ORSYS
Informatique et Management

TypeScript - Produire ses développements JavaScript

Intervenant :

Renaud Dubuis

Linked in

Renaud Dubuis

View profile

Approche pédagogique.

Participants. (Tour de table)

Développeurs et chefs de projets web.

Evaluation des pré-requis.

Le tour de table initial et les premiers exercices ont pour but l'évaluation effective des participants au regard des pré-requis.

Récapitulatif (matinal).

Chaque journée commence par un exercice collectif de restitution des concepts abordés la veille. L'objectif étant de renforcer (répétition) les connaissances acquises pour les utiliser comme socle lors de la journée à venir.

Concertation personnelle.

Le formateur passera assister les participants individuellement aussi souvent que possible.
Les participants sont invités à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Travaux Pratiques.

Les concepts seront illustrés par des démonstrations techniques et visuelles.

La mise en oeuvre des travaux pratiques se déroule en 3 phases, permettant la reproduction en autonomie des exercices.

> **1. Ennoncé > 2. Démonstration > 3. Manipulation**

Version numérique du support de formation.

Pour renforcer le confort de lecture et de manipulation **(copier/coller, liens cliquables, coloration du code)** le support est également distribué en version numérique. **au format PDF.**

Introduction :

Vous découvrirez le langage TypeScript, la surcouche JavaScript pour les développements Front-End. Vous comprendrez le typage, vous maîtriserez les classes et l'héritage pour la programmation orientée objet. Vous gagnerez en productivité afin de mettre en place des interfaces maintenables.

Objectif(s) pédagogique(s) :

- Prendre en main les outils de développement
- Créer des fonctions et des tableaux
- Maîtriser l'héritage avec TypeScript
- Assembler les codes pour le serveur

Pré-requis :

> Programmeurs ayant des connaissances JavaScript de base.

Le principe de TypeScript

Le principe de TypeScript.

> TypeScript c'est JavaScript avec une syntaxe pour les types.

TypeScript est un langage de programmation **fortement typé** qui s'appuie sur JavaScript et offre de meilleurs outils à n'importe quelle échelle.

Principe syntaxique

TypeScript ajoute une syntaxe supplémentaire à JavaScript pour favoriser une **intégration plus poussée avec votre éditeur.**

> Détectez les erreurs au plus tôt dans votre éditeur.

Exemple de code JavaScript

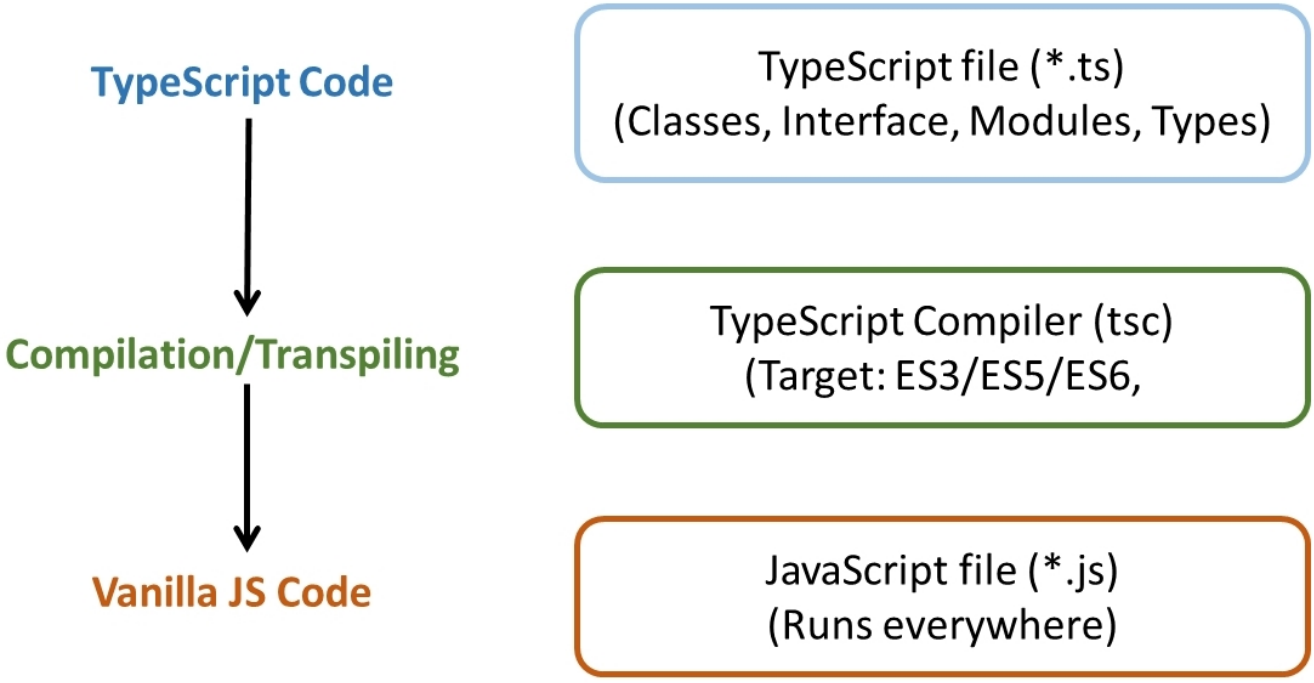
```
function addNumbers(a, b) {
    return a + b;
}
```

Exemple avec TypeScript

```
/*
A l'usage  TypeScript préviendra de mauvaises valeurs en arguments
*/
function addNumbers(a: number, b: number) {
    return a + b;
}
```

Principe de mise en oeuvre

Le code TypeScript peut être **converti en JavaScript**, qui peut être exécuté n'importe où où fonctionne JavaScript.



> TypeScript est donc une *surcouche* au language JavaScript ainsi qu'un programme de transformation du code (**transpilation**).

Principe de simplicité.

TypeScript comprend le JavaScript et utilise **l'inférence de type** pour vous donner des outils de qualité sans code supplémentaire.

Exemple avec TypeScript

```
/*
Par inférence TypeScript assigne le type "string" à la variable "day"
*/
let day = '';

/*
TypeScript détecte l'incohérence de type et produit une erreur.
*/
day = Date.now();
```

Pourquoi Microsoft a créé cette alternative ?

> JavaScript (aussi connu sous le nom ECMAScript) était à l'origine un simple langage de scripting pour navigateurs. Quand il fut inventé, il était utilisé pour de petits extraits de code dans une page web — aller au-delà d'une douzaine de ligne était inhabituel.

De ce fait, les navigateurs exécutaient du code JS assez lentement. Cependant, la popularité de JavaScript grandira avec le temps, et les développeurs web ont commencé à s'en servir pour créer des expériences interactives.

> Pour résumer, ce langage a été créé à l'origine pour répondre à des besoins simples, puis a évolué pour supporter l'exécution de millions de lignes.

Un site web moderne, de nos jours, contient des centaines de milliers de lignes de code. Ceci est en phase avec la façon dont le web a grandi, partant d'un simple ensemble de pages statiques, pour devenir une plateforme d'applications riches pour tout et sur tout.

> De plus, le JS est devenu assez populaire pour être utilisé en dehors de navigateurs, Node.js ayant marqué l'implémentation de JS dans un environnement côté serveur.

Origine de TypeScript.

En 2012 Microsoft lançait sous le nom de code "Strada" l'initiative TypeScript.

"With HTML5, the standards web platform has become significantly more compelling for delivering rich user experiences. At the same time, the reach of JavaScript has continued to expand, going beyond the browser to include native device apps (e.g. Windows Store apps for Windows 8), applications in the cloud (e.g., node.js running on Windows Azure), and more. With these developments, we're starting to see applications of unprecedented size written with JavaScript, despite the fact that creating large-scale JavaScript applications is hard. TypeScript makes it easier."

Soma Somasegar

> En résumé : Microsoft développe le "superset" TypeScript de JavaScript pour fournir un "enrichissement syntaxique" nécessaire pour créer de grandes applications et prendre en charge de grandes équipes de développement.

TypeScript, informations clés.

Paradigmes

> Multi-paradigm: functional, generic, imperative, object-oriented

Developpeur/Mainteneur

> Microsoft

Site Officiel

> https://www.typescriptlang.org/

Première apparition

> 1 October 2012

License

> Apache License 2.0

Extensiosn de fichiers

> .ts, .tsx

Typage

> Duck, gradual, structural

Lead Designer

> Anders Hejlsberg

La surcouche JavaScript.

