TypeScript : promesses et modules

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

TypeScript

Promesse

- Introduite dans **ES6**
- Utilisée souvent pour réaliser des traitements sur un résultat suite à une opération asynchrone
- Disposant d'une première méthode `then()` permettant de traiter le résultat une fois l'opération accomplie
- Disposant d'une deuxième méthode `catch()` exécutée en cas d'échec de l'opération
- Comme les fonctions : composée de deux parties : déclaration et utilisation

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise<void>((resolve, reject) => {
    if (test)
        resolve();
    else
        reject();
});
```

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise<void>((resolve, reject) => {
    if (test)
        resolve();
    else
        reject();
});
```

Dans la partie utilisation, on doit indiquer ce qu'il faut faire dans les deux cas : réussite (`resolve`) ou échec (`reject`)

```
promesse.then(() => console.log("test réussi") )
        .catch(() => console.log("erreur") );
// affiche test réussi car test = true
```

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise<void>((resolve, reject) => {
    if (test)
        resolve();
    else
        reject();
});
```

Dans la partie utilisation, on doit indiquer ce qu'il faut faire dans les deux cas : réussite (`resolve`) ou échec (`reject`)

```
promesse.then(() => console.log("test réussi") )
        .catch(() => console.log("erreur") );
// affiche test réussi car test = true
```

Le `catch()` n'est pas obligatoire.

TypeScript

Pour compiler, ajoutez dans `tsconfig.json` la propriété suivante

```
"target": "es6"
```

TypeScript

Pour compiler, ajoutez dans `tsconfig.json` la propriété suivante

```
"target": "es6"
```


Ou lancez la commande

```
tsc file -t es6
```

TypeScript

Une promesse peut être déclarée avec une fonction fléchée

```
var test = true;
var promesse = () => {
    return new Promise<void>((resolve, reject) => {
        if (test)
            resolve();
        else
            reject();
    });
};
```

TypeScript

Une promesse peut être déclarée avec une fonction fléchée

```
var test = true;
var promesse = () => {
    return new Promise<void>((resolve, reject) => {
        if (test)
            resolve();
        else
            reject();
    });
};
```


Pour l'utilisation, il faut appeler promesse comme une fonction

```
promesse().then(() => console.log("test réussi"))
          .catch(() => console.log("erreur"));
```

TypeScript

Remarque

Une promesse peut recevoir des paramètres et retourner un résultat

TypeScript

Exemple : déclaration

```
var division = (a: number, b: number) => {
    return new Promise((resolve, reject) => {
        if (b != 0)
            resolve(a / b);
        else
            reject("erreur : division par zéro");
    });
};
```

TypeScript

Exemple : déclaration

```
var division = (a: number, b: number) => {
    return new Promise((resolve, reject) => {
        if (b != 0)
            resolve(a / b);
        else
            reject("erreur : division par zéro");
    });
};
```


L'utilisation

```
division(10, 2).then((res) => console.log("résultat : " + res))
               .catch((error) => console.log(error));
// affiche résultat : 5

division(5, 0).then((res) => console.log("résultat : " + res))
              .catch((error) => console.log(error));
// affiche erreur : division par zéro
```

Remarque : les promesses s'exécutent d'une manière asynchrone

```
var division = (a: number, b: number) => {
  return new Promise((resolve, reject) => {
    if(b != 0)
      resolve(a / b);
    else
      reject("erreur : division par zéro");
  });
};

division(10, 2).then((res) => console.log("résultat : " + res))
               .catch((error) => console.log(error));

division(5, 0).then((res) => console.log("résultat : " + res))
              .catch((error) => console.log(error));

console.log("fin");
```

Le résultat est :

```
fin
résultat : 5
erreur : division par zéro
```

TypeScript

finally

- Introduit dans **ES2018**
- Prenant comme paramètre une fonction callback appelée lorsque l'exécution de la promesse est terminée : résolue ou rejetée.

TypeScript

Remarque : les promesses s'exécutent d'une manière asynchrone

```
division(10, 2)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error))
    .finally(() => console.log('fin'));

division(5, 0)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error))
    .finally(() => console.log('fin'));
```


Le résultat est :

```
résultat : 5
erreur : division par zéro
fin
fin
```

TypeScript

Le mot-clé `async` [ES2018]

Il permet de transformer une fonction en promesse

TypeScript

Le mot-clé `async` [ES2018]

Il permet de transformer une fonction en promesse

Considérons la fonction `somme()` suivante

