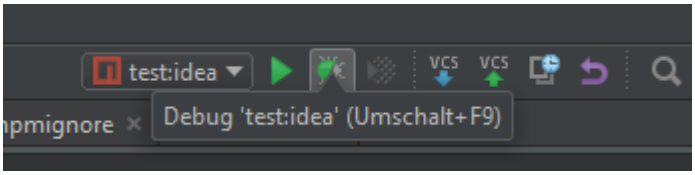
Ajoutez ce script à votre `package.json` :

```
"start:idea": "ts-node %NODE_DEBUG_OPTION% --ignore false index.ts",
```

Faites un clic droit sur le script et sélectionnez *Create 'test: idea' ...* et confirmez avec 'OK' pour créer la configuration de débogage:



Démarrez le débogueur en utilisant cette configuration:



Lire Le débogage en ligne: https://riptutorial.com/fr/typescript/topic/9131/le-debogage

Chapitre 15: Les fonctions

Remarques

Lien de documentation typographique pour les fonctions

Examples

Paramètres facultatifs et par défaut

Paramètres facultatifs

Dans TypeScript, chaque paramètre est supposé être requis par la fonction. Vous pouvez ajouter un **?** à la fin d'un nom de paramètre pour le définir comme facultatif.

Par exemple, le paramètre `lastName` de cette fonction est facultatif:

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

Les paramètres facultatifs doivent venir après tous les paramètres non facultatifs:

```
function buildName(firstName?: string, lastName: string) // Invalid
```

Paramètres par défaut

Si l'utilisateur passe `undefined` ou ne spécifie pas d'argument, la valeur par défaut sera attribuée. Ce sont les paramètres *initialisés par défaut* .

Par exemple, "Smith" est la valeur par défaut du paramètre `lastName` .

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
buildName('foo', 'bar');      // firstName == 'foo', lastName == 'bar'
buildName('foo');             // firstName == 'foo', lastName == 'Smith'
buildName('foo', undefined);  // firstName == 'foo', lastName == 'Smith'
```

Types de fonctions

Fonctions nommées

```
function multiply(a, b) {
    return a * b;
}
```

Fonctions anonymes

```
let multiply = function(a, b) { return a * b; };
```

Fonctions lambda / flèche

```
let multiply = (a, b) => { return a * b; };
```

Fonction comme paramètre

Supposons que nous voulons recevoir une fonction en tant que paramètre, nous pouvons le faire comme ceci:

```
function foo(otherFunc: Function): void {
    ...
}
```

Si on veut recevoir un constructeur en paramètre:

```
function foo(constructorFunc: { new() }) {
    new constructorFunc();
}

function foo(constructorWithParamsFunc: { new(num: number) }) {
    new constructorWithParamsFunc(1);
}
```

Ou pour faciliter la lecture, nous pouvons définir une interface décrivant le constructeur:

```
interface IConstructor {
    new();
}

function foo(contructorFunc: IConstructor) {
    new constructorFunc();
}
```

Ou avec des paramètres:

```
interface INumberConstructor {
    new(num: number);
}

function foo(contructorFunc: INumberConstructor) {
    new contructorFunc(1);
}
```

Même avec les génériques:

```
interface ITConstructor<T, U> {
    new(item: T): U;
```

```
}

function foo<T, U>(contructorFunc: ITConstructor<T, U>, item: T): U {
    return new contructorFunc(item);
}
```

Si nous voulons recevoir une fonction simple et non un constructeur, c'est presque pareil:

```
function foo(func: { (): void }) {
    func();
}

function foo(constructorWithParamsFunc: { (num: number): void }) {
    new constructorWithParamsFunc(1);
}
```

Ou pour faciliter la lecture, nous pouvons définir une interface décrivant la fonction:

```
interface IFunction {
    (): void;
}

function foo(func: IFunction ) {
    func();
}
```

Ou avec des paramètres:

```
interface INumberFunction {
    (num: number): string;
}

function foo(func: INumberFunction ) {
    func(1);
}
```

Même avec les génériques:

```
interface ITFunc<T, U> {
    (item: T): U;
}

function foo<T, U>(contructorFunc: ITFunc<T, U>, item: T): U {
    return func(item);
}
```

Fonctions avec types d'union

Une fonction TypeScript peut prendre en compte les paramètres de plusieurs types prédéfinis à l'aide de types d'union.

```
function whatTime(hour:number|string, minute:number|string):string{
    return hour+':'+minute;
```

```
}

whatTime(1,30)          //'1:30'
whatTime('1',30)        //'1:30'
whatTime(1,'30')        //'1:30'
whatTime('1','30')      //'1:30'
```

Typescript traite ces paramètres comme un type unique qui est une union des autres types, de sorte que votre fonction doit être capable de gérer les paramètres de tout type qui se trouve dans l'union.

```
function addTen(start:number|string):number{
    if(typeof number === 'string'){
        return parseInt(number)+10;
    }else{
        else return number+10;
    }
}
```

Lire Les fonctions en ligne: https://riptutorial.com/fr/typescript/topic/1841/les-fonctions

Chapitre 16: Mixins

Syntaxe

- classe BeetleGuy implémente Climbs, Bulletproof {}
- appliquerMixines (BeetleGuy, [Climbs, Bulletproof]);

Paramètres

Paramètre	La description
dérivéCtor	La classe que vous souhaitez utiliser comme classe de composition
baseCtors	Un tableau de classes à ajouter à la classe de composition

Remarques

Il y a trois règles à prendre en compte avec les mixins:

- Vous utilisez le mot-clé `implements` , pas le mot `extends` clé `extends` lorsque vous écrivez votre classe de composition
- Vous devez avoir une signature correspondante pour que le compilateur reste silencieux (mais il ne nécessite aucune implémentation réelle: il le récupérera depuis le mixin).
- Vous devez appeler `applyMixins` avec les arguments corrects.

Examples

Exemple de mixins

Pour créer des mixins, déclarez simplement des classes légères pouvant être utilisées comme "comportements".

```
class Flies {
    fly() {
        alert('Is it a bird? Is it a plane?');
    }
}

class Climbs {
    climb() {
        alert('My spider-sense is tingling.');
    }
}

class Bulletproof {
    deflect() {
        alert('My wings are a shield of steel.');
```

```
    }
}
```

Vous pouvez ensuite appliquer ces comportements à une classe de composition:

```
class BeetleGuy implements Climbs, Bulletproof {
        climb: () => void;
        deflect: () => void;
}
applyMixins (BeetleGuy, [Climbs, Bulletproof]);
```

La fonction `applyMixins` est nécessaire pour effectuer le travail de composition.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
             if (name !== 'constructor') {
                derivedCtor.prototype[name] = baseCtor.prototype[name];
            }
        });
    });
}
```

Lire Mixins en ligne: https://riptutorial.com/fr/typescript/topic/4727/mixins

Chapitre 17: Modules - exportation et importation

Examples

Bonjour tout le monde

```
//hello.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
function helloES(name: string){
    console.log(`Hola ${name}!`);
}
export {helloES};
export default hello;
```

Charger à l'aide de l'index du répertoire

Si le répertoire contient le fichier nommé `index.ts` il peut être chargé en utilisant uniquement le nom du répertoire (pour le fichier `index.ts` fichier est facultatif).

```
//welcome/index.ts
export function welcome(name: string){
    console.log(`Welcome ${name}!`);
}
```

Exemple d'utilisation de modules définis

```
import {hello, helloES} from "./hello";  // load specified elements
import defaultHello from "./hello";      // load default export into name defaultHello
import * as Bundle from "./hello";       // load all exports as Bundle
import {welcome} from "./welcome";       // note index.ts is omitted

hello("World");                          // Hello World!
helloES("Mundo");                        // Hola Mundo!
defaultHello("World");                   // Hello World!

Bundle.hello("World");                   // Hello World!
Bundle.helloES("Mundo");                 // Hola Mundo!

welcome("Human");                        // Welcome Human!
```

Exportation / importation de déclarations

Toute déclaration (variable, const, fonction, classe, etc.) peut être exportée à partir du module à importer dans un autre module.

Typecript offre deux types d'export: nommé et default.

Exportation nommée

```
// adams.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
export const unused = 0;
```

Lors de l'importation d'exportations nommées, vous pouvez spécifier les éléments à importer.

```
import {hello, answerToLifeTheUniverseAndEverything} from "./adams";
hello(answerToLifeTheUniverseAndEverything);   // Hello 42!
```

Export par défaut

Chaque module peut avoir une exportation par défaut

```
// dent.ts
const defaultValue = 54;
export default defaultValue;
```

qui peuvent être importés en utilisant

```
import dentValue from "./dent";
console.log(dentValue);        // 54
```

Importation groupée

Typecript propose une méthode pour importer un module entier dans une variable

```
// adams.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
```

```
import * as Bundle from "./adams";
Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything);  // Hello 42!
console.log(Bundle.unused);                                 // 0
```

Réexporter

Les caractères dactylographiés permettent de réexporter les déclarations.

```
//Operator.ts
interface Operator {
    eval(a: number, b: number): number;
}
export default Operator;
```

```
//Add.ts
import Operator from "./Operator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return a + b;
    }
}
```

```
//Mul.ts
import Operator from "./Operator";
export class Mul implements Operator {
    eval(a: number, b: number): number {
        return a * b;
    }
}
```

Vous pouvez regrouper toutes les opérations dans une seule bibliothèque

```
//Operators.ts
import {Add} from "./Add";
import {Mul} from "./Mul";

export {Add, Mul};
```

Les déclarations nommées peuvent être réexportées en utilisant une syntaxe plus courte

```
//NamedOperators.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
```

Les exportations par défaut peuvent également être exportées, mais aucune syntaxe courte n'est disponible. N'oubliez pas qu'une seule exportation par défaut par module est possible.

```
//Calculator.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
import Operator from "./Operator";

export default Operator;
```

Possible est la réexportation de l' **importation groupée**

```
//RepackedCalculator.ts
export * from "./Operators";
```

Lors de la réexportation d'un ensemble, les déclarations peuvent être remplacées lorsqu'elles sont déclarées explicitement.

```
//FixedCalculator.ts
export * from "./Calculator"
import Operator from "./Calculator";
export class Add implements Operator {
    eval(a: number, b: number): number {
```

```
        return 42;
    }
}
```

Exemple d'utilisation

```
//run.ts
import {Add, Mul} from "./FixedCalculator";

const add = new Add();
const mul = new Mul();

console.log(add.eval(1, 1)); // 42
console.log(mul.eval(3, 4)); // 12
```

Lire Modules - exportation et importation en ligne:
https://riptutorial.com/fr/typescript/topic/9054/modules---exportation-et-importation

Chapitre 18: Pourquoi et quand utiliser TypeScript

Introduction

Si vous trouvez que les arguments pour les systèmes de caractères sont convaincants en général, alors vous serez satisfait de TypeScript.

