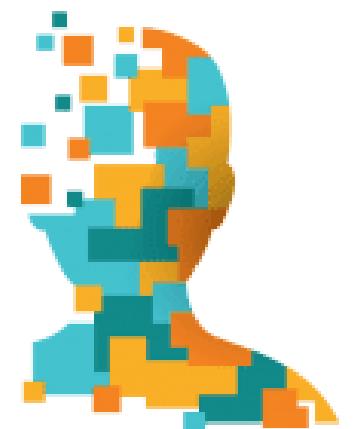




Groupe 2  
Dev WEB/Mobile  
Promo 6



# TSConfig

**Le fichier tsconfig.json représente quoi dans un projet typescript ?**

C'est un fichier dans lequel, on spécifie les options de compilations, de **gérer la structure du projet** et de **personnaliser le comportement du compilateur**

TypeScript en fonction des besoins spécifiques du projet.  
Explorons quelques-unes des clés importantes du fichier tsconfig.json et expliquons leur rôle :

# 1.include

**Fonctionnement:** La propriété **include** permet de spécifier un ensemble de fichiers ou de répertoires à inclure dans la compilation TypeScript. Cela permet de cibler les fichiers sources spécifiques que vous souhaitez compiler, en excluant les fichiers inutiles ou de test.

```
{  
  "compilerOptions": {  
    "target": "es2016",  
    "module": "commonjs",  
    "esModuleInterop": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "skipLibCheck": true  
  },  
  "include": [  
    "src/**/*.ts"  
  ]  
}
```

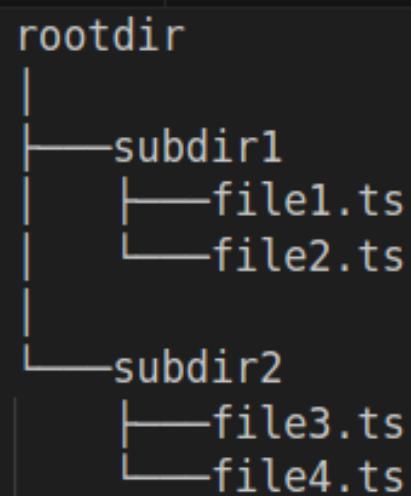
## 2.file

**Fonctionnement:** Similaire à include, la propriété **file** permet de spécifier des fichiers sources individuels à inclure dans la compilation. Elle est utile pour inclure des fichiers spécifiques qui ne correspondent pas aux modèles d'inclusion globaux.

```
[  
  "compilerOptions": {  
    "target": "es2016",  
    "module": "commonjs",  
    "esModuleInterop": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "skipLibCheck": true  
  },  
  "files": [  
    "src/main.ts",  
    "src/utils.ts"  
  ]  
}
```

## 3.rootDir

**Fonctionnement:** La propriété **rootDir** définit **le répertoire racine du projet TypeScript**. Elle est importante pour la résolution des chemins de fichiers et pour garantir que le compilateur utilise les chemins d'accès corrects lors de la compilation.



"rootDir": "./rootdir" spécifie que le répertoire racine pour la résolution des chemins est rootdir.  
"include": ["./rootdir/\*\*/\*.ts"] spécifie que tous les fichiers TypeScript (\*.ts) dans rootdir et ses sous-répertoires doivent être inclus dans la compilation.

```
"compilerOptions": {
  "rootDir": "./rootdir"
},
"include": [
  "./rootdir/**/*.ts"
]
```

# 4.removeComments

- Fonctionnement:** La propriété **removeComments** permet de supprimer les commentaires du code TypeScript pendant la compilation. Cela peut être utile pour réduire la taille du code généré ou pour simplifier la lecture du code compilé.

```
js driver.js
modules > JS driver.js > ...
1 "use strict";
2 Object.defineProperty(exports, "__esModule", {
3 const customer_1 = require("./customer");
4 let customer = new customer_1.Customer("Moon", "Sky", 44);
5 console.log(customer.firstname);
6 console.log(customer.lastname);
7 console.log(customer.age);
8 //Ajouter + 5 à l'age de l'utilisateur
9 function agePlus5(age) {
10   return age + 5;
11 }
12 const endAge = agePlus5(customer.age);
13 //Afficher l'age de l'utilisateur
14 console.log(endAge);
15

ts driver.ts
modules > TS driver.ts > ...
1 import { Customer } from "./customer";
2
3 let customer = new Customer("Moon", "Sky", 44);
4 console.log(customer.firstname);
5 console.log(customer.lastname);
6 console.log(customer.age);
7
8 //Ajouter + 5 à l'age de l'utilisateur
9 function agePlus5(age: number): number{
10   return age + 5;
11 }
12
13 const endAge = agePlus5(customer.age);
14 //Afficher l'age de l'utilisateur
15 console.log(endAge);

tsconfig.json
modules > TS tsconfig.json > ...
2 "compilerOptions": {
3   // "declaration": true,
4   // "declarationMap": true,
5   // "emitDeclarationOnly": true,
6   // "sourceMap": true,
7   // "outFile": "./",
8   // "outDir": "./",
9   "removeComments": false,
10  // "noEmit": true,
11  // "importHelpers": true,
12  // "importsNotUsedAsValues": "remove",
13  // "downlevelIteration": true,
14  // "sourceRoot": "",
15  // "mapRoot": "",
16  // "inlineSourceMap": true,
17  // "inlineSources": true,
```

```
js driver.js
modules > JS driver.js > ...
1 "use strict";
2 Object.defineProperty(exports, "__esModule", {
3 const customer_1 = require("./customer");
4 let customer = new customer_1.Customer("Moon", "Sky", 44);
5 console.log(customer.firstname);
6 console.log(customer.lastname);
7 console.log(customer.age);
8 function agePlus5(age) {
9   return age + 5;
10 }
11 const endAge = agePlus5(customer.age);
12 console.log(endAge);
13

ts driver.ts
modules > TS driver.ts > ...
1 import { Customer } from "./customer";
2
3 let customer = new Customer("Moon", "Sky", 44);
4 console.log(customer.firstname);
5 console.log(customer.lastname);
6 console.log(customer.age);
7
8 //Ajouter + 5 à l'age de l'utilisateur
9 function agePlus5(age: number): number{
10   return age + 5;
11 }
12
13 const endAge = agePlus5(customer.age);
14 //Afficher l'age de l'utilisateur
15 console.log(endAge);

tsconfig.json
modules > TS tsconfig.json > {} compilerOptions
2 "compilerOptions": {
3   // "declaration": true,
4   // "declarationMap": true,
5   // "emitDeclarationOnly": true,
6   // "sourceMap": true,
7   // "outFile": "./",
8   // "outDir": "./",
9   "removeComments": true,
10  // "noEmit": true,
11  // "importHelpers": true,
12  // "importsNotUsedAsValues": "remove",
13  // "downlevelIteration": true,
14  // "sourceRoot": "",
15  // "mapRoot": "",
16  // "inlineSourceMap": true,
17  // "inlineSources": true,
```

# 5.noEmitOnError

**Fonctionnement:** La propriété **noEmitOnError** contrôle le comportement du compilateur TypeScript lorsqu'il rencontre des erreurs. Si elle est définie sur **true**, le compilateur arrêtera la compilation et affichera les erreurs détectées, empêchant la génération de code JavaScript. Si elle est définie sur **false**, le compilateur continuera la compilation malgré les erreurs, générant un code JavaScript potentiellement incorrect.

```
EXPLORER ... JS driver.js ...
TYPESCRIPT ...
modules > JS driver.js > ...
1 "use strict";
2 Object.defineProperty(exports, "__esModule", { value: true });
3 const customer_1 = require("./customer");
4 let customer = new customer_1.Customer("Moon", "Sky");
5 console.log(customer.firstname);
6 console.log(customer.lastname);
7 console.log(customer.age);
8 function agePlus5(age) {
9     return age + 5;
10 }
11 const endAge = agePlus5(customer.age);
12 console.log(endAge);
13

TS driver.ts 1 ...
modules > TS driver.ts > ...
1
2 let customer = new Customer("Moon", "Sky");
3 console.log(customer.firstname);
4 console.log(customer.lastname);
5 console.log(customer.age);
6
7 //Ajouter + 5 à l'age de l'utilisateur
8 function agePlus5(age: number): number{
9     return age + 5;
10 }
11
12 const endAge = agePlus5(customer.age);
13 //Afficher l'age de l'utilisateur
14 console.log(endAge);
15

tsconfig.json ...
modules > TS tsconfig.json > {} compilerOptions > noEmitOnError
2 "compilerOptions": {
37     "noEmitOnError": false,
38     // "noResolve": true,
39     /* JavaScript Support */
40     // "allowJs": true,
41     // "checkJs": true,
42     // "maxNodeModuleJsDepth": 1,
43     /* Emit */
44     // "noEmitOnDeclaration": true,
45     // "declaration": true,
46     // "declarationMap": true,
47     // "emitDeclarationOnly": true,
48     // "sourceMap": true,
49     // "outFile": "./",
50 }
```

```
EXPLORER ... TS driver.ts 1 TS customer.ts ...
TYPESCRIPT ...
modules > TS customer.ts > Customer
1 export class Customer{
2     constructor(private _firstname: string, private _lastname: string, private _age: number) {
3         this._firstname = _firstname;
4         this._lastname = _lastname;
5         this._age = _age;
6     }
7     public get firstname(): string{
8         return this._firstname;
9     }
10    public get lastname(): string{
11        return this._lastname;
12    }
13    public get age(): number{
14        return this._age;
15    }
16 }

TS driver.ts 1 ...
modules > TS driver.ts > ...
1
2 let customer = new Customer("Moon", "Sky");
3 console.log(customer.firstname);
4 console.log(customer.lastname);
5 console.log(customer.age);
6
7 //Ajouter + 5 à l'age de l'utilisateur
8 function agePlus5(age: number): number{
9     return age + 5;
10 }
11
12 const endAge = agePlus5(customer.age);
13 //Afficher l'age de l'utilisateur
14 console.log(endAge);
15

tsconfig.json ...
modules > TS tsconfig.json > {} compilerOptions > noEmitOnError
2 "compilerOptions": {
37     "noEmitOnError": true,
38     // "noResolve": true,
39     /* JavaScript Support */
40     // "allowJs": true,
41     // "checkJs": true,
42     // "maxNodeModuleJsDepth": 1,
43     /* Emit */
44     // "noEmitOnDeclaration": true,
45     // "declaration": true,
46     // "declarationMap": true,
47     // "emitDeclarationOnly": true,
48     // "sourceMap": true,
49     // "outFile": "./",
50     // "outDir": "./",
51     // "removeComments": true,
```

- Partie 2

## Exclure les fichiers

```
05      /* Completeness */
06      // "skipDefaultLibCheck": true,
07      "skipLibCheck": true
08 },
09 "exclude": [
10   "fichierA.ts",
11   "fichierB.ts"
12 ]
13
14
```

## ● Partie 2

les fichiers node\_modules  
package et dépendances

✓ node\_modules \ babylonjs

JS babylon.js

JS babylon.max.js

JS babylon.max.js.map

TS babylon.module.d.ts

JS Oimo.js

{ } package.json

ⓘ readme.md

## • Partie 2

exclusion fichier .dev.ts

```
},
  "exclude": [
    "node_modules", // Exclut le répertoire node_modules
    "test/*.ts",    // Exclut tous les fichiers .ts dans le répertoire test
    "**/*.*.dev.ts" // Exclut tous les fichiers .dev.ts dans tous les répertoires
  ]
}
```

## • Partie 2

### Spread operator

```
11
12 // expansion d'element d'un tableau
13 const tableau1 = [1, 2, 3];
14 const tableau2 = [...tableau1, 4, 5, 6];
15 console.log(tableau2);
16
```

- Partie 2

## Spread operator

```
17
18 | 
19
20 // Expansion d'éléments dans un objet :
21 const objet1 = { a: 1, b: 2, c: 3};
22 const objet2 = { ...objet1, d: 4, e: 5 };
23 console.log(objet2);
```