```
let somme = (a: number, b: number) => a + b;
```

TypeScript

Le mot-clé `async` [ES2018]

Il permet de transformer une fonction en promesse

Considérons la fonction `somme()` suivante

```
let somme = (a: number, b: number) => a + b;
```


Le résultat est :

```
console.log(somme(2, 3));
// affiche 5
```

TypeScript

Pour transformer la fonction `somme()` en promesse, on ajoute le mot-clé `async` à sa déclaration

```
let somme = async (a: number, b: number) => a + b;
```

Maintenant, on peut l'utiliser comme une promesse

```
somme(2, 3).then(result => console.log(result));
// affiche 5
```

TypeScript

Pour transformer la fonction `somme()` en promesse, on ajoute le mot-clé `async` à sa déclaration

```
let somme = async (a: number, b: number) => a + b;
```

Maintenant, on peut l'utiliser comme une promesse

```
somme(2, 3).then(result => console.log(result));
// affiche 5
```

Appeler `somme()` comme une simple fonction JavaScript n'affiche pas le résultat

```
console.log(somme(2, 3));
// affiche Promise { 5 }
```

Le mot-clé `await`

- utilisable seulement dans des environnements asynchrones
- permettant d'interrompre l'exécution de la fonction asynchrone et attendre la résolution de la promesse

Le mot-clé `await`

- utilisable seulement dans des environnements asynchrones
- permettant d'interrompre l'exécution de la fonction asynchrone et attendre la résolution de la promesse

Considérons la promesse `somme()` qui attend 2 secondes pour retourner un résultat

```
let somme = (a: number, b: number) => {
    return new Promise((resolve) => {
        setTimeout(() => { resolve(a + b) }, 2000);
    });
};
```

Le mot-clé `await`

- utilisable seulement dans des environnements asynchrones
- permettant d'interrompre l'exécution de la fonction asynchrone et attendre la résolution de la promesse

Considérons la promesse `somme()` qui attend 2 secondes pour retourner un résultat

```
let somme = (a: number, b: number) => {
    return new Promise((resolve) => {
        setTimeout(() => { resolve(a + b) }, 2000);
    });
};
```

On veut implémenter une promesse `sommeCarre()` qui utilise la promesse `somme()`

```
let sommeCarre = async (a: number, b: number) => {
    let s: number = 0;
    somme(a, b).then(result => s = result as number);
    let result = Math.pow(s, 2);
    return result;
};
```

Pour utiliser la promesse avec les valeurs `2` et `3`

```
sommeCarre(2, 3).then(result => console.log(result));
// affiche 0 car on n'a pas obtenu le résultat de somme lorsqu'
   on a calculé le carré
```

Solution, utiliser `await` pour attendre la fin de la première promesse

```
let sommeCarre = async (a: number, b: number) => {
    let s: number = 0;
    await somme(a, b).then(result => s = result as number);
    let result = Math.pow(s, 2);
    return result;
};
```

Pour utiliser la promesse avec les valeurs `2` et `3`

```
sommeCarre(2, 3).then(result => console.log(result));
// affiche 0 car on n'a pas obtenu le résultat de somme lorsqu'
   on a calculé le carré
```

Solution, utiliser `await` pour attendre la fin de la première promesse

```
let sommeCarre = async (a: number, b: number) => {
    let s: number = 0;
    await somme(a, b).then(result => s = result as number);
    let result = Math.pow(s, 2);
    return result;
};
```

Si on teste maintenant

```
sommeCarre(2, 3).then(result => console.log(result));
// affiche 25
```

TypeScript

Considérons la promesse `carre()` prend un seul paramètre

```
var carre = (a: number) => {
    return new Promise<number>((resolve) => {
        resolve(a ** 2);
    });
};
```

TypeScript

Considérons la promesse `carre()` prend un seul paramètre

```
var carre = (a: number) => {
    return new Promise<number>((resolve) => {
        resolve(a ** 2);
    });
};
```

On peut enchaîner les `then`

```
carre(3)
    .then((res) => res + 1) //  .then((9) => 9 + 1)
    .then(carre)  // carre (10)
    .then((res) => console.log("résultat : " + res)); // affiche 100
```

TypeScript

Considérons la promesse `carre()` prend un seul paramètre

