

TypeScript

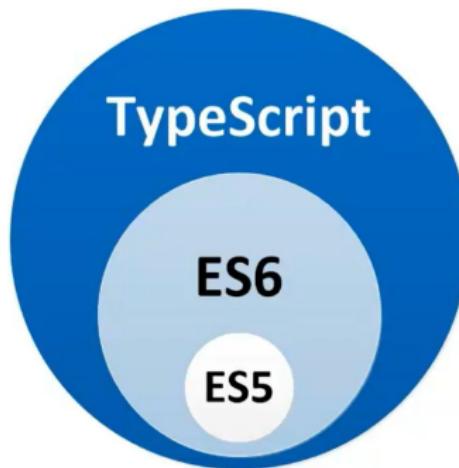
TypeScript

- TypeScript est un langage proposé par Microsoft en 2012.
- Il est open source, développé comme sur-ensemble de JS ainsi tout code valide en JS l'est également en TypeScript.
- Langage de programmation :
 - procédural et orienté-objet
 - supportant le typage statique, dynamique & générique
- TypeScript rend JS plus proche d'un langage orienté objet à typage statique comme (Java, C#,...).
- TypeScript peut introduire de nouvelles fonctionnalités de langage tout en conservant la compatibilité avec les moteurs JavaScript existants supportant ES3, ES5 et ES6

TypeScript

Principe de fonctionnement de TypeScript

- TypeScript intègre les différentes normes de JavaScript (ES5 et ES6) chose qui veut dire qu'il est possible de développer des applications TypeScript pour les vieux comme les nouveaux navigateurs



TypeScript

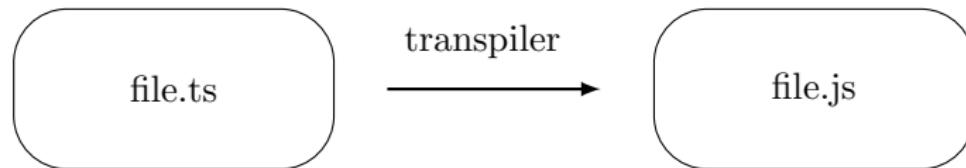
Le navigateur ne comprend pas **TypeScript**

Il faut le transcompiler (ou transpiler) en **JavaScript**

TypeScript

Le navigateur ne comprend pas **TypeScript**

Il faut le transcompiler (ou transpiler) en **JavaScript**



TypeScript

Comment va t-on procéder dans ce cours ?

file.ts

résultat

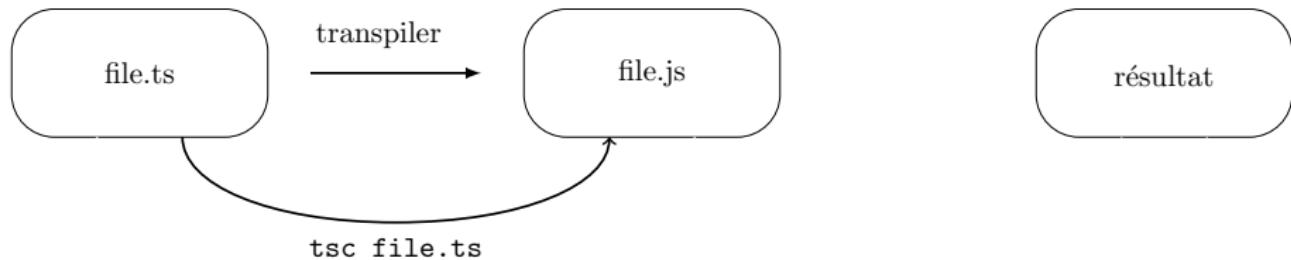
TypeScript

Comment va t-on procéder dans ce cours ?



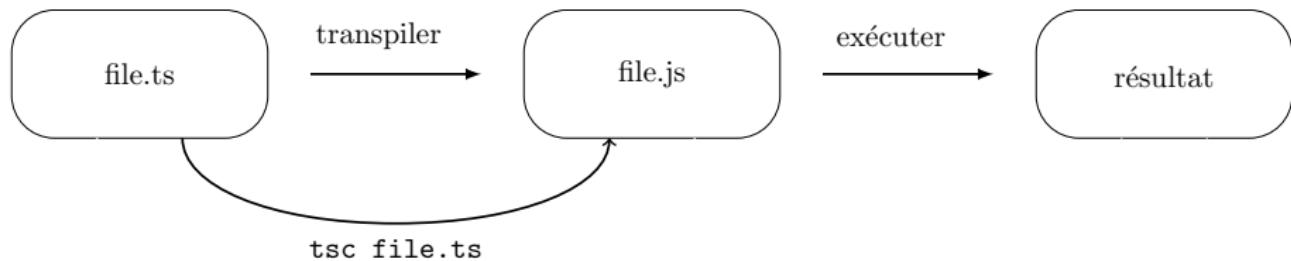
TypeScript

Comment va t-on procéder dans ce cours ?