- Partie 3

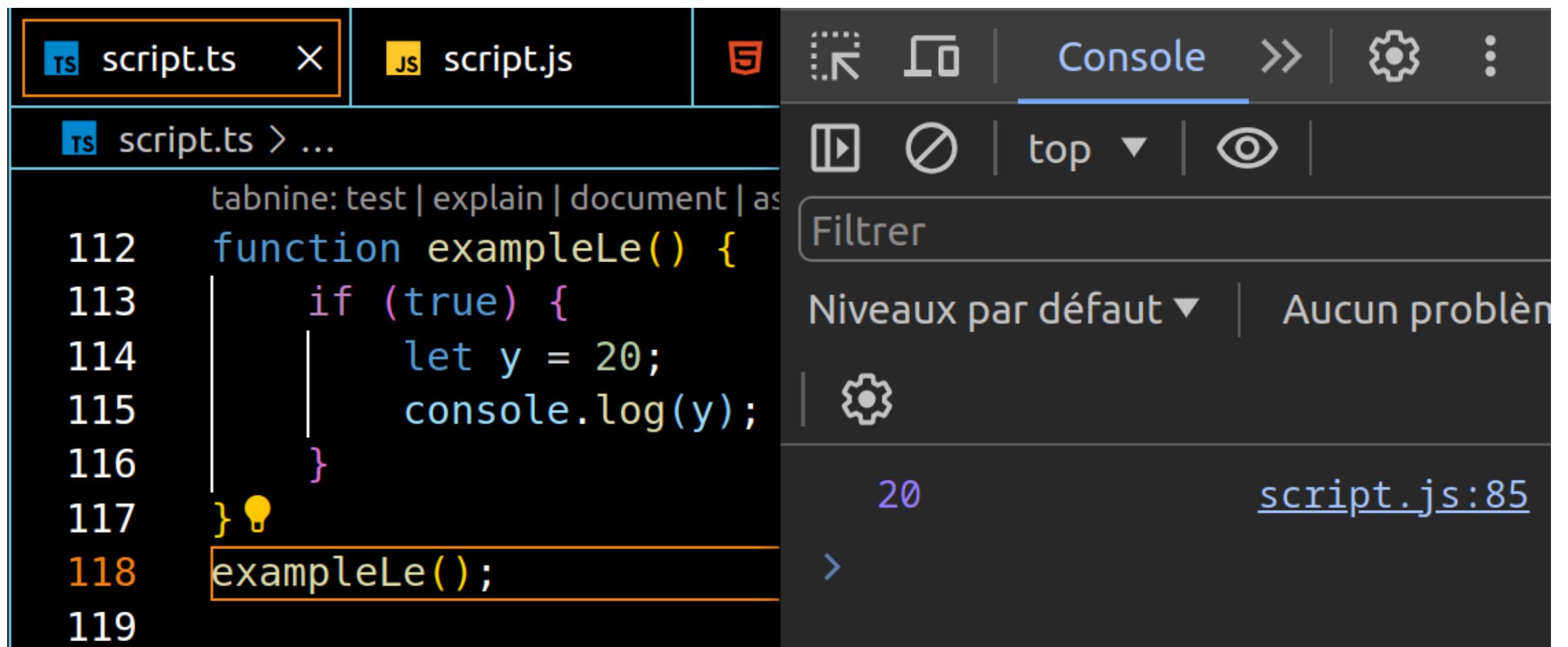
## **CONST – VAR – LET**

En typeScript comme en javascript, 'Var', 'Let' et 'Const' sont des mots clés utilisés pour déclarer des variables mais ils ont des comportements et des portés différents

## • Partie 3

### • LET

permet de déclarer une variable et sa portée se limite dans le bloc de code où il a été déclaré.



The screenshot shows a browser's developer tools with the "Console" tab selected. There are two tabs at the top: "script.ts" (selected) and "script.js". The "script.ts" tab contains the following TypeScript code:

```
function exampleLe() {
  if (true) {
    let y = 20;
    console.log(y);
  }
}
exampleLe();
```

The "script.js" tab shows the generated JavaScript code:

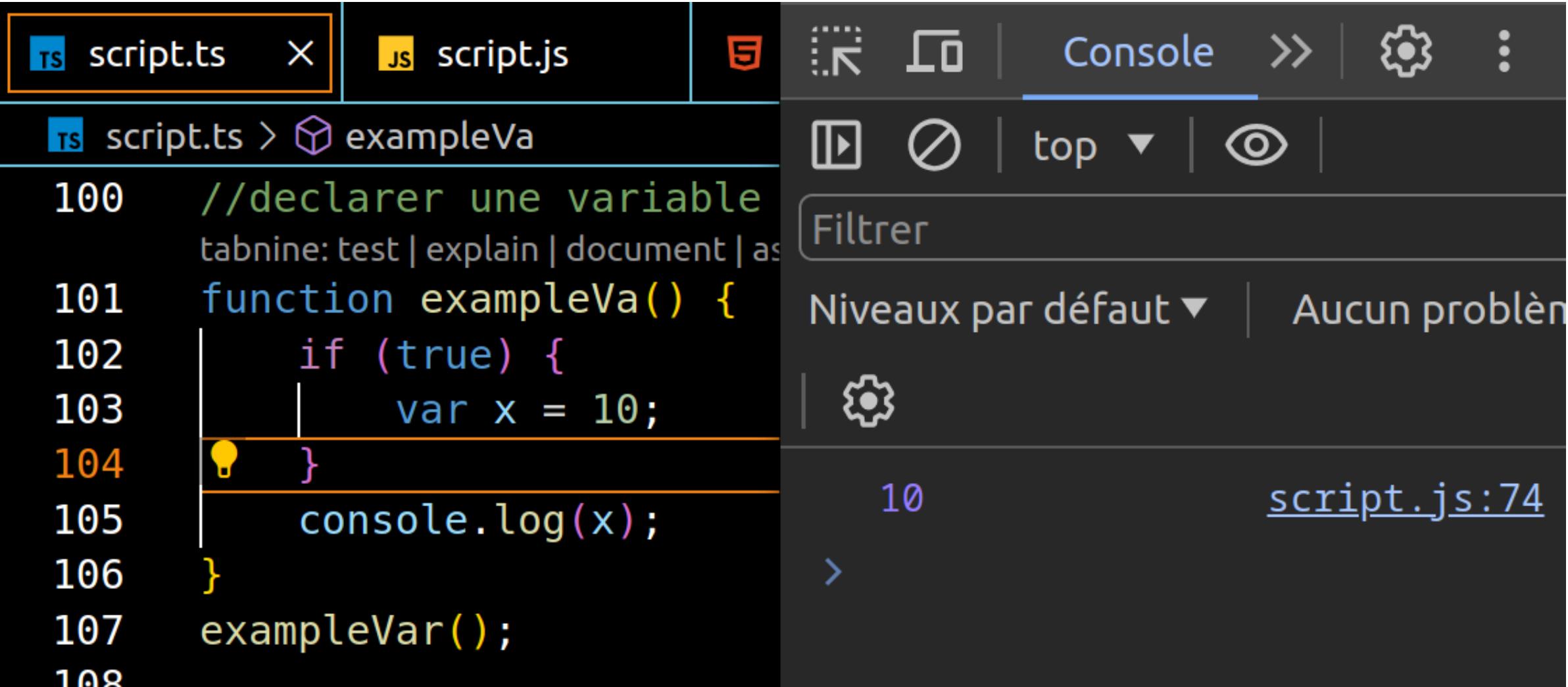
```
function exampleLe() {
  if (true) {
    let y = 20;
    console.log(y);
  }
}
exampleLe();
```

The console output shows the value 20, indicating that the variable y is only accessible within the scope of the if block. The status bar at the bottom right shows "script.js:85".

## • Partie 3

- VAR

permet de déclarer  
une variable et  
d'initialiser sa valeur



```
script.ts      script.js
script.ts > exampleVa
100 //déclarer une variable
101 function exampleVa() {
102     if (true) {
103         var x = 10;
104     }
105     console.log(x);
106 }
107 exampleVar();
108
```

The screenshot shows a code editor interface with two tabs: "script.ts" and "script.js". The "script.ts" tab is active, displaying the following TypeScript code:

```
//déclarer une variable
function exampleVa() {
    if (true) {
        var x = 10;
    }
    console.log(x);
}
exampleVar();
```

The "script.js" tab shows the generated JavaScript code:

```
function exampleVa() {
    if (true) {
        var x = 10;
    }
    console.log(x);
}
```

To the right of the code editor is a "Console" tab, which displays the output of the console.log statement: "10". The status bar at the bottom right indicates "script.js:74".

## • Partie 3

### • CONST

permet de déclarer une constante dont la valeur

ne peut être ni réaffectée ni modifiée.

C'est à dire une constante ne peut pas être déclarée à nouveau.

The screenshot shows a code editor interface with two tabs: 'script.ts' and 'script.js'. The 'script.ts' tab is active, displaying the following TypeScript code:

```
124 function exampleCons() {  
125     if (true) {  
126         const z = 30;  
127         console.log(z);  
128     }  
129 }  
130 exampleCons();  
131
```

The 'script.js' tab shows the transpiled JavaScript code:

```
function exampleCons() {  
    if (true) {  
        const z = 30;  
        console.log(z);  
    }  
}  
exampleCons();
```

To the right of the code editor is a browser developer tools console window titled 'Console'. It displays the output of the console.log statement: '30'. Below the console, there is a message 'Aucun problème' (No problems).

- Partie 3

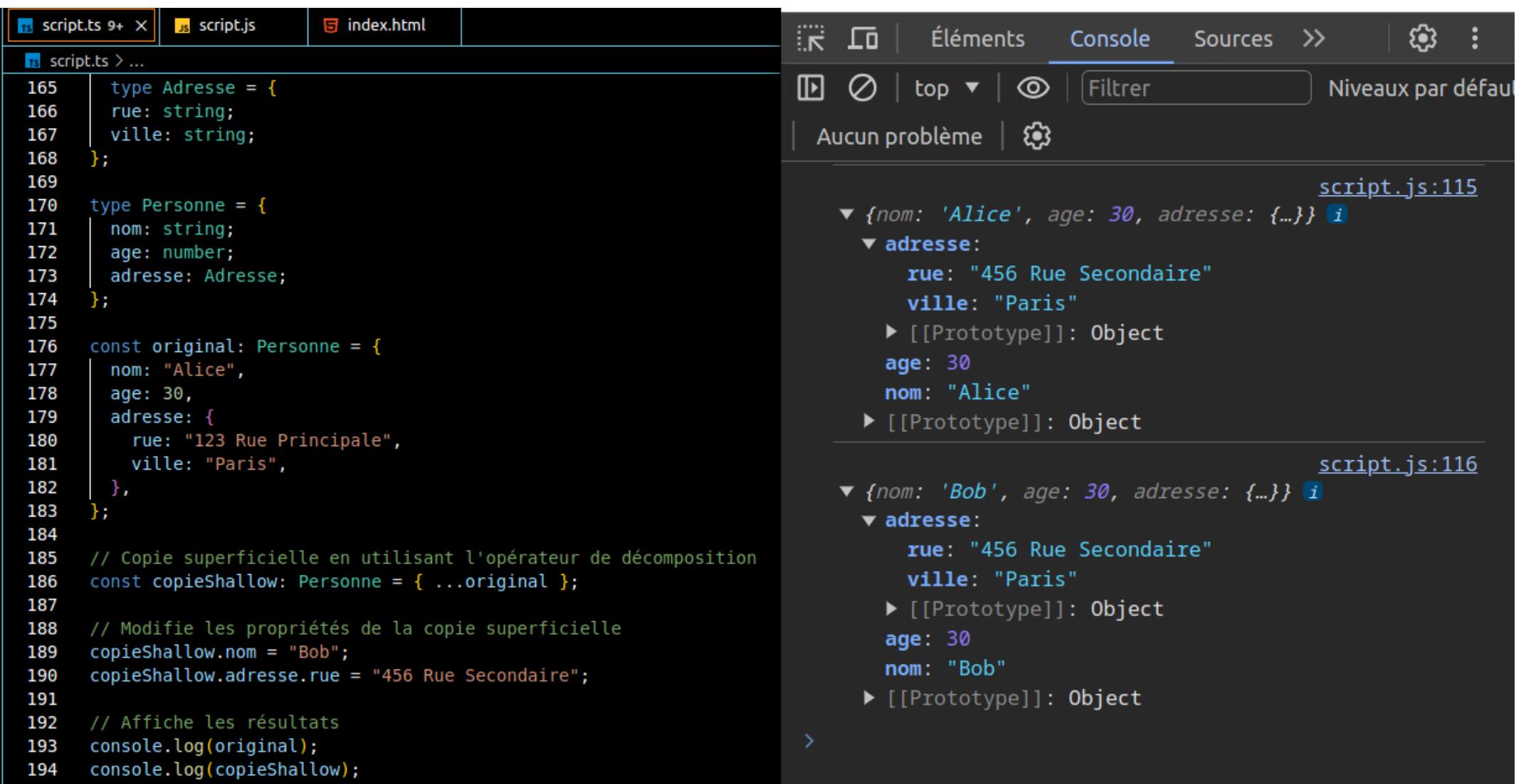
## **COPIER UN OBJET DE TYPE REFERENCE**

- COPIE SUPERFICIELLE
- COPIE PROFONDE

## ● Partie 3

### • COPIE SUPERFICIELLE

une copie superficielle d'un objet est une nouvelle instance de l'objet, mais copie uniquement les propriétés de premier niveau.



