TypeScript : variables et fonctions

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

Plan

Plan

TypeScript

Trois modes de typage en **TypeScript**

- Statique : le développeur précise le type à la déclaration de la variable
- Dynamique : le type d'une variable est déterminé à l'exécution (comme en JS)
- Générique : il permet au développeur de paramétrer un type complexe

TypeScript

Déclarer une variable

```
var nomVariable = valeurInitiale;
```

TypeScript

Déclarer une variable

```
var nomVariable = valeurInitiale;
```

Exemple

```
var x = "bonjour";
```

TypeScript

Déclarer une variable

```
var nomVariable = valeurInitiale;
```

Exemple

```
var x = "bonjour";
```


JavaScript détermine dynamiquement le type de la variable

```
console.log(typeof x);
// affiche string

x = 2;
console.log(typeof x);
// affiche number
```

TypeScript

Avec le typage dynamique, l'erreur suivante ne sera détectée qu'à l'exécution

```
var x = 2;
console.log(x.toUpperCase());
```

TypeScript

Avec le typage dynamique, l'erreur suivante ne sera détectée qu'à l'exécution

```
var x = 2;
console.log(x.toUpperCase());
```


Remarques

Avec le typage statique (en utilisant un IDE), cette erreur sera détectée à la compilation.

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

Initialiser une variable

```
x = 2;
```

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```


Initialiser une variable

```
x = 2;
```

Déclarer et initialiser une variable

```
var x: number = 2;
```

TypeScript

Cependant, ceci génère une erreur car une varaible ne change pas de type

```
x = "bonjour";
```

TypeScript

Quels types pour les variables en **TypeScript** ?

- `number` pour les nombres (entiers, réels, binaires, décimaux, hexadécimaux...)
- `string` pour les chaînes de caractère
- `boolean` pour les booléens
- `symbol` pour les symboles [**ES6**]
- `array` pour les tableaux non-statiques (taille variable)
- `tuple` pour les tableaux statiques (taille et type fixes)
- `object` pour les objets
- `any` pour les variables pouvant changer de type dans le programme
- `enum` pour les énumérations (tableau de constantes)

TypeScript

Quels types pour les variables en **TypeScript** ?

- `number` pour les nombres (entiers, réels, binaires, décimaux, hexadécimaux...)
- `string` pour les chaînes de caractère
- `boolean` pour les booléens
- `symbol` pour les symboles [**ES6**]
- `array` pour les tableaux non-statiques (taille variable)
- `tuple` pour les tableaux statiques (taille et type fixes)
- `object` pour les objets
- `any` pour les variables pouvant changer de type dans le programme
- `enum` pour les énumérations (tableau de constantes)

Les types `undefined` et `null` du **JavaScript** sont aussi disponibles.

TypeScript

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

TypeScript

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

On peut aussi utiliser `template strings`

```
var str3: string = `Bonjour ${ str2 } ${ str1 }
Que pensez-vous de TypeScript ?
`;
console.log(str3);
// affiche Bonjour john wick
Que pensez-vous de TypeScript ?
```

TypeScript

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

On peut aussi utiliser `template strings`

```
var str3: string = `Bonjour ${ str2 } ${ str1 }
Que pensez-vous de TypeScript ?
`;
console.log(str3);
// affiche Bonjour john wick
Que pensez-vous de TypeScript ?
```

L'équivalent de faire

```
var str3: string = "Bonjour " + str2 + " " + str1 + "\nQue
   pensez-vous de TypeScript ?";
```

TypeScript

Une première déclaration de tableau

```
var list: number[] = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

TypeScript

Une première déclaration de tableau

```
var list: number[] = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

Une deuxième déclaration

```
var list: Array<number> = new Array(1, 2, 3);
console.log(list);
// affiche [ 1, 2, 3 ]
```

TypeScript

Une première déclaration de tableau

```
var list: number[] = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

Une deuxième déclaration

```
var list: Array<number> = new Array(1, 2, 3);
console.log(list);
// affiche [ 1, 2, 3 ]
```

Ou encore plus simple

```
var list: Array<number> = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

TypeScript

Tuple

- structure de données multi-valeurs
- permettant de stocker une collection d'éléments de différents types dans un ordre spécifique

TypeScript

Pour les tuples, on initialise toutes les valeurs à la déclaration

```
var t: [number, string, string] = [100, "wick", 'john'];
```

TypeScript

Pour les tuples, on initialise toutes les valeurs à la déclaration

```
var t: [number, string, string] = [100, "wick", 'john'];
```

Pour accéder à un élément d'un tuple en lecture

```
console.log(t[0]);
// affiche 100
```