Il apporte de nombreux avantages de type système (sécurité, lisibilité, outils améliorés) à l'écosystème JavaScript. Il souffre également de certains inconvénients des systèmes de types (complexité et caractère incomplet).

Remarques

Les mérites des langages typés et non typés ont été débattus pendant des décennies. Les arguments pour les types statiques incluent:

1. Sécurité: les systèmes de type permettent de détecter de nombreuses erreurs sans exécuter le code. TypeScript peut être configuré pour permettre moins d'erreurs de programmation
2. Lisibilité: les types explicites facilitent la compréhension du code par l'homme. Comme Fred Brooks l'a écrit : «Montrez-moi vos organigrammes et cachez vos tables, et je continuerai à être mystifié. Montrez-moi vos tables et je n'aurai pas besoin de vos organigrammes, elles seront évidentes.
3. Tooling: les systèmes de type facilitent la compréhension du code par les ordinateurs. Cela permet aux outils comme les IDE et les linters d'être plus puissants.
4. Performances: les systèmes de types accélèrent le code en réduisant le besoin de vérifier le type à l'exécution.

La sortie de TypeScript étant indépendante de ses types , TypeScript n'a aucun impact sur les performances. L'argument pour utiliser TypeScript repose sur les trois autres avantages.

Les arguments contre les systèmes de type incluent:

1. Complexité ajoutée: les systèmes de type peuvent être plus complexes que le langage d'exécution qu'ils décrivent. Les fonctions d'ordre supérieur peuvent être faciles à mettre en œuvre correctement mais difficiles à taper . Traiter les définitions de type crée des obstacles supplémentaires à l'utilisation de bibliothèques externes.
2. Ajout de verbosité: les annotations de type peuvent ajouter du code à la chaîne, rendant la logique sous-jacente plus difficile à suivre.
3. Itération plus lente: en introduisant une étape de construction, TypeScript ralentit le cycle d'édition / sauvegarde / rechargement.
4. Incomplétude: Un système de type ne peut pas être à la fois sain et complet. Il existe des programmes corrects que TypeScript ne permet pas. Et les programmes acceptés par TypeScript peuvent toujours contenir des bogues. Un système de type ne réduit pas le

besoin de tests. Si vous utilisez TypeScript, vous devrez peut-être attendre plus longtemps pour utiliser les nouvelles fonctionnalités du langage ECMAScript.

TypeScript offre des moyens de résoudre tous ces problèmes:

1. Complexité supplémentaire Si tapant une partie d'un programme est trop difficile, tapuscrit peut être largement désactivé à l' aide d' un opaque `any` genre. La même chose est vraie pour les modules externes.
2. Ajout de verbosité. Cela peut être partiellement traité par des alias de type et la capacité de TypeScript à déduire des types. Ecrire du code clair est autant un art qu'une science: enlevez trop d'annotations de type et le code risque de ne plus être clair pour les lecteurs humains.
3. Itération lente: une étape de construction est relativement courante dans le développement JS moderne et TypeScript s'intègre déjà à la plupart des outils de construction . Et si TypeScript détecte une erreur à un stade précoce, il peut vous faire économiser tout un cycle d'itération!
4. Incomplétude Bien que ce problème ne puisse pas être complètement résolu, TypeScript a été capable de capturer de plus en plus de modèles JavaScript expressifs. Des exemples récents incluent l'ajout de types mappés dans TypeScript 2.1 et les mixins dans 2.2 .

Les arguments pour et contre les systèmes de type en général s'appliquent également à TypeScript. L'utilisation de TypeScript augmente le temps nécessaire pour démarrer un nouveau projet. Mais avec le temps, au fur et à mesure que le projet augmente de taille et gagne plus de contributeurs, l'espoir est que les avantages de l'utilisation (sécurité, lisibilité, outillage) deviennent plus forts et l'emportent sur les inconvénients. Cela se reflète dans la devise de TypeScript: "JavaScript qui évolue".

Examples

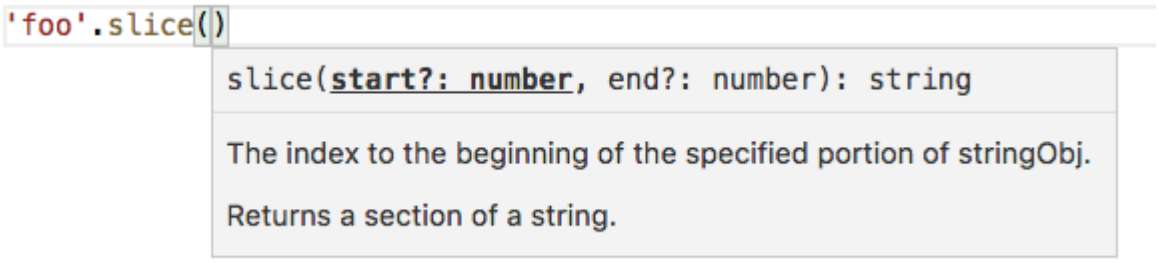
sécurité

TypeScript intercepte les erreurs de type au début de l'analyse statique:

```
function double(x: number): number {
  return 2 * x;
}
double('2');
//     ~~~ Argument of type '"2"' is not assignable to parameter of type 'number'.
```

Lisibilité

TypeScript permet aux éditeurs de fournir une documentation contextuelle:



Vous n'oublierez jamais si `String.prototype.slice` prend `(start, stop)` ou `(start, length)` nouveau!

Outillage

TypeScript permet aux éditeurs d'effectuer des refactorisations automatisées qui connaissent les règles des langages.

```
let foo = '123';

{
  const foo = (x: number) => {
    return 2 * x;
  }

  foo(2);
}
```

Ici, par exemple, Visual Studio Code est capable de renommer des références au `foo` interne sans modifier le `foo` externe. Cela serait difficile à faire avec un simple find / replace.

Lire Pourquoi et quand utiliser TypeScript en ligne:
https://riptutorial.com/fr/typescript/topic/9073/pourquoi-et-quand-utiliser-typescript

Chapitre 19: Publier des fichiers de définition TypeScript

Examples

Inclure le fichier de définition avec la bibliothèque sur npm

Ajouter des typages à votre package.json

```
{
...
"typings": "path/file.d.ts"
...
}
```

Désormais, chaque fois que cette bibliothèque est importée, typecript chargera le fichier de saisie

Lire Publier des fichiers de définition TypeScript en ligne:
https://riptutorial.com/fr/typescript/topic/2931/publier-des-fichiers-de-definition-typescript

Chapitre 20: Tableaux

Examples

Recherche d'objet dans un tableau

Utiliser find ()

```
const inventory = [
    {name: 'apples', quantity: 2},
    {name: 'bananas', quantity: 0},
    {name: 'cherries', quantity: 5}
];

function findCherries(fruit) {
    return fruit.name === 'cherries';
}

inventory.find(findCherries); // { name: 'cherries', quantity: 5 }

/* OR */

inventory.find(e => e.name === 'apples'); // { name: 'apples', quantity: 2 }
```

Lire Tableaux en ligne: https://riptutorial.com/fr/typescript/topic/9562/tableaux

Chapitre 21: Test d'unité

Examples

alsacien

Alsatian est un framework de test unitaire écrit en TypeScript. Il permet l'utilisation de cas de test et génère des balises compatibles TAP .

Pour l'utiliser, installez-le à partir de `npm` :

```
npm install alsatian --save-dev
```

Ensuite, configurez un fichier de test:

```
import { Expect, Test, TestCase } from "alsatian";
import { SomeModule } from "../src/some-module";

export SomeModuleTests {

    @Test()
    public statusShouldBeTrueByDefault() {
        let instance = new SomeModule();

        Expect(instance.status).toBe(true);
    }

    @Test("Name should be null by default")
    public nameShouldBeNullByDefault() {
        let instance = new SomeModule();

        Expect(instance.name).toBe(null);
    }

    @TestCase("first name")
    @TestCase("apples")
    public shouldSetNameCorrectly(name: string) {
        let instance = new SomeModule();

        instance.setName(name);

        Expect(instance.name).toBe(name);
    }

}
```

Pour une documentation complète, consultez le référentiel GsHub d'alsatian .

plugin chai-immuable

1. Installer à partir de npm chai, chai-immutable et ts-node

```
npm install --save-dev chai chai-immutable ts-node
```

2. Installez les types pour moka et chai

```
npm install --save-dev @types/mocha @types/chai
```

3. Ecrire un fichier de test simple:

```
  import {List, Set} from 'immutable';
  import * as chai from 'chai';
  import * as chaiImmutable from 'chai-immutable';

  chai.use(chaiImmutable);

  describe('chai immutable example', () => {
    it('example', () => {
      expect(Set.of(1,2,3)).to.not.be.empty;

      expect(Set.of(1,2,3)).to.include(2);
      expect(Set.of(1,2,3)).to.include(5);
    })
  })
```

4. Exécutez-le dans la console:

```
mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/**/*.spec.@(ts|tsx)'
```

ruban

La bande est un framework de test JavaScript minimaliste, elle produit un balisage compatible TAP .