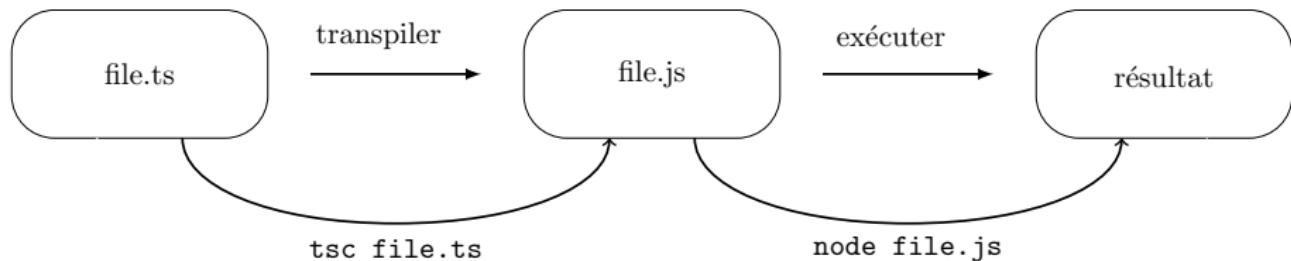
TypeScript

Comment va t-on procéder dans ce cours ?



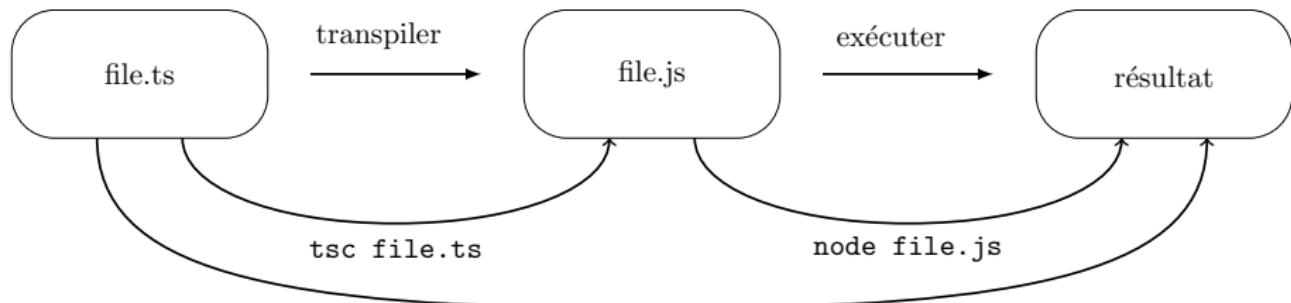
TypeScript

Comment va t-on procéder dans ce cours ?



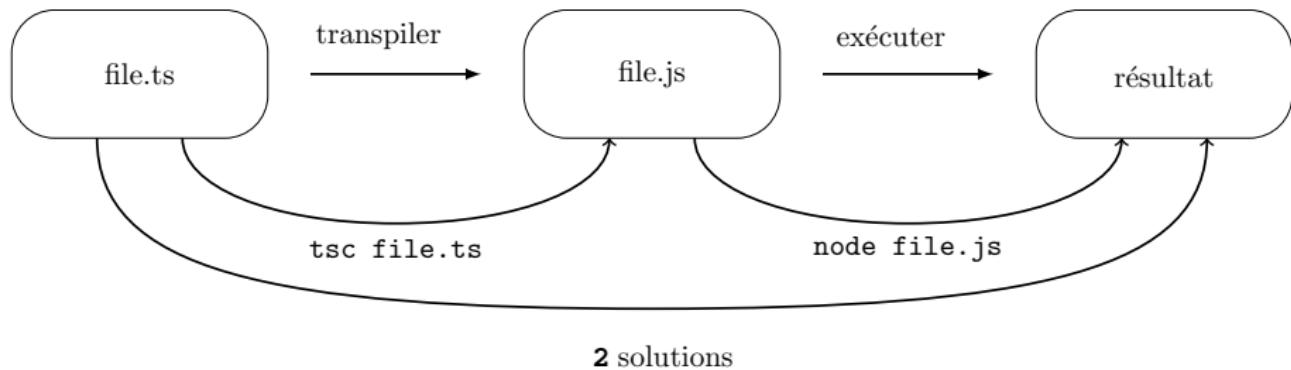
TypeScript

Comment va t-on procéder dans ce cours ?