TypeScript

Pour les tuples, on initialise toutes les valeurs à la déclaration

```
var t: [number, string, string] = [100, "wick", 'john'];
```

Pour accéder à un élément d'un tuple en lecture

```
console.log(t[0]);
// affiche 100
```


Ou en écriture

```
t[2] = "travolta";
console.log(t);
// affiche [ 100, 'wick', 'travolta' ]
```

TypeScript

Pour modifier toutes les valeurs d'un tuple en respectant les types indiqués à la déclaration

```
t = [ 200, "deepay", 'memphis' ];
console.log(t)
// affiche [ 200, 'deepay', 'memphis' ]
```

TypeScript

Pour modifier toutes les valeurs d'un tuple en respectant les types indiqués à la déclaration

```
t = [ 200, "deepay", 'memphis' ];
console.log(t)
// affiche [ 200, 'deepay', 'memphis' ]
```

Cependant, ceci génère une erreur

```
t = [100, 200, 'john'];
```

TypeScript

Pour modifier toutes les valeurs d'un tuple en respectant les types indiqués à la déclaration

```
t = [ 200, "deepay", 'memphis' ];
console.log(t)
// affiche [ 200, 'deepay', 'memphis' ]
```

Cependant, ceci génère une erreur

```
t = [100, 200, 'john'];
```

Une affectation partielle du tuple génère aussi une erreur

```
t = [200, 'Dalton']
```

TypeScript

Avec TypeScript 3.0, on peut utiliser `?` rendre certains éléments de tuple optionnels

```
var t: [number, string, string?];
```

TypeScript

Avec TypeScript 3.0, on peut utiliser ? rendre certains éléments de tuple optionnels

```
var t: [number, string, string?];
```

En faisant une affectation à un tuple, il faut indiquer une valeur pour chaque élément non optionnel

```
t = [200, 'Dalton']
console.log(t);
// affiche [ 100, 'Dalton' ]
```

TypeScript

Avec TypeScript 3.0, on peut utiliser `?` rendre certains éléments de tuple optionnels

```
var t: [number, string, string?];
```

En faisant une affectation à un tuple, il faut indiquer une valeur pour chaque élément non optionnel

```
t = [200, 'Dalton']
console.log(t);
// affiche [ 100, 'Dalton' ]
```


La valeur d'un élément optionnel non initialisé est `undefined`

```
console.log(t[2]);
// affiche undefined
```

TypeScript

Pour ajouter un élément

```
t[2] = 'john';
```

TypeScript

Pour ajouter un élément

```
t[2] = 'john';
```

Ceci génère une erreur

```
t[2] = 100;
```

TypeScript

Pour ajouter un élément

```
t[2] = 'john';
```

Ceci génère une erreur

```
t[2] = 100;
```


Et cette instruction aussi car on dépasse la taille du tuple

```
t[3] = 100;
```

TypeScript

Depuis TypeScript 4.2, on peut utiliser les `...` pour le restant d'éléments de même type

```
var student: [string, ...number[]] = ["wick"];
console.log(student);
// affiche [ 100 ]
```

TypeScript

Depuis TypeScript 4.2, on peut utiliser les ... pour le restant d'éléments de même type

```
var student: [string, ...number[]] = ["wick"];
console.log(student);
// affiche [ 100 ]
```

Ainsi, nous pouvons ajouter un nombre variable d'éléments de même type

```
student = ["wick", 10, 20, 15]
console.log(student);
// affiche [ 'wick', 10, 20, 15 ]

student = ["wick", 11, 13]
console.log(student);
// affiche [ 'wick', 11, 13 ]
```

TypeScript

Les éléments restants peuvent ne pas être à la dernière position si le l'élément suivant est ni optionnel ni restant et est de type différent

```
var student: [string, ...number[], string] = ["wick", "paris"];
console.log(student);
// affiche [ 'wick', 'paris' ]
```

TypeScript

Les éléments restants peuvent ne pas être à la dernière position si le l'élément suivant est ni optionnel ni restant et est de type différent

```
var student: [string, ...number[], string] = ["wick", "paris"];
console.log(student);
// affiche [ 'wick', 'paris' ]
```

Exemple d'ajout

```
student = ["wick", 10, 20, 15, "marseille"]
console.log(student);
// affiche [ 'wick', 10, 20, 15, 'marseille' ]

student = ["wick", 11, 13, "lyon"]
console.log(student);
// affiche [ 'wick', 11, 13, 'lyon' ]
```

TypeScript

Exemple avec `any`

```
var x: any;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```

TypeScript

Exemple avec `any`

```
var x: any;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```


Une variable de type `any` peut être affectée à n'importe quel autre type de variable

```
var x: any;
x = "bonjour";
x = 5;
var y: number = x;
```

TypeScript

Le type `unknown` (TypeScript 3.0) fonctionne comme `any` mais ne peut être affecté qu'à une variable de type `unknown` ou `any`

```
var x: unknown;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```

TypeScript

Le type `unknown` (TypeScript 3.0) fonctionne comme `any` mais ne peut être affecté qu'à une variable de type `unknown` ou `any`

```
var x: unknown;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```


Ceci génère donc une erreur

```
var x: unknown;
x = "bonjour";
x = 5;
var y: number = x;
```

TypeScript

Énumérations

- structure de données multi-valeurs
- permettant de stocker des valeurs nommées
- les noms et les valeurs étant accessibles via une syntaxe d'indexation.