Certains langages interdiraient l'exécution de code erroné.

La détection d'erreurs dans le code sans le lancer s'appelle **la vérification statique.**

La distinction entre ce qui est une erreur de ce qui ne l'est pas, en partant des valeurs avec lesquelles on travaille, s'appelle **la vérification statique de types.**

> TypeScript vérifie les erreurs d'un programme avant l'exécution, et fait cela en se basant sur les types de valeurs, **c'est un vérificateur statique.**

Exemple avec TypeScript

```
/*
Erreur la propriété "heigth" n'existe pas sur l'objet "obj"
*/
const obj = { width: 10, height: 15 };
const area = obj.width * obj.heigth;
```

Surcouche typée.

> TypeScript est une surcouche de JavaScript : une syntaxe JS légale est donc une syntaxe TS légale.
>
> TypeScript ne considère pas forcément du code JavaScript comme du code invalide.

Cela signifie que vous pouvez prendre du code JavaScript fonctionnel et le mettre dans un fichier TypeScript sans vous inquiéter de comment il est écrit exactement.

> TypeScript est une surcouche typée.

Cela veut dire que TS ajoute des règles régissant comment différents types de valeurs peuvent être utilisés.

Comportement à l'exécution.

> TypeScript préserve le comportement à l'exécution de JavaScript.

Cela veut dire que si vous déplacez du code de JavaScript à TypeScript, **il est garanti de s'exécuter de la même façon**, même si TS pense qu'il comporte des erreurs liées aux types.

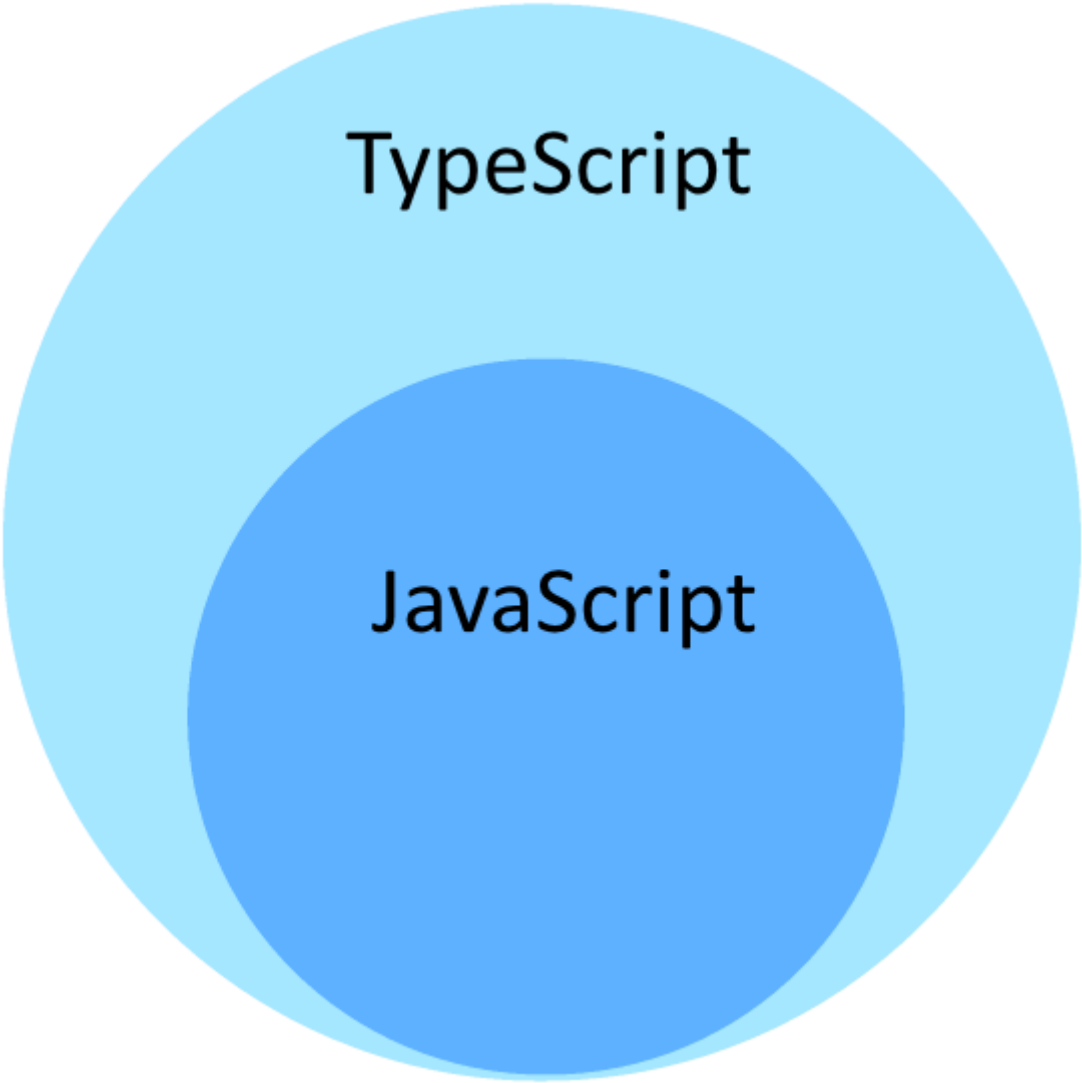
Surcouche : Effacement de types.

Une fois que le compilateur de TypeScript a fini de vérifier le code, **il efface** les types pour laisser le code résultant.

> Pour résumer, même si vous pouvez avoir des erreurs de type lors de la compilation, le système de types n'affecte aucunement la façon dont votre programme s'exécute.

Comparaison : JavaScript et TypeScript.

> Une question souvent posée est "Est-ce que je dois apprendre TypeScript ou JavaScript", à laquelle on répond qu'il n'est pas possible d'apprendre le TS sans apprendre le JS.



TypeScript possède la même syntaxe, et se comporte de la même façon que JavaScript, donc vous pourrez utiliser tout ce que vous apprenez avec JavaScript, dans TypeScript.

- Les ressources JavaScript de Microsoft
- Le guide JavaScript dans les Mozilla Web Docs

> TypeScript offre toutes les fonctionnalités de JavaScript, avec une couche supplémentaire de fonctionnalités : le système de typage.

JavaScript

- JavaScript fournit des primitives, comme string et number, mais aucune vérification n'est faite pour s'assurer que les assignations que vous faites sont correctes.

TypeScript

- **Types par Inférence**
  - TypeScriptgénére les types JavaScript par défaut.

- **Définition des Types**
  - TypeScript offre la possibilité de définir vos types.
- **Composition de Types**
  - Il est possible de combiner plusieurs types simples en un type complexe.
- **Types Génériques**
  - Les types génériques fournissent des variables aux types.
  - Vous pouvez utiliser les types génériques avec vos propres types.
- **Système Structurel de Types**
  - Au cœur de TypeScript c'est la vérification dela forme de la valeur.
  - Deux objets de même forme sont considérés comme étant du même type.
- **Interfaces**
  - Description et identification des types personnalisés.

Choix de l'éditeur, des plug-ins et l’installation.

TypeScript (tout comme JavaScript) peut être programmer au moyen de n'importe quel éditeur de texte.

> Installer les outils TypeScript

Certains éditeur et ou IDE offre cependant une meilleure exppérience de développement en le supportant par défaut.

On distingue notamment :

- Visual Studio
- Visual Studio Code
- WebStorm

Ecosystème d'outils.

Naviguer dans un nouvel écosystème d'outils pdemande du temps et un accès à l'information de qualité.

> Les **awesome list** sont des liste de ressources organisées par la communauté de développeurs.

Il existe des listes sur la plupart des écossytèmes informatique, des applications CLI aux livres.

Le référentiel principal sert de liste organisée de listes impressionnantes.

> http://awesome.re

Configuration de l'environnement de développement.

> https://www.typescriptlang.org/download

TypeScript peut être installé l'utilitaire NPM

```
> npm install -g typescript
```

Configuration.

Votre configuration peut alors être couplée :

- A votre éditeur si celui-ci le supporte
- A des instructions passée en ligne de commande (CLI)
- A des instructions fournies à un "transpilers" tiers.

