Formation Vue.js

Chapitre 3

TypeScript

Ibrahim ALAME

2025

Objectifs

Ce chapitre couvre les fondamentaux de TypeScript :

- Comprendre l'intérêt du typage statique
- Maîtriser les types de base et avancés
- Typer les fonctions et leurs paramètres
- Utiliser les classes avec les modificateurs d'accès
- Créer et utiliser des interfaces
- Travailler avec les types génériques
- Comprendre les types utilitaires et conditionnels

Prérequis : Chapitre 2 - JavaScript
Durée estimée : 10 à 12 heures

Contents

1 Introduction

1.1 Qu'est-ce que TypeScript ?

Définition

TypeScript est un sur-ensemble typé de JavaScript développé par Microsoft. Il ajoute un système de types statiques qui permet de détecter les erreurs à la compilation plutôt qu'à l'exécution.

1.1.1 Pourquoi TypeScript ?

- **Détection d'erreurs précoce** : les erreurs de type sont détectées avant l'exécution
- **Meilleure documentation** : les types servent de documentation vivante
- **Autocomplétion intelligente** : l'IDE connaît les propriétés et méthodes disponibles
- **Refactoring sécurisé** : les modifications sont vérifiées dans tout le projet
- **Code plus maintenable** : les types clarifient les intentions du développeur

1.1.2 Comment ça fonctionne ?

TypeScript est **transpilé** en JavaScript. Le navigateur n'exécute jamais de TypeScript directement.

```
fichier.ts  -->  Compilateur TypeScript (tsc)  -->  fichier.js
```

Listing 1: TypeScript vs JavaScript

```typescript
// TypeScript (fichier.ts)
function saluer(nom: string): string {
  return `Bonjour ${nom}`;
}

saluer('Alice');    // OK
saluer(42);         // Erreur a la compilation!

// JavaScript genere (fichier.js)
function saluer(nom) {
  return `Bonjour ${nom}`;
}
```

1.2 Installation et configuration

1.2.1 Installation

```
# Installation globale
npm install -g typescript
```

```
# Installation locale (recommandee)
npm install --save-dev typescript

# Verifier la version
tsc --version
```

1.2.2 Compilation

```
# Compiler un fichier
tsc fichier.ts

# Mode watch (recompile automatiquement)
tsc fichier.ts --watch

# Compiler tout le projet
tsc
```

1.2.3 Configuration : tsconfig.json

Listing 2: Exemple de tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "declaration": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Astuce

Utilisez `tsc -init` pour générer un fichier `tsconfig.json` avec toutes les options commentées.

2 Les types de base

2.1 Types primitifs

```typescript
// String
let nom: string = 'Alice';
let message: string = `Bonjour ${nom}`;

// Number (entier ou decimal)
let age: number = 25;
let prix: number = 19.99;
let hex: number = 0xff;

// Boolean
let estActif: boolean = true;
let estConnecte: boolean = false;

// Null et Undefined
let valeurNulle: null = null;
let nonDefini: undefined = undefined;

// Symbol
let id: symbol = Symbol('id');

// BigInt
let grandNombre: bigint = 9007199254740991n;
```

2.2 Le type any

```typescript
// any : desactive la verification de type
let variable: any = 'texte';
variable = 42;          // OK
variable = true;        // OK
variable.methode();     // OK (pas de verification)

// Utile pour la migration progressive de JavaScript
// Mais a eviter autant que possible!
```

Important

Le type any désactive tous les avantages de TypeScript. Utilisez-le uniquement en dernier recours, par exemple lors de la migration d'un projet JavaScript.

2.3 Le type unknown

```typescript
// unknown : type sur pour les valeurs inconnues
let valeur: unknown = 'texte';
valeur = 42;            // OK
```

```
4
5  // Mais impossible de l'utiliser directement
6  // let longueur = valeur.length;  // Erreur!
7
8  // Il faut verifier le type d'abord
9  if (typeof valeur === 'string') {
10   console.log(valeur.length);  // OK apres verification
11 }
12
13 // Ou utiliser une assertion de type
14 let str = valeur as string;
15 console.log(str.length);
```

À retenir

any vs unknown :

- `any` : tout est permis (dangereux)
- `unknown` : rien n'est permis sans vérification (sûr)

Préférez `unknown` quand vous ne connaissez pas le type.

