08 Stats project

Tags	
Created	@July 20, 2021 6:05 PM
Updated	@September 27, 2024 7:16 PM

dans un dossier vide

```
$ npm init -y
$ tsc --init
$ npm i nodemon concurrently
```

Creer 2 dossiers: build et src

creer le fichier index.ts dans le dossier src

```
>> console.log("Test")
```

dans tsconfig.json, decommenter et remplir les lignes suivantes

```
"outDir": "./build",
"rootDir": "./src",
```

dans packages.json, on ajoute les scripts suivants:

```
"start:build": "tsc -w",
"start:run": "nodemon build/index.js",
"start": "concurrently npm:start:*"
```

On teste avec la commande:

```
$ npm start
```

On doit avoir une erreur et c'est normal, car a ce moment la il a build, mais index.js n'etait pas encore present dans le build

car concurrently, lance en meme temps les 2 scripts du haut

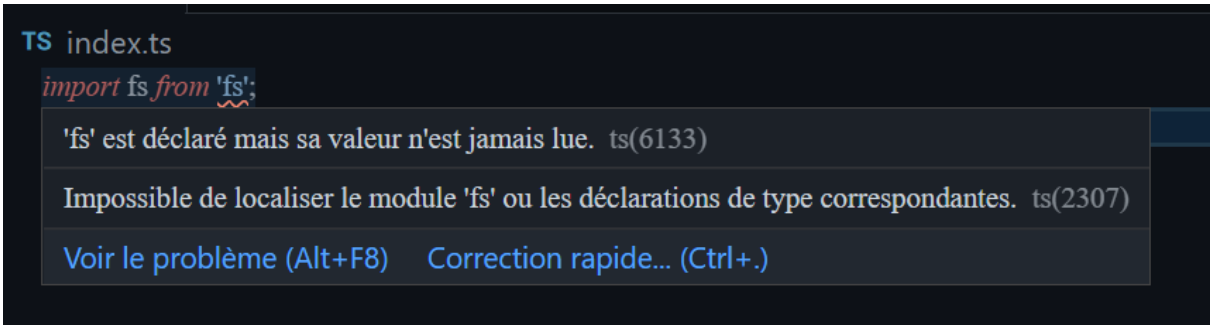
Si on relance npm start, ca va fonctionner

Si on voit le retour du console log dans le terminal, c'est que tout est okay

On telecharge le fichier csv football.csv et on le met a la racine du projet

nodejs.org/api

Files > fs.readFileSync



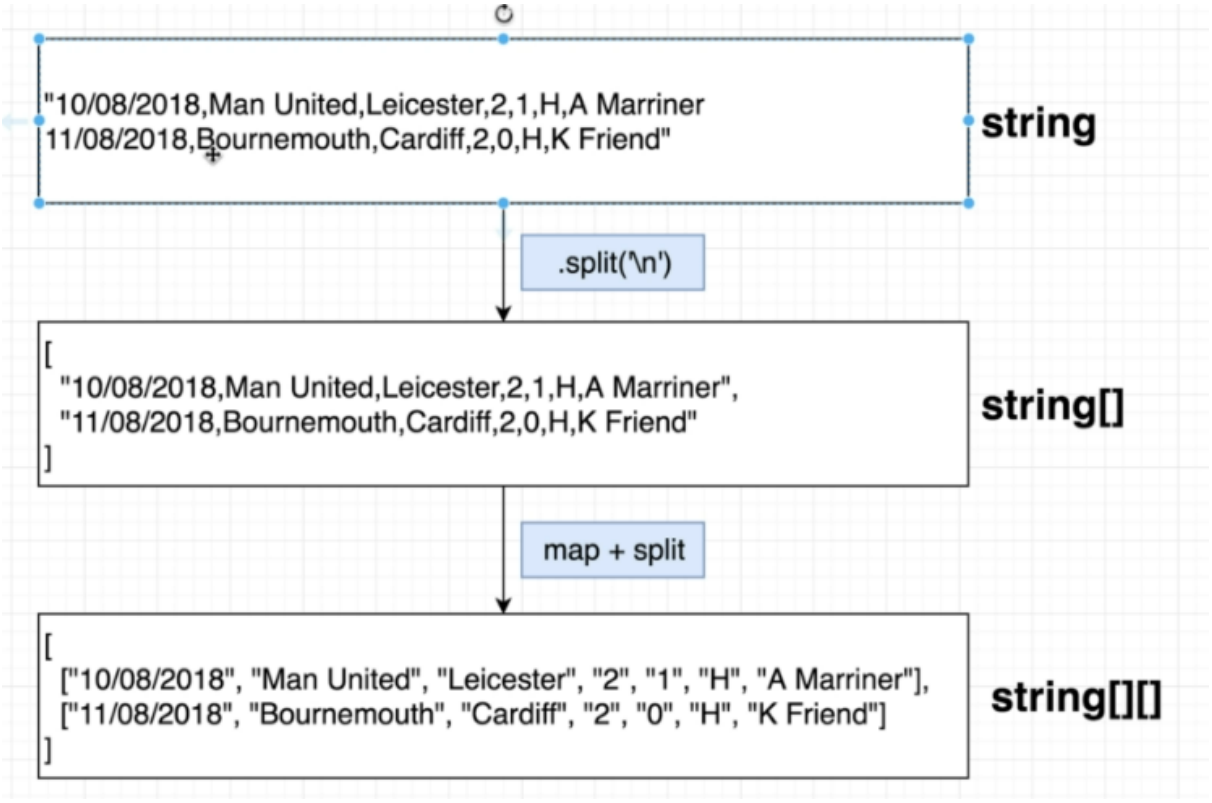
Il n'y a pas vraiment d'erreur, nous avons bien la librairie 'fs' mais nous n'avons pas son fichier de definition de type

```
npm install @types/node
```

```
// index.ts
import fs from 'fs';


const matches = fs
            .readFileSync('./src/football.csv', {encoding:
'utf-8'})


console.log(matches);
```

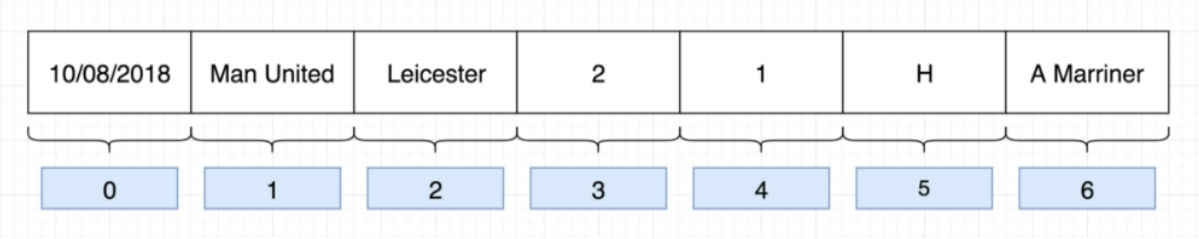
```
[start:run] 30/03/2019,Burnley,Wolves,2,0,H,C Kavanagh
[start:run] 30/03/2019,Crystal Palace,Huddersfield,2,0,H,L Probert
[start:run] 30/03/2019,Fulham,Man City,0,2,A,K Friend
[start:run] 30/03/2019,Leicester,Bournemouth,2,0,H,L Mason
[start:run] 30/03/2019,Man United,Watford,2,1,H,S Attwell
[start:run] 30/03/2019,West Ham,Everton,0,2,A,P Tierney
[start:run] 31/03/2019,Cardiff,Chelsea,1,2,A,C Pawson
[start:run] 31/03/2019,Liverpool,Tottenham,2,1,H,M Atkinson
[start:run] 01/04/2019,Arsenal,Newcastle,2,0,H,A Taylor
[start:run] 02/04/2019,Watford,Fulham,4,1,H,R East
[start:run] 02/04/2019,Wolves,Man United,2,1,H,M Dean
[start:run] 03/04/2019,Chelsea,Brighton,3,0,H,G Scott
[start:run] 03/04/2019,Man City,Cardiff,2,0,H,J Moss
[start:run] 03/04/2019,Tottenham,Crystal Palace,2,0,H,A Marriner
[start:run] 05/04/2019,Southampton,Liverpool,1,3,A,P Tierney
[start:run] 06/04/2019,Bournemouth,Burnley,1,3,A,M Atkinson
[start:run] 06/04/2019,Huddersfield,Leicester,1,4,A,D Coote
[start:run] 06/04/2019,Newcastle,Crystal Palace,0,1,A,S Attwell
[start:run] 07/04/2019,Everton,Arsenal,1,0,H,K Friend
[start:run]
[start:run] [nodemon] clean exit - waiting for changes before restart
```

Comment on va parser tout ca?




```
matches.split('\n').map(line => line.split(','))
```

```
import fs from 'fs';


const matches = fs
            .readFileSync('./src/football.csv', {encoding:
'utf-8'})
            .split('\n')
            .map((row: string): string[] => row.split
(','));


console.log(matches);
```




```
let manUnitedWins = 0;


matches.forEach((match: string[]): void => {
    if(match[1] === 'Man United' && match[5] === 'H') {
        manUnitedWins++;
    } else if(match[2] === 'Man United' && match[5] ===
'A') {
        manUnitedWins++;
    }
});


console.log(`Man United won ${manUnitedWins} games`);
```

En fait, il va avoir plusieurs problèmes à régler d'abord la comparaison de chaîne de caractère, la source de données et hardcoder, nos tableaux contiennent des data qui sont en chaîne de caractère, alors que parmi ces données on a des nombres du coup en typescript, c'est ce qu'on va régler comme problème

Si un dev ouvre notre code, il ne peut pas comprendre la signification de `H` et de `A` avant qu'il n'ouvre le CSV

```
matches.forEach((match: string[]) => {
    if(match[1] === 'Man United' && match[5] === 'H') {
        manUnitedWins++;
    } else if (match[2] === 'Man United' && match[5] === 'A') {
        manUnitedWins++;
    }
})
```

Donc l'idée ici est de le lui indiquer a l'ouverture du fichier

```
11
12      const homeWin = 'H';
13      const awayWIn = 'A';
14
15      let manUnitedWins = 0;
```

Mais ici on dit au dev qu'il y a deux possibilités gagnant ou perdant, or nous avons aussi le D pour match nul, la aussi le dev ne peut pas le deviner avant d'ouvrir un jour le csv