TypeScript

Déclarons une énumération (dans `file.ts`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
   AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

TypeScript

Déclarons une énumération (dans `file.ts`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
   AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

L'indice du premier élément est 0

```
console.log(mois.AVRIL)
// affiche 3
```

TypeScript

Déclarons une énumération (dans `file.ts`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
   AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

L'indice du premier élément est 0

```
console.log(mois.AVRIL)
// affiche 3
```


Pour accéder au nom via la valeur (l'indice)

```
console.log(mois[3])
// affiche AVRIL
```

TypeScript

Pour modifier l'indice (la valeur) du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

TypeScript

Pour modifier l'indice (la valeur) du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

En affichant maintenant, le résultat est

```
console.log(mois.AVRIL)
// affiche 4
```

TypeScript

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

TypeScript

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

En affichant, le résultat est

```
console.log(mois.MARS);
// affiche 3
console.log(mois.JUIN);
// affiche 12
console.log(mois.DECEMBRE);
// affiche 3
```

TypeScript

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

En affichant, le résultat est

```
console.log(mois.MARS);
// affiche 3
console.log(mois.JUIN);
// affiche 12
console.log(mois.DECEMBRE);
// affiche 3
```

Ceci est une erreur, on ne peut modifier une constante

```
mois.JANVIER = 3;
```

TypeScript

Pour déclarer un objet

```
var obj: {
    nom: string;
    numero: number;
};
```

TypeScript

Pour déclarer un objet

```
var obj: {
    nom: string;
    numero: number;
};
```

On peut initialiser les attributs de cet objet

```
obj = {
    nom: 'wick',
    numero: 100
};

console.log(obj);
// affiche { nom: 'wick', numero: 100 }

console.log(typeof obj);
// affiche object
```

TypeScript

On peut modifier les valeurs d'un objet ainsi

```
obj.nom = 'abruzzi';
obj['numero'] = 200;

console.log(obj);
// affiche { nom: 'abruzzi', numero: 200 }
```

TypeScript

On peut modifier les valeurs d'un objet ainsi

```
obj.nom = 'abruzzi';
obj['numero'] = 200;

console.log(obj);
// affiche { nom: 'abruzzi', numero: 200 }
```


Ceci est une erreur

```
obj.nom = 125;
```

TypeScript

Pour déclarer un symbole

```
const sym = Symbol(2);

console.log(sym);
// affiche Symbol(2)
```

TypeScript

Pour déclarer un symbole

```
const sym = Symbol(2);

console.log(sym);
// affiche Symbol(2)
```

Les symboles sont uniques

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

console.log(sym2 === sym3);
// affiche false
```

TypeScript

Un symbole peut-être utilisé comme une propriété d'objet (comme les chaînes de caractères)

```
const sym = Symbol(2);

let obj = {
    [sym]: "value"
};

console.log(obj);
// affiche { [Symbol(2)]: 'value' }

console.log(obj[sym]);
// affiche value

console.log(sym);
// affiche Symbol(2)
```

TypeScript

Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable

TypeScript

Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable

Déclarer une variable acceptant plusieurs types de valeur

```
var y: number | boolean | string;
```

TypeScript

Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable

Déclarer une variable acceptant plusieurs types de valeur

```
var y: number | boolean | string;
```

Affecter des valeurs de type différent

```
y = 2;
y = "bonjour";
y = false;
```

TypeScript

Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable

Déclarer une variable acceptant plusieurs types de valeur

```
var y: number | boolean | string;
```

Affecter des valeurs de type différent

```
y = 2;
y = "bonjour";
y = false;
```

Ceci génère une erreur

```
y = [2, 5];
```

TypeScript

Croisement de type

Il est possible qu'une variable ait les propriétés de plusieurs types différents

TypeScript

Croisement de type

Il est possible qu'une variable ait les propriétés de plusieurs types différents

Déclarer une variable ayant les propriétés de plusieurs types différents