2.4 Les types void, never et object

```
1  // void : absence de valeur de retour
2  function afficher(message: string): void {
3    console.log(message);
4    // pas de return
5  }
6
7  // never : fonction qui ne retourne jamais
8  function erreur(message: string): never {
9    throw new Error(message);
10 }
11
12 function boucleInfinie(): never {
13   while (true) {
14     // ...
15   }
16 }
17
18 // object : tout ce qui n'est pas primitif
19 let obj: object = { nom: 'Alice' };
20 let arr: object = [1, 2, 3];
21 let fn: object = () => {};
22
23 // Attention : object est tres permissif
24 // obj.nom;  // Erreur! 'nom' n'existe pas sur 'object'
```

2.5 Les tableaux (Array)

```typescript
// Deux syntaxes equivalentes
let nombres: number[] = [1, 2, 3, 4, 5];
let noms: Array<string> = ['Alice', 'Bob'];

// Tableau en lecture seule
let liste: readonly number[] = [1, 2, 3];
// liste.push(4);  // Erreur!

// Tableau de types mixtes
let mixte: (string | number)[] = ['Alice', 25, 'Bob', 30];

// Tableau multidimensionnel
let matrice: number[][] = [
  [1, 2, 3],
  [4, 5, 6]
];
```

2.6 Les tuples

Définition

Un **tuple** est un tableau de taille fixe où chaque élément a un type spécifique.

```typescript
// Tuple simple
let personne: [string, number] = ['Alice', 25];

// Acces par index
let nom = personne[0];    // string
let age = personne[1];    // number

// Erreurs de type
// personne[0] = 42;      // Erreur! string attendu
// personne[2] = 'test'; // Erreur! index hors limites

// Tuple nomme (TypeScript 4.0+)
type Coordonnees = [x: number, y: number, z?: number];
let point2D: Coordonnees = [10, 20];
let point3D: Coordonnees = [10, 20, 30];

// Tuple avec rest
type StringEtNombres = [string, ...number[]];
let data: StringEtNombres = ['valeurs', 1, 2, 3, 4];

// Destructuring
let [prenom, annees] = personne;
```

2.7 Les énumérations (enum)

```typescript
// Enum numerique (valeurs auto-incrementees)
enum Direction {
  Haut,       // 0
  Bas,        // 1
  Gauche,     // 2
  Droite      // 3
}

let dir: Direction = Direction.Haut;
console.log(dir);               // 0
console.log(Direction[0]);      // 'Haut' (reverse mapping)

// Enum avec valeurs personnalisees
enum StatusCode {
  OK = 200,
  NotFound = 404,
  ServerError = 500
}

// Enum de chaines
enum Couleur {
  Rouge = 'ROUGE',
  Vert = 'VERT',
  Bleu = 'BLEU'
}

let maCouleur: Couleur = Couleur.Rouge;

// const enum (inline au lieu de generer du code)
const enum Taille {
  Petit = 'S',
  Moyen = 'M',
  Grand = 'L'
}
let t = Taille.Moyen;  // Compile en: let t = 'M';
```

Astuce

Les `const enum` produisent du code JavaScript plus léger car ils sont remplacés par leurs valeurs à la compilation.

2.8 Inférence de types

```typescript
// TypeScript infere automatiquement les types
let nom = 'Alice';          // string (infere)
let age = 25;               // number (infere)
let actif = true;           // boolean (infere)

```

```typescript
// L'inference fonctionne aussi pour les structures complexes
let personne = {
  nom: 'Alice',
  age: 25
};
// Type infere: { nom: string; age: number }

// Et pour les fonctions
function ajouter(a: number, b: number) {
  return a + b;  // Type de retour infere: number
}

// Et pour les tableaux
let nombres = [1, 2, 3];  // number[]
let mixte = [1, 'deux'];  // (string | number)[]
```

À retenir

Quand annoter explicitement ?

- Paramètres de fonctions : **toujours**
- Variables initialisées : rarement nécessaire
- Retour de fonction : optionnel mais recommandé pour la documentation
- Cas ambigus ou complexes : oui

2.9 Assertions de types

```typescript
// Assertion avec 'as' (recommande)
let valeur: unknown = 'Hello World';
let longueur: number = (valeur as string).length;

// Assertion avec angle brackets (eviter en JSX/TSX)
let longueur2: number = (<string>valeur).length;

// Assertion sur des elements DOM
let input = document.getElementById('email') as HTMLInputElement;
input.value = 'test@example.com';

// Double assertion (rarement necessaire)
let x = 'hello' as unknown as number;

// Assertion non-null (!)
function traiter(valeur: string | null) {
  // On affirme que valeur n'est pas null
  console.log(valeur!.toUpperCase());
}
```

Important

Les assertions de type ne font **aucune vérification à l'exécution**. Utilisez-les avec précaution car elles peuvent masquer des erreurs.