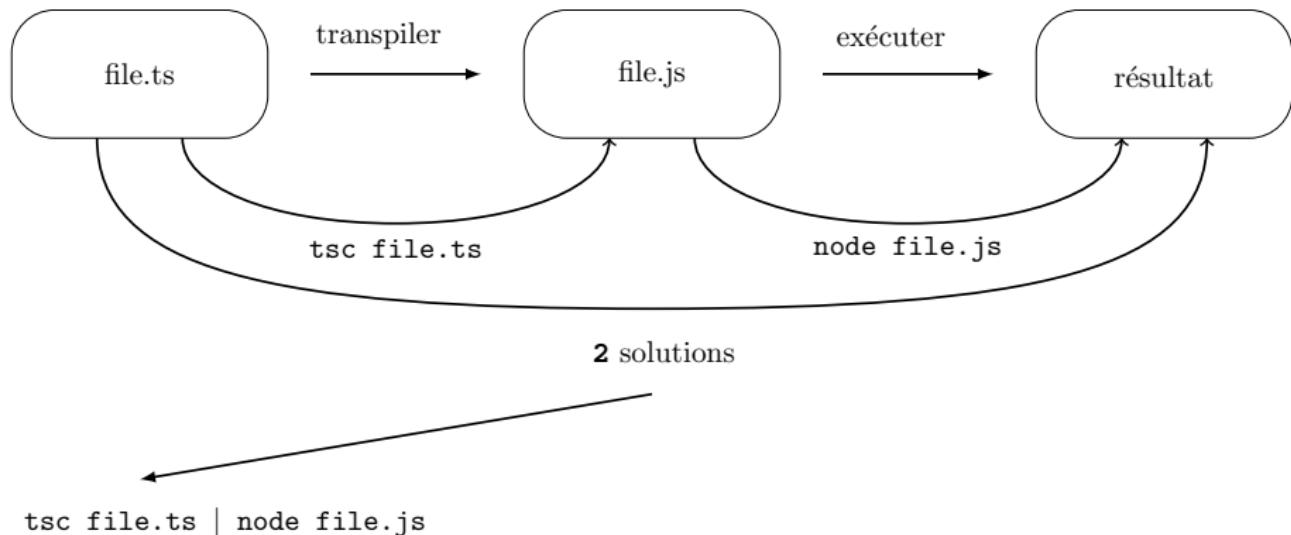
TypeScript

Comment va t-on procéder dans ce cours ?