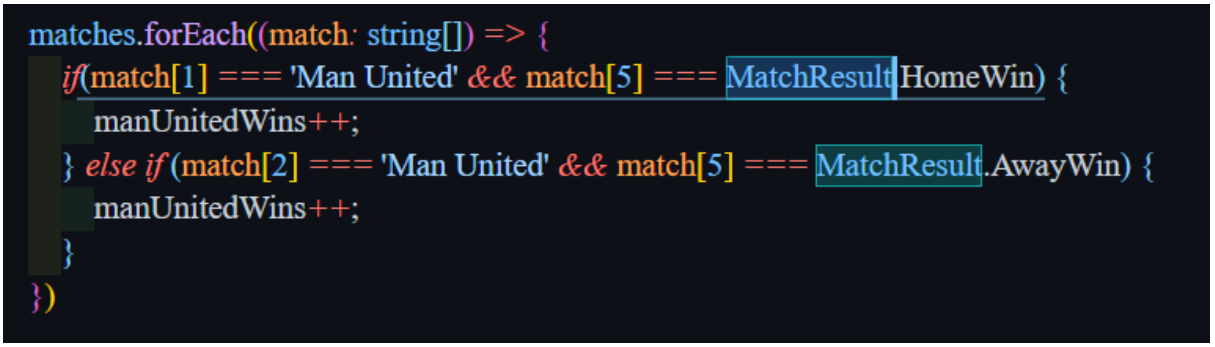
```
const homeWin = 'H';
const awayWIn = 'A';
const draw = 'D';
```

Mais la meilleur méthode lorsque nous avons plusieurs constantes possibles déjà définis est d'utiliser une classe `Enum`

Dans un fichier `MatchResult.ts` on va mettre

```
export enum MatchResult {
    HomeWin = 'H',
    AwayWin = 'A',
    Draw = 'D'
}
```

On va donc pouvoir mettre a jour notre `index.ts`

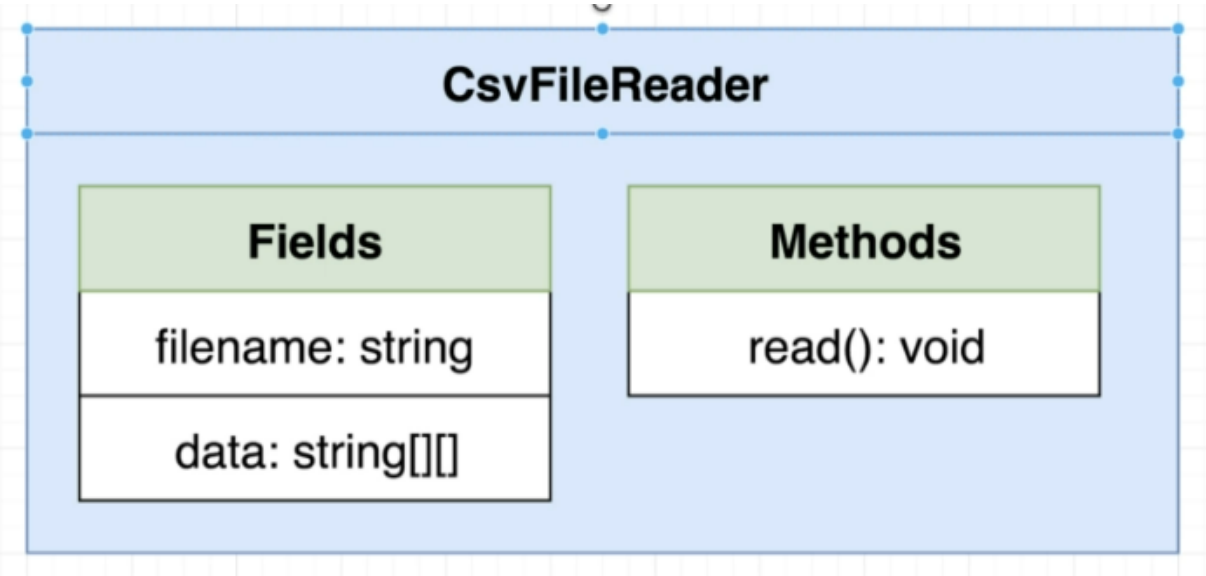


Avec les `Enums` on va pouvoir:

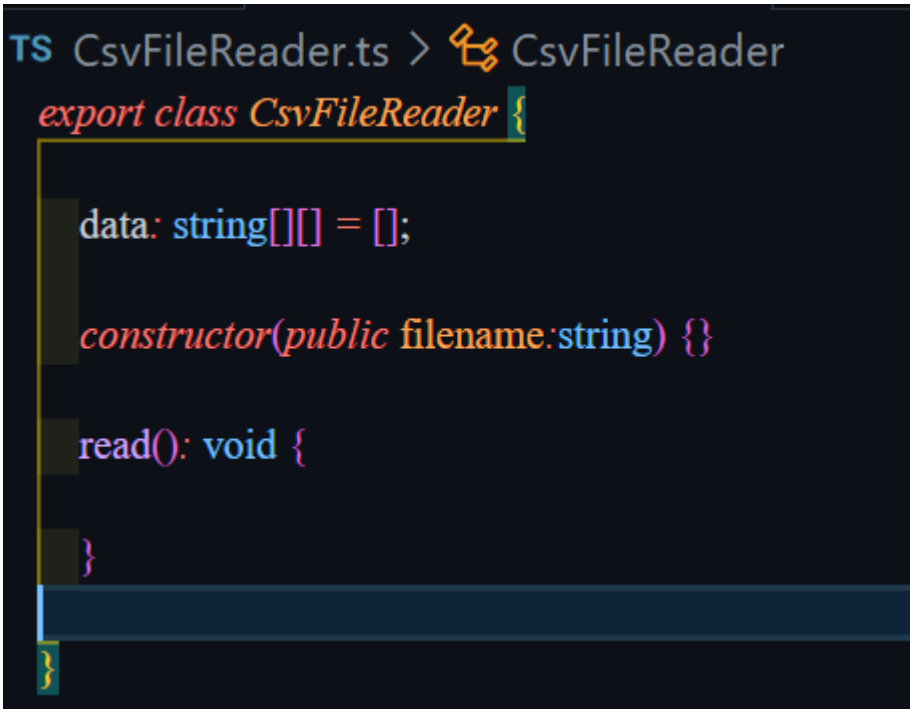
- suivre des règles de syntaxe presque identiques à celles des objets normaux
- Créer un objet avec les mêmes clés et valeurs lors de la conversion de typescript en javascript
- l'objectif principal est de signaler aux autres développeurs que ce sont toutes des valeurs étroitement liées
- et à utiliser chaque fois que nous avons un petit ensemble fixe de valeurs qui sont toutes étroitement liées et connues au moment de la compilation

CsvFileReader

Penchons nous sur la source des données



On va créer une classe pour les fichiers de type CSV




```
    read(): void {
        this.data = fs.readFileSync(this.filename, {
```

```
                encoding: 'utf-8'
            })
            .split('\n')
            .map(
                (line: string): string[] => line.split(',')
                );
    }
```

La méthode read() de notre classe CsvFileReader va effectuer plusieurs opérations sur le fichier CSV :

1. Lire le contenu du fichier avec fs.readFileSync()
2. Diviser le contenu en lignes avec split('\n')
3. Transformer chaque ligne en tableau en séparant par les virgules avec map() et split(',')

Dans `index.ts` on remplace le code qui lit le fichier csv par:

reader.read() va lancer la lecture du fichier qu'on passe en argument et va donc assigner a la variable data le contenu de notre fichier csv

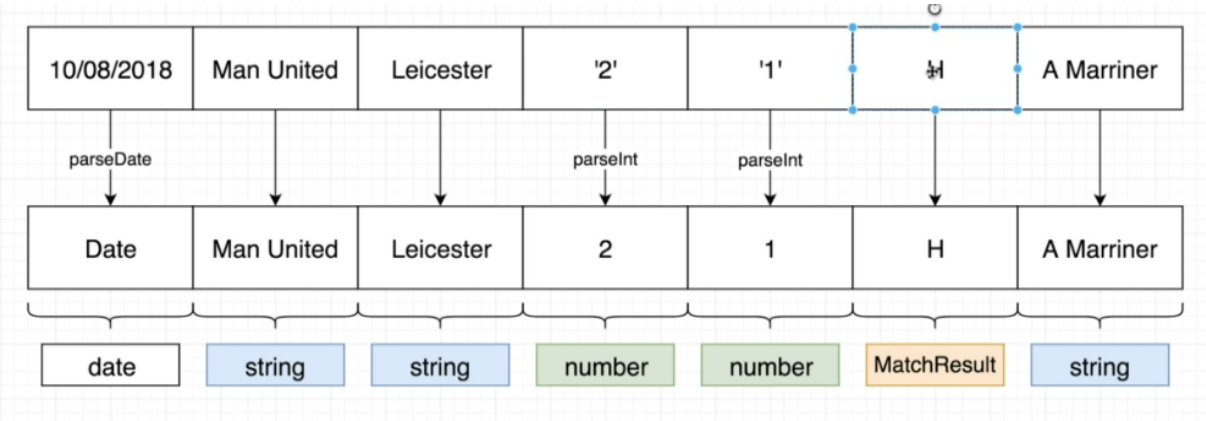
On peut donc lancer un forEach depuis `reader.data`

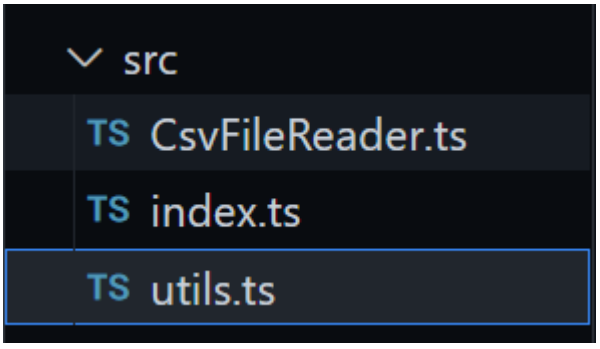
```
reader.data.forEach((match: string[]) => {
    if(match[1] === 'Man United' && match[5] === MatchResult.HomeWin) {
        manUnitedWins++;
    } else if (match[2] === 'Man United' && match[5] === MatchResult.AwayWin) {
        manUnitedWins++;
    }
})
```