3 TypeScript et les fonctions

3.1 Typer les arguments et la valeur de retour

```typescript
// Fonction avec types explicites
function additionner(a: number, b: number): number {
  return a + b;
}

// Fonction flechee
const multiplier = (a: number, b: number): number => a * b;

// Type de retour infere (mais explicite recommande)
function saluer(nom: string) {
  return `Bonjour ${nom}`;  // Retour infere: string
}

// Fonction sans retour
function afficher(message: string): void {
  console.log(message);
}

// Fonction qui lance une erreur
function erreur(message: string): never {
  throw new Error(message);
}
```

3.2 Typer une fonction avant sa déclaration

```typescript
// Type de fonction
type OperationMath = (a: number, b: number) => number;

// Utilisation
const addition: OperationMath = (a, b) => a + b;
const soustraction: OperationMath = (a, b) => a - b;

// Avec interface
interface Comparateur {
  (a: number, b: number): number;
}

const comparer: Comparateur = (a, b) => a - b;

// Type de callback
type Callback = (erreur: Error | null, resultat?: string) => void
  ;

function traiterAsync(callback: Callback): void {
  // ...
  callback(null, 'succes');
```

```
}
```

3.3 Paramètres optionnels et par défaut

```
// Parametre optionnel (?)
function saluer(nom: string, titre?: string): string {
  if (titre) {
    return `Bonjour ${titre} ${nom}`;
  }
  return `Bonjour ${nom}`;
}

saluer('Dupont');            // 'Bonjour Dupont'
saluer('Dupont', 'M.');      // 'Bonjour M. Dupont'

// Parametre par defaut
function creerUtilisateur(
  nom: string,
  role: string = 'utilisateur',
  actif: boolean = true
): object {
  return { nom, role, actif };
}

creerUtilisateur('Alice');
creerUtilisateur('Bob', 'admin');
creerUtilisateur('Charlie', 'moderateur', false);

// Combinaison optionnel et defaut
function configurer(
  options?: { debug?: boolean; timeout?: number }
): void {
  const config = {
    debug: options?.debug ?? false,
    timeout: options?.timeout ?? 5000
  };
  console.log(config);
}
```

3.4 L'opérateur rest

```
// Rest parameter (doit etre le dernier)
function somme(...nombres: number[]): number {
  return nombres.reduce((acc, n) => acc + n, 0);
}

somme(1, 2, 3);            // 6
somme(1, 2, 3, 4, 5);      // 15

```

```typescript
// Avec d'autres parametres
function afficherInfos(
  titre: string,
  ...elements: string[]
): void {
  console.log(titre);
  elements.forEach(e => console.log(`- ${e}`));
}

afficherInfos('Liste:', 'A', 'B', 'C');

// Tuple avec rest
function creerTuple(
  premier: string,
  ...reste: number[]
): [string, ...number[]] {
  return [premier, ...reste];
}
```

3.5 Surcharge de fonction

```typescript
// Signatures de surcharge
function formater(valeur: string): string;
function formater(valeur: number): string;
function formater(valeur: Date): string;

// Implementation (doit gerer tous les cas)
function formater(valeur: string | number | Date): string {
  if (typeof valeur === 'string') {
    return valeur.toUpperCase();
  }
  if (typeof valeur === 'number') {
    return valeur.toFixed(2);
  }
  return valeur.toISOString();
}

formater('hello');              // 'HELLO'
formater(3.14159);              // '3.14'
formater(new Date());           // '2025-...'

// Surcharge avec retours differents
function chercher(id: number): Utilisateur;
function chercher(email: string): Utilisateur;
function chercher(critere: number | string): Utilisateur {
  // Implementation...
  return {} as Utilisateur;
}
```

4 TypeScript et les classes

4.1 Généralités sur les classes

```typescript
class Personne {
  // Proprietes avec types
  nom: string;
  age: number;

  // Constructeur
  constructor(nom: string, age: number) {
    this.nom = nom;
    this.age = age;
  }

  // Methode avec type de retour
  sePresenter(): string {
    return `Je suis ${this.nom}, ${this.age} ans`;
  }

  // Getter
  get estMajeur(): boolean {
    return this.age >= 18;
  }

  // Setter
  set nouveauNom(nom: string) {
    if (nom.length > 0) {
      this.nom = nom;
    }
  }
}

const alice = new Personne('Alice', 25);
console.log(alice.sePresenter());
console.log(alice.estMajeur);  // true
```

4.2 Modificateurs d'accès

```typescript
class CompteBancaire {
  // public : accessible partout (par defaut)
  public titulaire: string;

  // private : accessible uniquement dans la classe
  private solde: number;

  // protected : accessible dans la classe et les sous-classes
  protected numeroCompte: string;

  // readonly : ne peut pas etre modifie apres l'initialisation
```

```
  readonly dateCreation: Date;

  constructor(titulaire: string, soldeInitial: number) {
    this.titulaire = titulaire;
    this.solde = soldeInitial;
    this.numeroCompte = this.genererNumero();
    this.dateCreation = new Date();
  }

  private genererNumero(): string {
    return 'FR' + Math.random().toString(36).substr(2, 9);
  }

  public deposer(montant: number): void {
    if (montant > 0) {
      this.solde += montant;
    }
  }

  public getSolde(): number {
    return this.solde;
  }
}

const compte = new CompteBancaire('Alice', 1000);
compte.titulaire;       // OK (public)
// compte.solde;        // Erreur! (private)
// compte.numeroCompte; // Erreur! (protected)
// compte.dateCreation = new Date();  // Erreur! (readonly)
```