The screenshot shows a browser's developer tools with the 'Console' tab selected. On the left, there is a code editor window for 'script.ts' containing TypeScript code that defines two types: 'Adresse' and 'Personne', and creates two objects, 'original' and 'copieShallow', demonstrating shallow copy behavior. On the right, the 'Console' tab displays the state of these variables as JSON objects. It shows that while 'nom' and 'age' are copied correctly, the 'adresse' property is a reference to the same memory location for both objects, as indicated by the identical addresses and modifications made to one reflecting in the other.

```
script.ts:115
{
  nom: 'Alice',
  age: 30,
  adresse: {
    rue: "456 Rue Secondaire",
    ville: "Paris"
  }
}

script.ts:116
{
  nom: 'Bob',
  age: 30,
  adresse: {
    rue: "456 Rue Secondaire",
    ville: "Paris"
  }
}
```

```
script.ts
165 type Adresse = {
166   rue: string;
167   ville: string;
168 };
169
170 type Personne = {
171   nom: string;
172   age: number;
173   adresse: Adresse;
174 };
175
176 const original: Personne = {
177   nom: "Alice",
178   age: 30,
179   adresse: {
180     rue: "123 Rue Principale",
181     ville: "Paris",
182   },
183 };
184
185 // Copie superficielle en utilisant l'opérateur de décomposition
186 const copieShallow: Personne = { ...original };
187
188 // Modifie les propriétés de la copie superficielle
189 copieShallow.nom = "Bob";
190 copieShallow.adresse.rue = "456 Rue Secondaire";
191
192 // Affiche les résultats
193 console.log(original);
194 console.log(copieShallow);
```

## ● Partie 3

### ● COPIE PROFONDE

Une copie profonde crée une nouvelle instance de l'objet ainsi que de tous les objets imbriqués en n'importe quel niveau.

The screenshot shows a browser's developer tools console tab. On the left, the code in script.ts is displayed:

```
132 type Adresse = {
133   rue: string;
134   ville: string;
135 };
136
137 type Personne = {
138   nom: string;
139   age: number;
140   adresse: Adresse;
141 };
142
143 const original: Personne = {
144   nom: "Alice",
145   age: 30,
146   adresse: {
147     rue: "123 Rue Principale",
148     ville: "Paris",
149   },
150 };
151 // Copie profonde en utilisant JSON.parse et JSON.stringify
152 const copieDeep: Personne = JSON.parse(JSON.stringify(original));
153
154 // Modifie les propriétés de la copie profonde
155 copieDeep.nom = "Bob";
156 copieDeep.adresse.rue = "456 Rue Secondaire";
157
158 // Affiche les résultats
159 console.log(original);
160 console.log(copieDeep);
```

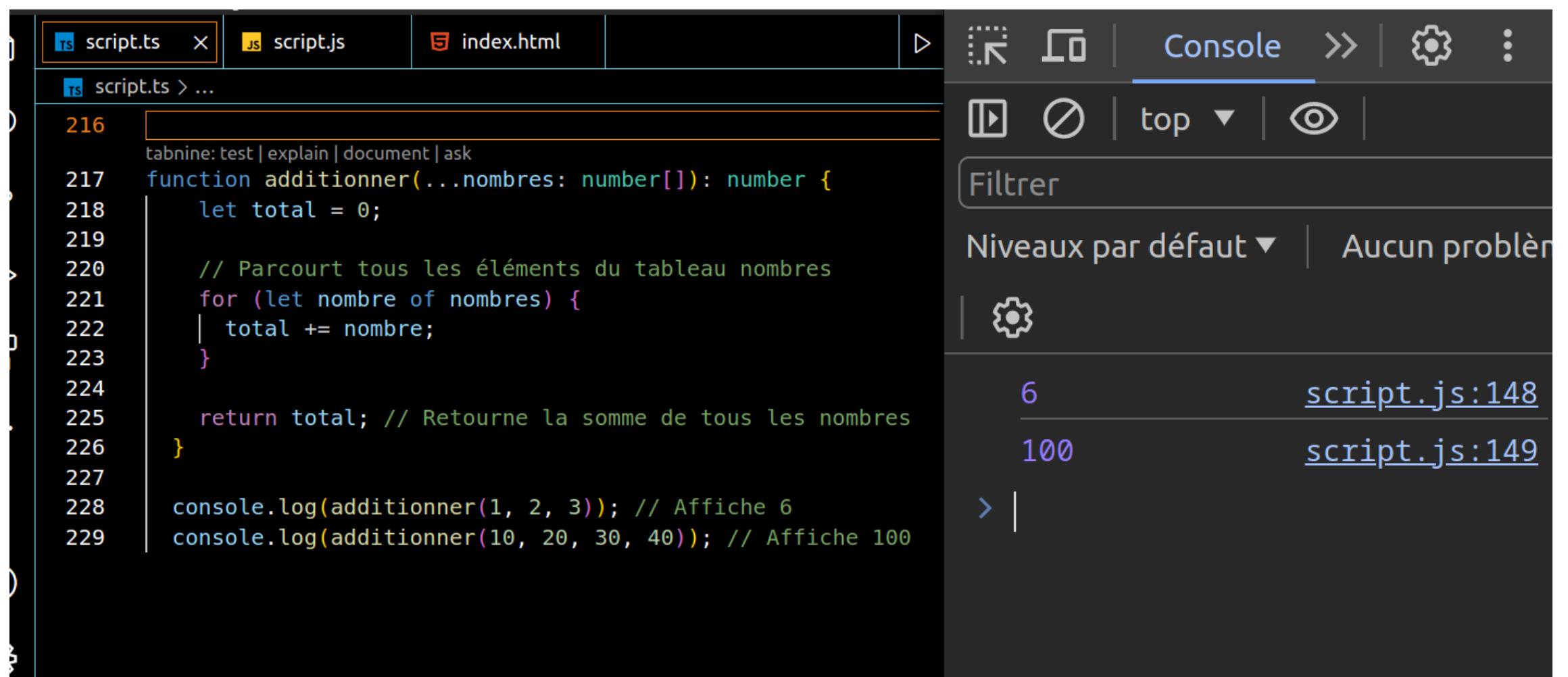
On the right, the console output shows two objects: 'original' and 'copieDeep'. Both objects have the same properties: 'nom' (Alice/Bob), 'age' (30), and an 'adresse' object. The 'adresse' object has 'rue' ('123 Rue Principale'/'456 Rue Secondaire') and 'ville' ('Paris'). The 'copieDeep' object is shown with its own prototype chain, indicating it is a deep copy.

```
script.js:106
▼ {nom: 'Alice', age: 30, adresse: {...}} ⓘ
  ▼ adresse:
    rue: "123 Rue Principale"
    ville: "Paris"
    ► [[Prototype]]: Object
    age: 30
    nom: "Alice"
    ► [[Prototype]]: Object
script.js:107
▼ {nom: 'Bob', age: 30, adresse: {...}} ⓘ
  ▼ adresse:
    rue: "456 Rue Secondaire"
    ville: "Paris"
    ► [[Prototype]]: Object
    age: 30
    nom: "Bob"
    ► [[Prototype]]: Object
```

## • Partie 3

### NBRE PARAMS VARIABLE

un nombre variable de paramètre signifie qu'une fonction peut accepter un nombre quelconque d'arguments, sans que ce nombre soit spécifié à l'avance



The screenshot shows a code editor interface with three tabs at the top: 'script.ts', 'script.js', and 'index.html'. The 'script.ts' tab is active, displaying the following TypeScript code:

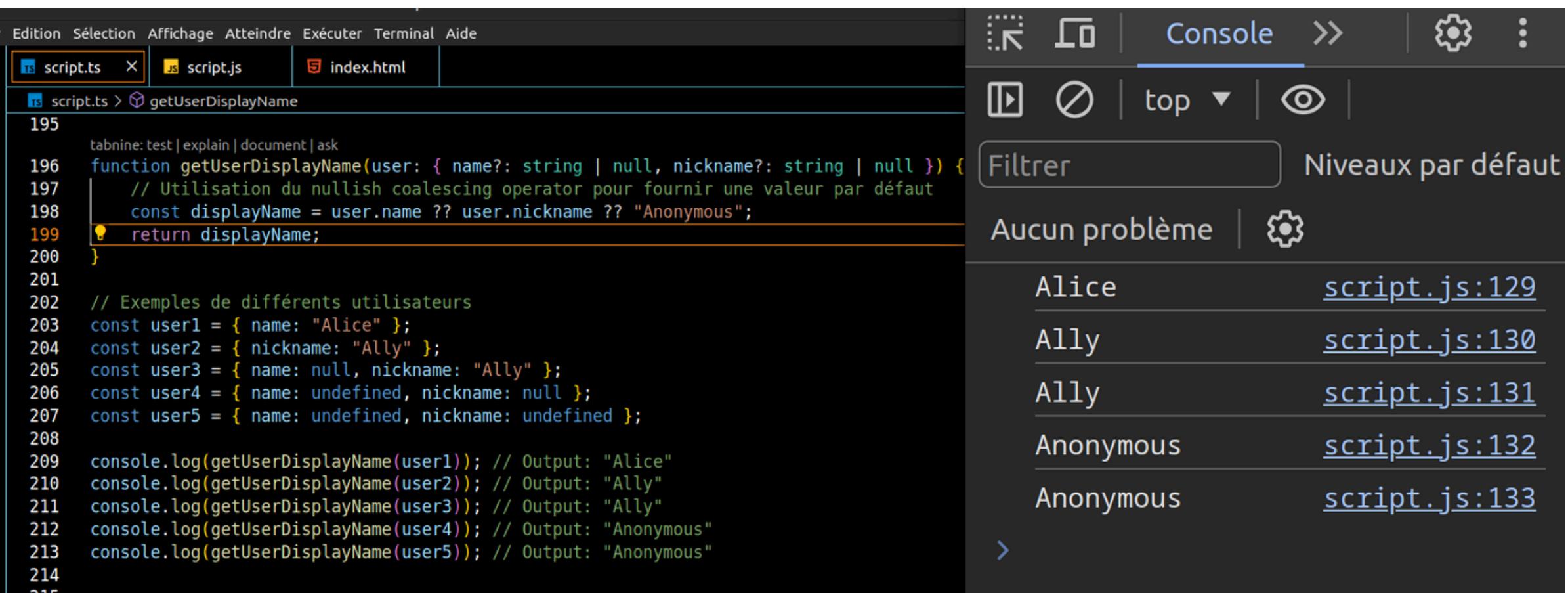
```
216 tabnine: test | explain | document | ask
217 function additionner(...nombres: number[]): number {
218     let total = 0;
219
220     // Parcourt tous les éléments du tableau nombres
221     for (let nombre of nombres) {
222         total += nombre;
223     }
224
225     return total; // Retourne la somme de tous les nombres
226 }
227
228 console.log(additionner(1, 2, 3)); // Affiche 6
229 console.log(additionner(10, 20, 30, 40)); // Affiche 100
```

The code defines a function named 'additionner' that takes a spread operator (...nombres) of numbers as its parameter. It initializes a variable 'total' to 0 and then iterates over each number in the array, adding it to 'total'. Finally, it returns the total sum. Two calls to the function are made using the 'console.log' statement, resulting in outputs of 6 and 100 respectively.