```
var enseignant: {
    nom: string;
    salaire: number;
};

var etudiant: {
    niveau: string;
};

var doctorant: typeof etudiant &  typeof enseignant = {
    nom: 'wick',
    salaire: 1700,
    niveau: 'master'
};
console.log(doctorant);
// affiche { nom: 'wick', salaire: 1700, niveau: 'master' }
```

TypeScript

Le mot-clé `let`

permet de donner une visibilité locale à une variable déclarée dans un bloc.

TypeScript

Le mot-clé `let`

permet de donner une visibilité locale à une variable déclarée dans un bloc.

Ceci génère une erreur car la variable `x` a une visibilité locale limitée au bloc `if`

```
if (5 > 2)
{
  let x = 1;
}
console.log(x);
// affiche ReferenceError: x is not defined
```

TypeScript

Pour convertir une chaîne de caractère en nombre

```
let x : string = "2";
let y: string = "3.5";

let a: number = Number(x);
let b: number = Number(y);

console.log(a);
// affiche 2

console.log(b);
// affiche 3.5
```

TypeScript

Pour convertir une chaîne de caractère en nombre

```
let x : string = "2";
let y: string = "3.5";

let a: number = Number(x);
let b: number = Number(y);

console.log(a);
// affiche 2

console.log(b);
// affiche 3.5
```

Il existe une fonction de conversion pour chaque type.

TypeScript

Le mot-clé `type` permet de définir un alias de type

```
type maStructure = [number, string, string];
```

TypeScript

Le mot-clé `type` permet de définir un alias de type

```
type maStructure = [number, string, string];
```

Ensuite, on peut utiliser `maStructure` comme un type

```
let first: maStructure = [100, "wick", 'john' ];

console.log(first);
// affiche [ 100, 'wick', 'john' ]
```

TypeScript

`type` **peut être utilisé pour indiquer un ensemble de valeurs autorisées**

```
type Color = "red" | "green" | "blue";

function setColor(color: Color) {
    console.log(`La couleur est ${color}`);
}
```

TypeScript

Ainsi les appels suivants passents

```
setColor("red");
// La couleur est red

setColor("green");
// La couleur est green

setColor("blue");
// La couleur est blue
```

TypeScript

Ainsi les appels suivants passents

```
setColor("red");
// La couleur est red

setColor("green");
// La couleur est green

setColor("blue");
// La couleur est blue
```


Et ce dernier génère une erreur

```
setColor("yellow");
```

TypeScript

Opérateurs **JavaScript** sur les variables : rappel

- $++$
- $--$
- $+=$
- $-=$
- $*=$
- $/=$
- ...

TypeScript

Coalescence nulle (? ?)

- Introduit dans **ES2020**.
- Intégré dans **TypeScript** depuis la version 3.7.

TypeScript

Coalescence nulle (? ?)

- Introduit dans **ES2020**.
- Intégré dans **TypeScript** depuis la version 3.7.

L'opérateur `??` permet d'éviter d'affecter la valeur `null` ou `undefined` à une variable

```
let nom: string;
let value: string = nom ?? "doe";
console.log(value);
// affiche doe
```

C'est équivalent à

```
let nom: string;
let value: string = (nom !== null && nom !== undefined) ? nom : 'doe';
console.log(value);
// affiche doe
```

TypeScript

Coalescence nulle et affectation (? ?=)

- Introduit dans **ES2021**.
- Intégré dans **TypeScript** depuis la version 4.0.

TypeScript

Coalescence nulle et affectation (? ?=)

- Introduit dans **ES2021**.
- Intégré dans **TypeScript** depuis la version 4.0.

L'opérateur `??=` permet d'affecter une valeur à la même variable si sa valeur actuelle est `null` ou `undefined`

```
let nom: string;
nom ??= "doe";
console.log(nom);
// affiche doe
```

C'est équivalent à

```
let nom: string;
if (nom === null || nom === undefined) {
    nom = 'doe';
}
console.log(nom);
// affiche doe
```

TypeScript

Déstructuration (ES6)

permet d'extraire les données d'un objet ou un tableau dans des variables.

TypeScript

Déstructuration (ES6)

permet d'extraire les données d'un objet ou un tableau dans des variables.

Exemple

```
var personne = { nom: 'wick', prenom: 'john' };
var { nom, prenom } = personne;

console.log(nom, prenom);
// affiche wick john
```

TypeScript

Puissance (**)

- Introduit dans **ES2016**.
- Équivalent de `Math.pow()` dans les versions précédentes.
- Intégré dans **TypeScript** depuis la version 4.0.

TypeScript

Puissance (**)

- Introduit dans **ES2016**.
- Équivalent de `Math.pow()` dans les versions précédentes.
- Intégré dans **TypeScript** depuis la version 4.0.

Exemple