4.3 Notation raccourcie du constructeur

```
// Version longue
class PersonneLongue {
  nom: string;
  age: number;

  constructor(nom: string, age: number) {
    this.nom = nom;
    this.age = age;
  }
}

// Version raccourcie (equivalente)
class PersonneCourte {
  constructor(
    public nom: string,
    public age: number,
    private email?: string,
    readonly id: string = crypto.randomUUID()
```

```
  ) {}
  // Les proprietes sont automatiquement declarees et assignees
}

const p = new PersonneCourte('Alice', 25);
console.log(p.nom, p.age, p.id);
```

4.4 Propriétés statiques

```
class Configuration {
  // Propriete statique
  static version: string = '1.0.0';
  static instances: number = 0;

  // Propriete statique privee
  private static config: Map<string, string> = new Map();

  // Methode statique
  static set(cle: string, valeur: string): void {
    Configuration.config.set(cle, valeur);
  }

  static get(cle: string): string | undefined {
    return Configuration.config.get(cle);
  }

  constructor() {
    Configuration.instances++;
  }
}

// Acces sans instanciation
console.log(Configuration.version);
Configuration.set('theme', 'dark');
console.log(Configuration.get('theme'));

new Configuration();
new Configuration();
console.log(Configuration.instances);  // 2
```

4.5 Classes abstraites

```
// Classe abstraite : ne peut pas etre instanciee directement
abstract class Forme {
  constructor(public nom: string) {}

  // Methode abstraite : doit etre implementee
  abstract calculerAire(): number;

```

```
  // Methode concrete : peut etre heritee
  decrire(): string {
    return `Forme: ${this.nom}, Aire: ${this.calculerAire()}`;
  }
}

class Rectangle extends Forme {
  constructor(
    public largeur: number,
    public hauteur: number
  ) {
    super('Rectangle');
  }

  // Implementation obligatoire
  calculerAire(): number {
    return this.largeur * this.hauteur;
  }
}

class Cercle extends Forme {
  constructor(public rayon: number) {
    super('Cercle');
  }

  calculerAire(): number {
    return Math.PI * this.rayon ** 2;
  }
}

// const forme = new Forme('test');  // Erreur! Classe abstraite
const rect = new Rectangle(10, 5);
console.log(rect.decrire());  // "Forme: Rectangle, Aire: 50"
```

4.6 Héritage et implements

```
// Heritage avec extends
class Animal {
  constructor(public nom: string) {}

  parler(): void {
    console.log(`${this.nom} fait un bruit`);
  }
}

class Chien extends Animal {
  constructor(nom: string, public race: string) {
    super(nom);  // Appel du constructeur parent
  }

```

```
15    // Surcharge de methode
16    parler(): void {
17      console.log(`${this.nom} aboie`);
18    }
19
20    // Nouvelle methode
21    courir(): void {
22      console.log(`${this.nom} court`);
23    }
24  }
25
26  // Implementation d'interface
27  interface Nageur {
28    nager(): void;
29  }
30
31  class Canard extends Animal implements Nageur {
32    parler(): void {
33      console.log(`${this.nom} fait coin-coin`);
34    }
35
36    nager(): void {
37      console.log(`${this.nom} nage`);
38    }
39  }
```

5 Les interfaces

5.1 Introduction aux interfaces

Définition

Une **interface** définit un contrat : la structure que doit respecter un objet. Elle décrit quelles propriétés et méthodes doivent être présentes.

```typescript
// Definition d'une interface
interface Utilisateur {
  id: number;
  nom: string;
  email: string;
  age: number;
}

// L'objet doit respecter le contrat
const alice: Utilisateur = {
  id: 1,
  nom: 'Alice',
  email: 'alice@example.com',
  age: 25
};

// Erreur si propriete manquante ou type incorrect
// const bob: Utilisateur = { id: 2, nom: 'Bob' };  // Erreur!

// Utilisation comme type de parametre
function afficherUtilisateur(user: Utilisateur): void {
  console.log(`${user.nom} (${user.email})`);
}
```

5.2 Propriétés optionnelles et en lecture seule

```typescript
interface Configuration {
  // Propriete obligatoire
  serveur: string;

  // Propriete optionnelle
  port?: number;

  // Propriete en lecture seule
  readonly version: string;

  // Propriete optionnelle et readonly
  readonly apiKey?: string;
}