Pour installer une `tape` aide de la commande `npm` run

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

Pour utiliser une `tape` avec Typescript, vous devez installer `ts-node` tant que package global, pour exécuter cette commande d'exécution

```
npm install -g ts-node
```

Maintenant, vous êtes prêt à écrire votre premier test

```
//math.test.ts
import * as test from "tape";

test("Math test", (t) => {
    t.equal(4, 2 + 2);
    t.true(5 > 2 + 2);

    t.end();
```

```
});
```

Pour exécuter la commande de test

```
ts-node node_modules/tape/bin/tape math.test.ts
```

En sortie, vous devriez voir

```
TAP version 13
# Math test
ok 1 should be equal
ok 2 should be truthy

1..2
# tests 2
# pass  2

# ok
```

Bon travail, vous venez de lancer votre test TypeScript.

Exécuter plusieurs fichiers de test

Vous pouvez exécuter plusieurs fichiers de test à la fois en utilisant des caractères génériques de chemin. Pour exécuter tous les tests Typescript dans la commande run du répertoire `tests`

```
ts-node node_modules/tape/bin/tape tests/**/*.ts
```

plaisanter (ts-blague)

jest est un framework de test JavaScript indolore par Facebook, avec ts-jest peut être utilisé pour tester le code TypeScript.

Pour installer jest à l'aide de la commande npm run

```
npm install --save-dev jest @types/jest ts-jest typescript
```

Pour plus de facilité, installez `jest` comme paquet global

```
npm install -g jest
```

Pour que `jest` fonctionne avec TypeScript, vous devez ajouter la configuration à `package.json`

```
//package.json
{
...
"jest": {
    "transform": {
      ".(ts|tsx)": "<rootDir>/node_modules/ts-jest/preprocessor.js"
    },
    "testRegex": "(/__tests__/.*|\\.(test|spec))\\.(ts|tsx|js)$",
```

```
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
}
```

Maintenant, la `jest` est prête. Supposons que nous ayons un échantillon de fizz buz à tester

```
//fizzBuzz.ts
export function fizzBuzz(n: number): string {
    let output = "";
    for (let i = 1; i <= n; i++) {
        if (i % 5 && i % 3) {
            output += i + ' ';
        }
        if (i % 3 === 0) {
            output += 'Fizz ';
        }
        if (i % 5 === 0) {
            output += 'Buzz ';
        }
    }
    return output;
}
```

Exemple de test pourrait ressembler

```
//FizzBuzz.test.ts
/// <reference types="jest" />

import {fizzBuzz} from "./fizzBuzz";
test("FizzBuzz test", () =>{
    expect(fizzBuzz(2)).toBe("1 2 ");
    expect(fizzBuzz(3)).toBe("1 2 Fizz ");
});
```

Pour exécuter un test

```
jest
```

En sortie, vous devriez voir

```
 PASS  ./fizzBuzz.test.ts
  ✓ FizzBuzz test (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.46s, estimated 2s
Ran all test suites.
```

Couverture de code

`jest` prend en charge la génération de rapports de couverture de code.

Pour utiliser la couverture de code avec TypeScript, vous devez ajouter une autre ligne de configuration à `package.json` .

```
{
...
  "jest": {
  ...
    "testResultsProcessor": "<rootDir>/node_modules/ts-jest/coverageprocessor.js"
  }
}
```

Pour exécuter des tests avec la génération de rapports de couverture

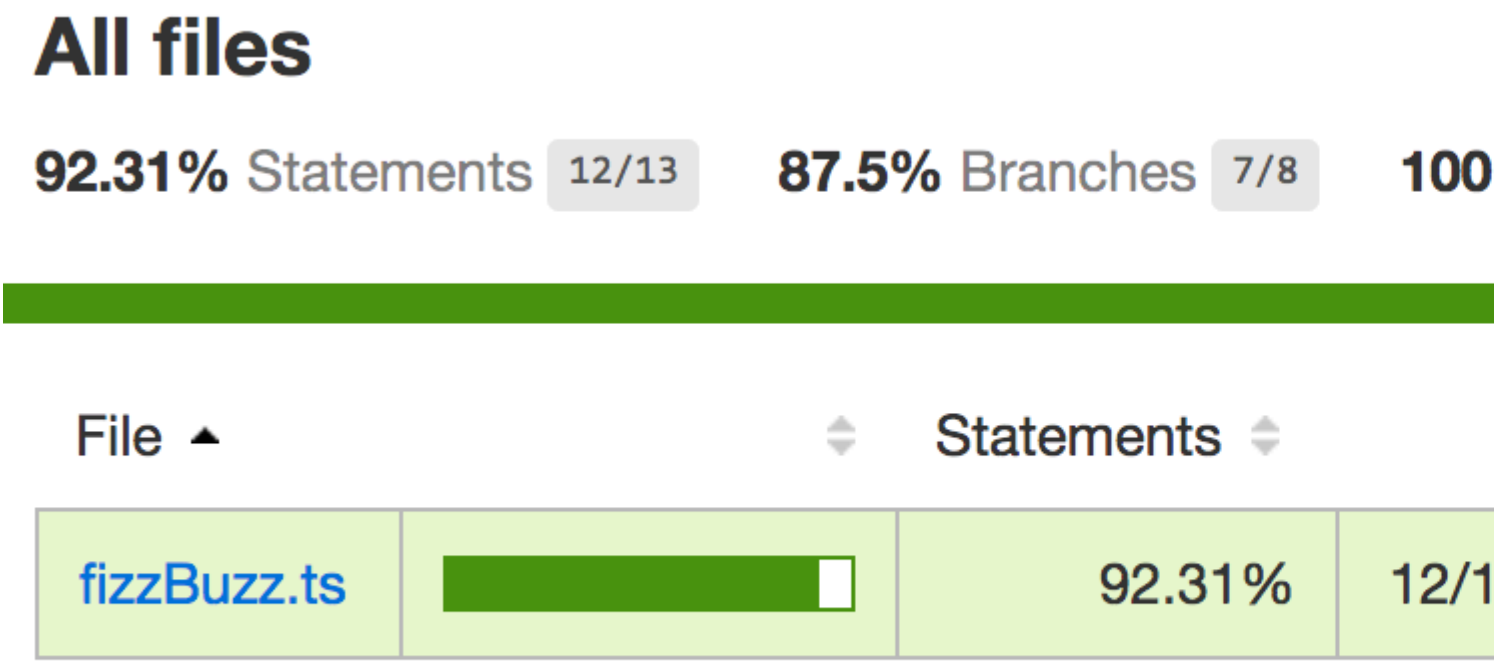
```
jest --coverage
```

Si utilisé avec notre exemple de fizz buzz, vous devriez voir

```
 PASS  ./fizzBuzz.test.ts
  ✓ FizzBuzz test (3ms)

-------------|----------|----------|----------|----------|----------------|
File         | % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-------------|----------|----------|----------|----------|----------------|
All files    |    92.31 |     87.5 |      100 |    91.67 |                |
 fizzBuzz.ts |    92.31 |     87.5 |      100 |    91.67 |             13 |
-------------|----------|----------|----------|----------|----------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.857s
Ran all test suites.
```

`jest` également créé la `coverage` dossier qui contient le rapport de couverture dans divers formats, y compris le rapport HTML convivial dans la `coverage/lcov-report/index.html`



Lire Test d'unité en ligne: https://riptutorial.com/fr/typescript/topic/7456/test-d-unite

Chapitre 22: tsconfig.json

Syntaxe

- Utilise le format de fichier JSON
- Peut également accepter les commentaires de style JavaScript

Remarques

Vue d'ensemble

La présence d'un fichier tsconfig.json dans un répertoire indique que le répertoire est la racine d'un projet TypeScript. Le fichier tsconfig.json spécifie les fichiers racine et les options de compilation requises pour compiler le projet.

Utiliser tsconfig.json

- En appelant tsc sans fichier d'entrée, le compilateur recherche le fichier tsconfig.json dans le répertoire en cours et continue la chaîne de répertoire parent.
- En appelant tsc sans fichiers d'entrée et une option de ligne de commande --project (ou juste -p) qui spécifie le chemin d'un répertoire contenant un fichier tsconfig.json. Lorsque les fichiers d'entrée sont spécifiés sur la ligne de commande, les fichiers tsconfig.json sont

Détails

La propriété `"compilerOptions"` peut être omise, auquel cas les valeurs par défaut du compilateur sont utilisées. Consultez notre liste complète des options de compilateur prises en charge.

Si aucune propriété `"files"` n'est présente dans un tsconfig.json, le compilateur inclut par défaut tous les fichiers TypeScript (* .ts ou * .tsx) dans le répertoire et les sous-répertoires. Lorsqu'une propriété "files" est présente, seuls les fichiers spécifiés sont inclus.

Si la propriété `"exclude"` est spécifiée, le compilateur inclut tous les fichiers TypeScript (* .ts ou * .tsx) dans le répertoire et les sous-répertoires, à l'exception des fichiers ou dossiers exclus.

La propriété `"files"` ne peut pas être utilisée conjointement avec la propriété "exclude". Si les deux sont spécifiés, la propriété "files" est prioritaire.