Typage des donnees

Attardons nous sur le type des lignes, ici toutes les lignes et le contenu des arrays sont de type string, qu'en serait-il si le type était différent

même si nous avons des chiffres dans nos données, elles sont de type string



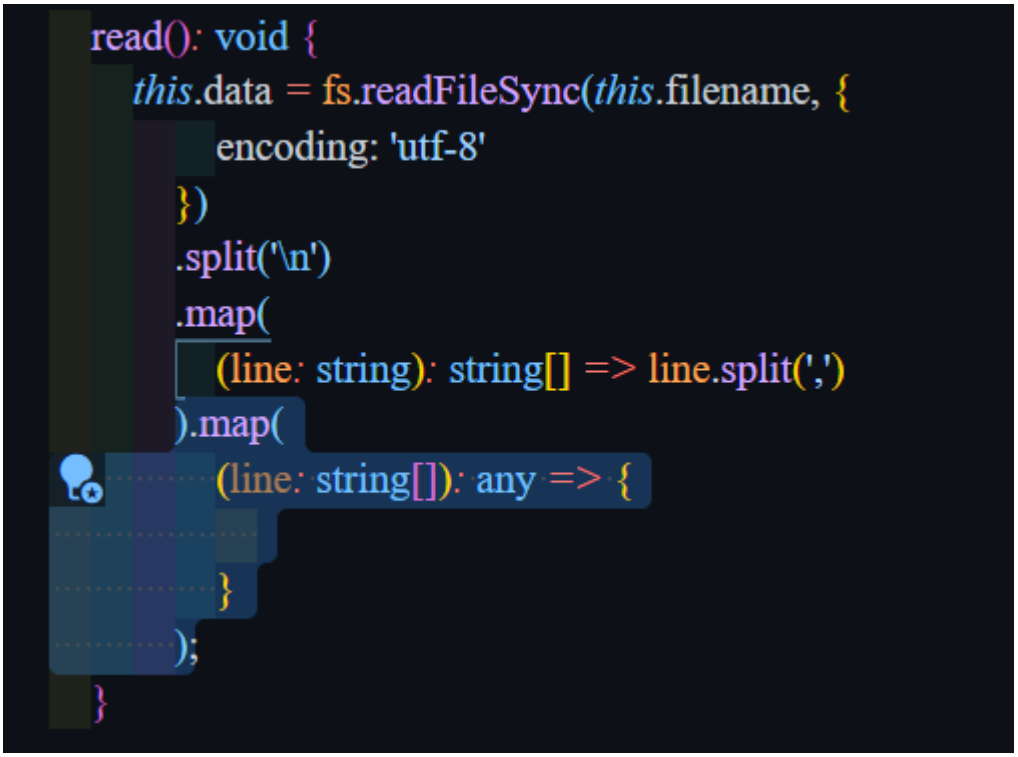


Et on va mettre notre classe `Helper` :

```
// utils.ts
export class DateUtils {
```

```
    static convertToDate(dateString: string): Date {
        const [day, month, year] = dateString.split('/').map(
        return new Date(Date.UTC(year, month - 1, day)); // D
    }
}
```

Dans `CsvFileReader.ts`, on va ajouter un nouveau `map()` qui va changer le type des données en utilisant en autres la fonction que l'on vient de créer




```
                .map(line => {
                return [
                    DateUtils.convertToDate(line[0]),
                    line[1],
                    line[2],
                    parseInt(line[3]),
                    parseInt(line[4]),
                    line[5] as MatchResult,
                    line[6]
```

```
                    ]
                })
```

Attention, on ne change pas les valeurs des données H en **HomeWin**, A en **AwayWin** etc..., on dit juste que l'élément 5 de la liste est de type **Enum MatchResult**

La preuve si on fait, dans index.ts :

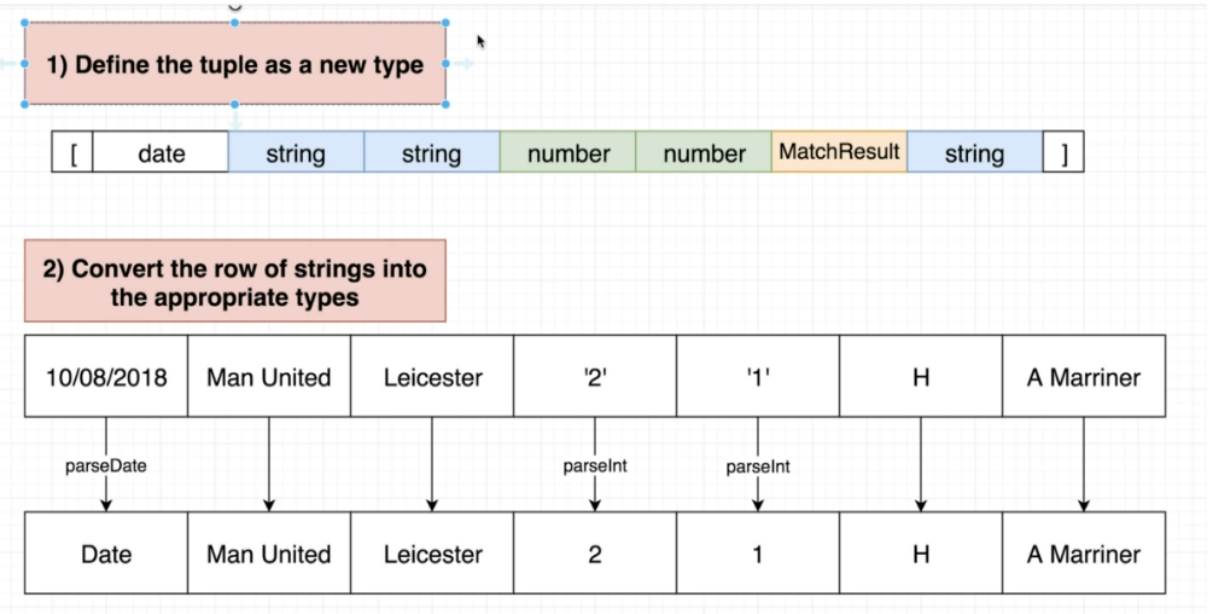

```
[start:run]    [ 2018-10-26T22:00:00.000Z, 'Liverpool', 'Cardiff', 4, 1, 'H' ],
[start:run]    [ 2018-10-26T22:00:00.000Z, 'Southampton', 'Newcastle', 0, 0, 'D' ],
[start:run]    [ 2018-10-26T22:00:00.000Z, 'Watford', 'Huddersfield', 3, 0, 'H' ],
[start:run]    [ 2018-10-27T22:00:00.000Z, 'Burnley', 'Chelsea', 0, 4, 'A' ],
[start:run]    [ 2018-10-27T22:00:00.000Z, 'Crystal Palace', 'Arsenal', 2, 2, 'D' ],
[start:run]    [ 2018-10-27T22:00:00.000Z, 'Man United', 'Everton', 2, 1, 'H' ],
[start:run]    [ 2018-10-28T23:00:00.000Z, 'Tottenham', 'Man City', 0, 1, 'A' ],
[start:run]    ... 225 more items
[start:run] ]
[start:run] [nodemon] clean exit - waiting for changes before restart
```

La propriete `data` va etre de type

```
(Date | string | number | MatchResult)[][]
```

Car c'est une liste qui pourrait contenir ces types

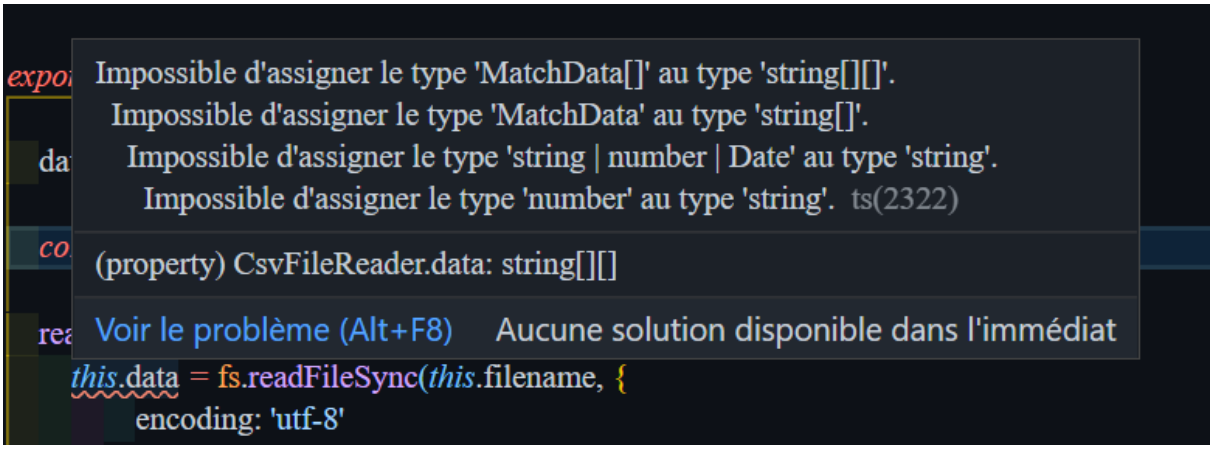
Au lieu de ca on va creer un nouveau type qui decrit ce qu'on a a l'issue du map



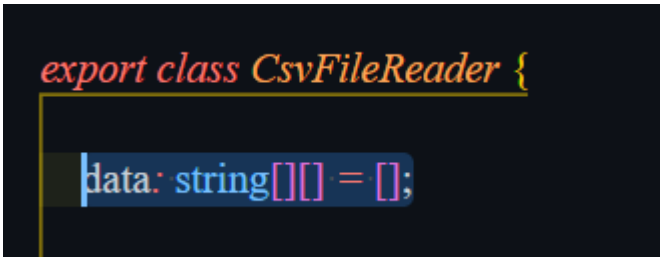
On va creer un fichier `MatchData.ts` et y mettre

```
import { MatchResult } from "./MatchResult";

export type MatchData = [Date, string, string, number, number
```

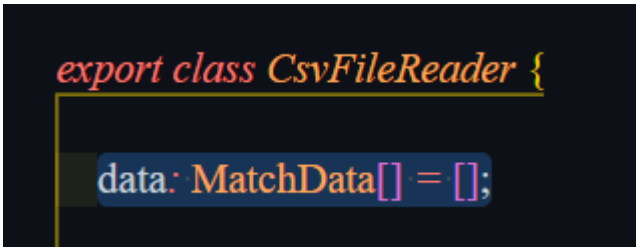
On a cependant une petite erreur qui subsiste



L'erreur vient d'ici



On corrige par:



Car `MatchData` est lui même un array, et on ajoute un deuxième array

Dernière chose a modifier, c'est le type dans le `forEach` de `index.ts`, ca doit être maintenant MatchData

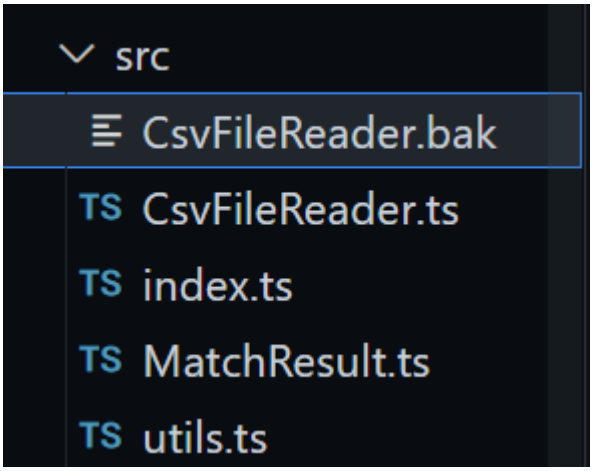
```ts
TS index.ts > reader.data.forEach() callback
import { CsvFileReader, MatchData } from "./CsvFileReader";
import { MatchResult } from "./MatchResult";

const reader = new CsvFileReader('football.csv');
reader.read();

let manUnitedWins = 0;

reader.data.forEach((match: MatchData) => {
    if(match[1] === 'Man United' && match[5] === MatchResult.HomeWin) {
        manUnitedWins++;
    } else if (match[2] === 'Man United' && match[5] === MatchResult.AwayWin) {
        manUnitedWins++;
    }
})

console.log(reader.data);
```

//////////////////////////////////

On peut encore aller plus loin, en se disant, est ce qu'un csv avec une autre structure pourrait être utiliser dans notre CsvFileReader

Est ce que d'autres csv auront un type MatchData, avec [Date, string, string....]