```

```typescript
const config: Configuration = {
  serveur: 'localhost',
  version: '1.0.0'
};

config.port = 8080;        // OK (optionnel mais modifiable)
// config.version = '2.0'; // Erreur! (readonly)

// Utility type Readonly pour rendre tout readonly
const configImmuable: Readonly<Configuration> = {
  serveur: 'prod.example.com',
  version: '2.0.0',
  port: 443
};
// configImmuable.port = 80;  // Erreur!
```

5.3 Types indexables

```typescript
// Index signature : cles dynamiques
interface Dictionnaire {
  [cle: string]: string;
}

const traductions: Dictionnaire = {
  hello: 'bonjour',
  goodbye: 'au revoir',
  thanks: 'merci'
};

traductions['yes'] = 'oui';  // OK

// Combinaison avec proprietes fixes
interface Configuration {
  version: string;
  [option: string]: string | number;  // Doit etre compatible
}

// Index numerique (comme un tableau)
interface TableauDeNoms {
  [index: number]: string;
  length: number;
}

const noms: TableauDeNoms = {
  0: 'Alice',
  1: 'Bob',
  length: 2
};
```

5.4 Interfaces pour les fonctions

```typescript
// Interface de fonction
interface Calculateur {
  (a: number, b: number): number;
}

const additionner: Calculateur = (a, b) => a + b;
const multiplier: Calculateur = (a, b) => a * b;

// Interface avec proprietes et appel
interface CompteurFonction {
  (): number;             // Signature d'appel
  compteur: number;       // Propriete
  reset(): void;          // Methode
}

function creerCompteur(): CompteurFonction {
  const fn = function() {
    return ++fn.compteur;
  } as CompteurFonction;

  fn.compteur = 0;
  fn.reset = () => { fn.compteur = 0; };

  return fn;
}

const compteur = creerCompteur();
compteur();             // 1
compteur();             // 2
compteur.reset();
compteur();             // 1
```

5.5 Extension et composition d'interfaces

```typescript
// Extension d'interface
interface Personne {
  nom: string;
  age: number;
}

interface Employe extends Personne {
  employeId: string;
  departement: string;
}

const employe: Employe = {
  nom: 'Alice',
  age: 30,
```

```
  employeId: 'E001',
  departement: 'IT'
};

// Extension multiple
interface Adresse {
  rue: string;
  ville: string;
}

interface Contact extends Personne, Adresse {
  telephone: string;
}

// Fusion d'interfaces (declaration merging)
interface Config {
  debug: boolean;
}

interface Config {
  timeout: number;
}

// Config a maintenant debug ET timeout
const conf: Config = {
  debug: true,
  timeout: 5000
};
```

6 Unions, intersections et alias

6.1 Union de types

```typescript
// Union : l'un OU l'autre
type StringOuNombre = string | number;

let valeur: StringOuNombre;
valeur = 'texte';     // OK
valeur = 42;          // OK
// valeur = true;     // Erreur!

// Union de litteraux
type Direction = 'haut' | 'bas' | 'gauche' | 'droite';
let dir: Direction = 'haut';
// dir = 'diagonal';  // Erreur!

type StatusHTTP = 200 | 201 | 400 | 404 | 500;

// Narrowing (affinage de type)
function traiter(valeur: string | number): string {
  if (typeof valeur === 'string') {
    return valeur.toUpperCase();  // TypeScript sait que c'est
       string
  }
  return valeur.toFixed(2);  // TypeScript sait que c'est number
}

// Union discriminee
interface Succes {
  type: 'succes';
  donnees: string[];
}

interface Erreur {
  type: 'erreur';
  message: string;
}

type Resultat = Succes | Erreur;

function traiterResultat(res: Resultat): void {
  if (res.type === 'succes') {
    console.log(res.donnees);  // TypeScript connait le type
  } else {
    console.log(res.message);
  }
}
```

6.2 Intersection de types

```typescript
// Intersection : l'un ET l'autre
interface Identifiable {
  id: string;
}

interface Nomme {
  nom: string;
}

interface Timestamped {
  createdAt: Date;
  updatedAt: Date;
}

// Combine toutes les proprietes
type Entite = Identifiable & Nomme & Timestamped;

const entite: Entite = {
  id: '123',
  nom: 'Test',
  createdAt: new Date(),
  updatedAt: new Date()
};

// Avec des types inline
type PersonneAvecAdresse = {
  nom: string;
  age: number;
} & {
  rue: string;
  ville: string;
};
```

6.3 Alias de types

```typescript
// Alias simple
type ID = string | number;
type Callback = (erreur: Error | null) => void;

// Alias d'objet
type Point = {
  x: number;
  y: number;
};

type Point3D = Point & { z: number };

// Alias recursif
```

```typescript
type ArbreJSON = {
  valeur: string;
  enfants?: ArbreJSON[];
};

const arbre: ArbreJSON = {
  valeur: 'racine',
  enfants: [
    { valeur: 'enfant1' },
    { valeur: 'enfant2', enfants: [{ valeur: 'petit-enfant' }] }
  ]
};

// Alias de tuple
type Coordonnees = [number, number];
type RGB = [number, number, number];
```

À retenir

Type vs Interface :

- **Interface** : pour les objets, extensible, peut fusionner
- **Type** : pour tout (unions, tuples, primitifs), ne peut pas fusionner

Préférez les interfaces pour les objets et les types pour le reste.