> La matérialisation des options de configuraton d'un projet TypeScript se fait dans le fichier **tsconfig.json**

Transpilation et débogage

- Les fichiers sources possèdent l'extension **.ts**
- Les fichiers "transpilés" possèdent l'extension **.js**
- L'utilitaire de compiliation (transformation du code ou transpilation) est nommé **tsc** (TypeScript Compiler)
    - Il est accessible en ligne de commande.

Transpilation example :

Fichier source (index.ts)

```
function greeter(person) {
  return "Hello, " + person;
}

let user = "Jane User";

document.body.textContent = greeter(user);
```

Ligne de commande

```
tsc greeter.ts
```

Fichier de sortie (index.js)

```
function greeter(person) {
  return "Hello, " + person;
}

let user = "Jane User";

document.body.textContent = greeter(user);
```

> A noter : dans ce cas le fichier source est égal au fichier de sortie.

Débogage

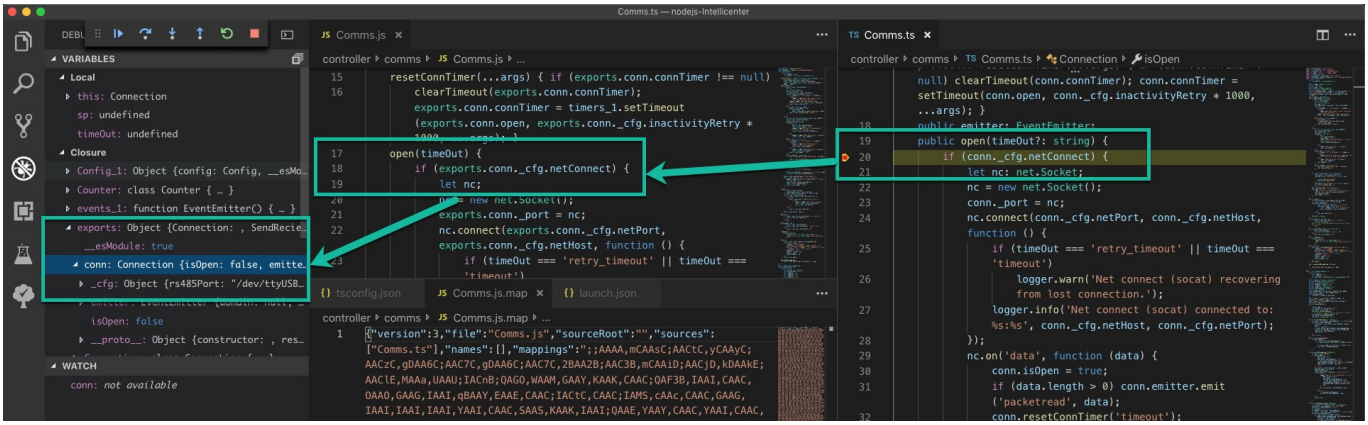
> Au mmoment de la compilation le compilateur TypeScript indique les erreurs détectées.

Source Map

> Les sources JavaScript exécutées par le navigateur sont souvent transformées d'une manière ou d'une autre à partir des sources originales créées par un développeur.

- Les sources sont souvent combinées et réduites pour rendre leur livraison depuis le serveur plus efficace.
- JavaScript s'exécutant dans une page est souvent généré par la machine, comme lorsqu'il est compilé à partir d'un langage comme CoffeeScript ou TypeScript

Un fichier "source-map" mappe la source transformée et la source d'origine, permettant au navigateur de reconstruire la source d'origine et de présenter l'original reconstruit dans le débogueur.



Ligne de commande avec fichier de débogage

```
tsc greeter.ts --sourceMap
```

Configurer le compilateur.

Configurer le compilateur.

L'exécution de **tsc** localement compilera le projet le plus proche défini par un **tsconfig.json**, vous pouvez compiler un ensemble de fichiers TypeScript.

```
/*
Exécutez une compilation basée sur une recherche d'un fichier
tsconfig.json*/
tsc

/*
Compiler uniquement le fichier index.ts avec les valeurs par défaut du
compilateur */
tsc index.ts

/*
Compiler tous les fichiers .ts du dossier avec les valeurs par défaut du
compilateur
*/
tsc src/*.ts

/*
Compiler tous les fichiers référencés selon les options du fichier
tsconfig.production.json
*/
tsc --project tsconfig.production.json

/*
Compiler des fichiers de déclaration d.ts pour un fichier JavaScript
*/
tsc index.js --declaration

/*
Compiler un "bundle" de deux fichiers de
en un seul
*/
tsc app.ts util.ts --target esnext --outfile index.js
```

> Toutes les nombreuses options du compilateurs TypeScript sont consultables à cette adresse : https://www.typescriptlang.org/docs/handbook/compiler-options.html

Ces options peuvent être persistées dans un fichier de projet.

Le fichier 'tsconfig.json'.

> Le nom d'un fichier de projet est par défaut **tsconfig.json**. Ce nom peut être personnalisé avec l'option **--project** du compilateur.

Génération du fichier **tsconfig.json**

La commande suivante génère un fichier de projet.

```
tsc --init
```

Pour des raison de simplicité et compréhension toutes les options de compilations sont présente de façon commentées dans le fichier généréré.

Référentiel du fichier **tsconfig.json**

> Ces options peuvent être supprimées du fichiers si elles ne sont pas utilisées.

Options esentielles.

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    // "rootDir": "./",
    // "moduleResolution": "node",
    // "paths": {},
    // "typeRoots": [],
    // "allowJs": true,
    // "checkJs": true,
    // "sourceMap": true,
    // "outFile": "./",
    // "outDir": "./",
    "esModuleInterop": true,
    "strict": true,
    "skipLibCheck": true,
    "include": [
        "scripts/**/*"
    ],
    "exclude": [
        "scripts/**/*"
    ]
  }
}
```

Options du compilateur.

Créer un 'build'.

En informatique, un **build** correspond à la vérification et la compilation de l'ensemble de codes sources relatifs à un projet, aboutissant idéalement à un produit executable.

> En d'autres termes un **build** TypeScript est constitué de l'ensemble des tâches que vous avez configurées.

L'installation de TypeScript nécessitant l'outil NPM, vous pouvez utilisez les **Scripts NPM** pour différencier vos **build**

- Build de développement.
- Build de production.

Générer un fichier de projet NPM (package.json)

Dans une invite de commande. `npm init`

Le fichier **package.json**

```
{
  "name": "suppport",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "Renaud Dubuis",
  "license": "ISC"
}
```

Editer la section script :

```
{
  ...
  "scripts": {
    "tsc:dev": "tsc --project ./tsconfig.dev.json",
    "tsc:prod": "tsc --project ./tsconfig.prod.json",
  },
  ...
}
```

La commande 'build'.

Une fonctionnalité attendue depuis longtemps est la construction incrémentielle intelligente pour les projets TypeScript.

```
> tsc --build
```

- Trouve tous les projets référencés
- Build les projets dans le bon ordre

```
 > tsc -b                            # Utilise le tsconfig.json dans le
répertoire courant
 > tsc -b src                        # Utilise src/tsconfig.json
 > tsc -b foo/prd.tsconfig.json bar  # Utilise foo/prd.tsconfig.json et
bar/tsconfig.json
```<!-- CONTENT ## Stratégies de projets à configuration multiples.-->

<break></break>

Stratégies de projets à configuration multiples.

Extension

Le fichier **tsconfig.json** à la possibilité d'étendre une configuration de base.

```json
{
  "extends": "path/to/baseconfiguration",
  // ...
}
```

TSConfig Bases

Selon l'environnement d'exécution JavaScript dans lequel vous avez l'intention d'exécuter votre code, il peut y avoir une configuration de base que vous pouvez utiliser sur https://github.com/tsconfig/bases/ .

```
{
  "extends": "@tsconfig/node12/tsconfig.json",
  "compilerOptions": {
    "preserveConstEnums": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

> La propriété "compilerOptions" peut être omise, auquel cas les valeurs par défaut du compilateur sont utilisées.

Project References

Les références de projet sont une nouvelle fonctionnalité de TypeScript 3.0 qui vous permet de structurer vos programmes TypeScript en plus petits morceaux.

> https://www.typescriptlang.org/docs/handbook/project-references.html

```
 // ...
"references": [
    { "path": "../src" }
]
// ...
```

> Les références de projet permettent l'import de fichiers entre projet.

