Un peu plus loin en ES6

Types dynamiques et statiques

Les applications Angular peuvent être écrites en ES5, ES6, ou TypeScript

Alors qu'est-ce que TypeScript, et qu'est-ce qu'il apporte de plus ?

JavaScript est dynamiquement typé :

```
// on déclare implicitement un type String
let pony = 'Rainbow Dash';

// maintenant on déclare dynamiquement un type Integer
pony = 2;
```

Cette nature dynamique est formidable, mais elle est aussi un handicap dans certains cas, comparée à *d'autres langages plus fortement typés*.

Le cas le plus évident est quand on doit appeler une fonction inconnue d'une autre API en JS : *on doit lire la documentation* (ou pire le code de la fonction) pour deviner à quoi doivent ressembler les paramètres.

Sans les informations de type, *les IDEs n'ont aucun indice* pour savoir si on écrit quelque chose de faux, et les outils ne peuvent pas nous aider à trouver des bugs dans notre code.

Angular a toujours facilité les tests, mais c'est pratiquement impossible d'avoir une parfaite couverture de tests.

Un peu plus loin en ES6

Cela nous amène au sujet de la **maintenabilité**.

Le code JS peut être difficile à maintenir, malgré les tests et la documentation.

Refactoriser une grosse application JS n'est pas chose aisée, comparativement à ce qui peut être fait dans des langages **statiquement typés**.

La *maintenabilité* est un sujet important, et les *types* aident les outils, ainsi que les développeurs, à éviter les erreurs lors de l'écriture et la modification de code.

Typescript

Langage de programmation scripté orienté objet à classes, open source influencé par C# et JavaScript développé et présenté par MicroSoft en 2012 :

- **typer** les variables
- définir des **classes** et des **interfaces**
- utiliser les **annotations** (les décorateurs)
- exporter et importer des **modules**

TYPESCRIPT

- TypeScript, qui existe depuis 2012, est un **sur-ensemble de JavaScript**, ajoutant quelques fonctionnalités à ES5.
- Depuis la version 1.5, sortie en 2015, cette bibliothèque essaie d'être un sur-ensemble d'ES6, incluant toutes les fonctionnalités vues précédemment, et quelques nouveautés, comme les décorateurs.
- Écrire du TypeScript ressemble à écrire du JavaScript.
- Par convention les fichiers sources TypeScript ont l'extension **.ts**, et sont compilés en JavaScript standard, en général lors du build, avec le compilateur TypeScript.
- Le code généré reste très lisible.

```
npm install -g typescript

tsc test.ts
```

Les types de TypeScript

La syntaxe pour ajouter des informations de type en TypeScript est basique :

```
let variable: type;
```

Les différents types sont simples à retenir :

```
const poneyNumber: number = 0;
const poneyName: string = 'Rainbow Dash';
```

Dans ces cas, les types sont facultatifs, car le compilateur TS peut les deviner depuis leur valeur (c'est ce qu'on appelle **l'inférence de type**).

Le type peut aussi être défini dans notre application, avec par exemple la classe suivante **Pony** :

```
const pony: Pony = new Pony();
```

Les types de TypeScript

TypeScript supporte aussi ce que certains langages appellent des types génériques, par exemple avec un Array :

```
const ponies: Array<Pony> = [new Pony()];
```

Cet Array ne peut contenir que des poneys, ce qu'indique la notation générique <>.

Ajouter de telles informations de type aidera le compilateur à détecter des erreurs :

```
ponies.push('hello'); // error TS2345
// Argument of type 'string' is not assignable to
parameter of type 'Pony'.
```

Les types de TypeScript

Et comment faire si on a besoin d'une variable pouvant recevoir plusieurs types ? TS a un type spécial pour cela, nommé any.

```
let changing: any = 2;
changing = true; // aucune erreur
```

Si la variable ne doit recevoir que des valeurs de type number ou boolean, on peut utiliser l'union des types :

```
let changing: number|boolean = 2;
changing = true; // aucune erreur
```

Valeurs énumérées (enum)

TypeScript propose aussi des valeurs énumérées : **enum**.