Tous les fichiers référencés par ceux spécifiés dans la propriété `"files"` sont également inclus. De même, si un fichier B.ts est référencé par un autre fichier A.ts, alors B.ts ne peut être exclu que si le fichier de référence A.ts est également spécifié dans la liste "exclude".

Un fichier `tsconfig.json` est autorisé à être complètement vide, ce qui compile tous les fichiers dans le répertoire contenant et les sous-répertoires avec les options du compilateur par défaut.

Les options du compilateur spécifiées sur la ligne de commande remplacent celles spécifiées dans le fichier tsconfig.json.

Schéma

Le schéma peut être trouvé à: http://json.schemastore.org/tsconfig

Examples

Créer un projet TypeScript avec tsconfig.json

La présence d'un fichier **tsconfig.json** indique que le répertoire en cours est la racine d'un projet compatible TypeScript.

L'initialisation d'un projet TypeScript, ou mieux, un fichier tsconfig.json, peut être effectuée à l'aide de la commande suivante:

```
tsc --init
```

À partir de TypeScript v2.3.0 et versions ultérieures, tsconfig.json sera créé par défaut:

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",                          /* Specify ECMAScript target version: 'ES3'
(default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify module code generation: 'commonjs',
'amd', 'system', 'umd' or 'es2015'. */
    // "lib": [],                             /* Specify library files to be included in the
compilation:  */
    // "allowJs": true,                       /* Allow javascript files to be compiled. */
    // "checkJs": true,                       /* Report errors in .js files. */
    // "jsx": "preserve",                     /* Specify JSX code generation: 'preserve',
'react-native', or 'react'. */
    // "declaration": true,                   /* Generates corresponding '.d.ts' file. */
    // "sourceMap": true,                     /* Generates corresponding '.map' file. */
    // "outFile": "./",                       /* Concatenate and emit output to single file.
*/
    // "outDir": "./",                        /* Redirect output structure to the directory.
*/
    // "rootDir": "./",                       /* Specify the root directory of input files.
Use to control the output directory structure with --outDir. */
    // "removeComments": true,                /* Do not emit comments to output. */
    // "noEmit": true,                        /* Do not emit outputs. */
    // "importHelpers": true,                 /* Import emit helpers from 'tslib'. */
    // "downlevelIteration": true,            /* Provide full support for iterables in 'for-
of', spread, and destructuring when targeting 'ES5' or 'ES3'. */
    // "isolatedModules": true,               /* Transpile each file as a separate module
(similar to 'ts.transpileModule'). */
```

```
    /* Strict Type-Checking Options */
    "strict": true                           /* Enable all strict type-checking options. */
    // "noImplicitAny": true,                /* Raise error on expressions and declarations
with an implied 'any' type. */
    // "strictNullChecks": true,             /* Enable strict null checks. */
    // "noImplicitThis": true,               /* Raise error on 'this' expressions with an
implied 'any' type. */
    // "alwaysStrict": true,                 /* Parse in strict mode and emit "use strict"
for each source file. */

    /* Additional Checks */
    // "noUnusedLocals": true,               /* Report errors on unused locals. */
    // "noUnusedParameters": true,           /* Report errors on unused parameters. */
    // "noImplicitReturns": true,            /* Report error when not all code paths in
function return a value. */
    // "noFallthroughCasesInSwitch": true,   /* Report errors for fallthrough cases in switch
statement. */

    /* Module Resolution Options */
    // "moduleResolution": "node",           /* Specify module resolution strategy: 'node'
(Node.js) or 'classic' (TypeScript pre-1.6). */
    // "baseUrl": "./",                      /* Base directory to resolve non-absolute module
names. */
    // "paths": {},                          /* A series of entries which re-map imports to
lookup locations relative to the 'baseUrl'. */
    // "rootDirs": [],                       /* List of root folders whose combined content
represents the structure of the project at runtime. */
    // "typeRoots": [],                      /* List of folders to include type definitions
from. */
    // "types": [],                          /* Type declaration files to be included in
compilation. */
    // "allowSyntheticDefaultImports": true,  /* Allow default imports from modules with no
default export. This does not affect code emit, just typechecking. */

    /* Source Map Options */
    // "sourceRoot": "./",                   /* Specify the location where debugger should
locate TypeScript files instead of source locations. */
    // "mapRoot": "./",                      /* Specify the location where debugger should
locate map files instead of generated locations. */
    // "inlineSourceMap": true,              /* Emit a single file with source maps instead
of having a separate file. */
    // "inlineSources": true,                /* Emit the source alongside the sourcemaps
within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. */

    /* Experimental Options */
    // "experimentalDecorators": true,       /* Enables experimental support for ES7
decorators. */
    // "emitDecoratorMetadata": true,        /* Enables experimental support for emitting
type metadata for decorators. */
  }
}
```

La plupart des options, sinon toutes, sont générées automatiquement avec uniquement le strict minimum nécessaire.

Les anciennes versions de TypeScript, comme par exemple v2.0.x et versions inférieures, généraient un tsconfig.json comme ceci:

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

compilerOnSave

Définir une propriété de niveau supérieur `compileOnSave` signale à l'EDI de générer tous les fichiers pour un **tsconfig.json** donné lors de l'enregistrement.

```
{
    "compileOnSave": true,
    "compilerOptions": {
        ...
    },
    "exclude": [
        ...
    ]
}
```

Cette fonctionnalité est disponible depuis TypeScript 1.8.4 et ultérieur, mais doit être directement prise en charge par les IDE. Actuellement, des exemples d'EDI pris en charge sont:

- Visual Studio 2015 avec la mise à jour 3
- JetBrains WebStorm
- Atome avec Atom -Typograph

commentaires

Un fichier tsconfig.json peut contenir des commentaires de ligne et de bloc, en utilisant les mêmes règles que ECMAScript.

```
//Leading comment
{
    "compilerOptions": {
        //this is a line comment
        "module": "commonjs", //eol line comment
        "target" /*inline block*/ : "es5",
        /* This is a
        block
        comment */
    }
}
/* trailing comment */
```

Configuration pour moins d'erreurs de programmation

Il existe de très bonnes configurations pour forcer les saisies et obtenir des erreurs plus utiles qui ne sont pas activées par défaut.

```
{
  "compilerOptions": {

    "alwaysStrict": true, // Parse in strict mode and emit "use strict" for each source file.

    // If you have wrong casing in referenced files e.g. the filename is Global.ts and you
have a /// <reference path="global.ts" /> to reference this file, then this can cause to
unexpected errors. Visite: http://stackoverflow.com/questions/36628612/typescript-transpiler-
casing-issue
    "forceConsistentCasingInFileNames": true, // Disallow inconsistently-cased references to
the same file.

    // "allowUnreachableCode": false, // Do not report errors on unreachable code. (Default:
False)
    // "allowUnusedLabels": false, // Do not report errors on unused labels. (Default: False)

    "noFallthroughCasesInSwitch": true, // Report errors for fall through cases in switch
statement.
    "noImplicitReturns": true, // Report error when not all code paths in function return a
value.

    "noUnusedParameters": true, // Report errors on unused parameters.
    "noUnusedLocals": true, // Report errors on unused locals.

    "noImplicitAny": true, // Raise error on expressions and declarations with an implied
"any" type.
    "noImplicitThis": true, // Raise error on this expressions with an implied "any" type.

    "strictNullChecks": true, // The null and undefined values are not in the domain of every
type and are only assignable to themselves and any.

    // To enforce this rules, add this configuration.
    "noEmitOnError": true     // Do not emit outputs if any errors were reported.
  }
}
```

Pas assez? Si vous êtes un codeur dur et que vous en voulez plus, alors vous pourriez être intéressé à vérifier vos fichiers TypeScript avec tslint avant de le compiler avec tsc. Vérifiez comment configurer tslint pour un code encore plus strict .

se préserverConstEnums

Typescript supporte les énumérables costant, déclarés via `const enum` .

Il s'agit généralement de sucre syntaxique, car les énumérations coûteuses sont incorporées dans le code JavaScript compilé.

Par exemple le code suivant

```
const enum Tristate {
    True,
    False,
    Unknown
}

var something = Tristate.True;
```

compile à

```
var something = 0;
```

Bien que la prestation de perfomance de inline, vous pouvez préférer garder énumérations même si costant (ex: vous pouvez souhaiter la lisibilité sur le code de développement), pour ce faire , vous devez définir en **tsconfig.json** les `preserveConstEnums` clausole dans les `compilerOptions` à `true` .

```
{
    "compilerOptions": {
        "preserveConstEnums" = true,
        ...
    },
    "exclude": [
        ...
    ]
}
```

De cette manière, l'exemple précédent serait compilé comme toute autre énumération, comme illustré dans l'extrait suivant.

```
var Tristate;
(function (Tristate) {
    Tristate[Tristate["True"] = 0] = "True";
    Tristate[Tristate["False"] = 1] = "False";
    Tristate[Tristate["Unknown"] = 2] = "Unknown";
})(Tristate || (Tristate = {}));

var something = Tristate.True
```

Lire tsconfig.json en ligne: https://riptutorial.com/fr/typescript/topic/4720/tsconfig-json

Chapitre 23: TSLint - Assurer la qualité et la cohérence du code

Introduction

TSLint effectue une analyse statique du code et détecte les erreurs et les problèmes potentiels dans le code.

Examples

Configuration de base de tslint.json

Ceci est une configuration de base de `tslint.json` qui

- empêche l'utilisation de `any`
- nécessite des accolades pour les instructions `if` / `else` / `for` / `do` / `while`
- exige que les guillemets (`"`) soient utilisés pour les chaînes

```
{
    "rules": {
        "no-any": true,
        "curly": true,
        "quotemark": [true, "double"]
    }
}
```

Configuration pour moins d'erreurs de programmation

Cet exemple tslint.json contient un ensemble de configuration pour appliquer davantage de typages, intercepter des erreurs communes ou des constructions déroutantes susceptibles de générer des bogues et de suivre davantage les directives de codage pour les contributeurs TypeScript .

Pour appliquer ces règles, incluez tslint dans votre processus de génération et vérifiez votre code avant de le compiler avec tsc.