Inclusion et exclusion de ressources.

Les options **include** et **exclude** spécifient un tableau de noms de fichiers ou de modèles à inclure/exclure du programme.

Ces noms de fichiers sont résolus par rapport au répertoire contenant le fichier tsconfig.json.

> https://www.typescriptlang.org/tsconfig#include

```
{
  "include": ["src/**/*", "tests/**/*"]
}
```

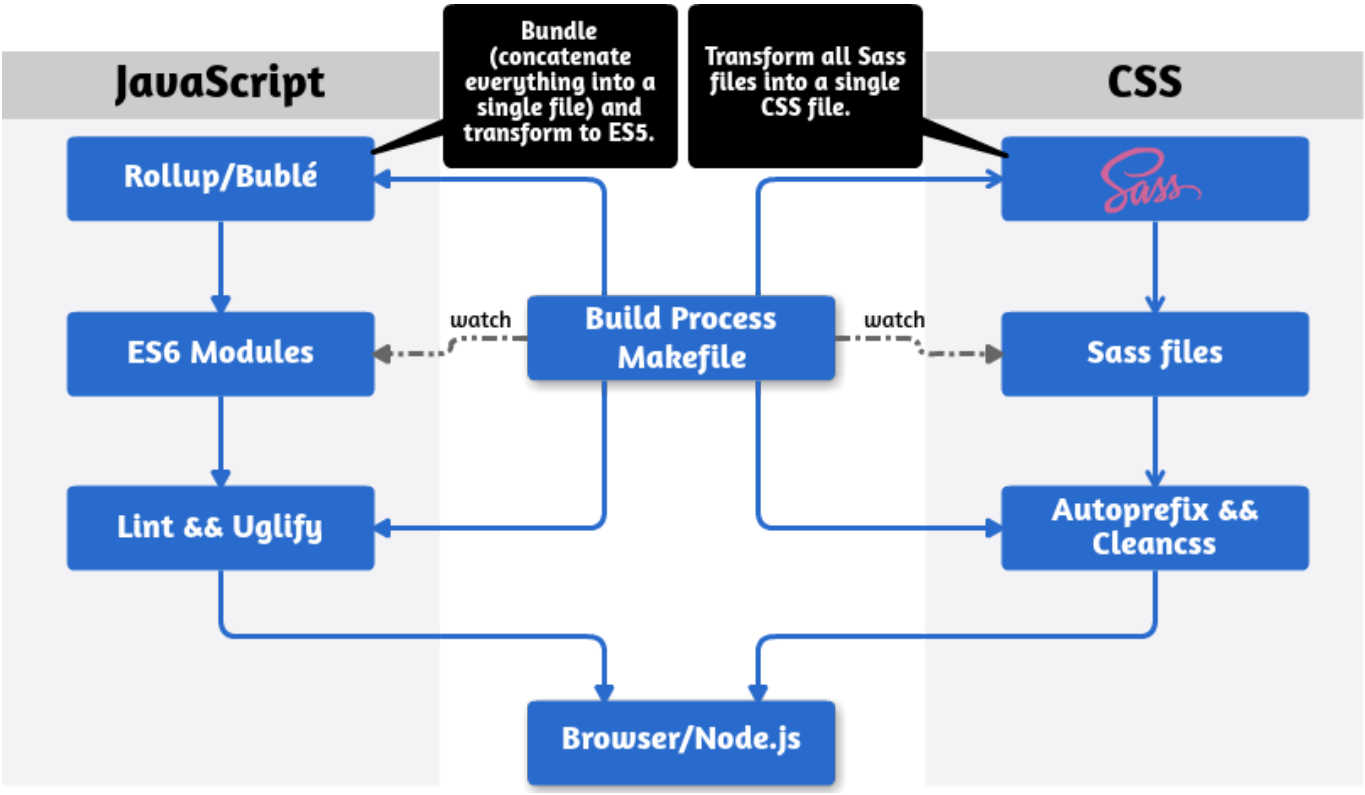
Donnerait la prise en compte suivante :

```
.
├── scripts                ×
│   ├── lint.ts            ×
│   ├── update_deps.ts     ×
│   └── utils.ts           ×
├── src                    ✓
│   ├── client             ✓
│   │   ├── index.ts       ✓
│   │   └── utils.ts       ✓
│   ├── server             ✓
│   │   └── index.ts       ✓
├── tests                  ✓
│   ├── app.test.ts        ✓
│   ├── utils.ts           ✓
│   └── tests.d.ts         ✓
├── package.json
├── tsconfig.json
└── yarn.lock
```

Création d'un workflow personnalisé.

En tant que programmeur JavaScript, vous pouvez utiliser plusieurs outils de "build" pour le développement.

> Exemple de workflow : https://www.npmjs.com/package/es6-sample-project



La création d'applications Web ou mobiles peut signifier que vous devez faire face à des tâches répétitives, mais nécessaires.

Les outils de "build" peuvent automatiser ces tâches :

- Installez la plupart des éléments liés au code pour vous.
- Automatisez les tâches subalternes et sujettes aux erreurs dans le développement Web.

Les plus connus sont :

- NPM
- Babel
- Webpack
- Gulp

> TypeScript s'intégre dans votre flux de de travail ("WorkFlow"). Selon l'outil choisit vous trouverez à cette adresse les étapes d'intégrations:
>
> https://www.typescriptlang.org/docs/handbook/integrating-with-build-tools.html

Exemple avec Webpack

```
> mkdir webpack-demo
> cd webpack-demo
> npm init -y
> npm install webpack webpack-cli ts-loader --save-dev
```

Créer le fichier de configuration **webpack.config.js**

```
const path = require('path');
module.exports = {
  entry: './src/index.ts',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

Créer le **build**

```
> npx webpack --config webpack.config.js
```

TypeScript Basics & Basic Types.

Les types TypeScript versus JavaScript.

Chaque valeur en JavaScript a un ensemble de comportements que vous pouvez observer en exécutant différentes opérations.

On peut se référer à ces ensembles par le terme **"types"**

Primitive

En JavaScript, une primitive (valeur primitive, type de données primitif) est une donnée qui n'est pas un objet et qui n'a ni méthode ni propriété. Il existe 7 types de données primitives :

- string
- number
- boolean
- bigint
- undefined
- symbol
- null

> https://developer.mozilla.org/en-US/docs/Glossary/Primitive

> L'opérateur JavaScript `typeof` appliqué sur une valeur renvoie l'unde ces types ou `object` pour les valeurs **non primitives** et `function` pour les fonctions.

Apports de TypeScript

TypeScript appporte un expression de types pour les types non considérés nativement.

- Function
- Array
- Specific Object

Ainis que de nouveaux types particuliers :

- never
- void
- any

> Le type spécial, `any` accepte tous les types, vous pouvez l'utiliser chaque fois que vous ne voulez pas qu'une valeur particulière provoque des erreurs de vérification de type.

Quand et quoi typer ?

TypeScript inclut tous les types primitifs de JavaScript - nombre, chaîne et booléen.

De plus l'inférence de type permet à TypeScript de deduire le type de la valeur.

> Assigner === typer
>
> Il convient de donc de renseigner les type pour les portions de code ambigües.

Pourquoi typer ?

- Meilleure deécouverte du code.
- Prévention des erreurs logiques.
- Documentation

Exemple de code ambigü

```
const player = {
    points
}
const bonus = 10;
let score; /* Que sera le type de score ? */

/* le paramètres person n'est pas identifiable */
function getScore(
    person
){
    /* Il n'est pas certain que person possède une propriété points de ce
fait la valeur de retour est incertaine */
    return person.points * bonus;
}

score = getScore(player)
```

Réécriture avec TypeScript

```
interface Person{
    points:number
}
const player:Person = {
    points
}
const bonus = 10;
let score:number;

function getScore(
    person:Person
):number{
    return person.points * bonus;
```

```
}

score = getScore(player);
```

Expression statique de types.

Annotations de type sur les variables.

Lorsque vous déclarez une variable à l'aide de `const, var ou let`, vous pouvez éventuellement ajouter une annotation de type pour spécifier explicitement le type de la variable.