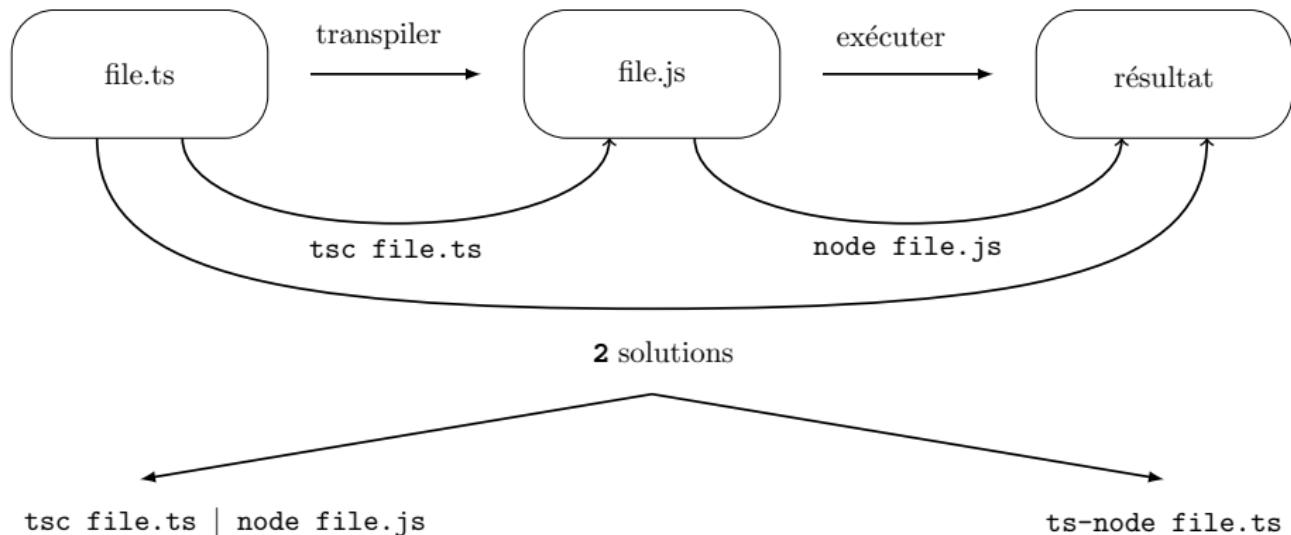
TypeScript

Comment va t-on procéder dans ce cours ?



TypeScript

Comment va t-on procéder dans ce cours ?



TypeScript

De quoi on a besoin ?

- **Node.js** pour exécuter la commande `node`
- **TypeScript** pour exécuter la commande `tsc`
- **ts-node** pour exécuter la commande `ts-node`

TypeScript

Pour **Node.js**, il faut

- aller sur <https://nodejs.org/en/>
- choisir la dernière version, télécharger et installer

TypeScript

Pour **Node.js**, il faut

- aller sur <https://nodejs.org/en/>
- choisir la dernière version, télécharger et installer

Pour vérifier l'installation depuis une console (invite de commandes), exédez

```
node -v
```

TypeScript

Pour installer TypeScript, ouvrez une console (invite de commandes) et exéutez

```
npm install -g typescript
```

TypeScript

Pour installer TypeScript, ouvrez une console (invite de commandes) et exéutez

```
npm install -g typescript
```

Pour vérifier l'installation depuis une console (invite de commandes), exéutez

```
tsc -v
```

TypeScript

Pour installer ts-node, ouvrez une console (invite de commandes) et exécutez

```
npm install -g ts-node
```

TypeScript

Pour installer ts-node, ouvrez une console (invite de commandes) et exécutez

```
npm install -g ts-node
```

Pour vérifier l'installation depuis une console (invite de commandes), exécutez

```
ts-node -v
```

TypeScript

Fichier de configuration : `tsconfig.json`

Situé à la racine du projet **TypeScript**

Consulté par le compilateur à chaque exécution de la commande `tsc`

Si le fichier n'existe pas, des valeurs par défaut seront utilisées
(<https://www.typescriptlang.org/docs/handbook/compiler-options.html>)

TypeScript

Fichier de configuration : `tsconfig.json`

Situé à la racine du projet **TypeScript**

Consulté par le compilateur à chaque exécution de la commande `tsc`

Si le fichier n'existe pas, des valeurs par défaut seront utilisées
(<https://www.typescriptlang.org/docs/handbook/compiler-options.html>)

Pour le générer, exécutez

```
tsc --init
```

TypeScript

Contenu de tsconfig.json généré (sans les commentaires)

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

TypeScript

Ajoutons l'instruction suivante dans file.ts

```
class Personne{
    _nom;
    _prenom;
    constructor(nom, prenom){
        this._nom = nom;
        this._prenom = prenom;
    }
    toString(){
        return `${this._nom} ${this._prenom}`;
    }
}
let p = new Personne("Dupont", "Jean");
console.log(p.toString());
```

TypeScript

Lancez la commande

```
tsc
```

TypeScript

Lancez la commande

```
tsc
```

Vérifions le contenu suivant dans file.js

```
var Personne = /** @class */ (function () {
    function Personne(nom, prenom) {
        this._nom = nom;
        this._prenom = prenom;
    }
    Personne.prototype.toString = function () {
        return "".concat(this._nom, " ").concat(this._prenom);
    };
    return Personne;
}());
var p = new Personne("Dupont", "Jean");
console.log(p.toString());
```

TypeScript

Modifions la valeur de la propriété target dans tsconfig.json

```
"target": "es2016"
```

TypeScript

Modifions la valeur de la propriété target dans tsconfig.json

```
"target": "es2016"
```

Relancez la commande

```
tsc
```

TypeScript

Modifions la valeur de la propriété target dans tsconfig.json

```
"target": "es2016"
```