On duplique le fichier `CsvFileReader` et on le renomme :



ca va nous servir de backup

On creer dans `src` un dossier `heritage` et on y met `CsvFileReader.ts`

Si on regarde bien, c'est pas la partie suivante qui pose problème:

```
            .map(line => {
                return [
                    DateUtils.convertToDate(line[0]),
                    line[1],
                    line[2],
                    parseInt(line[3]),
                    parseInt(line[4]),
                    line[5] as MatchResult,
                    line[6]
                ]
            })
```

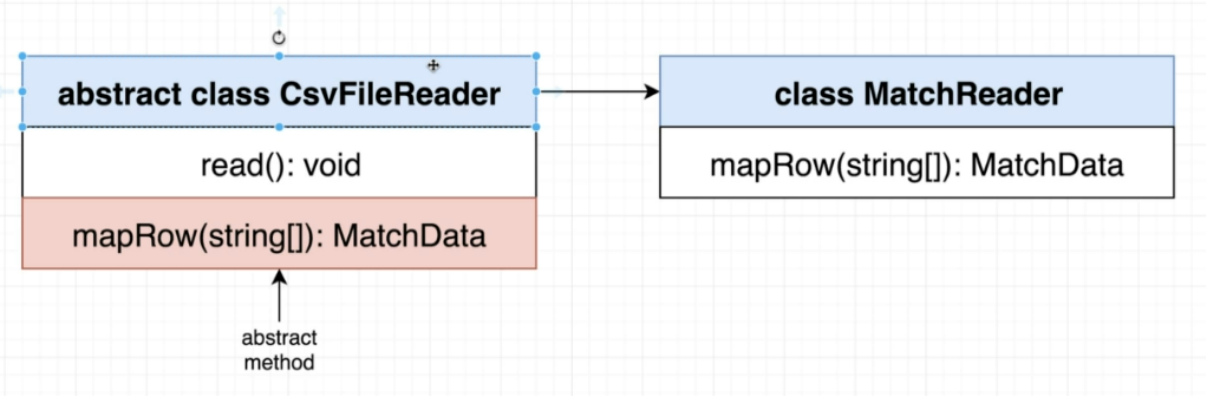
En effet, pour ce fichier `football.csv` le typage des colonnes va se faire de cette maniere, mais si demain on fournit un nouveau fichier csv avec des colonnes differentes, ce `map` ne fonctionnera pas

On va créer une méthode **mapRow**() a l'intérieur de la classe **CsvFileReader**

```
// CsvFileReader.ts
...
    mapRow(line: string[]): MatchData {
        return [
            DateUtils.convertToDate(line[0]),
            line[1],
            line[2],
            parseInt(line[3]),
            parseInt(line[4]),
            line[5] as MatchResult,
            line[6]
        ]
    }
```

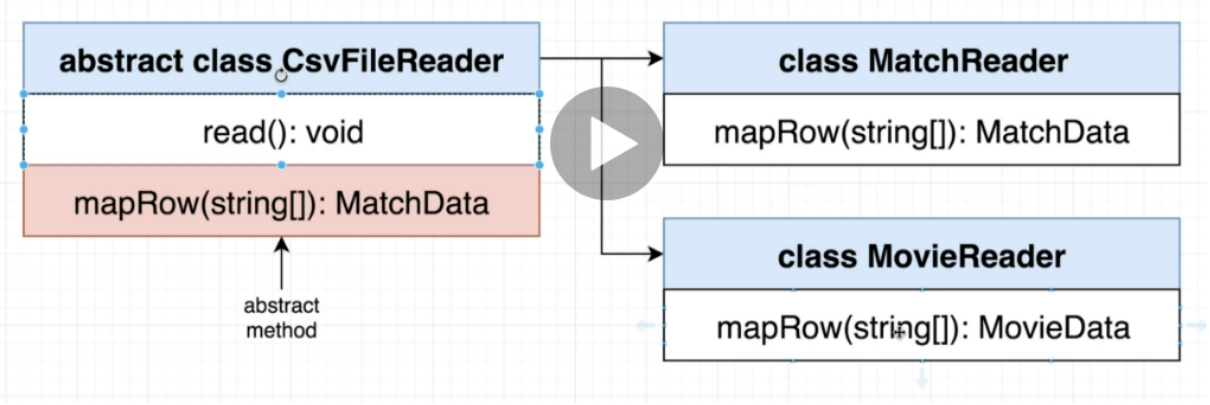
On modifie le dernier map() en conséquence dans la methode `read()`

```
read(): void {
    this.data = fs.readFileSync(this.filename, {
        encoding: 'utf-8'
    })
    .split('\n')
    .map(
        (line: string): string[] => line.split(',')
    ).map(this.mapRow);
}
```



On va créer une classe abstraite qui aura une méthode `mapRow` abstraite, donc a redéfinir

On pourra ainsi redéfinir la méthode mapRow() pour chaque type de Csv



Par exemple, un CSV contenant des films, etc...

- ---

Notre `CsvFileReader` devient la classe abstraite:

```
import fs from 'fs';
import { MatchData } from './MatchReader';


export abstract class CsvFileReader {
    protected data: MatchData[] = []; // protected permet a
ux classes enfants d'avoir acces a la propriete,

// mais interdit a l'exterieur d'y avoir acces

    constructor(public filename: string) {}


    abstract mapRow(row: string[]): MatchData;


    read(): void {
        this.data = fs
        .readFileSync(this.filename, {
            encoding: 'utf-8',
```

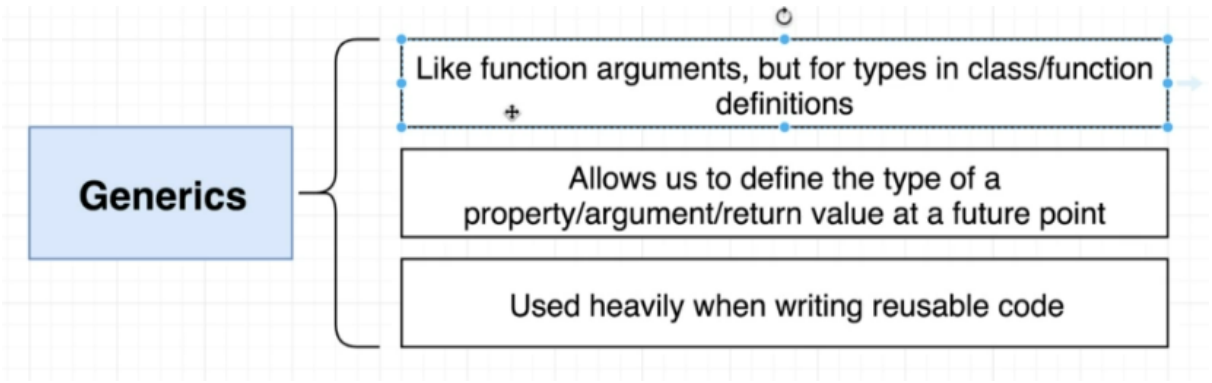
```
        })
        .split('\n')
        .map((line: string): string[] => line.split(','))
        .map(this.mapRow)
    }
}
```

Et on créer notre classe `MatchReader.ts` **(dans le dossier** `heritage` **)**qui va lire des fichiers CSV de matchs

```
import { MatchData } from "../MatchData";
import { MatchResult } from "../MatchResult";
import { DateUtils } from "../utils";
import { CsvFileReader } from "./CsvFileReader";


export class MatchReader extends CsvFileReader {
    mapRow(line: string[]) : MatchData { // ['10/08/201
8','Man United','Leicester','2','1','H','A Marriner']
        return [
            DateUtils.convertToDate(line[0]),
            line[1],
            line[2],
            parseInt(line[3]),
            parseInt(line[4]),
            line[5] as MatchResult,
            line[6]
        ]
    }
}
```

Notre classe abstraite est presque parfaite, le seul souci c'est qu'elle dépend de l'interface `MatchData`, qui est trop spécifique

Pour cela on va utiliser ce qu'on appelle les types generics



Voyons un exemple de generique avec des classes

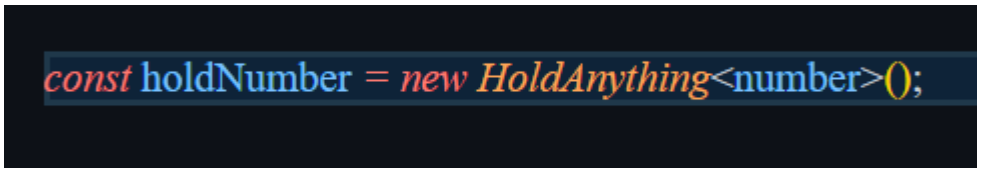
```
class HoldNumber {
    data: number;
}


class HoldString {
    data: string;
}


const holdNumber = new HoldNumber();
holdNumber.data = 123;


const holdString = new HoldString();
holdString.data = "Hello";
```

Ici j'ai créer deux classes pour gérer 2 types différents, mais les classes font exactement la meme chose

L'ideal serait de créer une classe générique, qui permet d'accepter n'importe quel type

```
class HoldAnything<T> {
    data: T
}
```

Et lorsqu'on créera un objet `HoldAnything` qui utilisera un type "number" on écrira



Donc notre classe `CsvFileReader` devient

```
import fs from 'fs'

export abstract class CsvFileReader<T> { // Class generique
    protected data: T[] = []

    constructor(public filename: string) {}

    read() {
        this.data = fs.readFileSync(this.filename, {encoding:
                .split('\n')
                .map(line => line.split(','))
                .map(this.mapRow)
    }

    abstract mapRow(line: string[]) : T
}
```

Ainsi elle ne dépend plus de rien

Notre classe `MatchReade` devient:

```
import { MatchData } from "../MatchData";
import { MatchResult } from "../MatchResult";
import { DateUtils } from "../utils";
import { CsvFileReader } from "./CsvFileReader";
```

```
export class MatchReader extends CsvFileReader<MatchData> {
// A chaque appel de CsvFileReader on doit preciser le type
    mapRow(line: string[]) : MatchData { // ['10/08/201
8','Man United','Leicester','2','1','H','A Marriner']
        return [
            DateUtils.convertToDate(line[0]),
            line[1],
            line[2],
            parseInt(line[3]),
            parseInt(line[4]),
            line[5] as MatchResult,
            line[6]
        ]
    }
}
```