## • Partie 3

### • NULLISH COALESING OPERATOR

L'opérateur de coalescence des nulls(ou nullish coalescing operator) est un opérateur logique introduit dans javascript pour gerer les valeurs nulles ou indéfinies.



The screenshot shows a code editor interface with a dark theme. On the left, there are tabs for 'script.ts', 'script.js', and 'index.html'. The 'script.ts' tab is active, displaying the following TypeScript code:

```
tabnine: test | explain | document | ask
196 function getUserDisplayName(user: { name?: string | null, nickname?: string | null }) {
197     // Utilisation du nullish coalescing operator pour fournir une valeur par défaut
198     const displayName = user.name ?? user.nickname ?? "Anonymous";
199     return displayName;
200 }

// Exemples de différents utilisateurs
203 const user1 = { name: "Alice" };
204 const user2 = { nickname: "Ally" };
205 const user3 = { name: null, nickname: "Ally" };
206 const user4 = { name: undefined, nickname: null };
207 const user5 = { name: undefined, nickname: undefined };

209 console.log(getUserDisplayName(user1)); // Output: "Alice"
210 console.log(getUserDisplayName(user2)); // Output: "Ally"
211 console.log(getUserDisplayName(user3)); // Output: "Ally"
212 console.log(getUserDisplayName(user4)); // Output: "Anonymous"
213 console.log(getUserDisplayName(user5)); // Output: "Anonymous"
```

The 'Console' tab is selected at the top right. It displays the following output:

Output	Line Number
Alice	script.js:129
Ally	script.js:130
Ally	script.js:131
Anonymous	script.js:132
Anonymous	script.js:133

En TypeScript, il existe plusieurs types primitifs qui sont utilisés pour représenter des valeurs simples. Voici les types primitifs disponibles en TypeScript :

**boolean** : représente une valeur booléenne, soit true soit false.

```
let isDone: boolean = false;
```

**number** : représente un nombre, entier ou décimal.  
TypeScript ne distingue pas entre les entiers et les nombres à virgule flottante.

```
let count: number = 42;  
let price: number = 5.99;
```

**string** : représente une chaîne de caractères.  
Les chaînes de caractères peuvent être délimitées par des guillemets simples ('') ou doubles ("").

```
let name: string = 'John Doe';  
let message: string = "Hello, world!";
```

**null** et **undefined** : représentent l'absence de valeur. null est une valeur intentionnellement absente, tandis que undefined est une valeur non initialisée ou inexistante.

```
let noValue: null = null;  
let undefinedValue: undefined = undefined;
```

**symbol (ES6) :** représente un identifiant unique et immuable. Les symboles sont créés à l'aide de la fonction `Symbol()`.

```
let uniqueId: symbol = Symbol('uniqueId');
```

**bigint (ES2020) :** représente des entiers de grande taille, qui dépassent la capacité de représentation des nombres en JavaScript. Les bigints sont créés en ajoutant le suffixe `n` à un entier.

```
let largeNumber: bigint = 9007199254740991n;
```

## Reference en typescript

En TypeScript, les références sont utilisées pour manipuler des objets ou des fonctions en se référant à leur emplacement dans la mémoire

Voici quelques exemples d'utilisation des références en TypeScript :

**Objets : Lorsque vous affectez un objet à une variable, la variable contient une référence à l'objet**

```
let obj1 = { prop: 1 };
let obj2 = obj1;

obj2.prop = 2;
console.log(obj1.prop); // Affiche 2, car obj1 et obj2 font référence au même objet
```

**Tableaux : Les tableaux fonctionnent de la même manière que les objets. Lorsque vous affectez un tableau à une variable, la variable contient une référence à ce tableau.**

```
let arr1 = [1, 2, 3];
let arr2 = arr1;

arr2.push(4);
console.log(arr1); // Affiche [1, 2, 3, 4], car arr1 et arr2 font référence au même tableau
```

**Fonctions : Les fonctions peuvent également être assignées à des variables, créant ainsi des références aux fonctions.**

```
function sayHello() {  
    console.log("Hello");  
}  
  
let funcRef = sayHello;  
funcRef(); // Affiche "Hello", car funcRef fait référence à la fonction sayHello
```

En TypeScript, les "index properties" (propriétés d'index) permettent de définir des propriétés dynamiques pour un objet en utilisant des indexeurs. Les indexeurs sont similaires aux tableaux, où vous pouvez accéder aux éléments en utilisant des indices numériques ou des clés de chaîne de caractères.

**Voici un exemple d'utilisation des propriétés d'index :**

**Dans cet exemple, l'interface NumberKeyObject accepte des clés numériques et des valeurs de chaîne de caractères.**

```
interface StringKeyObject {  
    [key: string]: string;  
}  
  
let obj: StringKeyObject = {  
    prop1: "value1",  
    prop2: "value2"  
};  
  
obj.prop3 = "value3";  
obj["prop4"] = "value4";  
  
console.log(obj.prop1); // Affiche "value1"  
console.log(obj["prop2"]); // Affiche "value2"  
console.log(obj.prop3); // Affiche "value3"  
console.log(obj["prop4"]); // Affiche "value4"
```

Dans cet exemple, nous avons défini une interface `StringKeyObject` avec une propriété d'index `[key: string]: string`. Cela signifie que l'objet peut avoir n'importe quel nombre de propriétés, à condition que leurs clés soient des chaînes de caractères et que leurs valeurs soient également des chaînes de caractères.

```
interface NumberKeyObject {
  [key: number]: string;
}

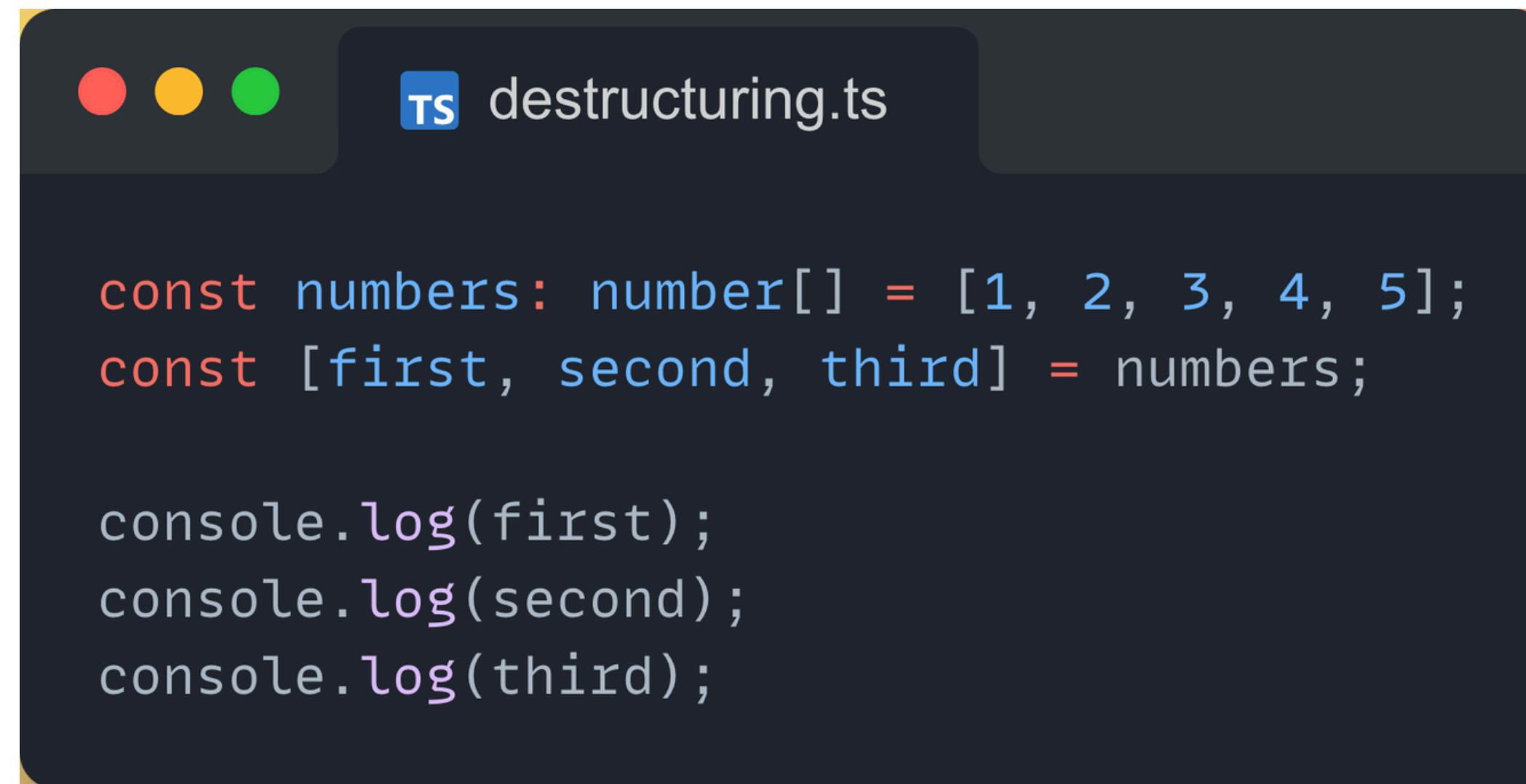
let obj: NumberKeyObject = {
  0: "value0",
  1: "value1"
};

obj[2] = "value2";
obj["3"] = "value3"; // Bien que cela fonctionne, il est préférable d'utiliser des clés numériques sans guillemets

console.log(obj[0]); // Affiche "value0"
console.log(obj[1]); // Affiche "value1"
console.log(obj[2]); // Affiche "value2"
console.log(obj[3]); // Affiche "value3"
```