6.4 Opérateurs de types

```typescript
// keyof : obtenir les cles d'un type
interface Personne {
  nom: string;
  age: number;
  email: string;
}

type ClesPersonne = keyof Personne;  // 'nom' | 'age' | 'email'

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K]
   {
  return obj[key];
}

const p: Personne = { nom: 'Alice', age: 25, email: 'a@b.com' };
const nom = getProperty(p, 'nom');      // string
const age = getProperty(p, 'age');      // number
// getProperty(p, 'invalid');           // Erreur!

// typeof : obtenir le type d'une valeur
const config = {
  url: 'https://api.example.com',
```

```
  timeout: 5000
};

type ConfigType = typeof config;
// { url: string; timeout: number }

// Indexed access types
type NomType = Personne['nom'];          // string
type NomOuAge = Personne['nom' | 'age'];  // string | number
```

7 Les types génériques

7.1 Introduction aux génériques

Définition

Les **types génériques** permettent de créer des composants réutilisables qui fonctionnent avec différents types tout en conservant la sécurité du typage.

```
// Probleme sans generiques
function identiteAny(valeur: any): any {
  return valeur;
}
let resultat = identiteAny('hello');  // Type: any (perdu!)

// Solution avec generiques
function identite<T>(valeur: T): T {
  return valeur;
}

let str = identite<string>('hello');    // Type: string
let num = identite<number>(42);         // Type: number
let auto = identite('inference');       // Type: string (infere)

// Plusieurs parametres de type
function paire<T, U>(premier: T, second: U): [T, U] {
  return [premier, second];
}

const p = paire<string, number>('age', 25);  // [string, number]
const p2 = paire('nom', 'Alice');            // [string, string]
```

7.2 Fonctions génériques

```
// Fonction generique avec tableau
function premier<T>(tableau: T[]): T | undefined {
  return tableau[0];
}

premier([1, 2, 3]);        // number | undefined
premier(['a', 'b']);       // string | undefined

// Fonction generique avec contrainte
interface Longueur {
  length: number;
}

function afficherLongueur<T extends Longueur>(element: T): number
    {
```

```
  console.log(element.length);
  return element.length;
}

afficherLongueur('hello');          // OK (string a length)
afficherLongueur([1, 2, 3]);        // OK (array a length)
// afficherLongueur(123);           // Erreur! (number n'a pas length)

// Contrainte avec keyof
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
```

7.3 Interfaces et classes génériques

```
// Interface generique
interface Reponse<T> {
  succes: boolean;
  donnees: T;
  erreur?: string;
}

const reponseUser: Reponse<{ nom: string }> = {
  succes: true,
  donnees: { nom: 'Alice' }
};

const reponseNombres: Reponse<number[]> = {
  succes: true,
  donnees: [1, 2, 3]
};

// Classe generique
class File<T> {
  private elements: T[] = [];

  enfiler(element: T): void {
    this.elements.push(element);
  }

  defiler(): T | undefined {
    return this.elements.shift();
  }

  taille(): number {
    return this.elements.length;
  }
}
```

```

const fileNombres = new File<number>();
fileNombres.enfiler(1);
fileNombres.enfiler(2);
fileNombres.defiler();  // 1

const fileStrings = new File<string>();
fileStrings.enfiler('hello');
```

7.4 Valeurs par défaut et contraintes

```
// Type generique avec valeur par defaut
interface Container<T = string> {
  valeur: T;
}

const c1: Container = { valeur: 'texte' };          // T = string
const c2: Container<number> = { valeur: 42 };      // T = number

// Contrainte avec interface
interface Identifiable {
  id: string;
}

class Repository<T extends Identifiable> {
  private items: Map<string, T> = new Map();

  ajouter(item: T): void {
    this.items.set(item.id, item);
  }

  trouver(id: string): T | undefined {
    return this.items.get(id);
  }
}

interface Produit extends Identifiable {
  nom: string;
  prix: number;
}

const repo = new Repository<Produit>();
repo.ajouter({ id: '1', nom: 'Livre', prix: 29.99 });
```