```
console.log(2 ** 3);
// affiche 8
```

TypeScript

Puissance (**=)

- Introduit dans **ES2016**.
- Intégré dans **TypeScript** depuis la version 4.0.

TypeScript

Puissance (**=)

- Introduit dans **ES2016**.
- Intégré dans **TypeScript** depuis la version 4.0.

Exemple

```
let a: number = 2, b: number = 3;
a **= b;
// l'équivalent de a = a ** b;
console.log(a);
// affiche 8
```

TypeScript

Opérateur de coupe-circuit (&&=)

- Introduit dans **ES2021**.
- Intégré dans **TypeScript** depuis la version 4.0.
- `x &&= y` : c'est-à-dire, `x` reçoit la valeur de `y` si et seulement si `x` est `truthy` (not `falsy`).
- Les valeurs `falsy` sont `false`, `0`, `""`, `null`, `undefined`, `NaN`.

L'écriture sans simplification

```
let a: string | null = "wick", b: string = "travolta";

if (a) {
    a = b;
}

console.log(a);
// affiche travolta
```

L'écriture sans simplification

```
let a: string | null = "wick", b: string = "travolta";

if (a) {
    a = b;
}

console.log(a);
// affiche travolta
```

L'équivalent en ES2020

```
let a: string | null = "wick", b: string = "travolta";
a = a && (a = b);

console.log(a);
// affiche travolta
```

L'écriture sans simplification

```
let a: string | null = "wick", b: string = "travolta";

if (a) {
    a = b;
}

console.log(a);
// affiche travolta
```

L'équivalent en ES2020

```
let a: string | null = "wick", b: string = "travolta";
a = a && (a = b);

console.log(a);
// affiche travolta
```

L'équivalent en ES2021 et TypeScript depuis la version 4.0

```
let a: string | null = "wick", b: string = "travolta";
a &&= b;

console.log(a);
// affiche travolta
```

TypeScript

Opérateur de coupe-circuit (||=)

- Introduit dans **ES2021**.
- Intégré dans **TypeScript** depuis la version 4.0.
- `x ||= y` : c'est-à-dire, `x` reçoit la valeur de `y` si et seulement si `x` est `falsy`.
- Les valeurs `falsy` sont `false`, `0`, `""`, `null`, `undefined`, `NaN`.

L'écriture sans simplification

```
let a: string | null = null, b: string = "travolta";

if (!a) {
    a = b;
}

console.log(a);
// affiche travolta
```

L'écriture sans simplification

```
let a: string | null = null, b: string = "travolta";

if (!a) {
    a = b;
}

console.log(a);
// affiche travolta
```

L'équivalent en ES2020

```
let a: string | null = null, b: string = "travolta";
a = a || (a = b);

console.log(a);
// affiche travolta
```

L'écriture sans simplification

```
let a: string | null = null, b: string = "travolta";

if (!a) {
    a = b;
}

console.log(a);
// affiche travolta
```

L'équivalent en ES2020

```
let a: string | null = null, b: string = "travolta";
a = a || (a = b);

console.log(a);
// affiche travolta
```

L'équivalent en ES2021 et TypeScript depuis la version 4.0

```
let a: string | null = null, b: string = "travolta";
a ||= b;

console.log(a);
// affiche travolta
```

TypeScript

Chaînage optionnel (?.)

- Introduit dans dans **TypeScript** depuis la version 3.7.
- Permettant à **TypeScript** d' arrêter immédiatement l'exécution d'une expression s'il rencontre `null` ou `undefined`.

TypeScript

L'écriture sans simplification

```
let obj = null;
let x = 2;
if (x < 0) {
    obj = { nom: "wick", prenom: "john" }
}
console.log(obj.nom);
// TypeError: Cannot read property 'nom' of null
```

TypeScript

L'écriture sans simplification

```
let obj = null;
let x = 2;
if (x < 0) {
    obj = { nom: "wick", prenom: "john" }
}
console.log(obj.nom);
// TypeError: Cannot read property 'nom' of null
```

Simplification avec TypeScript

```
let obj = null;
let x = 2;
if (x < 0) {
    obj = { nom: "wick", prenom: "john" }
}
console.log(obj?.nom);
// affiche undefined
```

TypeScript

L'écriture sans simplification

```
let obj = null;
let x = 2;
if (x < 0) {
    obj = { nom: "wick", prenom: "john" }
}
console.log(obj.nom);
// TypeError: Cannot read property 'nom' of null
```

Simplification avec TypeScript