Dans notre index.ts

Nous ne passons plus par une instance de **CsvFileReader**, mais de **MatchReader** (qui hérite de la classe abstraite **CsvFileReader**)

NB: On ne peut pas instancier une classe Abstraite


```
let manUnitedWins = 0;

reader.data.forEach((match: MatchData): void => {
    if(match[1] === 'Man United' && match[5] === MatchResult.HomeWin) {
        manUnitedWins++;
    } else if(match[2] === 'Man United' && match[5] === MatchResult.AwayWin) {
        manUnitedWins++;
    }
});

console.log(`Man United won ${manUnitedWins} games`);

```

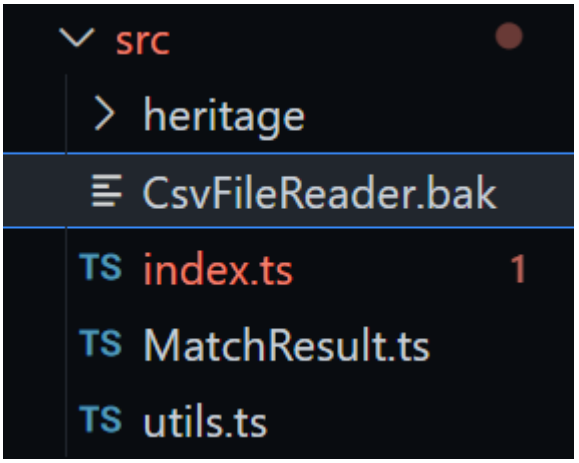
`data` est en `protected` , du coup on va mettre en place un getter dans `MatchReader.ts`

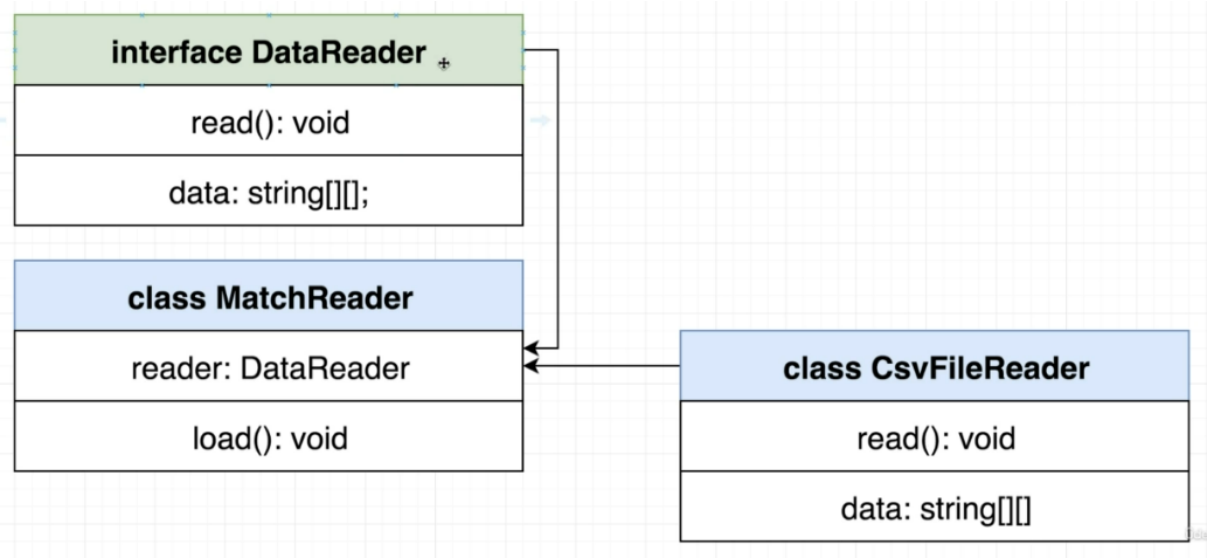
```ts
export class MatchReader extends CsvFileReader<MatchData> {

    getData(): MatchData[] {
        return this.data;
    }
```

```ts
let manUnitedWins = 0;

reader.getData().forEach((match: MatchData): void => {
    if(match[1] === 'Man United' && match[5] === MatchResult.HomeWin) {
        manUnitedWins++;
    } else if(match[2] === 'Man United' && match[5] === MatchResult.AwayWin) {
        manUnitedWins++;
    }
});

console.log(`Man United won ${manUnitedWins} games`);
```

Composition

On va créer un dossier "composition" dans `src` et mettre le fichier .bak que nous avions créer plus tot, on le renomme en ts et redevient notre base de déârt



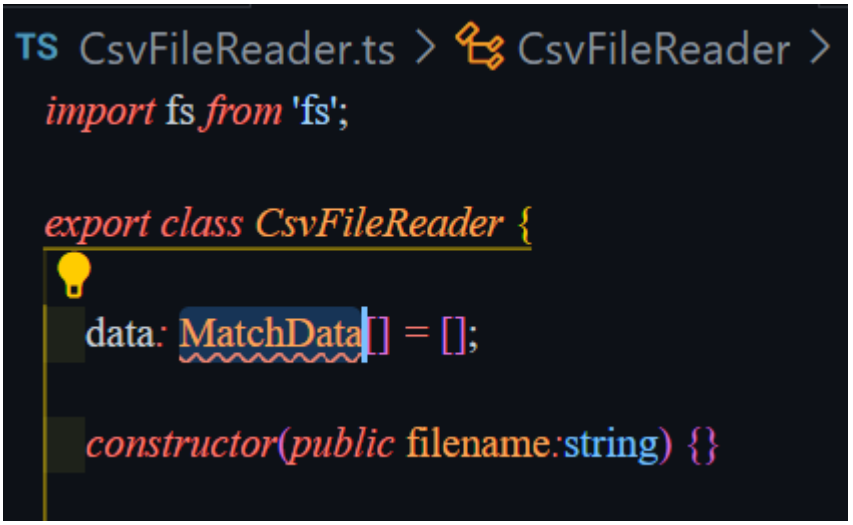


L'idée ici est que nous lisions des données de Match depuis un csv file, mais admettons que nous puissions récupérer ces mêmes données depuis un autre format qu'un csv, comme une API par exemple

On va creer le fichier `DataReader.interface.ts` dans le dossier `composition` et mettre:

```
export interface IDataReader {
    data: string[][]
    read() : void
}
```

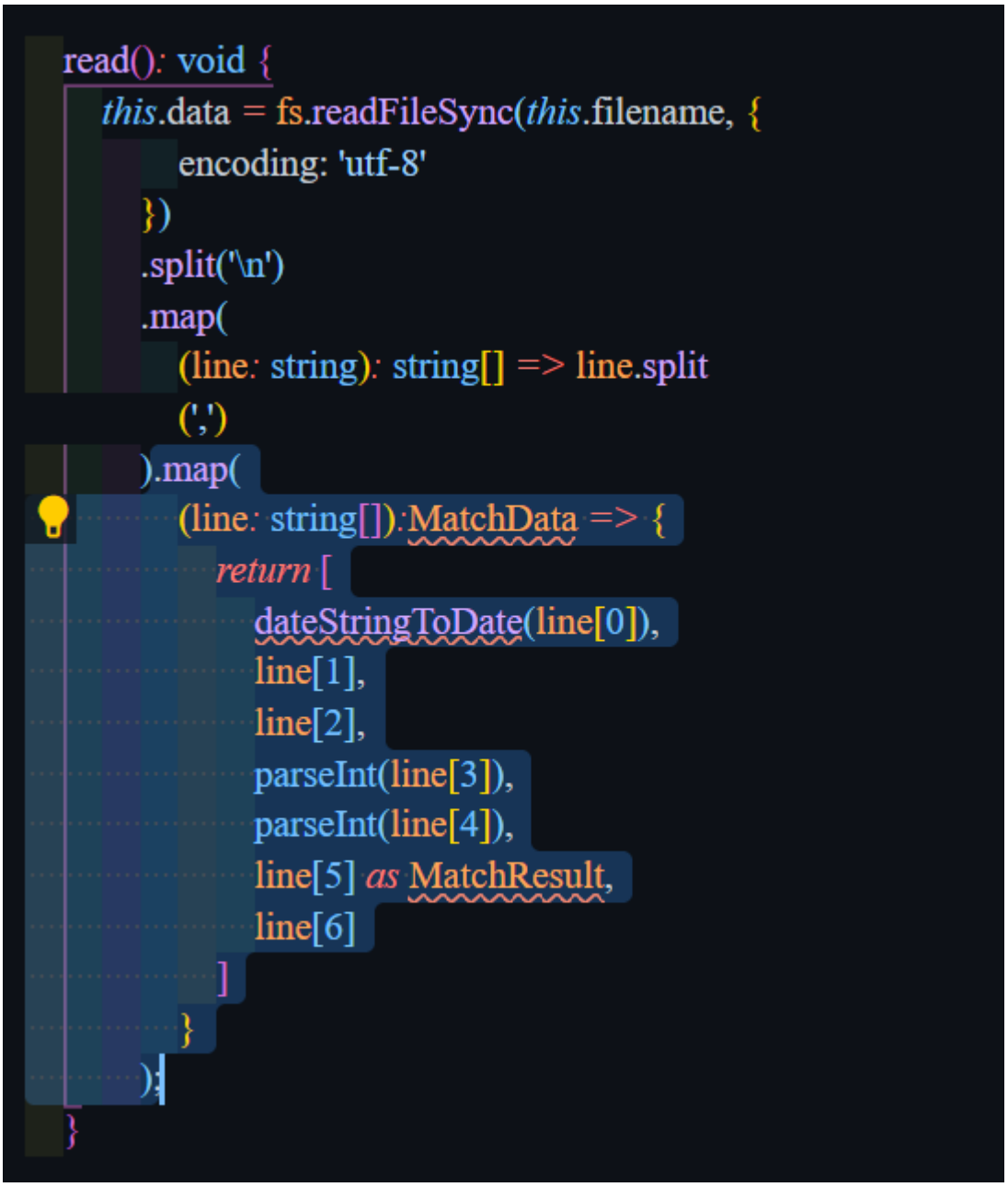
Dans `CsvFileReader.ts`



devient

```
data: string[][] = []
```

et le deuxieme `map()` de la methode `read()` est retirer sera pris en charge par la classe MatchReader



On le met en commentaire dans une classe `MatchReader` qu'on va mettre dans un fichier `MatchReader.ts` dans le dossier `composition`, on y reviendra plus tard

L'idée ici est de créer un `CsvFileReader` qui se charge de récupérer des données depuis un fichier csv,