Relancez la commande

```
tsc
```

Vérifions le contenu suivant dans file.js

```
class Personne{  
    constructor(nom, prenom){  
        this._nom = nom;  
        this._prenom = prenom;  
    }  
    toString(){  
        return `${this._nom} ${this._prenom}`;  
    }  
}  
let p = new Personne("Dupont", "Jean");  
console.log(p.toString());
```

TypeScript

Lancez la commande `tsc` en spécifiant le nom du fichier

```
tsc file.ts
```

TypeScript

Lancez la commande `tsc` en spécifiant le nom du fichier

```
tsc file.ts
```

Vérifions le contenu suivant dans `file.js`

```
var Personne = /** @class */ (function () {
    function Personne(nom, prenom) {
        this._nom = nom;
        this._prenom = prenom;
    }
    Personne.prototype.toString = function () {
        return "".concat(this._nom, " ").concat(this._prenom);
    };
    return Personne;
}());
var p = new Personne("Dupont", "Jean");
console.log(p.toString());
```

TypeScript

Lancez la commande `tsc` en spécifiant le nom du fichier

```
tsc file.ts
```

Vérifions le contenu suivant dans `file.js`

```
var Personne = /** @class */ (function () {
    function Personne(nom, prenom) {
        this._nom = nom;
        this._prenom = prenom;
    }
    Personne.prototype.toString = function () {
        return "".concat(this._nom, " ").concat(this._prenom);
    };
    return Personne;
}());
var p = new Personne("Dupont", "Jean");
console.log(p.toString());
```

Si on précise un nom de fichier après la commande `tsc`, le fichier `tsconfig.json` sera ignore.

TypeScript

Pour transpiler le code TypeScript en ES2016, exéutez

```
tsc file.ts --target es2016
```

TypeScript

Pour transpiler le code TypeScript en ES2016, exéutez

```
tsc file.ts --target es2016
```

Ou

```
tsc file.ts -t es2016
```

TypeScript

Créons un deuxième fichier file2.ts avec le contenu suivant

```
console.log("Hello World!");
```

TypeScript

Créons un deuxième fichier file2.ts avec le contenu suivant

```
console.log("Hello World!");
```

Lancez la commande

```
tsc
```

TypeScript

Créons un deuxième fichier file2.ts avec le contenu suivant

```
console.log("Hello World!");
```

Lancez la commande

```
tsc
```

Remarque

Vérifier que les deux fichiers file.ts et file2.ts ont été transpilé en file.js et file2.js.

Pour ne pas transpiler file2.ts, ajoutons la clé exclude dans tsconfig.json

```
{  
  "exclude": ["file2.ts"],  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

Pour ne pas transpiler file2.ts, ajoutons la clé exclude dans tsconfig.json

```
{  
  "exclude": ["file2.ts"],  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

Supprimez les fichiers .js avant de lancer

```
tsc
```

Pour ne pas transpiler file2.ts, ajoutons la clé exclude dans tsconfig.json

```
{  
  "exclude": ["file2.ts"],  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

Supprimez les fichiers .js avant de lancer

```
tsc
```

Remarque

Vérifier que seul le fichier file.ts a été transpilé en file.js.

Pour transpiler seulement file2.ts, ajoutons la clé include dans tsconfig.json

```
{  
  "include": ["file2.ts"],  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

Pour transpiler seulement file2.ts, ajoutons la clé include dans tsconfig.json

```
{  
  "include": ["file2.ts"],  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

Supprimez les fichiers .js avant de lancer

```
tsc
```

Pour transpiler seulement file2.ts, ajoutons la clé include dans tsconfig.json

```
{  
  "include": ["file2.ts"],  
  "compilerOptions": {  
    "target": "es5",  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "compileOnSave": true  
  }  
}
```

Supprimez les fichiers .js avant de lancer

```
tsc
```

Remarque

Vérifier que seul le fichier file2.ts a été transpilé en file2.js.

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 paramétriser un type complexe

TypeScript

Déclarer une variable

```
var nomVariable = valeurInitiale;
```

TypeScript

Déclarer une variable

```
var nomVariable = valeurInitiale;
```

Exemple

```
var x = 2
```

TypeScript

Déclarer une variable

```
var nomVariable = valeurInitiale;
```

Exemple

```
var x = 2
```

JavaScript détermine dynamiquement le type de la variable

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

x = "Bonjour";
// Ceci genere une erreur
```