## • Destructuration avec les tableaux

### Déstructuration de base des tableaux



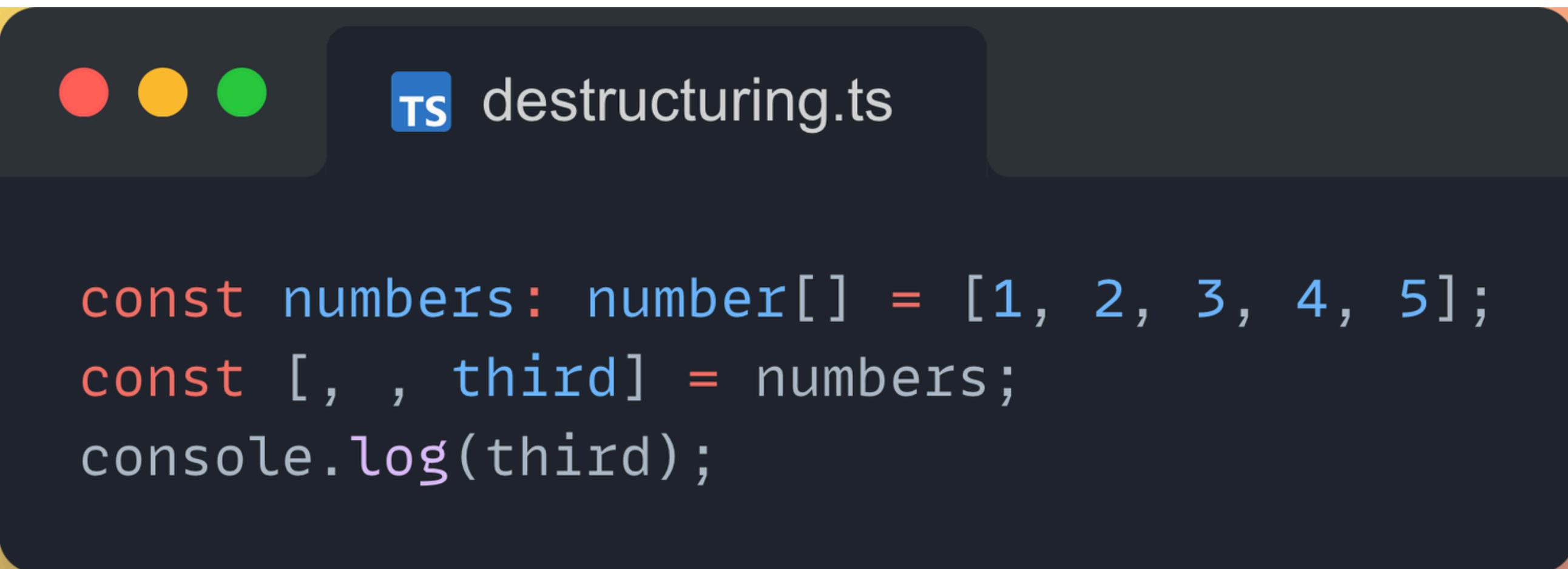
The image shows a screenshot of a code editor interface. At the top, there are three circular status indicators (red, yellow, green) followed by the text "destructuring.ts". The main area contains the following TypeScript code:

```
const numbers: number[] = [1, 2, 3, 4, 5];
const [first, second, third] = numbers;

console.log(first);
console.log(second);
console.log(third);
```

## • Destructuration avec les tableaux

Sauter des éléments :

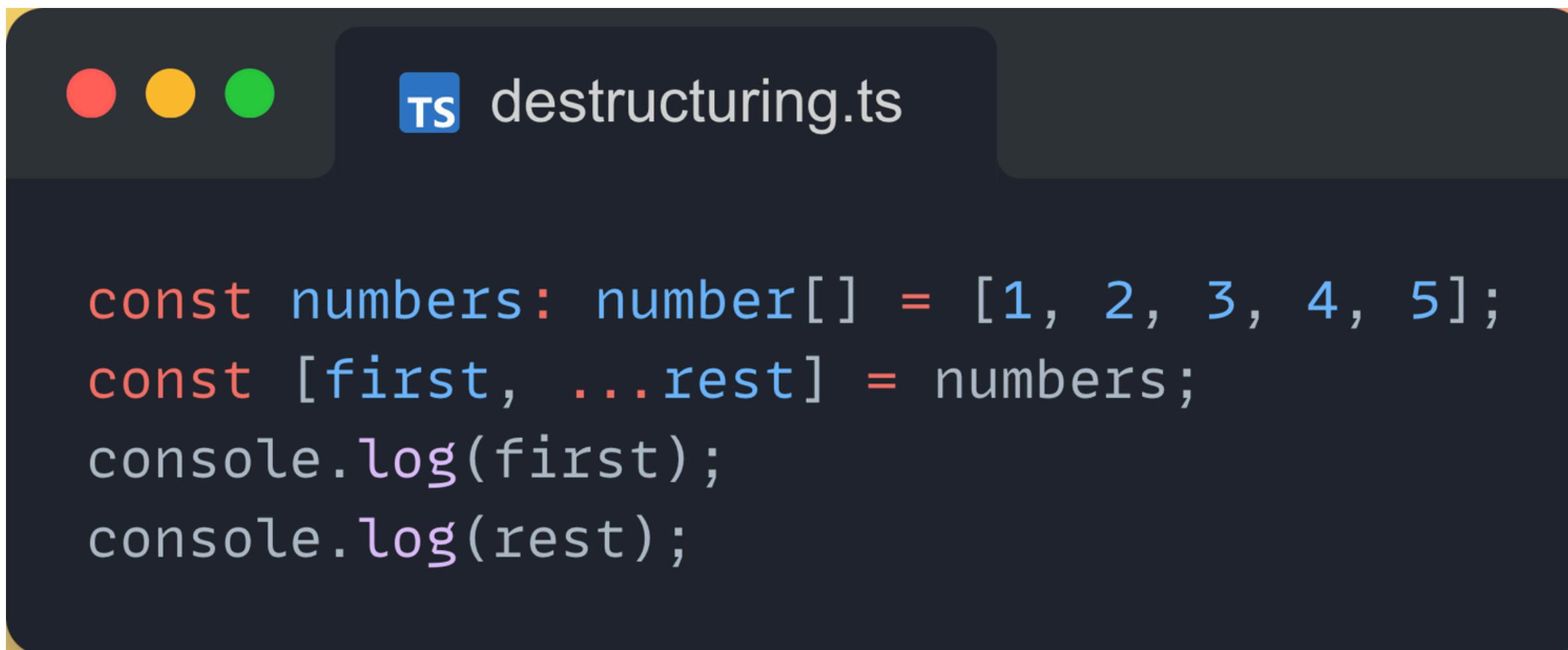


The image shows a screenshot of a code editor interface. At the top, there are three colored circular icons (red, yellow, green) followed by the text "destructuring.ts". The main area contains the following TypeScript code:

```
const numbers: number[] = [1, 2, 3, 4, 5];
const [, , third] = numbers;
console.log(third);
```

## • Destructuration avec les tableaux

### Le reste des éléments

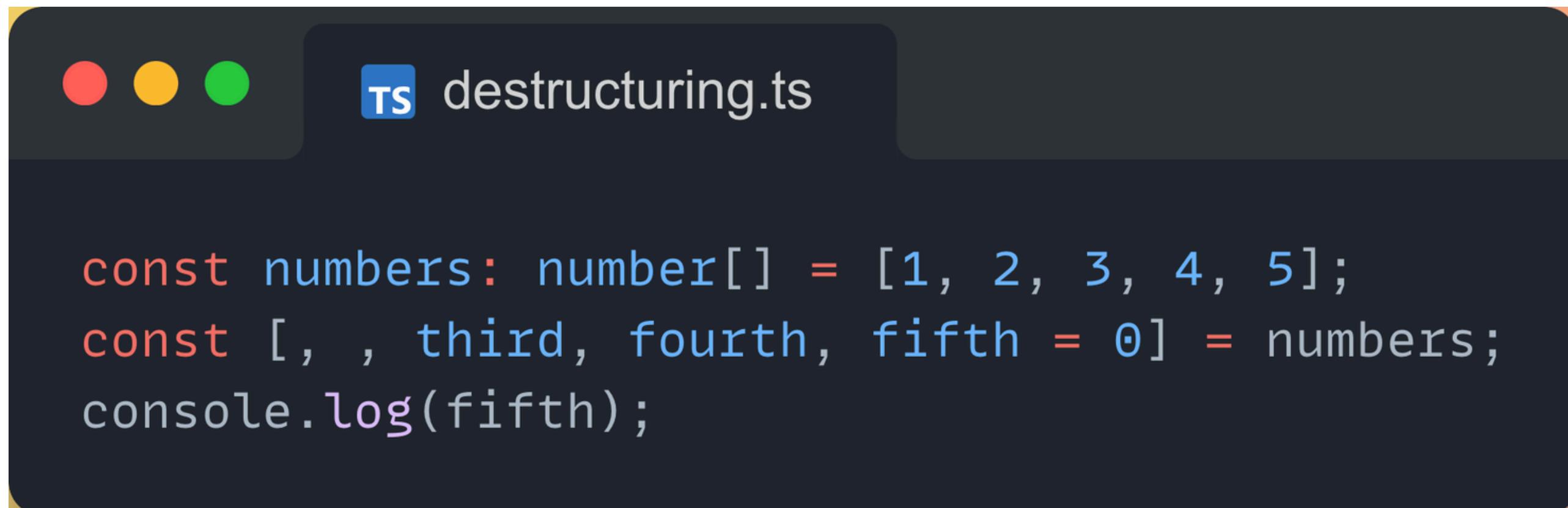


The image shows a screenshot of a code editor with a dark theme. At the top left, there are three colored circular icons: red, yellow, and green. To their right, the text "destructuring.ts" is displayed next to a blue "TS" icon, indicating the file name and its TypeScript nature. The main area of the editor contains the following code:

```
const numbers: number[] = [1, 2, 3, 4, 5];
const [first, ...rest] = numbers;
console.log(first);
console.log(rest);
```

## • Destructuration avec les tableaux

### Valeur par défaut

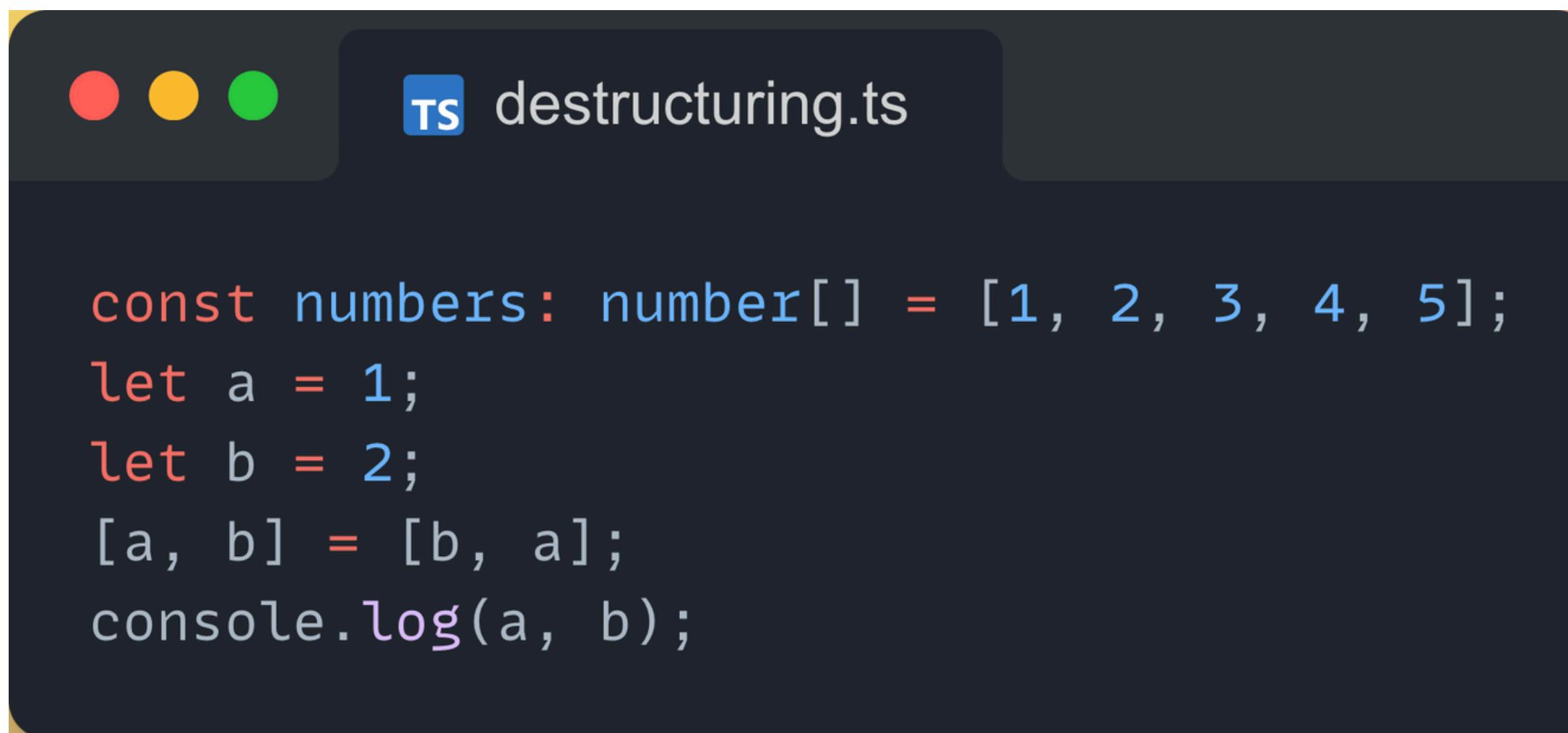


The image shows a dark-themed code editor interface. In the top left corner, there are three colored circular icons: red, yellow, and green. To the right of these icons is a blue square icon containing the letters 'TS', followed by the text 'destructuring.ts'. The main area of the editor displays the following TypeScript code:

```
const numbers: number[] = [1, 2, 3, 4, 5];
const [, , third, fourth, fifth = 0] = numbers;
console.log(fifth);
```