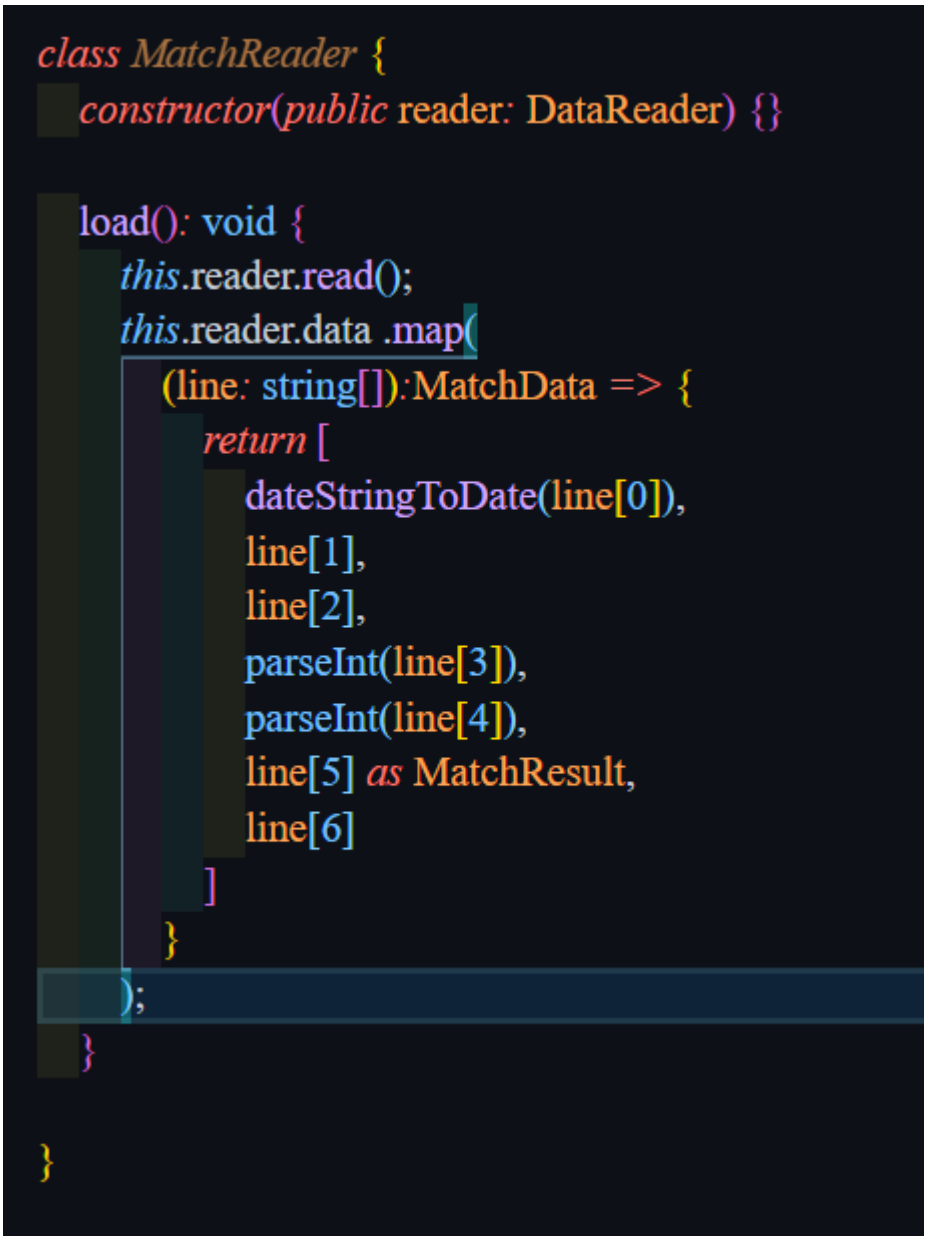
la classe MatchReader elle sera en charge de mapper chaque ligne en fonction du type souhaité

Et on obtient clairement ce qu'on souhaitait, une classe qui se charge de lire un fichier csv

Dans la classe MatchReader, on va créer une méthode qui va charger les données

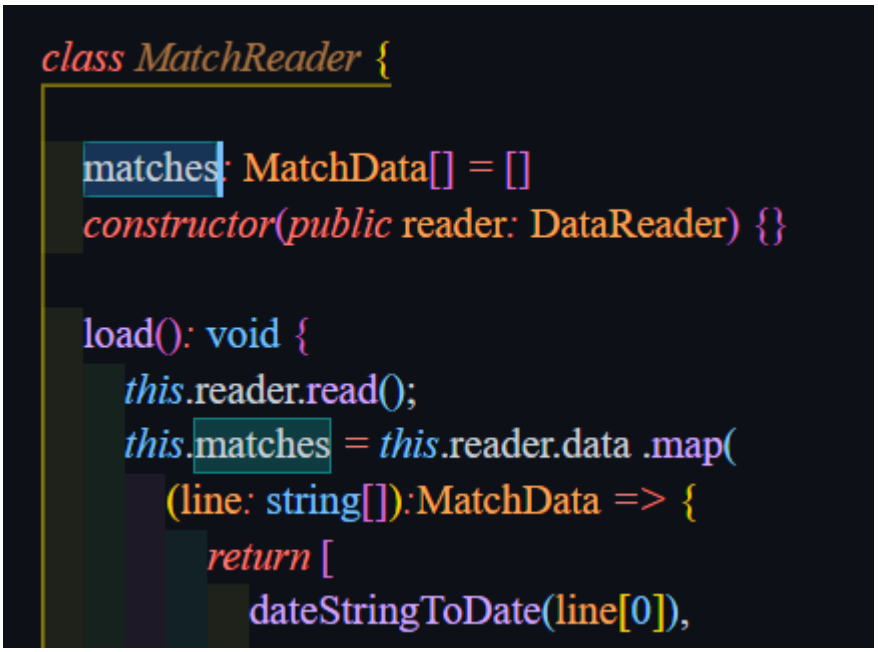
```
load(): void {
        this.reader.read()
        this.reader.data.map(line => {
            return [
                DateUtils.convertToDate(line[0]),
                line[1],
```

```
                line[2],
                parseInt(line[3]),
                parseInt(line[4]),
                line[5] as MatchResult,
                line[6]
            ]
        })
    }
```



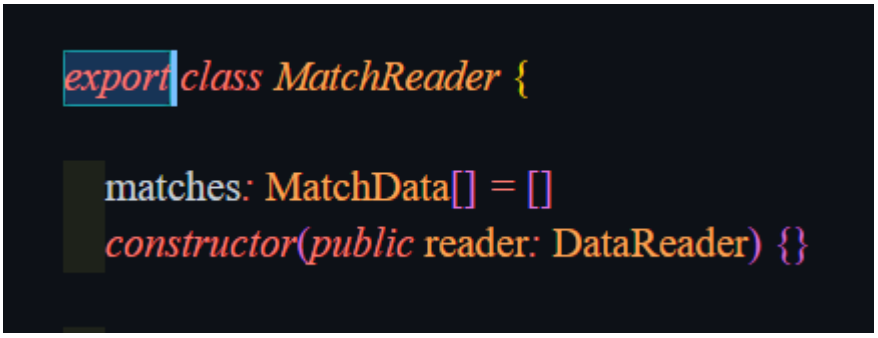
On lit d'abord les données et ensuite on les charge en redéfinissant les types de chaque élément

Ces données, on va les mettre dans une variable "matches"



On met a jour notre index.ts pour que cela fonctionne

On n'oublie pas avant d'exporter la classe MatchReader



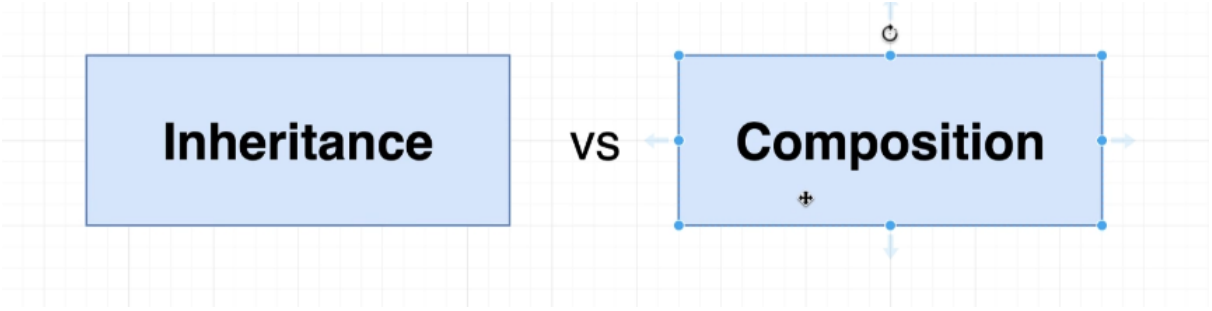
On modifie le index.ts comme suit:

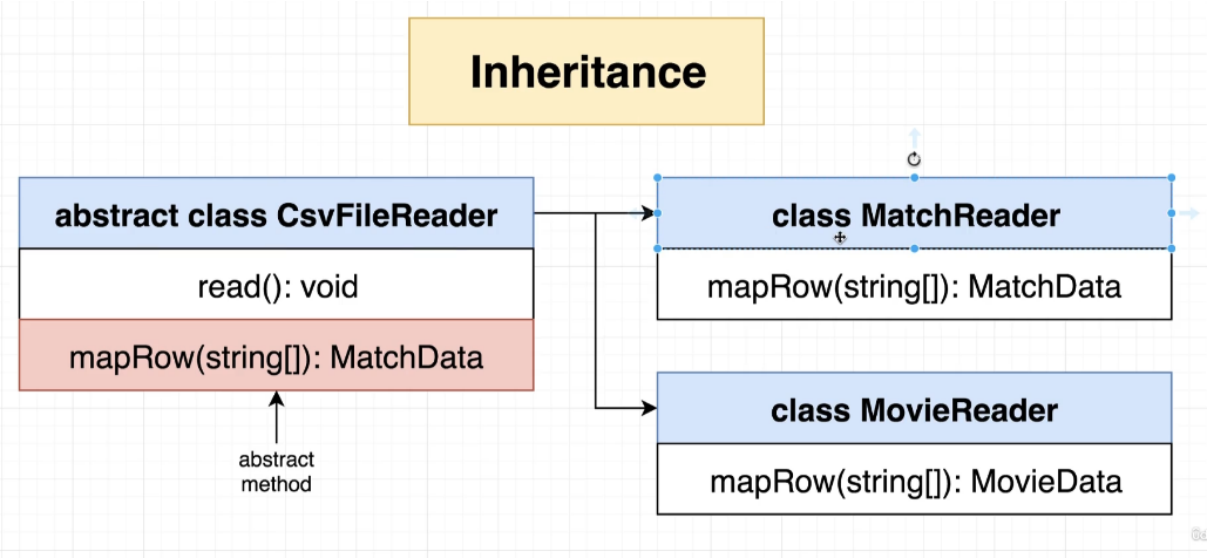
```ts
TS index.ts > ...
import { MatchReader } from "./MatchReader";
import { CsvFileReader } from "./CsvFileReader";
import { MatchResult } from "./MatchResult";
import { MatchData } from "./MatchReader";

// creation d'un objet qui satisfait l'interface DataReader
const csvFileReader = new CsvFileReader('football.csv');

// Creation d'une instance de MatchReader
const matchReader = new MatchReader(csvFileReader);
matchReader.load();
```