TypeScript

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

```
console.log(x.toUpperCase());
```

TypeScript

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

```
console.log(x.toUpperCase());
```

Remarques

Avec le typage statique (en utilisant un IDE), cette erreur sera détectée avant la transpilation

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 variable 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

`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

`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

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;
```

Déclarons une énumération (dans file.ts)

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

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
```

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 modifier l'indice du premier élément

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

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 modifier l'indice 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

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

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);
```

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

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 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

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

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

```
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

```
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

```
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"
```

TypeScript

Pour simplifier le code précédent en utilisant une boucle

```
//Premiere version
for(let e of f) {
    console.log(e); }

//Deuxieme version
while (true) {
    const value = f.next();
    if (value.done) {
        break;
    }
    console.log(value.value);
}
```

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

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();
//The containing arrow
function captures the global
value of 'this'.ts(7041)
```

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

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

On peut utiliser ? rendre certains éléments de tuple optionnels

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

TypeScript

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

TypeScript

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

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

On peut aussi utiliser les ... pour le restant d'éléments de même type

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

TypeScript

On peut aussi utiliser les ... pour le restant d'éléments de même type

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

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 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 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

```
var list: number[] = Array.from(Array(3).keys())
console.log(list);
// affiche [ 0, 1, 2 ]
```

TypeScript

Exemple

```
var list: number[] = Array.from(Array(3).keys())
console.log(list);
// affiche [ 0, 1, 2 ]
```

On peut utiliser la méthode `from` pour créer un tableau à partir d'un itérable

```
console.log(Array.from('wick'));
// affiche [ 'w', 'i', 'c', 'k' ]
```

TypeScript

Exemple

```
var list: number[] = Array.from(Array(3).keys())
console.log(list);
// affiche [ 0, 1, 2 ]
```

On peut utiliser la méthode `from` pour créer un tableau à partir d'un itérable

```
console.log(Array.from('wick'));
// affiche [ 'w', 'i', 'c', 'k' ]
```

On peut aussi utiliser la méthode `from` pour créer un tableau à partir d'un autre tableau

```
console.log(Array.from([1, 2, 3], elt => elt * 2));
// affiche [ 2, 4, 6 ]
```

TypeScript

On peut utiliser la méthode `of` pour créer un tableau à partir d'une liste de paramètres

```
console.log(Array.of(1, 2, 3));
// affiche [1, 2, 3]
```

TypeScript

On peut utiliser la méthode `of` pour créer un tableau à partir d'une liste de paramètres

```
console.log(Array.of(1, 2, 3));
// affiche [1, 2, 3]
```

L'écriture précédente peut être simplifiée

```
console.log(Array(1, 2, 3));
// affiche [1, 2, 3]
```

TypeScript

Les deux écritures suivantes sont différentes

```
console.log(Array.of(7));  
// affiche [7]
```

```
console.log(Array(7));  
// affiche [ <7 empty items> ]
```

TypeScript

La méthode `fill` permet de remplir les éléments d'un tableau entre deux index avec une valeur statique

```
let tab = Array(7);
tab.fill(2);
console.log(tab);
// affiche [ 2, 2, 2, 2, 2, 2, 2 ]  
  
tab.fill(0, 3);
console.log(tab);
// affiche [ 2, 2, 2, 0, 0, 0, 0 ]  
  
tab.fill(5, 2, 4);
console.log(tab);
// affiche [ 2, 2, 5, 5, 0, 0, 0 ]
```

TypeScript

Les fonctions fléchées sont utilisées pour réaliser les opérations suivant sur les tableaux

- `forEach()` : pour parcourir un tableau
- `map()` : pour appliquer une fonction sur les éléments d'un tableau
- `filter()` : pour filtrer les éléments d'un tableau selon un critère défini sous forme d'une fonction anonyme ou fléchée
- `reduce()` : pour réduire tous les éléments d'un tableau en un seul selon une règle définie dans une fonction anonyme ou fléchée
- `some()` : pour vérifier s'il existe au moins un élément qui respect une condition
- `every()` : pour vérifier si tous les éléments respectent une condition
- ...