```
let myName: string = "Alice";
```

Dans la plupart des cas, cependant, cela n'est pas nécessaire. Dans la mesure du possible, TypeScript essaie de déduire automatiquement les types dans votre code.

Fonctions

Les fonctions sont le principal moyen de transmettre des données en JavaScript. TypeScript vous permet de spécifier les types des valeurs d'entrée et de sortie des fonctions.

- Paramètres
- Valeur de retour
- Fonction anonymes

```
function greet(name: string):string {
  return "Hello, " + name.toUpperCase() + "!!";
}

const names = ["Alice", "Bob", "Eve"];

names.forEach( (s) => {
  console.log(s.toUpperCase());
});
```

Union Types

Le système de type de TypeScript vous permet de créer de nouveaux types à partir de ceux existants en utilisant une grande variété d'opérateurs.

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Erreur
printId({ myID: 22342 });
```

Tuples, les enums, les aliases (type personnalisés).

Tuples

TypeScript a une analyse spéciale des tableaux qui contiennent plusieurs types, et où l'ordre dans lequel ils sont indexés est important.

> Ceux-ci sont appelés **tuples**. Considérez-les comme un moyen de connecter certaines données, mais avec moins de syntaxe que les objets.

```
const order: [string, number] = ["computer", 200];
```

Enums

Les énumérations permettent à un développeur de définir un ensemble de constantes nommées.

> TypeScript fournit des énumérations numériques et basées sur des chaînes.
>
> https://www.typescriptlang.org/docs/handbook/enums.html

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}

enum UserResponse {
  No = 0,
  Yes = 1,
}
```

Type Aliases

Il est courant de vouloir utiliser le même type plus d'une fois et de s'y référer par un seul nom.

```
type Point = {
  x: number;
  y: number;
};

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

Cas de typage particulier (never, void...).

never

Le type **never** représente le type de valeurs qui ne se produisent jamais.

Par exemple, never est le type de retour d'une expression de fonction ou d'une expression de fonction fléchée qui lève toujours une exception ou qui ne revient jamais.

```
/*
Une  fonction retournant "never" ne doit pas avoir de valeur de retour
accessible
*/
function error(message: string): never {
  throw new Error(message);
}
```

void

void est un peu comme le contraire de **any**: l'absence d'avoir un type du tout.

```
function warnUser(): void {
  console.log("This is my warning message");
}
```

Null et Undefined

Dans TypeScript, **undefined** et **null** ont en fait leurs types nommés respectivement undefined et null.

> Toute variable, paramètre ou propriété initialisée avec **undefined** et **null** aura le type **any**

```
let score = null;
score = 10;
```

keyof

L'opérateur **keyof** prend un type d'objet et produit une chaîne ou une union littérale numérique de ses clés.

> https://www.typescriptlang.org/docs/handbook/2/keyof-types.html#the-keyof-type-operator

```
/* Le type suivant P est du même type que "x" | "y": */
type Point = { x: number; y: number };
type P = keyof Point;
```

typeof

TypeScript ajoute un opérateur typeof que vous pouvez utiliser dans un contexte de type pour faire référence au type d'une variable ou d'une propriété :

```
let s = "hello";
let n: typeof s;
```

Vous pouvez utiliser typeof pour exprimer facilement de nombreux modèles.

```
function f() {
  return { x: 10, y: 3 };
}
type P = ReturnType<typeof f>;
```

Fonctions et les tableaux.

Fonctions

Les fonctions sont le bloc de construction de base de toute application, qu'il s'agisse de fonctions locales, importées d'un autre module ou de méthodes sur une classe.

> Ce sont aussi des valeurs, et tout comme les autres valeurs, TypeScript a de nombreuses façons de décrire comment les fonctions peuvent être appelées.

```
/*
La manière la plus simple de décrire une fonction consiste à utiliser une
expression de type de fonction.
*/
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

Call Signatures

En JavaScript, les fonctions peuvent avoir des propriétés en plus d'être appelables.

Pour décrire quelque chose d'appelable avec des propriétés, nous pouvons écrire une **signature d'appel** dans un type d'objet.

```
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};

function doSomething(fn: DescribableFunction) {
  console.log(fn.description + " returned " + fn(6));
}
```

Construct Signatures

Les fonctions JavaScript peuvent également être appelées avec l'opérateur new.

> TypeScript les appelle **constructeurs** car ils créent généralement un nouvel objet.

Vous pouvez écrire une signature de construction en ajoutant le nouveau mot clé devant une signature d'appel :

```
type SomeConstructor = {
  new (s: string): SomeObject;
  /* Certains constructeurs, comme l'objet Date de JavaScript, peuvent
être appelés avec ou sans new. */
  (s: string): SomeObject;
};

function fn(ctor: SomeConstructor) {
  return new ctor("hello");
}
```

Generic Functions

Il est courant d'écrire une fonction où les types de l'entrée sont liés au type de la sortie, ou où les types de deux entrées sont liés d'une manière ou d'une autre.

> En TypeScript, les génériques sont utilisés pour décrire une correspondance entre deux valeurs.

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}
```

Constraints

Parfois, il est utile de n'opérer que sur un certain sous-ensemble de valeurs.

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}

// OK
const longerArray = longest([1, 2], [1, 2, 3]);
// OK
const longerString = longest("alice", "bob");
// Erreurr!
const notOK = longest(10, 100);
```

Tableaux

Les types d'objets génériques sont souvent une sorte de type de conteneur qui fonctionne indépendamment du type d'éléments qu'ils contiennent.

Les structures de données fonctionnant de cette façon sont réutilisables dans différents types de données.

> **number[] ou string[]**, sont en réalité un raccourci pour **Array** et **Array**.

```
const scores:number[] = []
/* Est équivalent  à */
const ages:Array<number> = []
```

ReadonlyArray

ReadonlyArray est un type spécial qui décrit les tableaux qui ne doivent pas être modifiés.

```
function doStuff(values: ReadonlyArray<string>) {
  const copy = values.slice();
  console.log(`The first value is ${values[0]}`);
  // Erreur !
  values.push("hello!");
}
```

Next-generation JavaScript.

Next-generation JavaScript.

JavaScript est considéré comme l'un des langages de script les plus largement utilisés et les plus populaires. Il a été inventé en 1995 et s'appelait à l'origine Mocha, mais est finalement devenu JavaScript. Le langage JavaScript a été inventé par BrendanEich et transformé en standard ECMA ; ES1, ES2, ES3, ES5 et ES6 sont de nombreuses versions d'ECMAScript.

JavaScript de nouvelle génération peut être considéré comme la dernière version d'ECMAScript, la spécification standardisée pour JavaScript.

> https://kangax.github.io/compat-table/esnext/

Chaque version s'efforce d'améliorer et d'ajouter de nouvelles fonctionnalités à ce langage de programmation de plus en plus populaire.

Impact sur la syntaxe JavaScript.

JavaScript ES6 est une norme garantissant l'interopérabilité des pages Web dans divers navigateurs Web.

ES6 est la 6ème version du langage de programmation JavaScript.

C'est une amélioration majeure du langage JavaScript, ajoutant plus de fonctionnalités pour faciliter le développement de logiciels à grande échelle.

Peu de fonctionnalités du script ES6 ne sont pas prises en charge par tous les navigateurs, mais la plupart sont prises en charge ; presque tous les navigateurs Web célèbres prennent en charge toutes les fonctionnalités d'ES6.

> https://kangax.github.io/compat-table/esnext/

Automatiser la documentation.

> https://typedoc.org/guides/installation/

TypeDoc convertit les commentaires du code source TypeScript en documentation HTML rendue ou en modèle JSON.

Il est extensible et prend en charge une variété de configurations. Disponible en tant que module CLI ou Node.

```
# Installation
npm install typedoc

# Exécution
npx typedoc src/index.ts
```

Utiliser TypeScript avec du code JavaScript standard.

Lorsque vous utilisez un éditeur qui utilise TypeScript, pour activer les erreurs dans vos fichiers JavaScript, ajoutez : **// @ts-check** à la première ligne de vos fichiers .js pour que TypeScript le signale comme une erreur.

> https://www.typescriptlang.org/docs/handbook/intro-to-js-ts.html#ts-check

```
// @ts-check
/** @type {number} */
var x;

x = 0; // OK
x = false; // Not OK
```

Il est alors possible d'utiliser les annotations JSDoc pour fournir des informations de type dans les fichiers JavaScript.

> https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html#type

Interopérabilité TypeScript et JavaScript.

Plusieur options du compilateur permettent l'interaction avec des fichier d'extension **.js**

allowJs

Autorisez l'importation de fichiers JavaScript dans votre projet, au lieu des seuls fichiers .ts et .tsx.