## • Destructuration avec les tableaux

Échange de valeurs :

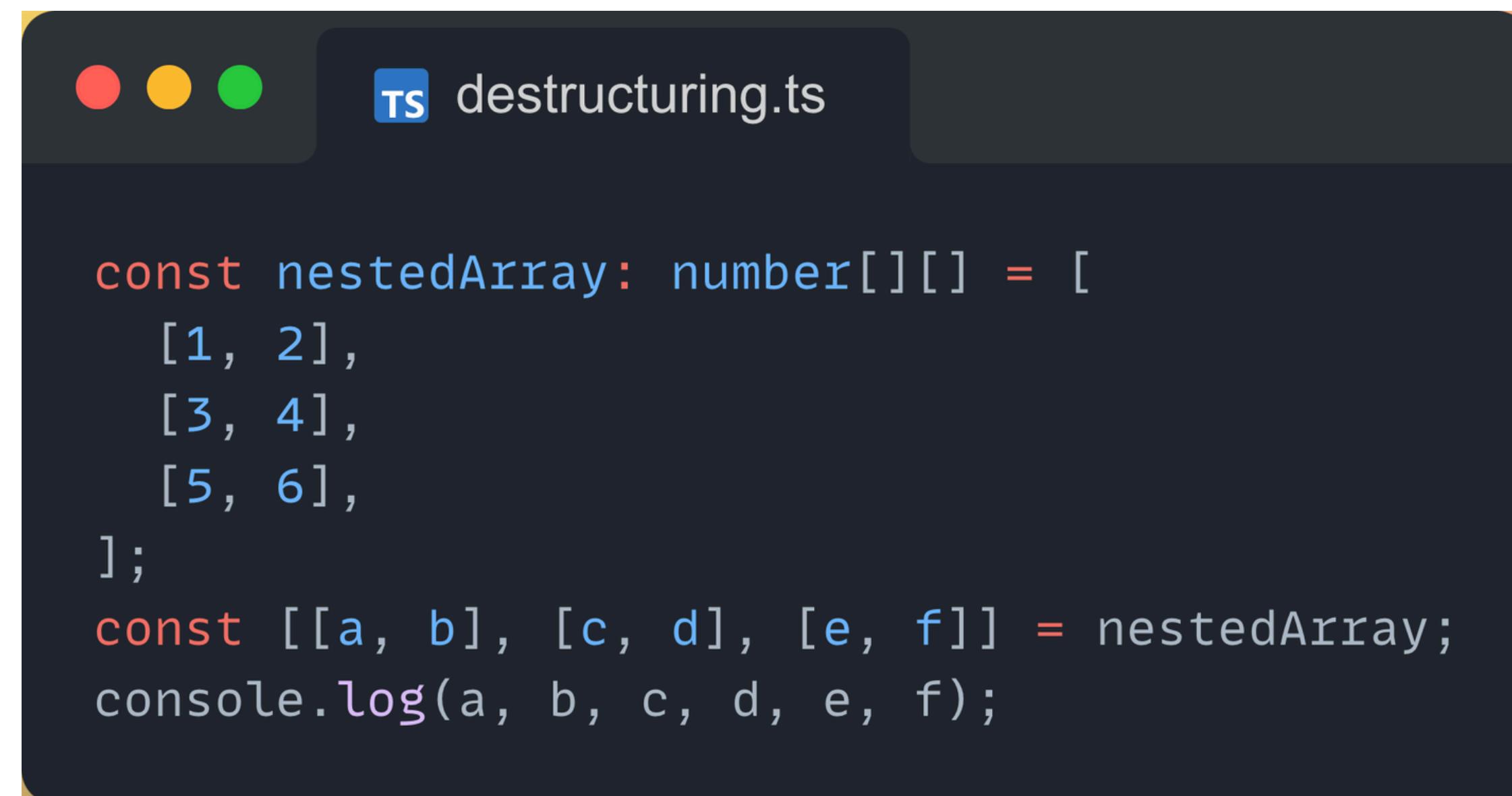


The image shows a dark-themed code editor window. In the top bar, there are three circular icons (red, yellow, green) on the left and a 'TS destructuring.ts' tab on the right. The main editor area contains the following TypeScript code:

```
const numbers: number[] = [1, 2, 3, 4, 5];
let a = 1;
let b = 2;
[a, b] = [b, a];
console.log(a, b);
```

## • Destructuration avec les tableaux

Déstructuration imbriquée:



The image shows a screenshot of a code editor with a dark theme. At the top, there are three circular icons (red, yellow, green) followed by the text "destructuring.ts". The code itself is as follows:

```
const nestedArray: number[][][] = [
  [1, 2],
  [3, 4],
  [5, 6],
];
const [[a, b], [c, d], [e, f]] = nestedArray;
console.log(a, b, c, d, e, f);
```

## • Destructuration avec les tableaux

Ignorer les éléments de fin :



ts destructuring.ts

```
const numbers: number[] = [1, 2, 3, 4, 5];
const [first, second] = numbers;
console.log(first, second);
```

- Destructuration avec les Objets

### Syntaxe générale

```
const { propriete1, propriete2, ..., proprieteN } = objet;
```

## • Destructuration avec les Objets

### Destructuration d'objets avec des valeurs par défaut

```
const person = { name: 'John' };
const { name, age = 25 } = person;

console.log(name); // Affiche: John
console.log(age); // Affiche: 25 (valeur par défaut car non présente dans l'objet)
```

## • Destructuration avec les Objets

### Destructuration d'objets de base

```
demo.ts  x  index.html
demo.ts > ...
1  const person = { name: 'John', age: 30 };
2  const { name: personName, age: personAge } = person;
3
4  console.log(personName); // Affiche: John
5  console.log(personAge); // Affiche: 30
6  |
```

## • Destructuration avec les Objets

### Destructuration d'objets avec des propriétés imbriquées

```
const user = {  
    name: 'Alice',  
    address: {  
        city: 'Paris',  
        country: 'France'  
    }  
};  
  
const { name, address: { city, country } } = user;  
  
console.log(name); // Affiche: Alice  
console.log(city); // Affiche: Paris  
console.log(country); // Affiche: France
```

## • Destructuration avec les Objets

### Destructuration d'objets avec des propriétés facultatives

```
demo.ts  ✘  index.html
demo.ts > ...
1  interface Person {
2    name: string;
3    age?: number;
4  }
5
6  const person: Person = { name: 'Jane' };
7  const { name: personName, age: personAge = 30 } = person;
8
9  console.log(personName);
10 console.log(personAge);
```

## • Les décorateurs

C'est une méthode de programmation orienté aspect qui permet de créer des points d'injection

Configuration de base



```
{  
  "compilerOptions": {  
    "target": "es2016",  
    "experimentalDecorators": true,  
  }  
}
```

- Les décorateurs

Les décorateurs utilisent la forme `@expression` ou `@expression()`, où l'expression doit évaluer une fonction qui sera appelée au moment de l'exécution avec des informations sur la déclaration décorée.

## • Les décorateurs

Il peut être attaché à une déclaration de classe, une méthode, un accesseur, une propriété ou un paramètre.

### Mots clés

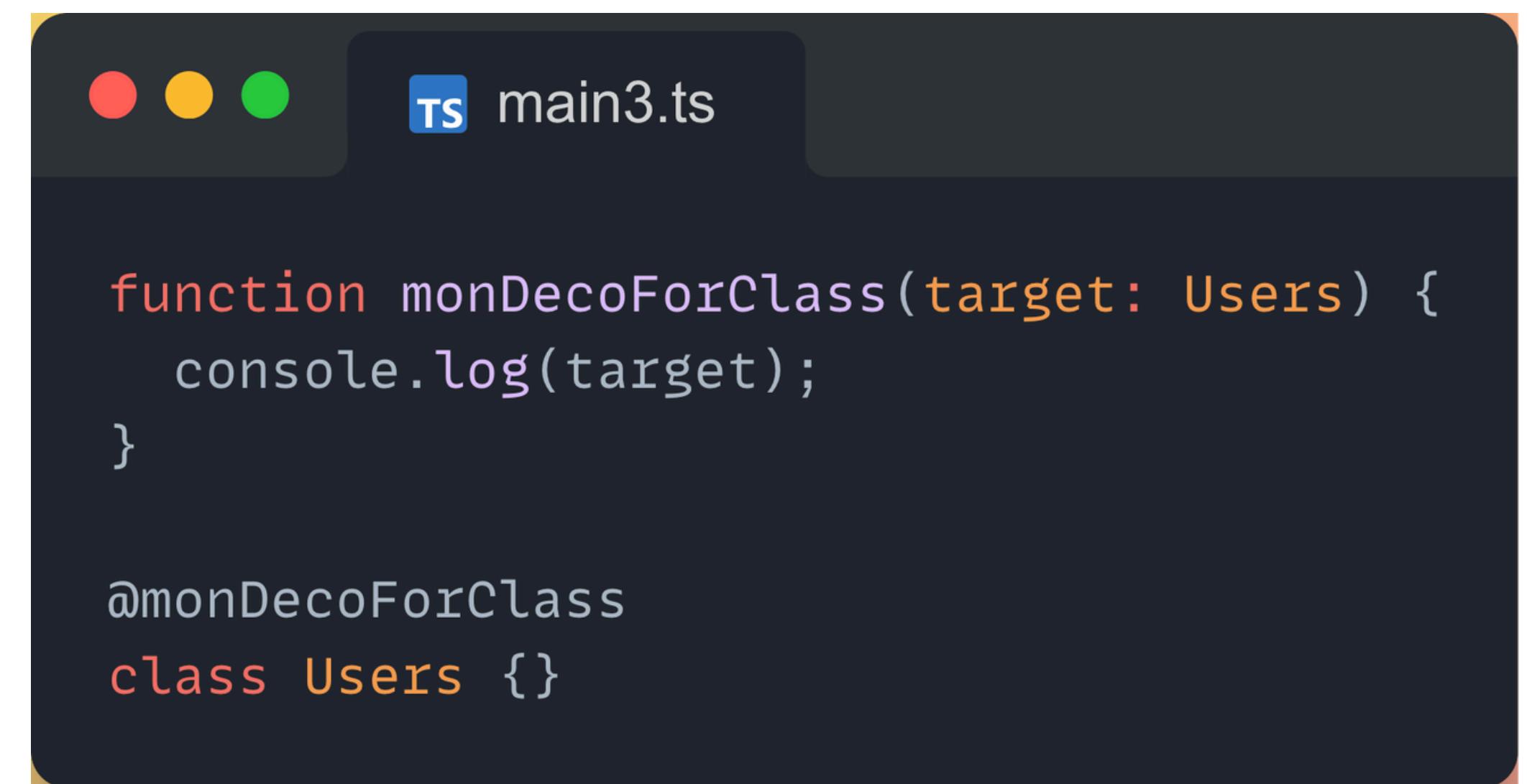
**descriptor**: Un objet qui contient les descripteurs de la propriété, tels que value, writable, enumerable, configurable, get et set