```
var carre = (a: number) => {
    return new Promise<number>((resolve) => {
        resolve(a ** 2);
    });
};
```

On peut enchaîner les `then`

```
carre(3)
    .then((res) => res + 1) //  .then((9) => 9 + 1)
    .then(carre)  // carre (10)
    .then((res) => console.log("résultat : " + res)); // affiche 100
```

Remarques

Si une valeur est retournée, la méthode `then()` suivante est appelée avec cette valeur.

TypeScript

Module

- Introduit dans **ES6**
- Un fichier pouvant contenir des variables ; fonctions, classes, interfaces...

TypeScript

Module

- Introduit dans **ES6**
- Un fichier pouvant contenir des variables ; fonctions, classes, interfaces...

Propriétés

- Il est possible d'utiliser des éléments définis dans un autre fichier : une variable, une fonction, une classe, une interface...
- Pour cela, il faut l'importer là où on a besoin de l'utiliser
- Pour importer un élément, il faut l'exporter dans le fichier source
- En transpilant le fichier contenant les `import`, les fichiers contenant les éléments importés seront aussi transpilés.

TypeScript

Étant donné le fichier `fonctions.ts` dont le contenu est

```
function somme(a: number = 0, b: number = 0) {
    return a + b;
}

function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

TypeScript

Pour exporter les deux fonctions `somme` **et** `produit` **de** `fonction.ts`

```
export function somme(a: number = 0, b: number = 0) {
    return a + b;
}

export function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

TypeScript

Pour exporter les deux fonctions `somme` et `produit` de `fonction.ts`

```
export function somme(a: number = 0, b: number = 0) {
    return a + b;
}

export function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

Ou aussi

```
function somme(a:number = 0, b:number = 0) {
    return a + b;
}

function produit(a: number = 0, b: number = 1) {
    return a * b;
}
export { somme, produit };
```

TypeScript

Pour importer et utiliser une fonction

```
import { somme } from './fonctions';

console.log(somme(2, 5));
// affiche 7
```

TypeScript

Pour importer et utiliser une fonction

```
import { somme } from './fonctions';

console.log(somme(2, 5));
// affiche 7
```

Pour importer plusieurs éléments

```
import { somme, produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

TypeScript

On peut aussi utiliser des alias

```
import  { somme as s, produit as p } from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(p(2, 5));
// affiche 10
```

TypeScript

On peut aussi utiliser des alias

```
import  { somme as s, produit as p } from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(p(2, 5));
// affiche 10
```

Ou aussi

```
import  * as f from './fonctions';

console.log(f.somme(2, 5));
// affiche 7

console.log(f.produit(2, 5));
// affiche 10
```

TypeScript

Les alias peuvent être attribués à l'exportation

```
function somme(a:number = 0, b:number = 0) {
    return a + b;
}

function produit(a: number = 0, b: number = 1) {
    return a * b;
}
export { produit as p, somme as s } ;
```

TypeScript

Les alias peuvent être attribués à l'exportation

```
function somme(a:number = 0, b:number = 0) {
    return a + b;
}

function produit(a: number = 0, b: number = 1) {
    return a * b;
}
export { produit as p, somme as s } ;
```


Pour importer

```
import * as f from './fonctions';

console.log(f.s(2, 5));
// affiche 7

console.log(f.p(2, 5));
// affiche 10
```

TypeScript

Le mot-clé import peut être utilisé comme une fonction afin d'importer dynamiquement un module (import conditionnel...). Lorsqu'il est utilisé ainsi, il renvoie une promesse :

```
import('./fonctions')
    .then((module) => {
        console.log(module.somme(2, 5));
        // affiche 7

        console.log(module.produit(2, 5));
        // affiche 10
    });
```

TypeScript

On peut aussi utiliser le `export default` (un seul par fichier)

```
export default function somme(a: number = 0, b: number = 0) {
    return a + b;
}

export function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

TypeScript

On peut aussi utiliser le `export default` (un seul par fichier)

```
export default function somme(a: number = 0, b: number = 0) {
    return a + b;
}

export function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

Pour importer, pas besoin de { } pour les éléments exporter par défaut

```
import somme from './fonctions';
import { produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

TypeScript

Attention, ici on a importé `somme` avec deux alias différents

```
import s from './fonctions';
import produit from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 7
```