Par exemple, une course de poneys dans l'application peut être soit **ready**, **started** ou **done**.

```
enum RaceStatus {Ready, Started, Done}
const race = new Race();
race.status = RaceStatus.Ready;
```

Un **enum** est en fait une valeur numérique, commençant à 0.

On peut cependant définir la valeur que l'on veut :

```
enum Medal {Gold = 1, Silver, Bronze}
```

Return types

On peut aussi spécifier le type de retour d'une fonction :

```
function startRace(race: Race): Race {
    race.status = RaceStatus.Started;
    return race;
}
```

Si la fonction ne retourne rien, on peut la déclarer avec void :

```
function startRace(race: Race): void {
    race.status = RaceStatus.Started;
}
```

Interfaces

Une fonction fonctionnera si elle reçoit un objet possédant la bonne propriété :

```
function addPointsToScore(player, points) {
    player.score += points;
}
```

Cette fonction peut être appliquée à n'importe quel objet ayant une propriété score.

Comment traduit-on cela en TypeScript ? On définit une interface, un peu comme la "forme" de l'objet.

```
function startRace(race: Race): void {
    race.status = RaceStatus.Started;
}
```

Interfaces

Cela signifie que le paramètre doit avoir une propriété nommée *score* de type number.

On peut aussi nommer ces interfaces :

```
interface HasScore {
      score: number;
}

function addPointsToScore(player: HasScore, points: number): void {
      player.score += points;
}
```

Paramètre optionnel

En JavaScript nous avons les **paramètres optionnels**.

Si on ne les passe pas à l'appel de la fonction, leur valeur sera **undefined**.

Mais en TypeScript, si on déclares une fonction avec des paramètres typés, le compilateur générera une erreur :

```
addPointsToScore(player); // error TS2346
// Supplied parameters do not match any signature of call target
```

Pour montrer qu'un paramètre est optionnel dans une fonction (ou une propriété dans une interface), on ajoute un **?** après le paramètre.

Ici, le paramètre ***points*** est optionnel :

```
function addPointsToScore(player: HasScore, points?: number): void {
    points = points || 0;
    player.score += points;
}
```

Fonctions en propriété

On peut décrire un paramètre comme devant posséder une fonction spécifique plutôt qu'une propriété :

```
interface CanRun {
      run(meters: number): void;
}

function startRunning(pony: CanRun): void {
      pony.run(10);
}

const pony = {
      run: (meters) => logger.log(`pony runs ${meters}m`)
};

startRunning(pony);
```

Copier/Coller ce code dans es6.com puis transpiler le code et exécuter le code transpilé dans la zone gauche (remplacer le code Typescript).

Classes

Une classe peut implémenter une interface.

Pour l'application que nous allons créer, un poney peut courir :

```
class Pony implements CanRun {
    run(meters) {
      logger.log(`pony runs ${meters}m`);
    }
}
```

Maintenant le compilateur nous *obligera* à implémenter la méthode run dans la classe.

Si nous l'implémentons mal, par exemple en attendant une string au lieu d'un number, le compilateur va crier :

```
class IllegalPony implements CanRun {
     run(meters: string) {
          console.log(`pony runs ${meters}m`);
     }
}
// error TS2420: Class 'IllegalPony' incorrectly implements interface 'CanRun'
// Types of property 'run' are incompatible
```

Classes

On peut aussi implémenter plusieurs interfaces :

```
class HungryPony implements CanRun, CanEat {
     run(meters) {
          logger.log(`pony runs ${meters}m`);
     }
     eat() {
          logger.log(`pony eats`);
     }
}
```

Et une interface peut en étendre une ou plusieurs autres :

```
interface Animal extends CanRun, CanEat {}

class Pony implements Animal {
     // ...
}
```

On peut utiliser le mot-clé private pour cacher une propriété ou une méthode.