Et nous itérons maintenant sur la propirété `matches` de `mathcReader`

```ts
matchReader.matches.forEach((match: MatchData) => {
    if(match[1] === 'Man United' && match[5] === MatchResult.
    HomeWin) {
        manUnitedWins++;
    } else if (match[2] === 'Man United' && match[5] ===
    MatchResult.AwayWin) {
        manUnitedWins++;
    }
})
```

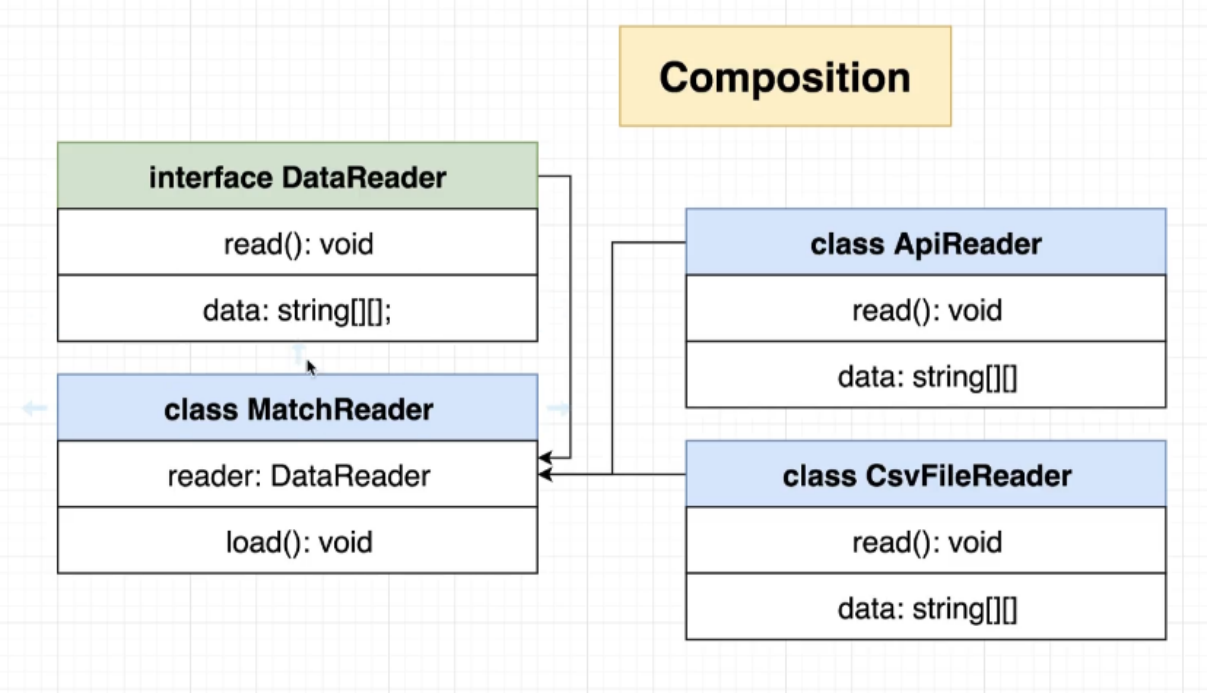
Parlons maintenant de la différence entre Heritage et Composition





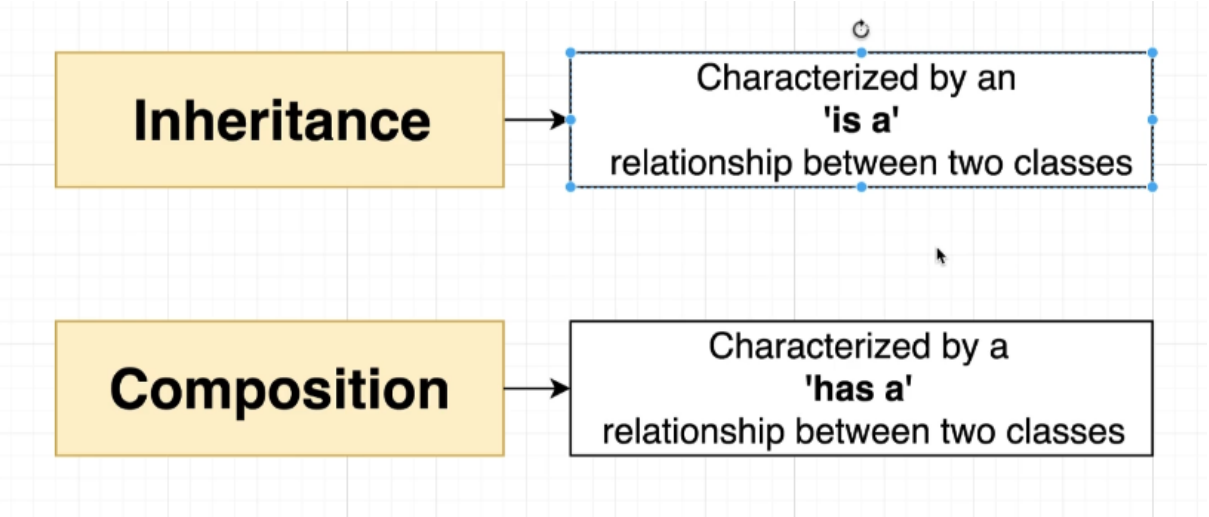
les classes **MatchReader** et **MovieReader** héritent de la classe abstraite **CsvFileReader**

L'heritage traduit une relation de type `is-a` (est-un)



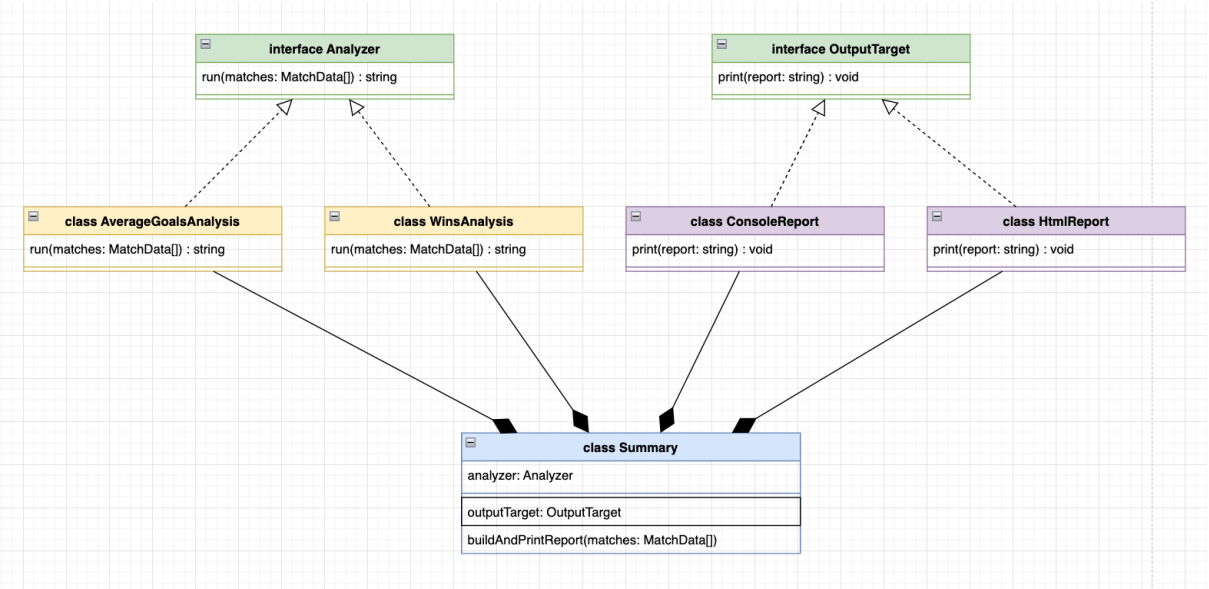
La composition traduit une relation de type `has-a` (possede-un)

ApiReader et CsvFIleReader a une relation avec la classe MatchReader



Pour bien comprendre la différence entre les 2, on dit que l'héritage `EST` une relation, la composition `A` une relation

On va mettre en place la structure suivante:



On va creer la classe `Summary` qui aura un `analyser` qui doit satisfaire l'interface `Analyser` et un objet `outputTarget` qui devra satisfaire l'interface `OutputTarget`

Toujours dans le dossier `composition` On va donc creer la classe `Summary`

On a dit que notre classe `Summary` doit comporter 2 objets qui satisfassent 2 interfaces

```
import { MatchData } from "./MatchData";

// Vous pouvez regrouper les interfaces dans un fichier dif
ferent
export interface Analyzer {
    run(matches: MatchData[]): string;
}


export interface OutputTarget {
    print(report: string): void;
}

export class Summary {
    constructor(
        public analyzer: Analyzer,
        public outputTarget: OutputTarget)
        {}
}
```

On va mettre en place les classes qui vont analyser les resultats des matches de foot

On va creer pour ca un dossier `analyzers` et creer dedans le fichier `WinsAnalysis.ts` et `AverageGoalsAnalysis`

Dans le fichier `WinsAnalysis.ts`

```
import { MatchData } from "../MatchData";
import { MatchResult } from "../MatchResult";
import { Analyzer } from "../Summary";


export class WinsAnalysis implements Analyzer {
```

```
    constructor(public team: string) {}


    run(matches: MatchData[]): string {
        let wins = 0;


        for (let match of matches) {
            if (match[1] === this.team && match[5] === Matc
hResult.HomeWin) {
                wins++;
            } else if (match[2] === this.team && match[5] =
== MatchResult.AwayWin) {
                wins++;
            }
        }


        return `Team ${this.team} won ${wins} games`;
    }
}
```

```
import { MatchData } from "../extends/MatchData";
import { Analyzer } from "./interfaces/Analyzer.interface";


export class GoalAverageAnalysis implements Analyzer {
    constructor(
        public team: string
    ) {}


    run(matches: MatchData[]): string {
        let result: number = 0;
        let totalMatches = 0;
        for (const match of matches) {
            if (match[1] === this.team) {
                result += match[3];
```