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

Dans `forEach`, on peut aussi appeler une fonction `afficher`

```
tab.forEach(elt => afficher(elt));

function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach_elt => console.log_elt);
// affiche 2 3 5
```

Dans `forEach`, on peut aussi appeler une fonction `afficher`

```
tab.forEach_elt => afficher_elt);

function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

On peut simplifier l'écriture précédente en utilisant les callback

```
tab.forEach(afficher);

function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

TypeScript

La fonction `afficher` peut accepter deux paramètres : le premier est la valeur de l'itération courante et le deuxième est son indice dans le tableau

```
tab.forEach(afficher);  
  
function afficher(value: number, key: number) {  
    console.log(key, value);  
}  
  
/* affiche  
0 2  
1 3  
2 5  
*/
```

TypeScript

La fonction `afficher` peut accepter un troisième paramètre qui correspond au tableau

```
tab.forEach(afficher);
```

```
function afficher(value: number, key: number, t:  
    Array<number>) {  
    console.log(key, value, t);  
}
```

```
/* affiche  
0 2 [ 2, 3, 5 ]  
1 3 [ 2, 3, 5 ]  
2 5 [ 2, 3, 5 ]  
*/
```

TypeScript

On peut utiliser `map` pour effectuer un traitement sur chaque élément du tableau puis `forEach` pour afficher le nouveau tableau

```
tab.map(elt => elt + 3)
    .forEach(elt => console.log(elt));
// affiche 5 6 8
```

TypeScript

On peut aussi utiliser `filter` pour filtrer des éléments

```
tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .forEach(elt => console.log(elt));
// affiche 6 8
```

TypeScript

Remarque

Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer.

TypeScript

Remarque

Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer.

Exemple avec `reduce` : permet de réduire les éléments d'un tableau en une seule valeur

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) => sum + elt);

console.log(somme);
// affiche 14
```

TypeScript

Remarques

- Le premier paramètre de `reduce` correspond au résultat de l'itération précédente
- Le deuxième correspond à l'élément du tableau de l'itération courante
- Le premier paramètre est initialisé par la valeur du premier élément du tableau
- On peut changer la valeur initiale du premier paramètre en l'ajoutant à la fin de la méthode

TypeScript

Si on a plusieurs instructions, on doit ajouter les accolades

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) => {
        return sum + elt;
});
console.log(somme);
// affiche 14
```

TypeScript

Dans cet exemple, on initialise le premier paramètre de `reduce` par la valeur 0

```
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) => sum + elt, 0);

console.log(somme);
// affiche 14
```

TypeScript

Dans cet exemple, on vérifie s'il existe un élément pair dans le tableau, après modification

```
var tab = [2, 3, 5];

let result = tab.map(elt => elt + 3)
    .some(elt => elt % 2 == 0);

console.log(result);
// affiche true
```

TypeScript

Dans cet exemple, on vérifie si tous les éléments du tableau sont pairs, après modification

```
var tab = [2, 3, 5];

let result = tab.map(elt => elt + 3)
    .every(elt => elt % 2 == 0);

console.log(result);
// affiche false
```

TypeScript

Map (dictionnaire)

- Type fonctionnant avec un couple (clé,valeur)
- La clé doit être unique
- Chaque élément est appelé entrée (entry)
- Les éléments sont stockés et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

TypeScript

Pour créer un Map

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

TypeScript

Pour créer un Map

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

Ajouter un élément à un Map

```
map.set('html', 18);
```

TypeScript

Pour créer un Map

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

Ajouter un élément à un Map

```
map.set('html', 18);
```

Pour ajouter plusieurs éléments à la fois

```
map.set('html', 18)
    .set('css', 12);
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

Pour vérifier l'existence d'une clé

```
console.log(map.has('php'));
// affiche true
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

Pour vérifier l'existence d'une clé

```
console.log(map.has('php'));
// affiche true
```

Pour supprimer un élément selon la clé

```
map.delete('php');
```

TypeScript

Pour récupérer la liste des clés d'un Map

```
console.log(map.keys());  
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

TypeScript

Pour récupérer la liste des clés d'un Map

```
console.log(map.keys());  
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

Pour récupérer la liste des valeurs d'un Map

```
console.log(map.values());  
// affiche [Map Iterator] { 10, 12, 20, 12 }
```

TypeScript

Pour récupérer la liste des clés d'un Map

```
console.log(map.keys());  
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

Pour récupérer la liste des valeurs d'un Map

```
console.log(map.values());  
// affiche [Map Iterator] { 10, 12, 20, 12 }
```

Pour récupérer la liste des entrées d'un Map