```
// @filename: card.js
export const defaultCardDeck = "Heart";
```

```
// @filename: index.ts
import { defaultCardDeck } from "./card";

console.log(defaultCardDeck);
```

> Cette option peut être utilisée comme moyen d'ajouter progressivement des fichiers TypeScript dans des projets JS en permettant aux fichiers .ts et .tsx de coexister avec des fichiers JavaScript existants.

checkJs

Fonctionne en tandem avec **allowJs**. Lorsque **checkJs** est activé, les erreurs sont signalées dans les fichiers JavaScript. Cela équivaut à inclure `// @ts-check` en haut de tous les fichiers JavaScript inclus dans votre projet.

```

```

```ts
// @filename: constants.ts
// Erreur : string expected !
module.exports.pi = parseFloat(3.124);
```

```ts
// @filename: index.ts
import { pi } from "./constants";
console.log(pi);
```

Particuarité

> https://www.typescriptlang.org/docs/handbook/type-checking-javascript-files.html#handbook-content

Inclure/générer des fichiers de définition pour la compatibilité.

Les fichiers d'extension **.d.ts** sont des fichiers de déclaration qui contiennent uniquement des informations de type.

- Ces fichiers ne produisent pas de . js sorties
- Ils ne sont utilisés que pour la vérification de type.

Générer un fichier de déclaration

```
npx  tsc src/**/*.js --declaration --allowJs --emitDeclarationOnly --
outDir types
```

Ajouter un fichier de déclaration

> https://www.typescriptlang.org/docs/handbook/declaration-files/dts-from-js.html

TypeScript réplique la résolution pour les modules depuis le **package.json**, avec une étape supplémentaire pour trouver les fichiers .d.ts.

La résolution vérifiera d'abord le champ des **types** optionnel, puis le champ **main**, et enfin essaiera de trouver **index.d.ts** à la racine.

Référentiel de rédaction

> https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html

Bénéfice des 'getter,setter' implicites.

En ES6 les classes peuvent également avoir des accesseurs :

```
class C {
  _length = 0;
  get length() {
    return this._length;
  }
  set length(value) {
    this._length = value;
  }
}
```

TypeScript a des règles d'inférence spéciales pour les accesseurs :

- Si get existe mais n'est pas défini, la propriété est automatiquement en lecture seule
- Si le type du paramètre setter n'est pas spécifié, il est déduit du type de retour du getter
- Les getters et les setters doivent avoir la même visibilité

> Depuis TypeScript 4.3, il est possible d'avoir des accesseurs avec différents types pour obtenir et définir.

Programmation 'Ahead-of-Time', réduire la dette technique.

Un logiciel Transpiler peut être utilisé pour transformer le code ES6 en ES5, qui est l'ancienne version de JavaScript et donc plus compatible avec les navigateurs.

> En informatique , la compilation anticipée (compilation AOT) est l'acte de compiler un langage de programmation (souvent) de niveau supérieur dans un langage (souvent) de niveau inférieur avant l'exécution d'un programme.

L'usage de TypeScript permet une approche dérivée **Ahead-of-Time** Progamming :

> Progammer sur la version la plus récente du language tout en présevant la compatilibilité.

Cette option est exprimée par l'option **target** du compilateur TypeScript.

```
{
    "target":"ES6"
}
```

Les navigateurs modernes prennent en charge toutes les fonctionnalités ES6, donc ES6 est un bon choix.

> https://www.typescriptlang.org/tsconfig#target

Vous pouvez choisir de définir une cible inférieure si votre code est déployé dans des environnements plus anciens, ou une cible supérieure s'il est garanti que votre code s'exécute dans des environnements plus récents.

> En combinant les fichier de projets il est possible de compiler pour plusieurs cibles.

Modules et les espaces de noms.

Modules

> https://www.typescriptlang.org/docs/handbook/modules.html

Depuis ECMAScript 2015, JavaScript a un concept de modules. TypeScript partage ce concept.

Les modules sont exécutés dans leur propre portée, pas dans la portée globale ; cela signifie que les variables, fonctions, classes, etc. déclarées dans un module ne sont pas visibles à l'extérieur du module sauf si elles sont explicitement exportées à l'aide d'un des formulaires d'exportation.

```
export interface StringValidator {
  isAcceptable(s: string): boolean;
}
```

A l'inverse, pour consommer une variable, une fonction, une classe, une interface, etc. exportée depuis un module différent, il faut l'importer à l'aide d'un des formulaires d'import.

```
import { StringValidator } from "./StringValidator";
export const numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

Espaces de noms

> https://www.typescriptlang.org/docs/handbook/namespaces.html

Les « modules internes » TypeScript aussi appelés « espaces de noms » sont une façon d'organiser votre code.

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }
}
```

```
/// <reference path="Validation.ts" />
namespace Validation {
  const lettersRegexp = /^[A-Za-z]+$/;
  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
```

```
        return lettersRegexp.test(s);
      }
    }
  }
```

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />

const lettersOnly = new Validation.LettersOnlyValidator();
```

Améliorer la POO avec TypeScript

Améliorer la POO avec TypeScript

> TypeScript utilise le même runtime que JavaScript !!!

Des langages tels que C# et Java sont ce que nous pourrions appeler des langages OOP (Obejct Oriented Programming) obligatoires.

> Dans ces langages, la classe est l'unité de base de l'organisation du code, ainsi que le conteneur de base de toutes les données et de tous les comportements lors de l'exécution.

Forcer toutes les fonctionnalités et toutes les données à être conservées dans des classes peut être un bon modèle de domaine pour certains problèmes.

Quatre piliers de la POO

Abstraction Le premier concept de la POO est l'Abstraction.

L'abstraction en POO signifie n'exposer que les détails nécessaires à l'utilisateur de la classe. Tout ce qui est sous-jacent n'a pas d'importance.

> Pour atteindre l'abstraction dans TypeScript, vous disposez de plusieurs moyens : classe/méthode abstraite, interfaces et types.

```
abstract class Character {
  public name: string;
  public damage: number;
  public attackSpeed: number;

  constructor(name: string, damage: number, speed: number) {
    this.name = name;
    this.damage = damage;
    this.attackSpeed = speed;
  }

  public abstract damagePerSecond(): number;
}

class Goblin extends Character {
  constructor(name: string, damage: number, speed: number) {
    super(name, damage, speed);
  }

  public damagePerSecond(): number {
    return this.damage * this.attackSpeed;
  }
}
```

Encapsulation

Le deuxième concept de la POO est l'encapsulation. L'encapsulation est construite sur l'idée de cacher des données. C'est la restrcition l'accès à des propriétés ou méthodes spécifiques.

```
class Character {
  private _name: string;

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

  public get name(): string {
    return this._name;
  }

  public set name(value: string) {
    this._name = value;
  }
}
```

Inheritance

L'héritage est une fonctionnalité intéressante et vous permet de réutiliser le code.

Une une nouvelle classe utilise les propriétés ou les méthodes d'une classe existante.

C'est unesous-classe, classe enfant ou classe inférieure.

Ces classes s'étendent de leur classe supérieure, souvent appelée super-classe, classe de base ou classe parente.

```
class Character {
  public name: string;
  public damage: number;

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

  public talk(): void {
    console.log('Says something ...');
  }
}

class Orc extends Character {
  public weapon: string;

  constructor(name: string, damage: number, weapon: string) {
    super(name, damage);
```

```
    this.weapon = weapon;
  }

  public attack(): void {
    console.log(`Attacks his target with his ${this.weapon}.`);
  }
}
```

Polymorphism

Le polymorphisme est la possibilité de créer une classe avec plus d'une forme.

> Ou en d'autres termes, les classes ont les mêmes méthodes mais des implémentations différentes.

```
class Character {
  public name: string;
  public damage: number;

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

  public talk(): void {
    console.log('Says something ...');
  }

  public attack(): void {
    console.log(`Attacks his target with his fists.`);
  }
}

class Orc extends Character {
  public weapon: string;

  constructor(name: string, damage: number, weapon: string) {
    super(name, damage);

    this.weapon = weapon;
  }

  public talk(): void {
    console.log('Says something but in orcish ...');
  }

  public attack(): void {
    console.log(`Attacks his target with his ${this.weapon}.`);
  }
}
```

Rappels des Pattern POO : Singleton, Factory

> Par définition, un design pattern (patrons de conception) est une solution à un problème récurrent dans la conception d'application orientée objet.