```
let obj = null;
let x = 2;
if (x < 0) {
    obj = { nom: "wick", prenom: "john" }
}
console.log(obj?.nom);
// affiche undefined
```

Une expression contenant un chaînage optionnel ne peut être à gauche d'une affectation.

TypeScript

Les constantes

- se déclare avec le mot-clé `const`
- permet à une variable de ne pas changer de valeur

TypeScript

Les constantes

- se déclare avec le mot-clé `const`
- permet à une variable de ne pas changer de valeur

Ceci génère une erreur car une constante ne peut changer de valeur

```
const X: any = 5;
X = "bonjour";
// affiche TypeError: Assignment to constant
   variable.
```

TypeScript

Avec TypeScript 3.4, on peut définir une constante avec une assertion sans préciser le type

```
let X = "bonjour" as const;

console.log(X);
// affiche bonjour

console.log(typeof X);
// affiche string

let Y: string = "bonjour";
console.log(X == y);
//affiche true
```

TypeScript

Avec TypeScript 3.4, on peut définir une constante avec une assertion sans préciser le type

```
let X = "bonjour" as const;

console.log(X);
// affiche bonjour

console.log(typeof X);
// affiche string

let Y: string = "bonjour";
console.log(X == y);
//affiche true
```

Ceci génère une erreur car une constante ne peut changer de valeur

```
X = "hello";
```

TypeScript

Avec TypeScript 3.4, on peut aussi définir une constante ainsi

```
let X = <const>"bonjour";

console.log(X);
// affiche bonjour

console.log(typeof X);
// affiche string

let y: string = "bonjour";
console.log(X == y);
//affiche true
```

TypeScript

Avec TypeScript 3.4, on peut aussi définir une constante ainsi

```
let X = <const>"bonjour";

console.log(X);
// affiche bonjour

console.log(typeof X);
// affiche string

let y: string = "bonjour";
console.log(X == y);
//affiche true
```

Ceci génère une erreur car une constante ne peut changer de valeur

```
X = "hello";
```

TypeScript

Déclarer une fonction

```
function nomFonction([les paramètres]){
  les instructions de la fonction
}
```

TypeScript

Déclarer une fonction

```
function nomFonction([les paramètres]){
  les instructions de la fonction
}
```

Exemple

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

TypeScript

Déclarer une fonction

```
function nomFonction([les paramètres]){
  les instructions de la fonction
}
```

Exemple

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

Appeler une fonction

```
let resultat: number = somme (1, 3);
console.log(resultat);
// affiche 4
```

TypeScript

Le code suivant génère une erreur

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

TypeScript

Le code suivant génère une erreur

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

Celui-ci aussi

```
let resultat: number = somme ("1", 3);
```

TypeScript

Le code suivant génère une erreur

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

Celui-ci aussi

```
let resultat: number = somme ("1", 3);
```

Et même celui-ci

```
let resultat: string = somme(1, 3);
```

TypeScript

Une fonction qui ne retourne rien a le type `void`

```
function direBonjour(): void {
    console.log("bonjour");
}
```

TypeScript

Une fonction qui ne retourne rien a le type `void`

```
function direBonjour(): void {
    console.log("bonjour");
}
```

Une fonction qui n'atteint jamais sa fin a le type `never`

```
function boucleInfinie(): never {
    while (true){

    }
}
```

TypeScript

Il est possible d'attribuer une valeur par défaut aux paramètres d'une fonction

TypeScript

Il est possible d'attribuer une valeur par défaut aux paramètres d'une fonction

```
function division(x: number, y: number = 1) : number
    {
     return x / y;
}

console.log(division(10));
// affiche 10

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

TypeScript

Il est possible de rendre certains paramètres d'une fonction optionnels

TypeScript

Il est possible de rendre certains paramètres d'une fonction optionnels

```
function division(x: number, y?: number): number {
    if(y)
        return x / y;
    return x;
}

console.log(division(10));
// affiche 10

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

TypeScript

Il est possible de définir une fonction prenant un nombre indéfini de paramètres

TypeScript

Il est possible de définir une fonction prenant un nombre indéfini de paramètres

```
function somme(x: number, ...tab: number[]): number {
    for (let elt of tab)
        x += elt;
    return x;
}

console.log(somme(10));
// affiche 10

console.log(somme(10, 5));
// affiche 15

console.log(somme(10, 1, 6));
// affiche 17
```

TypeScript

Il est possible d'autoriser plusieurs types pour un paramètre

TypeScript

Il est possible d'autoriser plusieurs types pour un paramètre

```
function stringOrNumber(param1: string | number,
   param2: number): number {
     if (typeof param1 == "string")
         return param1.length + param2;
     return param1 + param2;
}

console.log(stringOrNumber("bonjour", 3));
// affiche 10

console.log(stringOrNumber(5, 3));
// affiche 8
```

TypeScript

Le mot-clé `ReadonlyArray` (TypeScript 3.4) indique qu'un paramètre de type tableau est en lecture seule (non-modifiable)