**propertyKey**: contient le nom de la méthode, de l'accesseur, de la propriété ou du paramètre

**target**: peut faire référence à la fonction constructeur de la classe ou le prototype de la classe selon les cas.

## • Les décorateurs

### Décorateur appliquer aux classes



```
function monDecoForClass(target: Users) {
    console.log(target);
}

@monDecoForClass
class Users {}
```

## • Les décorateurs

### Décorateur appliquer aux méthodes



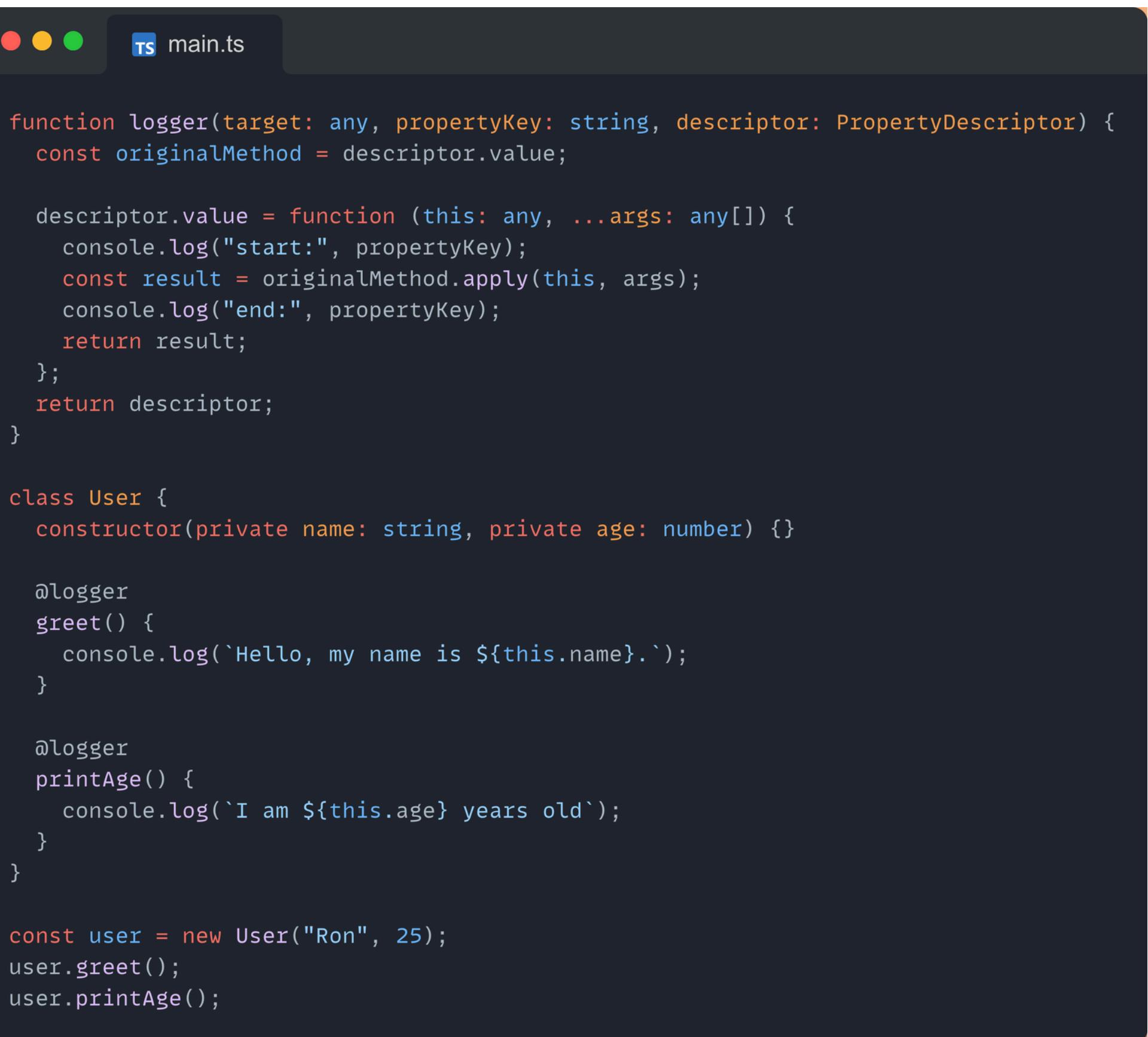
The screenshot shows a code editor window titled "main3.ts". The file contains the following TypeScript code:

```
function monDecoForMethod() {
  return function (target: Users | typeof Users, propertyKey: string, descriptor: PropertyDescriptor) {
    console.log(target, propertyKey, descriptor);
  };
}

class Users {
  @monDecoForMethod()
  maFonction(param1: string, param2: string) {}
  @monDecoForMethod()
  static maFonction2(param1: string, param2: string) {}
}
```

## • Les décorateurs

### Décorateur appliquer aux méthodes



The screenshot shows a code editor window with a dark theme. The title bar says "main.ts". The code implements a logger decorator:

```
function logger(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function (this: any, ...args: any[]) {
    console.log("start:", propertyKey);
    const result = originalMethod.apply(this, args);
    console.log("end:", propertyKey);
    return result;
  };
  return descriptor;
}

class User {
  constructor(private name: string, private age: number) {}

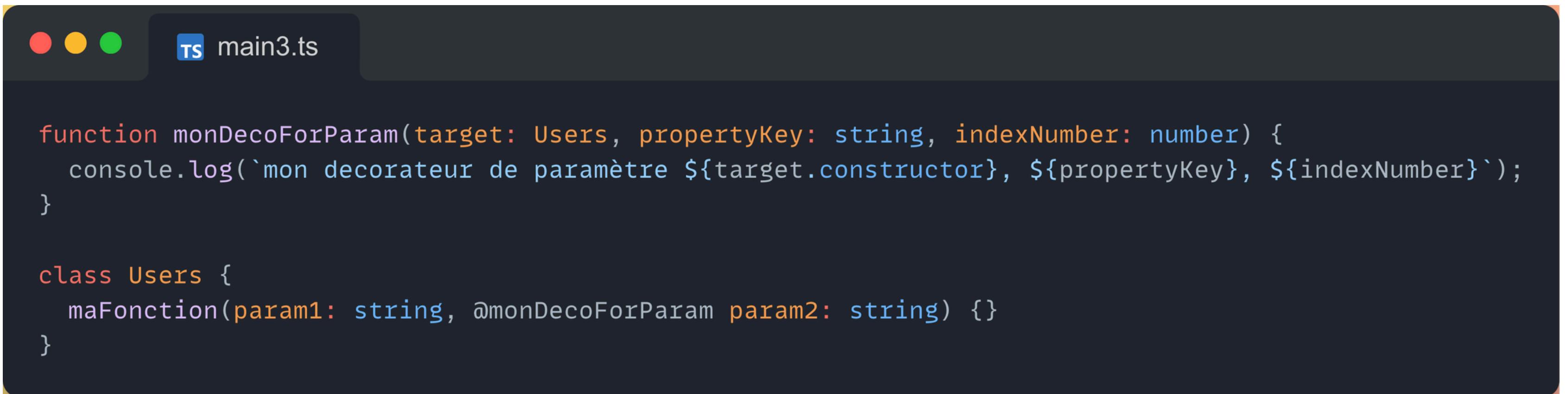
  @logger
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }

  @logger
  printAge() {
    console.log(`I am ${this.age} years old`);
  }
}

const user = new User("Ron", 25);
user.greet();
user.printAge();
```

## • Les décorateurs

### Décorateur appliquer aux paramètres



The screenshot shows a code editor window with a dark theme. The title bar says "main3.ts". The code editor displays the following TypeScript code:

```
function monDecoForParam(target: Users, propertyKey: string, indexNumber: number) {
  console.log(`mon decorateur de paramètre ${target.constructor}, ${propertyKey}, ${indexNumber}`);
}

class Users {
  maFonction(param1: string, @monDecoForParam param2: string) {}
}
```

## • Les décorateurs

Décorateur appliquer aux propriétés

```
● ○ ● TS main3.ts

function isValidEmail(email: string): boolean {
  const emailRegex = /^[^ \s@]+@[^\s@]+\.[^\s@]+$/;
  return emailRegex.test(email);
}
```

```
● ○ ● TS main3.ts

function IsEmail(target: any, propertyKey: string) {
  let value: string;

  const getter = function () {
    console.log(`Getting value of property ${propertyKey}`);
    return value;
  };

  const setter = function (newVal: string) {
    console.log(`Setting value of property ${propertyKey} to ${newVal}`);

    if (!isValidEmail(newVal)) {
      throw new Error(`Invalid email address: ${newVal}`);
    }

    value = newVal;
  };

  Object.defineProperty(target, propertyKey, {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true,
  });
}
```

## • Les décorateurs

### Décorateur appliquer aux propriétés



```
class Users {
  @IsEmail
  email: string;

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



```
const users = new Users('john.doe@example.com');

try {
  users.email = 'invalid-email'; // Le setter lèvera une exception
} catch (errors) {
  console.error(errors);
}

console.log(users.email);
```

## • Les décorateurs

### Décorateur appliquer aux propriétés

```
~/Desktop/Sonatel Academy/typescript/teste1
ts-node main3.ts
Setting value of property email to john.doe@example.com
Setting value of property email to invalid-email
Error: Invalid email address: invalid-email
    at Users.setter [as email] (/Users/cheikhbrahimadieng/Desktop/Sonatel Academy/typescript/teste1/main3.ts:61:19)
    at Object.<anonymous> (/Users/cheikhbrahimadieng/Desktop/Sonatel Academy/typescript/teste1/main3.ts:93:16)
    at Module._compile (node:internal/modules/cjs/loader:1358:14)
    at Module.m._compile (/usr/local/lib/node_modules/ts-node/src/index.ts:1618:23)
    at Module._extensions..js (node:internal/modules/cjs/loader:1416:10)
    at Object.require.extensions.<computed> [as .ts] (/usr/local/lib/node_modules/ts-node/src/index.ts:1621:12)
    at Module.load (node:internal/modules/cjs/loader:1208:32)
    at Function.Module._load (node:internal/modules/cjs/loader:1024:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:174:12)
    at phase4 (/usr/local/lib/node_modules/ts-node/src/bin.ts:649:14)
Getting value of property email
john.doe@example.com
```

## • Les décorateurs

Décorateur appliquer aux propriétés

```
● ○ ● TS main3.ts

class Users {
    @Contrainte({ min: 0, max: 30 })
    age: number = 10;
}

const users = new Users();
users.age = 10;
```

## Exemple 2

```
● ○ ● TS main3.ts

function Contrainte({min, max} : {min: number, max: number}){
    return function<T> (target: T, keys: keyof T){
        let val = target[keys] as any
        const setter = (v: unknown) => {
            console.log(typeof v = 'number' && v > min && v < max)
            if (typeof v = 'number' && v > min && v < max){
                val = v;
                return;
            }
            console.log("erreur")
        }

        const getter = () => val
        Object.defineProperty(target, keys, {
            get: getter,
            set: setter
        })
    }
}
```

- Map

Les objets Map sont des collections de paires clé-valeur.

Vérification de l'existence d'une clé

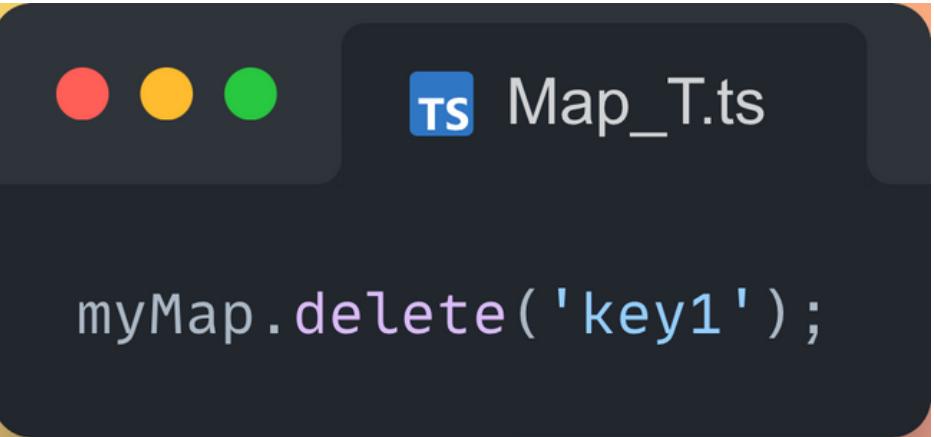
Ajout et récupération de valeurs :

The screenshot shows a code editor window with a dark theme. At the top, there are three colored circular icons (red, yellow, green) followed by a blue square icon containing the letters 'TS' and the text 'Map\_T.ts'. The code in the editor is as follows:

```
const myMap = new Map();
myMap.set('key1', 'value1');
myMap.set('key2', 'value2');
console.log(myMap.get('key1'));
console.log(myMap.has('key1'));
```

- Map

## Suppression des paires clé-valeur



```
myMap.delete('key1');
```

## Itération sur les entrées de la Map



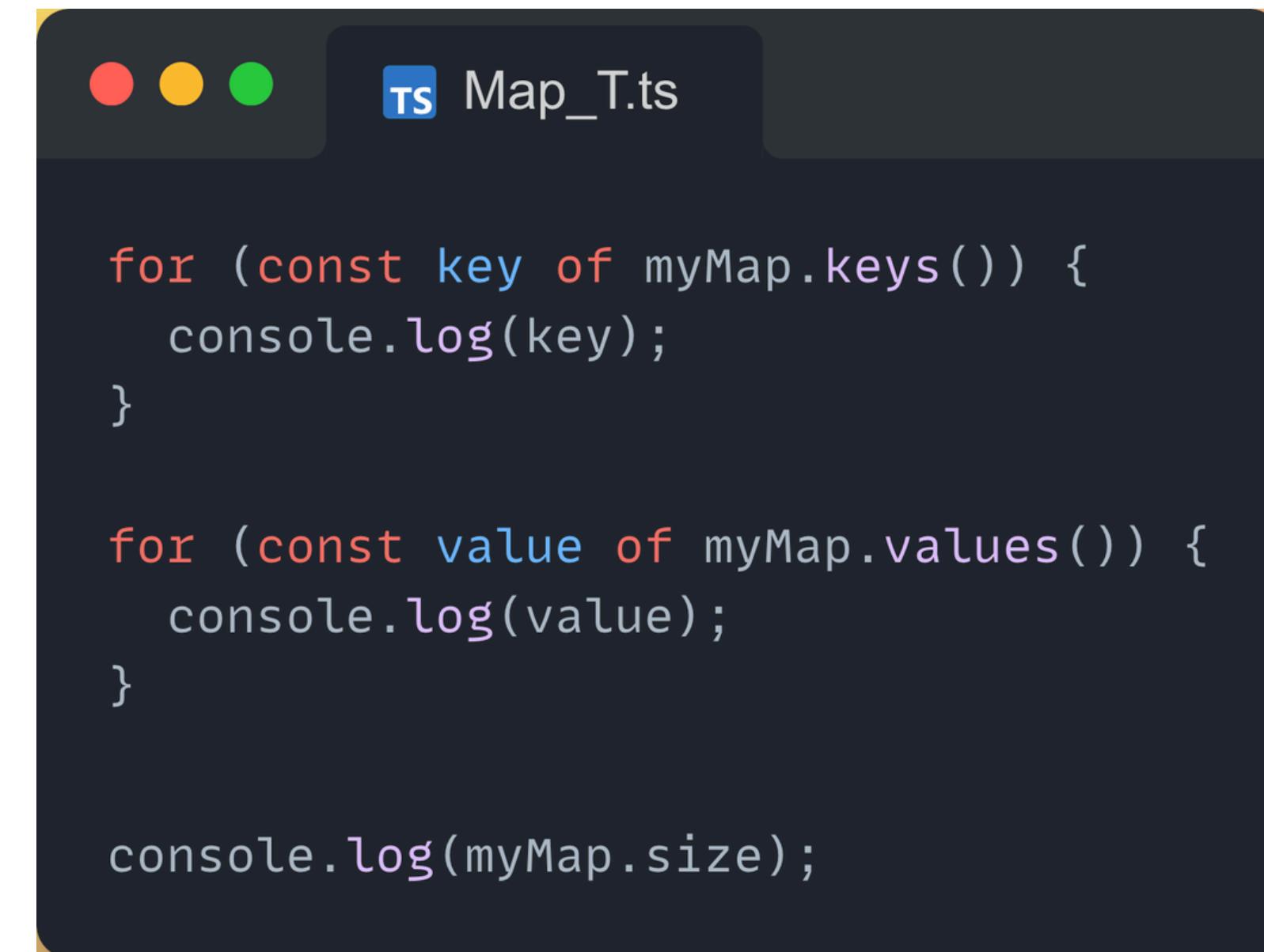
```
for (const [key, value] of myMap.entries()) {
  console.log(` ${key}: ${value}`);
}
```

- Map

Itération sur les clés de la Map

Itération sur les valeurs de la Map

La taille de la Map



Map\_T.ts

```
for (const key of myMap.keys()) {
    console.log(key);
}

for (const value of myMap.values()) {
    console.log(value);
}

console.log(myMap.size);
```

• Map



The screenshot shows a code editor window with a dark theme. At the top, there are three small colored circles (red, yellow, green) and a 'TS' icon followed by the filename 'Map\_T.ts'. The code itself is written in TypeScript:

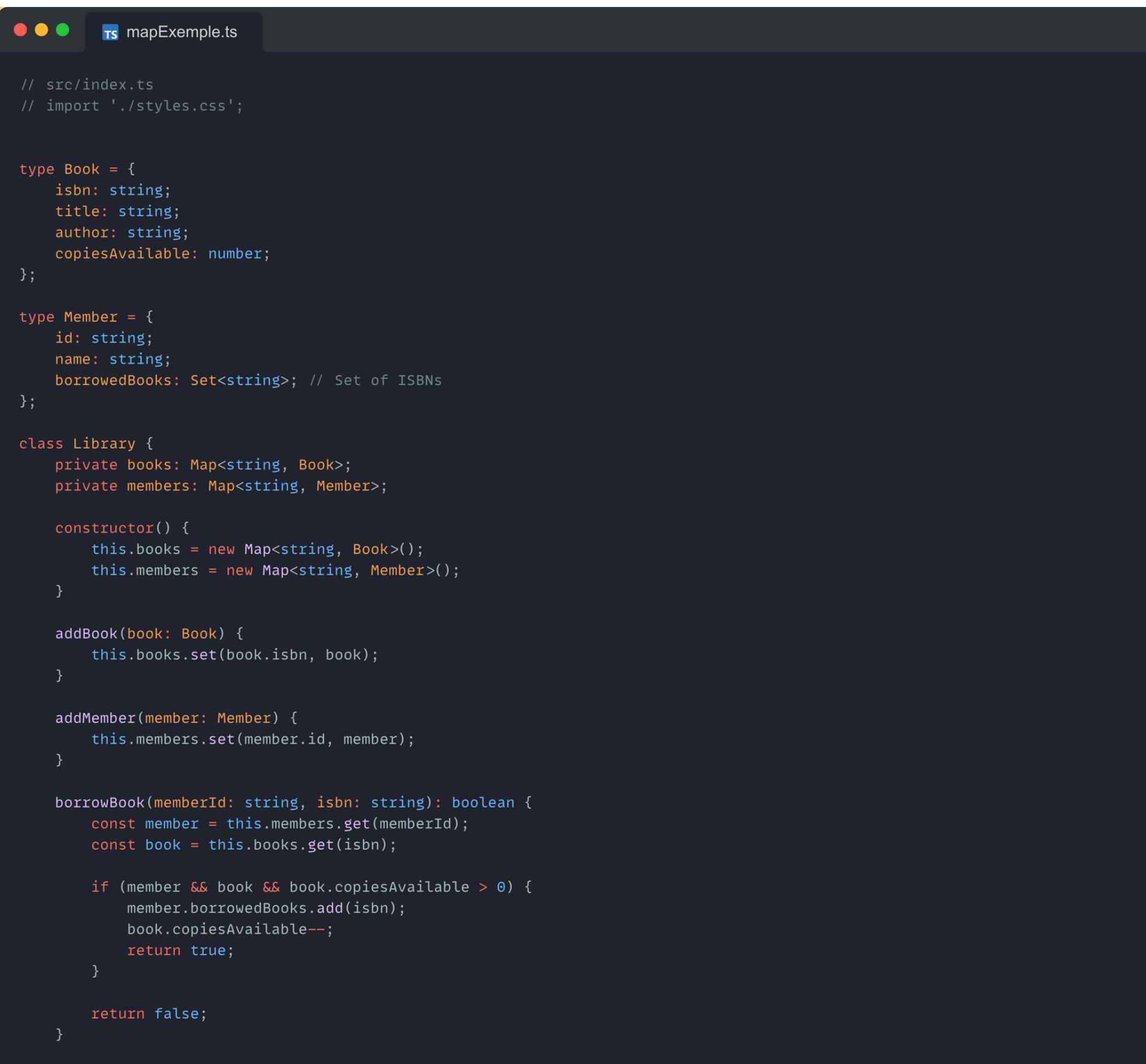
```
const task = new Map<string, string>([
    ['task', 'Code'],
    ['task1', 'Code1'],
    ['false', 'Code']
]);

const valuesIterator = task.values();

const firstValue = valuesIterator.next().value;

console.log(firstValue);
```

## ● Map



```
// src/index.ts
// import './styles.css';

type Book = {
    isbn: string;
    title: string;
    author: string;
    copiesAvailable: number;
};

type Member = {
    id: string;
    name: string;
    borrowedBooks: Set<string>; // Set of ISBNs
};

class Library {
    private books: Map<string, Book>;
    private members: Map<string, Member>;

    constructor() {
        this.books = new Map<string, Book>();
        this.members = new Map<string, Member>();
    }

    addBook(book: Book) {
        this.books.set(book.isbn, book);
    }

    addMember(member: Member) {
        this.members.set(member.id, member);
    }

    borrowBook(memberId: string, isbn: string): boolean {
        const member = this.members.get(memberId);
        const book = this.books.get(isbn);

        if (member && book && book.copiesAvailable > 0) {
            member.borrowedBooks.add(isbn);
            book.copiesAvailable--;
            return true;
        }

        return false;
    }
}
```

## ● Map

```
        }

    returnBook(memberId: string, isbn: string): boolean {
    const member = this.members.get(memberId);
    const book = this.books.get(isbn);

    if (member && book && member.borrowedBooks.has(isbn)) {
        member.borrowedBooks.delete(isbn);
        book.copiesAvailable++;
        return true;
    }

    return false;
}

listAvailableBooks(): void {
    console.log("Available Books:");
    for (const [isbn, book] of this.books.entries()) {
        if (book.copiesAvailable > 0) {
            console.log(`- ${book.title} by ${book.author} (ISBN: ${isbn}, Copies Available: ${book.copiesAvailable})`);
        }
    }
}

const library = new Library();

library.addBook({ isbn: "978-3-16-148410-0", title: "Book One", author: "Author A", copiesAvailable: 3 });
library.addBook({ isbn: "978-1-61-729054-0", title: "Book Two", author: "Author B", copiesAvailable: 2 });

console.log(library)

// library.addMember({ id: "1", name: "Alice", borrowedBooks: new Set<string>() });
// library.addMember({ id: "2", name: "Bob", borrowedBooks: new Set<string>() });

// library.borrowBook("1", "978-3-16-148410-0");
// library.borrowBook("2", "978-1-61-729054-0");

// library.returnBook("1", "978-3-16-148410-0");

// // Listing available books
// library.listAvailableBooks();
```