8 Namespaces et fichiers de déclaration

8.1 Les namespaces

```typescript
// Namespace pour organiser le code
namespace Validation {
  // Export pour rendre accessible
  export interface Validateur {
    estValide(s: string): boolean;
  }

  export class ValidateurEmail implements Validateur {
    estValide(s: string): boolean {
      return /^[\w-]+@[\w-]+\.\w+$/.test(s);
    }
  }

  export class ValidateurTelephone implements Validateur {
    estValide(s: string): boolean {
      return /^0[1-9]\d{8}$/.test(s);
    }
  }

  // Fonction privee (non exportee)
  function helper(): void {}
}

// Utilisation
const emailValidator = new Validation.ValidateurEmail();
emailValidator.estValide('test@example.com');  // true

// Namespace imbrique
namespace App {
  export namespace Models {
    export interface User {
      id: string;
      nom: string;
    }
  }

  export namespace Services {
    export function getUser(id: string): Models.User {
      return { id, nom: 'Test' };
    }
  }
}
```

Important

Les namespaces sont moins utilisés aujourd'hui. Préférez les **modules ES6** (`import`/`export`) pour organiser votre code.

8.2 Fichiers de déclaration (.d.ts)

```typescript
// fichier: types.d.ts

// Declaration d'une variable globale
declare const VERSION: string;

// Declaration d'une fonction globale
declare function log(message: string): void;

// Declaration d'un module
declare module 'ma-bibliotheque' {
  export function init(): void;
  export function process(data: string): string;
  export interface Config {
    debug: boolean;
    timeout: number;
  }
}

// Augmentation de module existant
declare module 'express' {
  interface Request {
    utilisateur?: {
      id: string;
      nom: string;
    };
  }
}
```

8.3 Le mot-clé declare

```typescript
// declare : informe TypeScript de l'existence de quelque chose
// sans generer de code JavaScript

// Variable globale (ex: definie dans un script externe)
declare const jQuery: any;
declare const $: typeof jQuery;

// Fonction globale
declare function ga(command: string, ...params: any[]): void;

// Classe globale
declare class ExternalClass {
```

```
  constructor(config: object);
  method(): void;
}

// Namespace global
declare namespace MonApp {
  interface Config {
    apiUrl: string;
  }
  function init(config: Config): void;
}

// Utilisation (TypeScript fait confiance a ces declarations)
MonApp.init({ apiUrl: 'https://api.example.com' });
```

8.4 Créer ses fichiers de types

```
// fichier: @types/mon-module/index.d.ts

// Pour un module sans types
declare module 'bibliotheque-sans-types' {
  export function doSomething(value: string): number;
  export interface Options {
    timeout?: number;
    retries?: number;
  }
  export default function main(options?: Options): void;
}

// Pour etendre les types globaux
declare global {
  interface Window {
    maVariable: string;
  }

  interface Array<T> {
    customMethod(): T[];
  }
}

// Export vide necessaire pour les declarations globales
export {};
```

Astuce

Pour les bibliothèques populaires, cherchez d'abord les types sur **DefinitelyTyped** :

```
npm install --save-dev @types/lodash
npm install --save-dev @types/express
```

9 Types utilitaires et conditionnels

9.1 Types utilitaires natifs

```typescript
interface Utilisateur {
  id: string;
  nom: string;
  email: string;
  age: number;
}

// Partial<T> : toutes les proprietes optionnelles
type UtilisateurPartiel = Partial<Utilisateur>;
// { id?: string; nom?: string; email?: string; age?: number }

// Required<T> : toutes les proprietes obligatoires
type UtilisateurComplet = Required<UtilisateurPartiel>;

// Readonly<T> : toutes les proprietes en lecture seule
type UtilisateurReadonly = Readonly<Utilisateur>;

// Pick<T, K> : selectionner certaines proprietes
type UtilisateurResume = Pick<Utilisateur, 'id' | 'nom'>;
// { id: string; nom: string }

// Omit<T, K> : exclure certaines proprietes
type UtilisateurSansEmail = Omit<Utilisateur, 'email'>;
// { id: string; nom: string; age: number }

// Record<K, T> : creer un objet avec cles et valeurs typees
type Scores = Record<string, number>;
// { [key: string]: number }

type PageViews = Record<'home' | 'about' | 'contact', number>;
// { home: number; about: number; contact: number }
```

9.2 Types utilitaires pour les unions

```typescript
type T1 = string | number | boolean;
type T2 = string | number;

// Exclude<T, U> : exclure les types de T qui sont dans U
type T3 = Exclude<T1, boolean>;  // string | number

// Extract<T, U> : extraire les types de T qui sont dans U
type T4 = Extract<T1, string | boolean>;  // string | boolean

// NonNullable<T> : exclure null et undefined
type T5 = NonNullable<string | null | undefined>;  // string