```
{
  "rules": {
     // TypeScript Specific
     "member-access": true, // Requires explicit visibility declarations for class members.
     "no-any": true, // Diallows usages of any as a type declaration.
     // Functionality
     "label-position": true, // Only allows labels in sensible locations.
     "no-bitwise": true, // Disallows bitwise operators.
     "no-eval": true, // Disallows eval function invocations.
     "no-null-keyword": true, // Disallows use of the null keyword literal.
     "no-unsafe-finally": true, // Disallows control flow statements, such as return,
continue, break and throws in finally blocks.
```

```
    "no-var-keyword": true, // Disallows usage of the var keyword.
    "radix": true, // Requires the radix parameter to be specified when calling parseInt.
    "triple-equals": true, // Requires === and !== in place of == and !=.
    "use-isnan": true, // Enforces use of the isNaN() function to check for NaN references
instead of a comparison to the NaN constant.
    // Style
    "class-name": true, // Enforces PascalCased class and interface names.
    "interface-name": [ true, "never-prefix" ], // Requires interface names to begin with a
capital 'I'
    "no-angle-bracket-type-assertion": true, // Requires the use of as Type for type
assertions instead of <Type>.
    "one-variable-per-declaration": true, // Disallows multiple variable definitions in the
same declaration statement.
    "quotemark": [ true, "double", "avoid-escape" ], // Requires double quotes for string
literals.
    "semicolon": [ true, "always" ], // Enforces consistent semicolon usage at the end of
every statement.
    "variable-name": [true, "ban-keywords", "check-format", "allow-leading-underscore"] //
Checks variable names for various errors. Disallows the use of certain TypeScript keywords
(any, Number, number, String, string, Boolean, boolean, undefined) as variable or parameter.
Allows only camelCased or UPPER_CASED variable names. Allows underscores at the beginning
(only has an effect if "check-format" specified).
  }
}
```

Utiliser un ensemble de règles prédéfini par défaut

`tslint` peut étendre un ensemble de règles existant et est livré avec les valeurs `tslint:recommended` défaut `tslint:recommended` et `tslint:latest` .

> `tslint:recommended` est un ensemble de règles stable, quelque peu motivé, que nous encourageons pour la programmation TypeScript générale. Cette configuration suit un demi-cycle, de sorte qu'elle n'aura pas de changements de rupture entre les versions mineures ou les correctifs.
>
> `tslint:latest` extension est tslint: recommandée et est continuellement mise à jour pour inclure la configuration des dernières règles dans chaque version de TSLint. L'utilisation de cette configuration peut entraîner des changements de rupture dans des versions mineures, car de nouvelles règles sont activées, ce qui entraîne des défaillances de peluches dans votre code. Lorsque TSLint atteint un bump de version majeur, tslint: la version recommandée sera mise à jour pour être identique à tslint: latest.

Docs et code source du jeu de règles prédéfini

On peut donc simplement utiliser:

```
{
  "extends": "tslint:recommended"
}
```

d'avoir une configuration de départ raisonnable.

On peut alors écraser les règles à partir de ce préréglage via des `rules` , par exemple pour les développeurs de noeuds, il était logique de définir `no-console` sur `false` :

```
{
  "extends": "tslint:recommended",
  "rules": {
    "no-console": false
  }
}
```

Installation et configuration

Pour installer la commande tslint run

```
npm install -g tslint
```

Tslint est configuré via le fichier `tslint.json` . Pour initialiser la commande d'exécution de la configuration par défaut

```
tslint --init
```

Pour vérifier le fichier pour les erreurs possibles dans la commande d'exécution de fichier

```
tslint filename.ts
```

Ensembles de règles TSLint

- tslint-microsoft-contrib
- tslint-eslint-rules
- codelyzer

Yeoman Genearator supporte tous ces presets et peut être étendu aussi:

- générateur-tslint

Lire TSLint - Assurer la qualité et la cohérence du code en ligne:
https://riptutorial.com/fr/typescript/topic/7457/tslint---assurer-la-qualite-et-la-coherence-du-code

Chapitre 24: Typecript-installation-typescript-and-running-the-typecript-compiler-tsc

Introduction

Comment installer TypeScript et exécuter le compilateur TypeScript sur un fichier .ts à partir de la ligne de commande.

Examples

Pas.

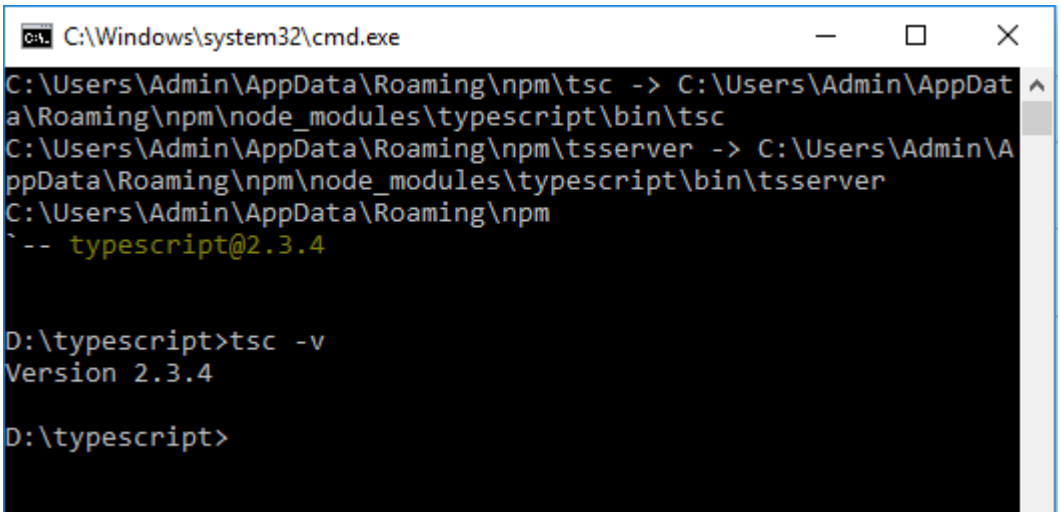
Installation de Typescript et exécution du compilateur typecript.

Pour installer Typecript Comiler