```
function incrementAll(tab: ReadonlyArray<number>): void {
  for (let i = 0; i < tab.length; i++){
    // la ligne suivante génère une erreur
    tab[i]++;
  }
}
```

TypeScript

Le mot-clé `ReadonlyArray` (TypeScript 3.4) indique qu'un paramètre de type tableau est en lecture seule (non-modifiable)

```
function incrementAll(tab: ReadonlyArray<number>): void {
  for (let i = 0; i < tab.length; i++){
    // la ligne suivante génère une erreur
    tab[i]++;
  }
}
```


On peut aussi utiliser le mot-clé `readonly` qui s'applique sur les tableaux et les tuples

```
function incrementAll(tab: readonly number[]): void {
  for (let i = 0; i < tab.length; i++){
    // la ligne suivante génère une erreur
    tab[i]++;
  }
}
```

TypeScript

Fonction génératrice

- déclarée avec `function*`
- utilise le mot-clé `yield` pour générer plusieurs valeurs

TypeScript

Fonction génératrice

- déclarée avec `function*`
- utilise le mot-clé `yield` pour générer plusieurs valeurs

Exemple

```
function* generateur() {
  for (let i = 0; i < 3; i++) {
    yield i;
  }
}
```

TypeScript

Lorsqu'on appelle une fonction génératrice, son corps n'est pas exécuté immédiatement, c'est un itérateur qui est renvoyé.

```
var f = generateur();
```

TypeScript

Lorsqu'on appelle une fonction génératrice, son corps n'est pas exécuté immédiatement, c'est un itérateur qui est renvoyé.

```
var f = generateur();
```

La méthode `next` de l'itérateur

- En appelant la méthode `next` de l'itérateur, la fonction génératrice est exécutée jusqu'à ce que la première expression `yield` soit trouvée.
- La méthode `next` renvoie un objet ayant deux propriétés :
  - `value` : contient la valeur générée ou `undefined` si le générateur ne produit plus de valeurs.
  - `done` : contient `true` si le générateur a produit sa dernière valeur, `false` sinon.

Exemple

```
console.log(f.next());
// affiche { value: 0, done: false }

console.log(f.next().value);
// affiche 1

console.log(f.next().value);
// affiche 2

console.log(f.next());
// affiche { value: undefined, done: true }
```

Exemple

```
console.log(f.next());
// affiche { value: 0, done: false }

console.log(f.next().value);
// affiche 1

console.log(f.next().value);
// affiche 2

console.log(f.next());
// affiche { value: undefined, done: true }
```


Avant de compiler, vérifiez dans `tsconfig.json` les propriétés suivantes

- `"target": "es6"`
- `"strictPropertyInitialization": false`

TypeScript

Exercice

- Simplifiez le code précédent en utilisant une boucle
- N'affichez que les valeurs générées par le générateur (pas le `undefined`)

TypeScript

Exercice

En utilisant `yield`, écrire une fonction génératrice **TypeScript**

- acceptant comme paramètre un entier positif `x`
- qui retourne à chaque appel un nombre premier inférieur à `x`

Solution

```
var nbr = 10;
var gen = getAllPremier(nbr);
var next = gen.next();
while (!next.done && next.value != undefined){
    console.log(next);
    next = gen.next();
}

function estPremier(n: number) {
    for (let i = 2; i < n; i++) {
        if (n % i == 0)
            return false;
    }
    return true;
}

function* getAllPremier(n: number) {
    for (let i = 1; i <= n; i++) {
        if (estPremier(i))
            yield i;
    }
}
```

TypeScript

Il est possible de déclarer une fonction en utilisant les expressions fléchées

```
let nomFonction = ([les paramètres]): typeValeurRetour => {
  les instructions de la fonction
}
```

TypeScript

Il est possible de déclarer une fonction en utilisant les expressions fléchées

```
let nomFonction = ([les paramètres]): typeValeurRetour => {
  les instructions de la fonction
}
```

Exemple

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

TypeScript

Il est possible de déclarer une fonction en utilisant les expressions fléchées

```
let nomFonction = ([les paramètres]): typeValeurRetour => {
  les instructions de la fonction
}
```

Exemple

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

Ou en plus simple

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

TypeScript

Il est possible de déclarer une fonction en utilisant les expressions fléchées

```
let nomFonction = ([les paramètres]): typeValeurRetour => {
  les instructions de la fonction
}
```

Exemple

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

Ou en plus simple

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

Appeler une fonction fléchée

```
let resultat: number = somme (1, 3);
```

TypeScript

Cas d'une fonction fléchée à un seul paramètre