Classes

Ajouter public ou private à un paramètre de constructeur est un raccourci pour créer et initialiser un membre privé ou public :

```
class NamedPony {
     constructor(public name: string, private speed: number) { }
     run() {
          logger.log(`pony runs at ${this.speed}m/s`);
     }
 }
const pony = new NamedPony('Rainbow Dash', 10);
// on définit une propriété publique avec 'Rainbow Dash' et une
propriété privée avec une valeur 10 pour ‘speed’
```

Son équivalent plus verbeux :

```
class NamedPonyWithoutShortcut {
     public name: string;
     private speed: number;

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

     run() {
          logger.log(`pony runs at ${this.speed}m/s`);
     }
}
```

Ces raccourcis sont très pratiques et très utilisés en Angular.

Décorateurs

- C’est une fonctionnalité toute nouvelle, ajoutée seulement en TypeScript 1.5.
- En effet, les composants Angular peuvent être décrits avec des **décorateurs**.
- Un décorateur est une façon de faire de la méta-programmation. Ils ressemblent beaucoup aux **annotations**, qui sont utilisées en Java.
- Généralement, les annotations ne sont pas vraiment utiles au langage lui- même, mais plutôt aux frameworks et aux bibliothèques.
- Les décorateurs sont vraiment puissants: ils peuvent modifier leur cible (classes, méthodes, etc.).
- En Angular, on utilisera les annotations fournies par le framework.
- Leur rôle est assez simple: ils ajoutent des métadonnées à nos classes, propriétés ou paramètres pour par exemple indiquer "cette classe est un composant", "cette dépendance est optionnelle", "ceci est une propriété spéciale du composant", etc...
- En TypeScript, les annotations sont préfixées par @, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction.
- Pas sur un constructeur en revanche, mais sur ses paramètres.

Décorateurs

Pour mieux comprendre ces décorateurs, essayons d'en construire un très simple par nous-mêmes, @Log(), qui va écrire le nom de la méthode à chaque fois qu'elle sera appelée.

```
class RaceService {
     @Log()
     getRaces() {
          // call API
     }

     @Log()
     getRace(raceId) {
          // call API
     }
}
```

Pour le définir, nous devons écrire une méthode renvoyant une fonction comme celle-ci :

```
const Log = function () {
     return (target: any, name: string, descriptor: any) => {
          logger.log(`call to ${name}`);
          return descriptor;
     };
};
```

Décorateurs

Selon l'emplacement du décorateur, la fonction n'aura pas exactement les mêmes arguments. Ici nous avons un décorateur de méthode, qui prend 3 paramètres :

- **target** : la méthode ciblée par notre décorateur
- **name** : le nom de la méthode ciblée.
- **descriptor** : le descripteur de la méthode ciblée, par exemple est-ce que la méthode est énumérable, etc…

Ici nous voulons simplement écrire le nom de la méthode, mais nous pourrions faire pratiquement ce que l'on veut : *modifier les paramètres, le résultat, appeler une autre fonction, etc...*

Dans notre exemple basique, chaque fois que les méthodes **getRace()** ou **getRaces()** sont exécutées, nous verrons une nouvelle trace dans la console du navigateur :

```
raceService.getRaces();
// logs: call to getRaces

raceService.getRace(1);
// logs: call to getRace
```

Décorateurs

En tant qu'utilisateur Angular, jetons un oeil à ces annotations :

```
@Component({ selector: 'app-home' })
class HomeComponent {
      constructor(@Optional() hello: HelloService {
            logger.log(hello);
      }
}
```

L'annotation @Component est ajoutée à la classe Home.

Quand Angular chargera cette application, il trouvera la classe Home, et va comprendre que c'est un composant grâce au décorateur.

Comme on le voit, une annotation peut recevoir des paramètres, ici un objet de configuration => selector