```
npm install -g typescript
```

Pour vérifier avec la version dactylographiée

```
tsc -v
```

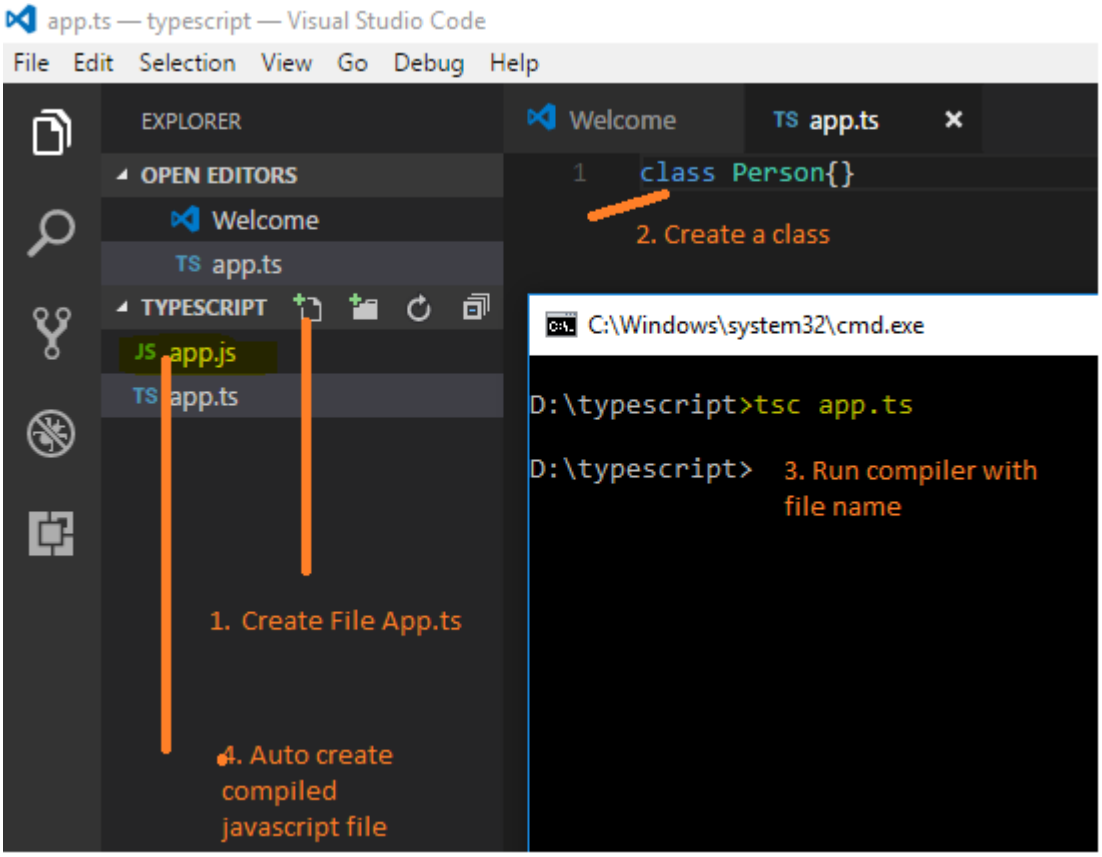


Télécharger le code Visual Studio pour Linux / Windows

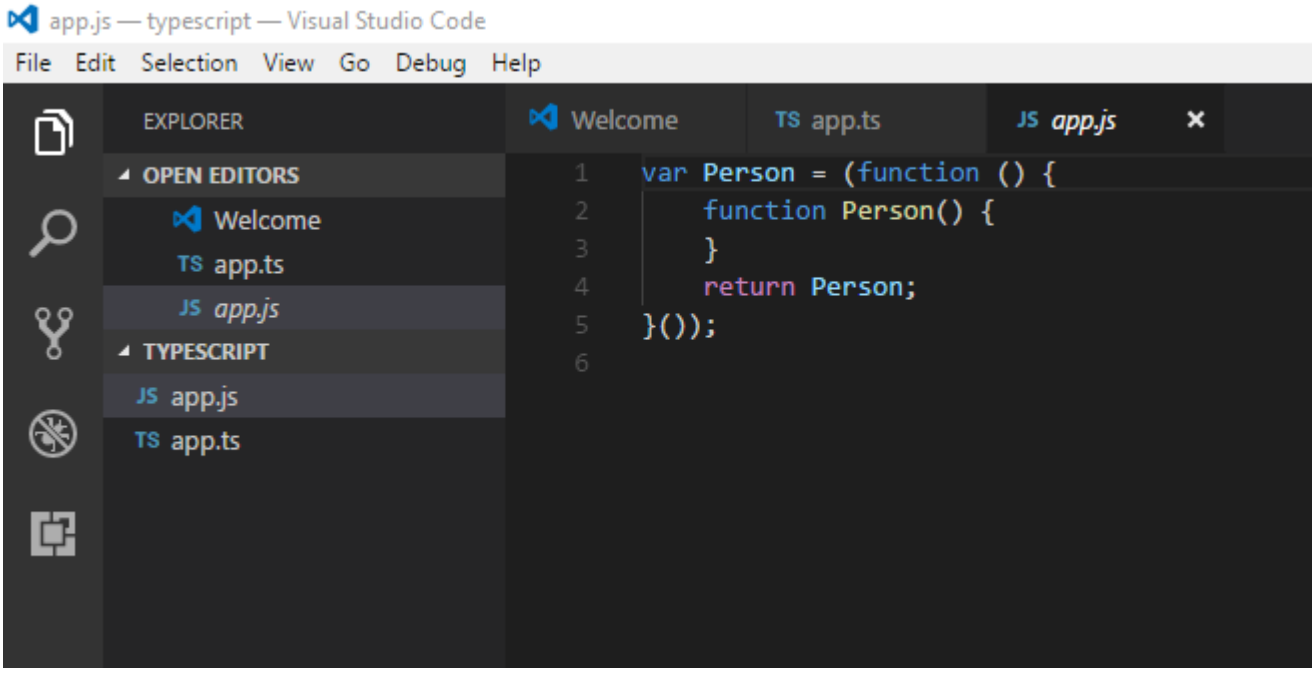
Lien de téléchargement du code visuel

1. Ouvrez le code Visual Studio
2. Open Same Folde où vous avez installé le compilateur Typescript
3. Ajouter un fichier en cliquant sur l'icône plus sur le volet gauche

4. Créez une classe de base.
5. Compilez votre fichier de script de type et générez une sortie.



Voir le résultat dans le javascript compilé du code dactylographié écrit.



Je vous remercie.

Lire Typecript-installation-typescript-and-running-the-typecript-compiler-tsc en ligne: https://riptutorial.com/fr/typescript/topic/10503/typecript-installation-typescript-and-running-the-typecript-compiler-tsc

Chapitre 25: Types de base TypeScript

Syntaxe

- let variableName: VariableType;
- function functionName (parameterName: VariableType, parameterWithDefault: VariableType = ParameterDefault, optionalParameter ?: VariableType, ... variardicParameter: VariableType []): ReturnType {/*...*/};

Examples

Booléen

Un booléen représente le type de données le plus élémentaire dans TypeScript, dans le but d'attribuer des valeurs true / false.

```
// set with initial value (either true or false)
let isTrue: boolean = true;

// defaults to 'undefined', when not explicitely set
let unsetBool: boolean;

// can also be set to 'null' as well
let nullableBool: boolean = null;
```

Nombre

Comme JavaScript, les nombres sont des valeurs à virgule flottante.

```
let pi: number = 3.14;           // base 10 decimal by default
let hexadecimal: number = 0xFF;  // 255 in decimal
```

ECMAScript 2015 permet les binaires et les octaux.

```
let binary: number = 0b10;   // 2 in decimal
let octal: number = 0o755;   // 493 in decimal
```

Chaîne

Type de données textuelles:

```
let singleQuotes: string = 'single';
let doubleQuotes: string = "double";
let templateString: string = `I am ${ singleQuotes }`; // I am single
```

Tableau

Un tableau de valeurs:

```
let threePigs: number[] = [1, 2, 3];
let genericStringArray: Array<string> = ['first', '2nd', '3rd'];
```

Enum

Un type pour nommer un ensemble de valeurs numériques:

Les valeurs numériques sont par défaut à 0:

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
let bestDay: Day = Day.Saturday;
```

Définir un numéro de départ par défaut:

```
enum TenPlus { Ten = 10, Eleven, Twelve }
```

ou attribuer des valeurs:

```
enum MyOddSet { Three = 3, Five = 5, Seven = 7, Nine = 9 }
```

Tout

En cas de doute sur un type, `any` est disponible:

```
let anything: any = 'I am a string';
anything = 5; // but now I am the number 5
```

Vide

Si vous n'avez pas de type du tout, couramment utilisé pour les fonctions qui ne renvoient rien:

```
function log(): void {
    console.log('I return nothing');
}
```

`void` types de `void` Ne peuvent être assignés à `null` ou `undefined` .

Tuple

Type de tableau avec des types connus et éventuellement différents:

```
let day: [number, string];
day = [0, 'Monday'];      // valid
day = ['zero', 'Monday']; // invalid: 'zero' is not numeric
console.log(day[0]); // 0
console.log(day[1]); // Monday
```

```
day[2] = 'Saturday'; // valid: [0, 'Saturday']
day[3] = false;      // invalid: must be union type of 'number | string'
```

Types dans les arguments de fonction et la valeur de retour. Nombre

Lorsque vous créez une fonction dans TypeScript, vous pouvez spécifier le type de données des arguments de la fonction et le type de données pour la valeur de retour

Exemple:

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

Ici, la syntaxe `x: number, y: number` signifie que la fonction peut accepter deux arguments `x` et `y` et qu'ils ne peuvent être que des nombres et `(...): number {` signifie que la valeur renvoyée ne peut être qu'un nombre

Usage:

```
sum(84 + 76) // will be return 160
```

Remarque:

Vous ne pouvez pas le faire

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

ou

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

il recevra les erreurs suivantes:

`error TS2322: Type 'string' is not assignable to type 'number'` et l' `error TS2322: Type 'number' is not assignable to type 'string'` respectivement

Types dans les arguments de fonction et la valeur de retour. Chaîne

Exemple:

```
function hello(name: string): string {
    return `Hello ${name}!`;
}
```

Ici, le `name: string` la syntaxe `name: string` signifie que la fonction peut accepter un argument de `name` et que cet argument ne peut être que string et `(...): string {` signifie que la valeur renvoyée ne peut être qu'une chaîne.

Usage:

```
hello('StackOverflow Documentation') // will be return Hello StackOverflow Documentation!
```

Types littéraux de chaîne

Les types littéraux de chaîne vous permettent de spécifier la valeur exacte qu'une chaîne peut avoir.

```
let myFavoritePet: "dog";
myFavoritePet = "dog";
```

Toute autre chaîne donnera une erreur.

```
// Error: Type '"rock"' is not assignable to type '"dog"'.
// myFavoritePet = "rock";
```

Avec les alias de type et les types d'union, vous obtenez un comportement semblable à un enum.

```
type Species = "cat" | "dog" | "bird";

function buyPet(pet: Species, name: string) : Pet { /*...*/ }

buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");

// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" |
"bird". Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");
```

Les types de littéral de chaîne peuvent être utilisés pour distinguer les surcharges.

```
function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet { /*...*/ }

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)
```

Ils fonctionnent bien pour les gardes de type défini par l'utilisateur.

```
interface Pet {
    species: Species;
    eat();
    sleep();
}
```

```
interface Cat extends Pet {
    species: "cat";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet){
    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)
        pet.eat();
        pet.sleep();
    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)
        pet.eat();
        pet.sing();
        pet.sleep();
    }
}
```

Exemple de code complet

```
let myFavoritePet: "dog";
myFavoritePet = "dog";

// Error: Type '"rock"' is not assignable to type '"dog"'.
// myFavoritePet = "rock";

type Species = "cat" | "dog" | "bird";

interface Pet {
    species: Species;
    name: string;
    eat();
    walk();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Dog extends Pet {
    species: "dog";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}
```

```
// Error: Interface 'Rock' incorrectly extends interface 'Pet'. Types of property 'species'
are incompatible. Type '"rock"' is not assignable to type '"cat" | "dog" | "bird"'. Type
'"rock"' is not assignable to type '"bird"'.
// interface Rock extends Pet {
//      type: "rock";
// }

function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet {
    if(pet === "cat") {
        return {
            species: "cat",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }
        } as Cat;
    } else if(pet === "dog") {
        return {
            species: "dog",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }
        } as Dog;
    } else if(pet === "bird") {
        return {
            species: "bird",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }, sing: function () {
                console.log(`${this.name} sings.`);
            }
        } as Bird;
    } else {
        throw `Sorry we don't have a ${pet}. Would you like to buy a dog?`;
    }
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsDog(pet: Pet): pet is Dog {
```

```
    return pet.species === "dog";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet) {
    console.log(`Hey ${pet.name}, let's play.`);

    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)

        pet.eat();
        pet.sleep();

        // Error: Type '"bird"' is not assignable to type '"cat"'.
        // pet.type = "bird";

        // Error: Property 'sing' does not exist on type 'Cat'.
        // pet.sing();

    } else if(petIsDog(pet)) {
        // pet is now from type Dog (pet: Dog)

        pet.eat();
        pet.walk();
        pet.sleep();

    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)

        pet.eat();
        pet.sing();
        pet.sleep();
    } else {
        throw "An unknown pet. Did you buy a rock?";
    }
}

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)

// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" |
"bird". Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");

playWithPet(dog);
// Output: Hey Rocky, let's play.
//         Rocky eats.
//         Rocky walks.
//         Rocky sleeps.
```