```

```typescript
// Exemple pratique
type Reponse =
  | { status: 'succes'; data: string }
  | { status: 'erreur'; message: string }
  | null;

type ReponseNonNull = NonNullable<Reponse>;
type ReponsesSucces = Extract<ReponseNonNull, { status: 'succes' }>;
```

9.3 Types utilitaires pour les fonctions

```typescript
function creerUtilisateur(
  nom: string,
  age: number,
  email?: string
): { id: string; nom: string; age: number } {
  return { id: '123', nom, age };
}

// Parameters<T> : tuple des types de parametres
type Params = Parameters<typeof creerUtilisateur>;
// [nom: string, age: number, email?: string]

// ReturnType<T> : type de retour de la fonction
type Retour = ReturnType<typeof creerUtilisateur>;
// { id: string; nom: string; age: number }

// Pour les classes
class MaClasse {
  constructor(public nom: string) {}
}

// InstanceType<T> : type de l'instance
type Instance = InstanceType<typeof MaClasse>;
// MaClasse

// ConstructorParameters<T> : parametres du constructeur
type CtorParams = ConstructorParameters<typeof MaClasse>;
// [nom: string]
```

9.4 Types conditionnels

```typescript
// Syntaxe : T extends U ? X : Y
type EstString<T> = T extends string ? true : false;

type A = EstString<string>;   // true
type B = EstString<number>;   // false

```

```typescript
// Type conditionnel pratique
type NonNullable<T> = T extends null | undefined ? never : T;

// Avec infer pour extraire des types
type RetourDeFonction<T> = T extends (...args: any[]) => infer R
  ? R
  : never;

type R1 = RetourDeFonction<() => string>;         // string
type R2 = RetourDeFonction<(x: number) => boolean>;  // boolean
type R3 = RetourDeFonction<string>;               // never

// Extraire le type des elements d'un tableau
type ElementDeTableau<T> = T extends (infer E)[] ? E : never;

type E1 = ElementDeTableau<string[]>;   // string
type E2 = ElementDeTableau<number[]>;   // number

// Type conditionnel distribue sur les unions
type ToArray<T> = T extends any ? T[] : never;

type TA = ToArray<string | number>;  // string[] | number[]
```

9.5 Types mappés

```typescript
// Transformer toutes les proprietes d'un type
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

type Partial<T> = {
  [P in keyof T]?: T[P];
};

type Nullable<T> = {
  [P in keyof T]: T[P] | null;
};

// Exemple personnalise
interface Personne {
  nom: string;
  age: number;
}

type PersonneNullable = Nullable<Personne>;
// { nom: string | null; age: number | null }

// Avec modificateurs
type Mutable<T> = {
  -readonly [P in keyof T]: T[P];  // Supprime readonly
```

```
};

type Obligatoire<T> = {
  [P in keyof T]-?: T[P];  // Supprime optionnel
};

// Renommer les cles
type Getters<T> = {
  [P in keyof T as `get${Capitalize<string & P>}`]: () => T[P];
};

type PersonneGetters = Getters<Personne>;
// { getNom: () => string; getAge: () => number }
```

10 Résumé et points clés

À retenir

Types de base :

- Primitifs : `string`, `number`, `boolean`, `null`, `undefined`
- Spéciaux : `any` (éviter), `unknown` (préférer), `void`, `never`
- Collections : `T[]`, `Array<T>`, tuples `[T, U]`
- Énumérations : `enum` pour les constantes nommées

À retenir

Fonctions :

- Toujours typer les paramètres
- Le type de retour est souvent inféré mais peut être explicite
- Paramètres optionnels avec `?`, par défaut avec `=`
- Opérateur rest `...args: T[]`
- Surcharge pour différentes signatures

À retenir

Classes :

- Modificateurs : `public`, `private`, `protected`, `readonly`
- Notation raccourcie dans le constructeur
- Membres `static` partagés entre instances
- Classes `abstract` non instanciables

À retenir

Interfaces et types :

- `interface` : pour les objets, extensibles
- `type` : pour unions, intersections, tuples
- Unions : `A | B`, intersections : `A & B`
- `keyof` pour les clés, `T[K]` pour les types de propriétés

À retenir

Génériques :

- Syntaxe : `<T>`, `<T, U>`
- Contraintes : `<T extends Interface>`
- Permettent la réutilisation avec sécurité de type

À retenir

Types utilitaires courants :

- `Partial<T>`, `Required<T>`, `Readonly<T>`
- `Pick<T, K>`, `Omit<T, K>`
- `Record<K, T>`
- `Parameters<T>`, `ReturnType<T>`

Prochain chapitre : Vue.js

Nous allons maintenant découvrir Vue.js, le framework JavaScript progressif !