La refactorisation est un procédé modulable qui permet d'améliorer le code sans y ajouter de nouvelles fonctionnalités.

Les patrons de conception sont des solutions classiques à des problèmes récurrents de la conception de logiciels. Ce sont des plans ou des schémas que l'on peut personnaliser afin de résoudre un problème récurrent dans notre code.

> https://refactoring.guru/fr/design-patterns

Que trouve-t-on dans un patron de conception ?

La majorité des patrons sont présentés de façon très générale, afin qu'ils soient reproductibles dans tous les contextes.

Voici les différentes sections que vous retrouverez habituellement dans la description d'un patron :

- L'Intention du patron permet de décrire brièvement le problème et la solution.
- La Motivation explique en détail la problématique et la solution offerte par le patron.
- La Structure des classes montre les différentes parties du patron et leurs relations.

> Les patrons de conception sont une boîte à outils de solutions fiables et éprouvées utilisées en réponse à des problèmes classiques de la conception de logiciels.

Classification des patrons de conception

> Tous les patrons de conception peuvent être catégorisés selon leur intention ou leur objectif. Ce livre couvre les trois groupes principaux de patron

- Les Patrons de **création** fournissent des mécanismes de création d'objets, ce qui augmente la flexibilité et la réutilisation du code.
- Les Patrons **structurels** expliquent comment assembler des objets et des classes en de plus grandes structures, tout en les gardant flexibles et efficaces.
- Les Patrons **comportementaux** mettent en place une communication efficace et répartissent les responsabilités entre les objets.

Usage des interfaces TypeScript 'Duck Typing'

> L'un des principes fondamentaux de TypeScript est que la vérification de type se concentre sur la forme des valeurs. Ceci est parfois appelé "typage de canard" ou "typage structurel".

Dans un système de type structurel, si deux objets ont la même forme, ils sont considérés comme étant du même type.

```
interface Point {
  x: number;
  y: number;
}

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}

const point = { x: 12, y: 26 };
logPoint(point);
```

Interfaces

Une déclaration d'interface est une autre façon de nommer un type d'objet :

```
interface Point {
  x: number;
  y: number;
}

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

propriétés et méthodes

Une interface peut décrire les propriété aussi bien que les méthodes.

```
interface Point {
  x: number;
  y: number;
  getCoord():{x:number,y:number};
}

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
```

```
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

Alias vs Interfaces

Les alias de type et les interfaces sont très similaires et, dans de nombreux cas, vous pouvez choisir librement entre eux.

Presque toutes les fonctionnalités d'une interface sont disponibles en type, la principale distinction est qu'un type ne peut pas être rouvert pour ajouter de nouvelles propriétés par rapport à une interface qui est toujours extensible.

Extension

```
/* Etendre une interface */
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}
```

```
/* Etendre une type */
type Animal = {
  name: string
}

type Bear = Animal & {
  honey: boolean
}
```

Modification

```
interface Window {
  title: string
}

interface Window {
  ts: TypeScriptAPI
}
```

```
//  Erreur : Un type ne peut pas être modifié après avoir été créé
type Window = {
  title: string
}

type Window = {
```

```
    ts: TypeScriptAPI
  }
```

Type Assertions

> Parfois, vous aurez des informations sur le type d'une valeur que TypeScript ne peut pas connaître.

Dans cette situation, vous pouvez utiliser une assertion de type pour spécifier un type plus spécifique :

```
const myCanvas =
<HTMLCanvasElement>document.getElementById("main_canvas");
```

Classes & les interfaces.

TypeScript offre une prise en charge complète du mot-clé class introduit dans ES2015.

```
class Point {
  x = 0;
  y = 0;
}
```

> Le paramètre **strictPropertyInitialization** contrôle si les champs de classe doivent être initialisés dans le constructeur.

Notez que le champ doit être initialisé dans le constructeur lui-même.

```
class GoodGreeter {
  name: string;
  // Si vous avez l'intention d'initialiser un champ par des moyens autres
que le constructeur
  age!: string;
  // Les champs peuvent être préfixés avec le modificateur readonly.
  readonly code = 123;

  constructor() {
    this.name = "hello";
  }
}
```

> Vous pouvez utiliser une clause implements pour vérifier qu'une classe satisfait une interface particulière. Une erreur sera émise si une classe ne parvient pas à l'implémenter correctement.

```
interface Soldier {
    name: string;
    damage: number;
}

class Character {
  public name: string;
  public damage: number;

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

class Orc extends Character implements Soldier {
  public weapon: string;
```

```
  public attack(): void {
    console.log(`Attacks his target with his ${this.weapon}.`);
  }
}
```

static Blocks

Les blocs statiques vous permettent d'écrire une séquence d'instructions avec leur propre portée qui peuvent accéder à des champs privés dans la classe contenante.

```
class Foo {
    static #count = 0;

    get count() {
        return Foo.#count;
    }

    static {
        try {
            const lastInstances = loadLastInstances();
            Foo.#count += lastInstances.length;
        }
        catch {}
    }
}
```

Generic Classes

Les classes, tout comme les interfaces, peuvent être génériques. Lorsqu'une classe générique est instanciée avec new, ses paramètres de type sont déduits de la même manière que dans un appel de fonction.

```
class Box<Type> {
  contents: Type;
  constructor(value: Type) {
    this.contents = value;
  }
}

const b = new Box("hello!");
```

abstract Classes and Members

Les classes, les méthodes et les champs de TypeScript peuvent être abstraits.

Une méthode abstraite ou un champ abstrait est celui qui n'a pas eu d'implémentation fournie. Ces membres doivent exister à l'intérieur d'une classe abstraite, qui ne peut pas être directement instanciée.

```
abstract class Base {
  abstract getName(): string;

  printName() {
    console.log("Hello, " + this.getName());
  }
}
// Erreur
const b = new Base();
```

Modificateurs de classe : 'public, private, readonly...'

Les champs peuvent être préfixés avec des modificateurs.

- **static** Définit une propriété/méthode sur l'object Class.
- **public** est la visibilité par défaut des membres de la classe. Un membre public est accessible partout.
- **protected** les membres ne sont visibles que pour les sous-classes de la classe dans laquelle ils sont déclarés.
- **private** est comme protected, mais n'autorise pas l'accès au membre même à partir de sous-classes.
- **readonly** Empêche les affectations au champ en dehors du constructeur.

```
class GoodGreeter {
  public name: string;
  private  age!: string;
  readonly code = 123;

  constructor() {
    this.name = "hello";
  }
}
```

Méthodes: 'private, protected, overrides...'

Les modificateurs de visibilité fonctionne de la même manière pour les méthodes.

Overriding Methods

Une classe dérivée peut également remplacer un champ ou une propriété de classe de base.

Vous pouvez utiliser le mot clé **super** pour accéder aux méthodes de la classe de base.

> Notez que parce que les classes JavaScript sont un simple objet de recherche, il n'y a pas de notion de "super champ".
>
> https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Object_prototypes

```
class Base {
  greet() {
    console.log("Hello, world!");
  }
}

class Derived extends Base {
  override greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name.toUpperCase()}`);
    }
  }
}

const d = new Derived();
d.greet();
d.greet("reader");
```

> L'option du compilateur **--noImplicitOverride** vérifie que les membres de remplacement dans les classes dérivées sont marqués avec un modificateur de remplacement.

Héritage simple, multiple.

Héritage simple.