```
console.log(map.entries());  
/* affiche  
[Map Entries] {  
  [ 'java', 10 ],  
  [ 'c', 12 ],  
  [ 'html', 20 ],  
  [ 'css', 12 ]  
}
```

TypeScript

Pour parcourir un Map, on peut utiliser entries() (solution ES5)

```
for (let elt of map.entries())
    console.log(elt[0] + " " + elt[1]);
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

Pour parcourir un Map, on peut utiliser entries()

```
for (let elt of map.entries())
    console.log(elt[0] + " " + elt[1]);
/* affiche
java 10
c 12
html 20
css 12
*/
```

Ou

```
for (let [key, value] of map) {
    console.log(key, value);
}
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

On peut le faire aussi avec keys()

```
for (let key of map.keys()) {  
    console.log(key + " " + map.get(key));  
}  
/* affiche  
java 10  
c 12  
html 20  
css 12  
*/
```

TypeScript

Une deuxième solution consiste à utiliser `forEach` (**affiche seulement les valeurs**)

```
map.forEach(elt => console.log(elt));
```

```
/* affiche  
10  
12  
20  
12  
*/
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach` (**affiche seulement les valeurs**)

```
map.forEach(elt => afficher(elt));  
  
function afficher(elt: number) {  
    console.log(elt)  
}
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach` (**affiche seulement les valeurs**)

```
map.forEach(elt => afficher(elt));  
  
function afficher(elt: number) {  
    console.log(elt)  
}
```

On peut encore simplifier l'appel de la fonction avec les callback
(affiche seulement les valeurs)

```
map.forEach(afficher);  
  
function afficher(elt: number) {  
    console.log(elt)  
}
```

TypeScript

Pour afficher les clés et les valeurs

```
map.forEach(afficher);  
  
function afficher (value: number, key: string) {  
    console.log(value, key)  
}  
  
/* affiche  
10 java  
12 c  
20 html  
12 css  
*/
```

TypeScript

Ou aussi

```
map.forEach((v: number, k: string) => console.log(k, v));  
  
/* affiche  
10 java  
12 c  
20 html  
12 css  
*/
```

TypeScript

Set

- Une collection ne contenant pas de doublons
- Acceptant les types simples et objets
- Les éléments sont stockées et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

TypeScript

Pour créer un Set

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

TypeScript

Pour créer un Set

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

Ajouter un élément à un Set

```
marques.add('citroen');
```

TypeScript

Pour créer un Set

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

Ajouter un élément à un Set

```
marques.add('citroen');
```

Pour ajouter plusieurs éléments à la fois

```
marques.add('citroen')
    .add('renault');
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', 
  'citroen', 'renault' }
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', 'citroen', 'renault' }
```

Pour vérifier l'existence d'un élément

```
console.log(marques.has('fiat'));
// affiche true
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', 'citroen', 'renault' }
```

Pour vérifier l'existence d'un élément

```
console.log(marques.has('fiat'));
// affiche true
```

Pour supprimer un élément

```
marques.delete('ford');
console.log(marques);
// affiche Set { 'peugeot', 'fiat', 'mercedes', 'citroen', 'renault' }
```

TypeScript

Autres méthodes sur les Set

- A.subSet (B) : retourne true si A est un sous-ensemble de B, false sinon.
- A.union (B) : retourne un **Set** regroupant les éléments de A et de B.
- A.intersection (B) : retourne un **Set** contenant les éléments de A qui sont dans B.
- A.difference (B) : retourne un **Set** contenant les éléments de A qui ne sont pas dans B.

TypeScript

Pour parcourir un Set

```
for(let marque of marques) {  
    console.log(marque)  
}  
/* affiche  
peugeot  
fiat  
mercedes  
citroen  
renault  
*/
```

TypeScript

Une deuxième solution consiste à utiliser `forEach`

```
marques.forEach(elt => console.log(elt));
```

```
/* affiche
peugeot
fiat
mercedes
citroen
renault
*/
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach`

```
marques.forEach(elt => afficher(elt));  
  
function afficher(elt: string) {  
    console.log(elt)  
}
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach`

```
marques.forEach(elt => afficher(elt));  
  
function afficher(elt: string) {  
    console.log(elt)  
}
```

On peut encore simplifier l'appel de la fonction avec les callback

```
marques.forEach(afficher);  
  
function afficher(elt: string) {  
    console.log(elt)  
}
```