```
let carre = (a: number): number => a * a;
console.log(carre(2)); // affiche 4
```

TypeScript

Cas d'une fonction fléchée à un seul paramètre

```
let carre = (a: number): number => a * a;
console.log(carre(2)); // affiche 4
```

Sans typage, la fonction peut être écrite ainsi

```
let carre = a => a * a;
console.log(carre(2)); // affiche 4
```

TypeScript

Cas d'une fonction fléchée à un seul paramètre

```
let carre = (a: number): number => a * a;
console.log(carre(2)); // affiche 4
```

Sans typage, la fonction peut être écrite ainsi

```
let carre = a => a * a;
console.log(carre(2)); // affiche 4
```


Déclaration d'une fonction fléchée sans paramètre

```
let sayHello = (): void => console.log('Hello');
sayHello(); // affiche Hello
```

TypeScript

Remarque

- Il est déconseillé d'utiliser les fonctions fléchées dans un objet
- Le mot-clé `this` est inutilisable dans les fonctions fléchées

TypeScript

Remarque

- Il est déconseillé d'utiliser les fonctions fléchées dans un objet
- Le mot-clé `this` est inutilisable dans les fonctions fléchées

Sans les fonctions fléchées

```
let obj = {
  nom: 'wick',
  afficherNom: function() {
    console.log(this.nom)
  }
}
obj.afficherNom();
// affiche wick
```

Avec les fonctions fléchées

```
let obj = {
  nom: 'wick',
  afficherNom: () => {
    console.log(this.nom)
  }
}
obj.afficherNom();
// affiche undefined
```

TypeScript

Curryfication (Currying)

- concept introduit par le mathématicien russe **Moses Schonfinkel** puis repris par le mathématicien américain **Haskell Curry**
- permettant de créer des fonctions pures
- transformant une fonction à plusieurs paramètres en une fonction à un seul paramètre retournant une fonction sur le reste des paramètres

TypeScript

Considérons la fonction fléchée `somme`

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

TypeScript

Considérons la fonction fléchée `somme`

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

Pour appeler la fonction `somme`

```
console.log(somme(2, 3, 5));
// affiche 10
```

TypeScript

Considérons la fonction fléchée `somme`

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

Pour appeler la fonction `somme`

```
console.log(somme(2, 3, 5));
// affiche 10
```

La version currifiée de la fonction `somme`

```
let sommeCur = (a: number) => (b: number) => (c: number) => a + b + c;
```

TypeScript

Considérons la fonction fléchée `somme`

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

Pour appeler la fonction `somme`

```
console.log(somme(2, 3, 5));
// affiche 10
```

La version currifiée de la fonction `somme`

```
let sommeCur = (a: number) => (b: number) => (c: number) => a + b + c;
```

Pour appeler la fonction `sommeCur`

```
console.log(sommeCur(2)(3)(5));
// affiche 10
```

TypeScript

Currifier facilite la composition des fonctions

```
const $ = x => k =>
    $(k(x))

const somme  = x => y =>
    x + y

const produit = x => y =>
    x * y

$(1)                  // 1
    (somme(2))        // + 2 = 3
    (produit(6))      // * 6 = 18
    (console.log)     // 18

$(7)                  // 7
    (somme(1))        // + 1 = 8
    (produit(8))      // * 8 = 64
    (produit(2))      // * 2 = 128
    (produit(2))      // * 2 = 256
    (console.log)     // 256
```

TypeScript

Opérateur d'assertion non-null (!)

- Introduit dans **TypeScript** depuis la version 2.0.
- Utilisé pour indiquer au compilateur **TypeScript** qu'une variable n'est pas nulle.

TypeScript

Considérons la fonction suivante

```
function incrementIfEqualToOne(i: number): number | null {
    return i == 1 ? ++i : null;
}
```

TypeScript

Considérons la fonction suivante

```
function incrementIfEqualToOne(i: number): number | null {
    return i == 1 ? ++i : null;
}
```

La variable `j` doit être de type `null | number` à cause de type de retour de la fonction

```
let j: null | number = incrementIfEqualToOne(1);
```

TypeScript

Considérons la fonction suivante

```
function incrementIfEqualToOne(i: number): number | null {
    return i == 1 ? ++i : null;
}
```

La variable `j` doit être de type `null | number` à cause de type de retour de la fonction

```
let j: null | number = incrementIfEqualToOne(1);
```


Le compilateur n'acceptera pas cette affectation même si on avait appelé la fonction `incrementIfEqualToOne` avec la valeur `1`

```
let k: number = j;
```

TypeScript

Dans ce cas, on peut utiliser l'opérateur `!` pour garantir au compilateur que la variable `k` ne peut pas être `null`

```
let k: number = j!;
console.log(k);
// affiche 2
```