Types d'intersection

Un type d'intersection combine le membre de deux types ou plus.

```
interface Knife {
    cut();
```

```
}

interface BottleOpener{
    openBottle();
}

interface Screwdriver{
    turnScrew();
}

type SwissArmyKnife = Knife & BottleOpener & Screwdriver;

function use(tool: SwissArmyKnife){
    console.log("I can do anything!");

    tool.cut();
    tool.openBottle();
    tool.turnScrew();
}
```

const Enum

Un const Enum est identique à un Enum normal. Sauf que aucun objet n'est généré au moment de la compilation. Au lieu de cela, les valeurs littérales sont substituées là où le const Enum est utilisé.

```
// Typescript: A const Enum can be defined like a normal Enum (with start value, specifig
values, etc.)
const enum NinjaActivity {
    Espionage,
    Sabotage,
    Assassination
}

// Javascript: But nothing is generated

// Typescript: Except if you use it
let myFavoriteNinjaActivity = NinjaActivity.Espionage;
console.log(myFavoritePirateActivity); // 0

// Javascript: Then only the number of the value is compiled into the code
// var myFavoriteNinjaActivity = 0 /* Espionage */;
// console.log(myFavoritePirateActivity); // 0

// Typescript: The same for the other constant example
console.log(NinjaActivity["Sabotage"]); // 1

// Javascript: Just the number and in a comment the name of the value
// console.log(1 /* "Sabotage" */); // 1

// Typescript: But without the object none runtime access is possible
// Error: A const enum member can only be accessed using a string literal.
// console.log(NinjaActivity[myFavoriteNinjaActivity]);
```

Pour comparaison, un Enum normal

```
// Typescript: A normal Enum
```

```
enum PirateActivity {
    Boarding,
    Drinking,
    Fencing
}

// Javascript: The Enum after the compiling
// var PirateActivity;
// (function (PirateActivity) {
//     PirateActivity[PirateActivity["Boarding"] = 0] = "Boarding";
//     PirateActivity[PirateActivity["Drinking"] = 1] = "Drinking";
//     PirateActivity[PirateActivity["Fencing"] = 2] = "Fencing";
// })(PirateActivity || (PirateActivity = {}));

// Typescript: A normale use of this Enum
let myFavoritePirateActivity = PirateActivity.Boarding;
console.log(myFavoritePirateActivity); // 0

// Javascript: Looks quite similar in Javascript
// var myFavoritePirateActivity = PirateActivity.Boarding;
// console.log(myFavoritePirateActivity); // 0

// Typescript: And some other normale use
console.log(PirateActivity["Drinking"]); // 1

// Javascript: Looks quite similar in Javascript
// console.log(PirateActivity["Drinking"]); // 1

// Typescript: At runtime, you can access an normal enum
console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"

// Javascript: And it will be resolved at runtime
// console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"
```

Lire Types de base TypeScript en ligne: https://riptutorial.com/fr/typescript/topic/2776/types-de-base-typescript

Chapitre 26: TypeScript avec AngularJS

Paramètres

prénom	La description
`controllerAs`	est un nom d'alias auquel des variables ou des fonctions peuvent être affectées. @voir: https://docs.angularjs.org/guide/directive
`$inject`	Dépendances Liste d'injection, elle est résolue par angulaire et en passant en argument aux fonctions constantes.

Remarques

Lors de l'exécution de la directive dans TypeScript, gardez à l'esprit la puissance de ce langage de type personnalisé et des interfaces que vous pouvez créer. Ceci est extrêmement utile pour développer des applications énormes. La complétion de code supportée par de nombreux IDE vous montrera la valeur possible par type correspondant avec lequel vous travaillez, donc il y a beaucoup moins de choses à garder à l'esprit (comparé à VanillaJS).

"Code contre interfaces, pas implémentations"

Examples

Directif

```
interface IMyDirectiveController {
    // specify exposed controller methods and properties here
    getUrl(): string;
}

class MyDirectiveController implements IMyDirectiveController {

    // Inner injections, per each directive
    public static $inject = ["$location", "toaster"];

    constructor(private $location: ng.ILocationService, private toaster: any) {
        // $location and toaster are now properties of the controller
    }

    public getUrl(): string {
        return this.$location.url(); // utilize $location to retrieve the URL
    }
}

/*
 * Outer injections, for run once controll.
 * For example we have all templates in one value, and we wan't to use it.
 */
```

```
export function myDirective(templatesUrl: ITemplates): ng.IDirective {
    return {
        controller: MyDirectiveController,
        controllerAs: "vm",

        link: (scope: ng.IScope,
               element: ng.IAugmentedJQuery,
               attributes: ng.IAttributes,
               controller: IMyDirectiveController): void => {

            let url = controller.getUrl();
            element.text("Current URL: " + url);

        },

        replace: true,
        require: "ngModel",
        restrict: "A",
        templateUrl: templatesUrl.myDirective,
    };
}

myDirective.$inject = [
    Templates.prototype.slug,
];

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
    directive(myDirective.prototype.slug, myDirective);
```

Exemple simple

```
export function myDirective($location: ng.ILocationService): ng.IDirective {
    return {

        link: (scope: ng.IScope,
            element: ng.IAugmentedJQuery,
            attributes: ng.IAttributes): void => {

            element.text("Current URL: " + $location.url());

        },

        replace: true,
        require: "ngModel",
        restrict: "A",
        templateUrl: templatesUrl.myDirective,
    };
}

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
```

```
    directive(myDirective.prototype.slug, [
        Templates.prototype.slug,
        myDirective
    ]);
```

Composant

Pour faciliter la transition vers Angular 2, il est recommandé d'utiliser `Component` , disponible depuis Angular 1.5.8.

myModule.ts

```
import { MyModuleComponent } from "./components/myModuleComponent";
import { MyModuleService } from "./services/MyModuleService";

angular
    .module("myModule", [])
    .component("myModuleComponent", new MyModuleComponent())
    .service("myModuleService", MyModuleService);
```

composants / myModuleComponent.ts

```
import IComponentOptions = angular.IComponentOptions;
import IControllerConstructor = angular.IControllerConstructor;
import Injectable = angular.Injectable;
import { MyModuleController } from "../controller/MyModuleController";

export class MyModuleComponent implements IComponentOptions {
    public templateUrl: string = "./app/myModule/templates/myComponentTemplate.html";
    public controller: Injectable<IControllerConstructor> = MyModuleController;
    public bindings: {[boundProperty: string]: string} = {};
}
```

templates / myModuleComponent.html

```
<div class="my-module-component">
    {{$ctrl.someContent}}
</div>
```

contrôleur / MyModuleController.ts

```
import IController = angular.IController;
import { MyModuleService } from "../services/MyModuleService";

export class MyModuleController implements IController {
    public static readonly $inject: string[] = ["$element", "myModuleService"];
    public someContent: string = "Hello World";

    constructor($element: JQuery, private myModuleService: MyModuleService) {
        console.log("element", $element);
    }

    public doSomething(): void {
        // implementation..
```

```
    }
}
```

services / MyModuleService.ts

```
export class MyModuleService {
    public static readonly $inject: string[] = [];

    constructor() {
    }

    public doSomething(): void {
        // do something
    }
}
```

quelque part.html

```
<my-module-component></my-module-component>
```

Lire TypeScript avec AngularJS en ligne: https://riptutorial.com/fr/typescript/topic/6569/typescript-avec-angularjs

Chapitre 27: TypeScript avec SystemJS

Examples

Hello World dans le navigateur avec SystemJS

Installer systemjs et plugin-typecript

```
npm install systemjs
npm install plugin-typescript
```

NOTE: ceci installera le compilateur typecript 2.0.0 qui n'est pas encore sorti.