Comme d'autres langages avec des fonctionnalités orientées objet, les classes en JavaScript peuvent hériter des classes de base.

```
class Character {
  public name: string;
  public damage: number;

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

class Orc extends Character {
  public weapon: string;

  public attack(): void {
    console.log(`Attacks his target with his ${this.weapon}.`);
  }
}
```

Héritage multiple - Mixins

> Lorsqu'un objet ou une classe hérite des caractéristiques et fonctionnalités de plusieurs classes parentes, ce type d'héritage est appelé héritage multiple. **TypeScript ne prend pas en charge l'héritage multiple.**

Outre les hiérarchies OO traditionnelles, un autre moyen populaire de créer des classes à partir de composants réutilisables consiste à les créer en combinant des classes partielles plus simples.

```
class Sprite {
  name = "";
  x = 0;
  y = 0;

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

type Constructor = new (...args: any[]) => {};


function ScaleMixin<TBase extends Constructor>(Base: TBase) {
  return class Scaling extends Base {
```

```
    _scale = 1;

    setScale(scale: number) {
      this._scale = scale;
    }

    get scale(): number {
      return this._scale;
    }
  };
}

const EightBitSprite = Scale(Sprite);

const flappySprite = new EightBitSprite("Bird");
flappySprite.setScale(0.8);
console.log(flappySprite.scale);
```

Advanced types.

Advanced types.

Les s manières les plus avancées de modéliser des types elle fonctionnent en tandem avec les types utilitaires inclus dans TypeScript.

> https://www.typescriptlang.org/docs/handbook/utility-types.html

Gestion des types personnalisés.

Type Guards

Pour définir une garde de type personnalisée, il suffit de définir une fonction dont le type de retour est un prédicat de type :

```
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

let pet = getSmallPet();

if (isFish(pet)) {
  pet.swim();
} else {
  pet.fly();
}
```

Custom String Type

```
type MarkerTime =`${number| ''}${number}:${number}${number}`

let a: MarkerTime = "0-00" // Erreur
let b: MarkerTime = "0:00" // OK
let c: MarkerTime = "09:00" // OK
```

Restrictions des types génériques.

Vous pouvez déclarer un paramètre de type qui est contraint par un autre paramètre de type.

```
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
  return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m");
```

> https://www.typescriptlang.org/docs/handbook/2/generics.html#generic-constraints

Décorateurs.

Décorateurs.

Avec l'introduction des classes dans TypeScript et ES6, certains scénarios nécessitent désormais des fonctionnalités supplémentaires pour prendre en charge l'annotation ou la modification des classes et des membres de classe. Les décorateurs fournissent un moyen d'ajouter à la fois des annotations et une syntaxe de méta-programmation pour les déclarations de classe et les membres.

> Les décorateurs sont une proposition d'étape 2 pour JavaScript et sont disponibles en tant que fonctionnalité expérimentale de TypeScript.

Pour activer la prise en charge expérimentale des décorateurs, vous devez activer l'option du compilateur **experimentalDecorators** sur la ligne de commande ou dans votre **tsconfig.json**

Les décorateurs et les 'MetaData'.

Certains exemples utilisent la bibliothèque reflect-metadata qui ajoute un polyfill pour une API de métadonnées expérimentale.

Cette bibliothèque ne fait pas encore partie du standard ECMAScript (JavaScript). Cependant, une fois que les décorateurs seront officiellement adoptés dans le cadre de la norme ECMAScript, ces extensions seront proposées pour adoption.

Vous pouvez installer cette bibliothèque via npm :

```
> npm i reflect-metadata --save
```

TypeScript inclut un support expérimental pour émettre certains types de métadonnées pour les déclarations qui ont des décorateurs.

Pour activer cette prise en charge expérimentale, vous devez définir l'option du compilateur **emitDecoratorMetadata** sur la ligne de commande ou dans votre **tsconfig.json** :

```
> tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

Comprendre les 'factories' de décorateurs.

> Une fabrique de décorateurs est simplement une fonction qui renvoie l'expression qui sera appelée par le décorateur lors de l'exécution.

```
function color(value: string) {
  // this is the decorator factory, it sets up
  // the returned decorator function
  return function (target) {
    // this is the decorator
    // do something with 'target' and 'value'...
  };
}
```

Exemple

```
function first() {
  console.log("first(): factory evaluated");
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    console.log("first(): called");
  };
}

function second() {
  console.log("second(): factory evaluated");
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    console.log("second(): called");
  };
}

class ExampleClass {
  @first()
  @second()
  method() {}
}
```

Décorateurs de classe, propriété, méthode, paramètres.

Class Decorators

Un décorateur de classe est déclaré juste avant une déclaration de classe. Le décorateur de classe est appliqué au constructeur de la classe et peut être utilisé pour observer, modifier ou remplacer une définition de classe.

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}

@sealed
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }
}
```

Method Decorators

Un décorateur de méthode est déclaré juste avant une déclaration de méthode. Le décorateur est appliqué au descripteur de propriété de la méthode et peut être utilisé pour observer, modifier ou remplacer une définition de méthode.

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

Accessor Decorators

Un accesseur décorateur est déclaré juste avant une déclaration d'accesseur. Le décorateur d'accesseur est appliqué au descripteur de propriété de l'accesseur et peut être utilisé pour observer, modifier ou remplacer les définitions d'un accesseur.

```
function configurable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
    descriptor.configurable = value;
  };
}

class Point {
  private _x: number;
  private _y: number;
  constructor(x: number, y: number) {
    this._x = x;
    this._y = y;
  }

  @configurable(false)
  get x() {
    return this._x;
  }

  @configurable(false)
  get y() {
    return this._y;
  }
}
```

Property Decorators

Un décorateur de propriété est déclaré juste avant une déclaration de propriété.

```
import "reflect-metadata";
const formatMetadataKey = Symbol("format");
function format(formatString: string) {
  return Reflect.metadata(formatMetadataKey, formatString);
}
function getFormat(target: any, propertyKey: string) {
  return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}

class Greeter {
  @format("Hello, %s")
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
```

```
    let formatString = getFormat(this, "greeting");
    return formatString.replace("%s", this.greeting);
  }
}
```

Parameter Decorators

Un décorateur de paramètre est déclaré juste avant une déclaration de paramètre. Le décorateur de paramètre est appliqué à la fonction pour un constructeur de classe ou une déclaration de méthode.

```
import "reflect-metadata";
const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol,
parameterIndex: number) {
  let existingRequiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata( requiredMetadataKey, existingRequiredParameters,
target, propertyKey);
}

function validate(target: any, propertyName: string, descriptor:
TypedPropertyDescriptor<Function>) {
  let method = descriptor.value!;

  descriptor.value = function () {
    let requiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyName);
    if (requiredParameters) {
      for (let parameterIndex of requiredParameters) {
        if (parameterIndex >= arguments.length ||
arguments[parameterIndex] === undefined) {
          throw new Error("Missing required argument.");
        }
      }
    }
    return method.apply(this, arguments);
  };
}

class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }

  @validate
  print(@required verbose: boolean) {
```

```
    if (verbose) {
      return `type: ${this.type}\ntitle: ${this.title}`;
    } else {
     return this.title;
    }
  }
}
```

Créer des décorateurs personnalisés.

Les décorateurs sont un moyen de décorer les membres d'une classe, ou une classe elle-même, avec des fonctionnalités supplémentaires tout en réduisant et réduire le code générique.

> https://www.npmjs.com/package/utils-decorators

Il peut être utile d'abstraire des fontion utilitaires.

Implémenter les abstractions utiles.

Grâce au processus d'abstraction, un programmeur cache tout sauf les données pertinentes sur un objet afin de **réduire la complexité et d'augmenter l'efficacité**.

- Vous n'avez pas besoin de comprendre au niveau technique chaque l'utiliser.
- Vous n'avez besoin que d'une représentation mentale de base.
- L'abstraction ne fait une têche clairement identifiable.

TypeScript en pratique.

D'autre environnement que le navigateur surpporte TypeScript, parfois nativement.

On distingue :

- NodeJS https://nodejs.org/en/
- Bun https://bun.sh/ (support natif)
- Deno https://deno.land/ (support natif)

Usage avec Node.js.

Créer un projet nodejs

```
mkdir project-name
cd project-name
npm init -y
npm i -D typescript
npx tsc --init
mkdir src
```

Editer le fichier **tsconfig.json**

```
{
  "compilerOptions": {
    "module": "commonjs",
    "esModuleInterop": true,
    "target": "es6",
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist"
  },
  "include": ["src/**/*"],
  "lib": ["es2015"]
}
```

Editer le fichier **package.json**

```
{
  "name": "project-name",
  "version": "1.0.0",
  "main": "dist/index.js",
  "scripts": {
    "prestart": "npm run tsc",
    "start": "node dist/index.js",
    "tsc": "tsc"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
```

```
    "typescript": "2.1.6"
  }
}
```