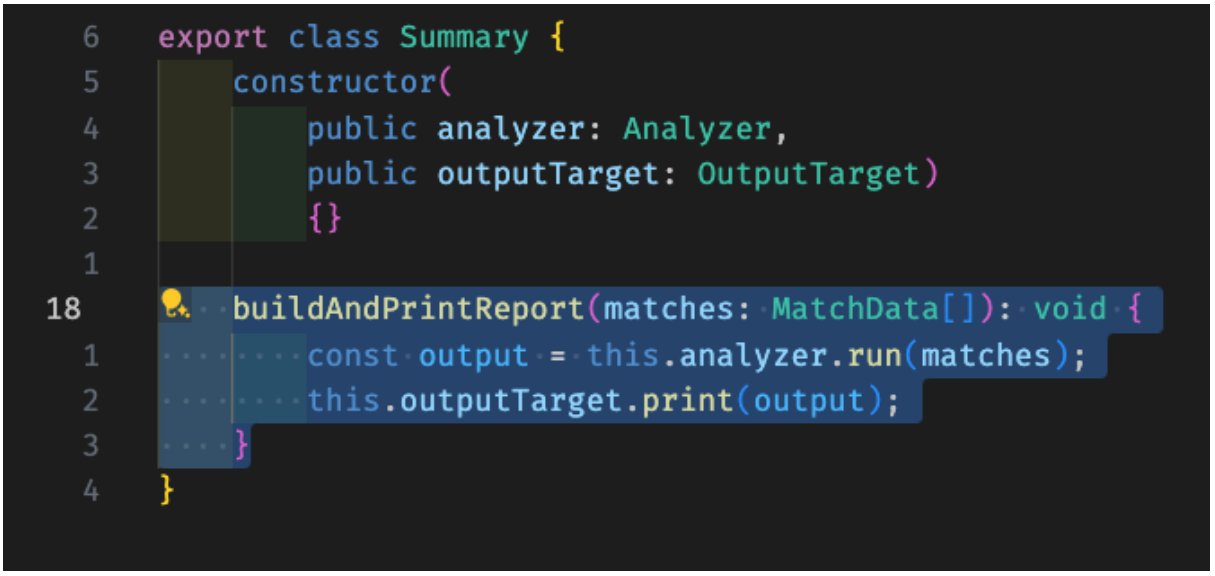
```
                totalMatches++;
            }
            if (match[2] === this.team) {
                result += match[4];
                totalMatches++;
            }
        }
        return `L'équipe: ${this.team} a marqué ${Math.roun
d(result/totalMatches)} buts par match en moyenne.`
    }
}
```

On va creer un dossier `reports` (toujours dans le dossier `composition`) et y mettre `ConsoleReport` et `HtmlRe`

Et on va y mettre:

```
export class ConsoleReport implements OutputTarget{
    print(report: string): void {
        console.log(report);
    }
}
```

On revient a notre classe `Summary`



```
export class Summary {
    constructor(
        public analyzer: Analyzer,
        public outputTarget: OutputTarget)
        {}

    buildAndPrintReport(matches: MatchData[]): void {
        const output = this.analyzer.run(matches);
        this.outputTarget.print(output);
    }
}
```

On revient donc a notre `index.ts`

On a besoin d'une analyse et d'un moyen de reporter les resultats

Ca devient:

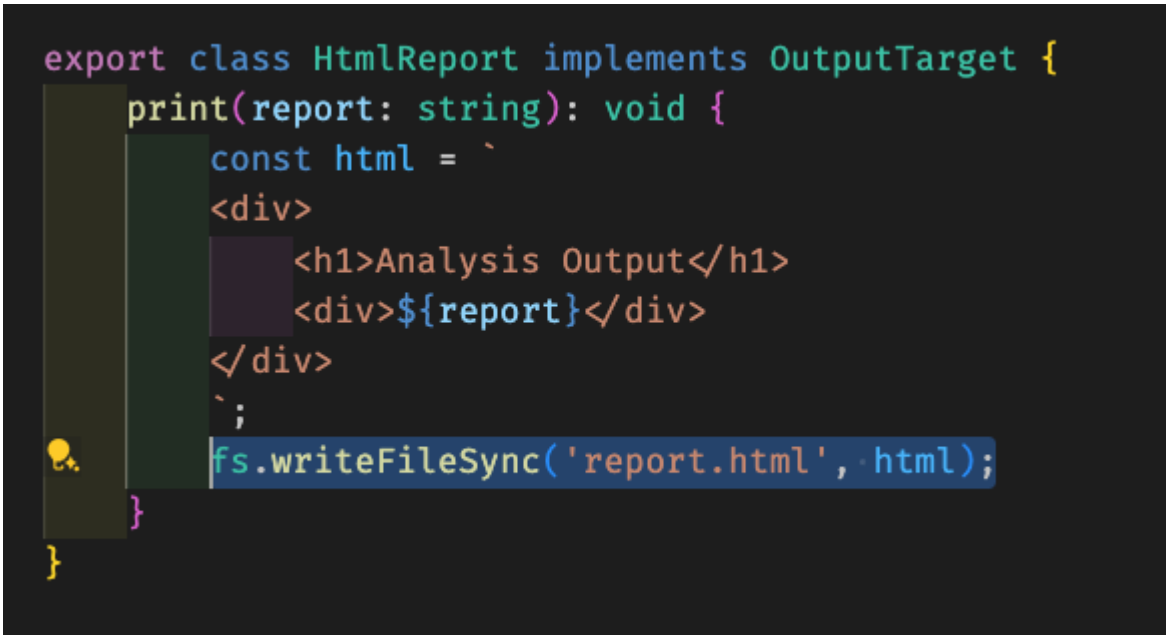
```
import { CsvFileReader } from './CsvFileReader';
import { MatchReader } from './MatchReader';
import { Summary } from './Summary';
import { WinsAnalysis } from './analyzers/WinsAnalysis';
import { ConsoleReport } from './reports/ConsoleReport';


const csvFileReader = new CsvFileReader('./src/football.cs
v');
const matchReader = new MatchReader(csvFileReader);
// on charge les donnees
matchReader.load();


const summary = new Summary(new WinsAnalysis('Man United'),
new ConsoleReport());
summary.buildAndPrintReport(matchReader.matches);
```

On va creer un second moyen de report : `HtmlReport.ts`

```
export class HtmlReport implements OutputTarget {
    print(report: string): void {
        const html = `
        <div>
            <h1>Analysis Output</h1>
            <div>${report}</div>
        </div>
        `;
    }
}
```

Du coup ca va generer du `html`

Ce qu'on veut c'est le coller dans un fichier HTML, pour cela on va utiliser `fs`



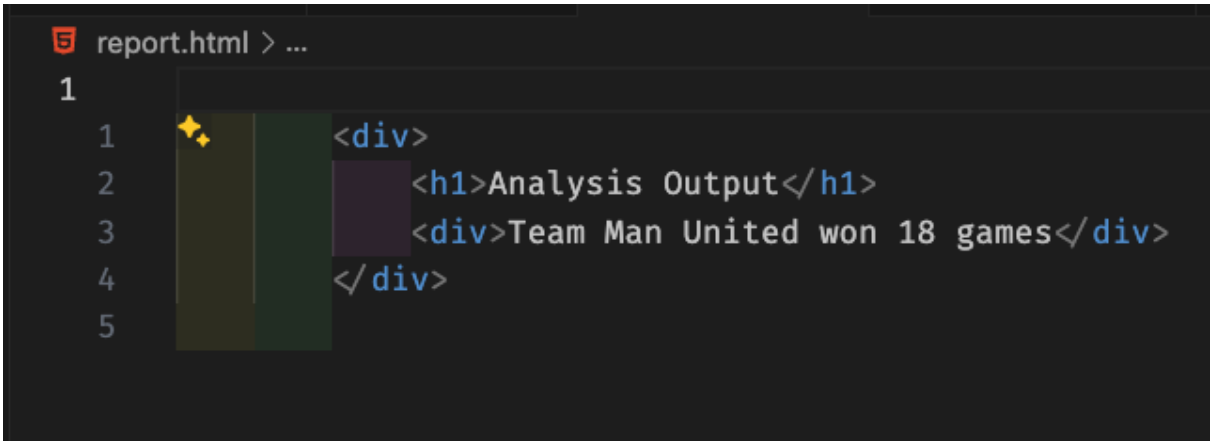
Le nom du fichier est en dur, on va utiliser le constructeur du coup

```ts
export class HtmlReport implements OutputTarget {

    constructor(public filename: string) {}

    print(report: string): void {
        const html = `
        <div>
            <h1>Analysis Output</h1>
            <div>${report}</div>
        </div>
        `;
        fs.writeFileSync(this.filename, html);
    }
}
```

On revient a `index.ts`

```ts
const summary = new Summary(
    new WinsAnalysis('Man United'),
    new HtmlReport("report.html")
);

summary.buildAndPrintReport(matchReader.matches);
```

Et des que j'enregistre les modifications, on voit un fichier `report.html` qui s'affiche a gauche dans l'explorateur de fichiers, contenant:



Une derniere chose pour que ce soit parfait

A chaque fois on a dut creer une nouvelle instance de `Summary` et lui passer les 2 instances : `WinsAnalysis` et `ConsoleReport`

C'est assez verbeux car a chaque fois qu'on va creer une instance de `Summary` on va devoir lui passer `WinsAnalysis` et `ConsoleReport` ou en tout cas 2 instances qui satisfassent les interfaces `Analyzer` et `OutputTarget`

Dans la classe `Summary` on a une methode `buildAntPrintReport`, on va ajouter une deuxieme methode static

```
    static winsAnalysisWithConsoleReport(team: string) {
        return new Summary(
            new WinsAnalysis(team),
            new ConsoleReport()
        )
    }
```

Donc dans notre `index.ts`

```
const csvFileReader = new CsvFileReader('./src/football.csv');
const matchReader = new MatchReader(csvFileReader);
matchReader.load();

const summary = Summary.winsAnalysisWithHtmlReport('Man United', 'report.html');
summary.buildAndPrintReport(matchReader.matches);
```

Je peux faire la meme chose avec la partie qui lit les donnees, en effet on peut se dire que la majoirté du temps, on chargera un fichier CSV

Dans `MatchReader.ts`

```
export class MatchReader {
    matches: MatchData[] = [];

    constructor(public reader: DataReader) {}

    static fromCsv(filename: string): MatchReader {
        return new MatchReader(new CsvFileReader(filename));
    }

    load(): void {
```

Je reviens a l'index

```
src > index.ts > ...
import { MatchReader } from './MatchReader';
import { Summary } from './Summary';

const matchReader = MatchReader.fromCsv('./src/football.csv');
matchReader.load();


const summary = Summary.winsAnalysisWithHtmlReport('Man United', 'report.html');
summary.buildAndPrintReport(matchReader.matches);
```