Pour TypeScript 1.8, vous devez utiliser le plugin-typescript 4.0.16

Créer `hello.ts` fichier `hello.ts`

```
export function greeter(person: String) {
    return 'Hello, ' + person;
}
```

Créer `hello.html` fichier `hello.html`

```
<!doctype html>
<html>
<head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>

    <script>
        window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

Créer `config.js` - Fichier de configuration SystemJS

```
System.config({
    packages: {
        "plugin-typescript": {
            "main": "plugin.js"
        },
```

```
        "typescript": {
            "main": "lib/typescript.js",
            "meta": {
                "lib/typescript.js": {
                    "exports": "ts"
                }
            }
        }
    },
    map: {
        "plugin-typescript": "node_modules/plugin-typescript/lib/",
        /* NOTE: this is for npm 3 (node 6) */
        /* for npm 2, typescript path will be */
        /* node_modules/plugin-typescript/node_modules/typescript */
        "typescript": "node_modules/typescript/"
    },
    transpiler: "plugin-typescript",
    meta: {
        "./hello.ts": {
            format: "esm",
            loader: "plugin-typescript"
        }
    },
    typescriptOptions: {
        typeCheck: 'strict'
    }
});
```

REMARQUE: si vous ne voulez pas vérifier le type, supprimez `loader: "plugin-typescript"` et `typescriptOptions` de `config.js` . Notez également qu'il ne vérifiera jamais le code JavaScript, en particulier le code de la `<script>` dans l'exemple html.

Essaye-le

```
npm install live-server
./node_modules/.bin/live-server --open=hello.html
```

Construis-le pour la production

```
npm install systemjs-builder
```

Créez le fichier `build.js` :

```
var Builder = require('systemjs-builder');
var builder = new Builder();
builder.loadConfig('./config.js').then(function() {
    builder.bundle('./hello.ts', './hello.js', {minify: true});
});
```

compiler hello.js depuis hello.ts

```
node build.js
```

Utilisez-le en production

Chargez simplement hello.js avec une balise script avant la première utilisation

Fichier `hello-production.html` :

```
<!doctype html>
<html>
<head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>
    <script src="hello.js"></script>
    <script>
        window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

Lire TypeScript avec SystemJS en ligne: https://riptutorial.com/fr/typescript/topic/6664/typescript-avec-systemjs

Chapitre 28: Utilisation de TypScript avec React (JS & native)

Examples

Composant ReactJS écrit en Typescript

Vous pouvez utiliser les composants de ReactJS facilement dans TypeScript. Renommez simplement l'extension de fichier 'jsx' en 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Mais pour utiliser pleinement la fonctionnalité principale de TypeScript (vérification de type statique), vous devez effectuer plusieurs opérations:

1) convertir React.createClass en une classe ES6:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Pour plus d'informations sur la conversion en ES6, cliquez ici

2) Ajouter des interfaces et des interfaces d'état:

```
interface Props {
    name:string;
    optionalParam?:number;
}

interface State {
  //empty in our case
}

class HelloMessage extends React.Component<Props, State> {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

```
// TypeScript will allow you to create without the optional parameter
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
// But it does check if you pass in an optional parameter of the wrong type
ReactDOM.render(<HelloMessage name="Sebastian" optionalParam='foo' />, mountNode);
```

Désormais, TypeScript affichera une erreur si le programmeur oublie de transmettre des accessoires. Ou si vous essayez de transmettre des accessoires qui ne sont pas définis dans l'interface.

Dactylographier et réagir et webpack

Installer des typescript, typings et webpack globalement

```
npm install -g typescript typings webpack
```

Installation de chargeurs et liaison de texte dactylographié

```
npm install --save-dev ts-loader source-map-loader npm link typescript
```

> Lier TypeScript permet à ts-loader d'utiliser votre installation globale de TypeScript au lieu d'avoir besoin d'un document distinct pour la copie de type local.

installer des fichiers `.d.ts` avec typeScript 2.x

```
npm i @types/react --save-dev
npm i @types/react-dom --save-dev
```

installer des fichiers `.d.ts` avec le typecript 1.x

```
typings install --global --save dt~react
typings install --global --save dt~react-dom
```

fichier de configuration `tsconfig.json`

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  }
}
```

fichier de configuration `webpack.config.js`

```
module.exports = {
    entry: "<path to entry point>",// for example ./src/helloMessage.tsx
    output: {
        filename: "<path to bundle file>", // for example ./dist/bundle.js
    },

    // Enable sourcemaps for debugging webpack's output.
```

```
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-
loader'.
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },

    // When importing a module whose path matches one of the following, just
    // assume a corresponding global variable exists and use that instead.
    // This is important because it allows us to avoid bundling all of our
    // dependencies, which allows browsers to cache those libraries between builds.
    externals: {
        "react": "React",
        "react-dom": "ReactDOM"
    },
};
```

enfin exécuter `webpack` ou `webpack -w` (pour le mode montre)

Remarque : React et ReactDOM sont marqués comme externes

Lire Utilisation de TypScript avec React (JS & native) en ligne:
https://riptutorial.com/fr/typescript/topic/1835/utilisation-de-typscript-avec-react--js--amp--native-

Chapitre 29: Utilisation de TypScript avec RequireJS

Introduction

RequireJS est un chargeur de fichiers et de modules JavaScript. Il est optimisé pour une utilisation dans le navigateur, mais il peut être utilisé dans d'autres environnements JavaScript, tels que Rhino et Node. L'utilisation d'un chargeur de script modulaire tel que RequireJS améliorera la vitesse et la qualité de votre code.

L'utilisation de TypeScript avec RequireJS nécessite la configuration de tsconfig.json et l'inclusion d'un extrait dans tout fichier HTML. Le compilateur traduira les importations de la syntaxe de TypeScript au format RequireJS.

Examples

Exemple HTML utilisant requireJS CDN pour inclure un fichier TypeScript déjà compilé.

```
<body onload="__init();">
    ...
    <script src="http://requirejs.org/docs/release/2.3.2/comments/require.js"></script>
    <script>
      function __init() {
        require(["view/index.js"]);
      }
    </script>
</body>
```

Exemple avec tsconfig.json à compiler pour afficher le dossier en utilisant le style d'importation requireJS.

```
{
  "module": "amd",    // Using AMD module code generator which works with requireJS
  "rootDir": "./src", // Change this to your source folder
  "outDir": "./view",
    ...
}
```

Lire Utilisation de TypScript avec RequireJS en ligne:
https://riptutorial.com/fr/typescript/topic/10773/utilisation-de-typscript-avec-requirejs

Chapitre 30: Utiliser TypeScript avec webpack

Examples

webpack.config.js

installer les chargeurs `npm install --save-dev ts-loader source-map-loader`

tsconfig.json

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react" // if you want to use react jsx
  }
}
```

```
module.exports = {
    entry: "./src/index.ts",
    output: {
        filename: "./dist/bundle.js",
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-
loader'.
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },
    /****************************
     *  If you want to use react *
     ****************************/


    // When importing a module whose path matches one of the following, just
    // assume a corresponding global variable exists and use that instead.
```

```
    // This is important because it allows us to avoid bundling all of our
    // dependencies, which allows browsers to cache those libraries between builds.
    // externals: {
    //     "react": "React",
    //     "react-dom": "ReactDOM"
    // },
};
```

Lire Utiliser TypeScript avec webpack en ligne:
https://riptutorial.com/fr/typescript/topic/2024/utiliser-typescript-avec-webpack

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec TypeScript	2426021684, Alec Hansen, Blackus, BrunoLM, cdbajorin, ChanceM, Community, danvk, Florian Hämmerle, Fylax, goenning, islandman93, jengeb, Joshua Breeden, k0pernikus, Kiloku, KnottytOmo, Kuba Beránek, Lekhnath, Matt Lishman, Mikhail, mleko, RationalDev, Roy Dictus, Saiful Azad, Sam, samAlvin, Wasabi Fan, zigzag
2	Comment utiliser une bibliothèque javascript sans fichier de définition de type	Bruno Krebs, Kevin Montrose
3	Configurez le projet typecript pour compiler tous les fichiers en texte dactylographié.	Rahul
4	Contrôles Nuls Strict	bnieland, danvk, JKillian, Yaroslav Admin
5	Décorateur de classe	bruno, Remo H. Jansen, Stefan Rein
6	Des classes	adamboro, apricity, Cobus Kruger, Equiman, hansmaad, James Monger, Jeff Huijsmans, Justin Niles, KnottytOmo, Robin
7	Enums	dimitrisli, Florian Hämmerle, Kevin Montrose, smnbbrv
8	Exemples de base de texte	vashishth
9	Gardes de type définis par l'utilisateur	Kevin Montrose
10	Génériques	danvk, hansmaad, KnottytOmo, Mathias Rodriguez, Muhammad Awais, Slava Shpitalny, Taytay
11	Importer des bibliothèques externes	2426021684, Almond, artem, Blackus, Brutus, Dean Ward, dublicator, Harry, islandman93, JKillian, Joel Day, KnottytOmo, lefb766, Rajab Shakirov, Slava Shpitalny, tuvokki

12	Intégration avec les outils de construction	Alex Filatov, BrunoLM, Dan, dublicator, John Ruddell, mleko, Protectator, smnbbrv, void
13	Interfaces	ABabin, Aminadav, Aron, artem, Cobus Kruger, Fabian Lauer, islandman93, Joshua Breeden, Paul Boutes, Robin, Saiful Azad , Slava Shpitalny, Sunnyok
14	Le débogage	Peopleware
15	Les fonctions	br4d, hansmaad, islandman93, KnottytOmo, muetzerich, SilentLupin, Slava Shpitalny
16	Mixins	Fenton
17	Modules - exportation et importation	mleko
18	Pourquoi et quand utiliser TypeScript	danvk
19	Publier des fichiers de définition TypeScript	2426021684
20	Tableaux	Udlei Nati
21	Test d'unité	James Monger, leonidv, Louie Bertoncin, Matthew Harwood, mleko
22	tsconfig.json	bnieland, Fylax, goenning, Magu, Moriarty, user3893988
23	TSLint - Assurer la qualité et la cohérence du code	Alex Filatov, James Monger, k0pernikus, Magu, mleko
24	Typecript-installation-typescript-and-running-the-typecript-compiler-tsc	Rahul
25	Types de base TypeScript	dublicator, Fenton, Fylax, Magu, Mikhail, Moriarty, RationalDev
26	TypeScript avec AngularJS	Chic, Roman M. Koss, Stefan Rein
27	TypeScript avec	artem

	SystemJS	
28	Utilisation de TypScript avec React (JS & native)	Aleh Kashnikau, irakli khitarishvili, islandman93, Rajab Shakirov , tBX
29	Utilisation de TypScript avec RequireJS	lilezek
30	Utiliser TypeScript avec webpack	BrunoLM, irakli khitarishvili, John Ruddell