Exemples de Types et Fonctions en TypeScript

Ce document fournit des exemples clairs et concis pour chaque type et concept de fonction mentionné dans le diagramme.

1. Types Primitifs

1.1 Number

1.2 Boolean

1.3 String

2. Types Avancés

2.1 Le type FUNCTION

2.2 Enum

2.3 Type union

2.4 Type never

2.5 Type object

2.6 Alias ou type personnalisé

2.7 Type objet anonyme

2.8 Type objet imbriqué

2.9 Array

3. Fonctions

3.1 Surcharge de méthode et fonction

3.2 Fonction générique (Extends)

3.3 Type de retour de fonction

3.4 Find

3.5 Fn sans retour : void - undefined

3.6 Fn avec retour

1.1 Number

Le type `number` en TypeScript représente les nombres à virgule flottante (float) et les entiers (integer). Tous les nombres en JavaScript sont des nombres à virgule flottante double précision (64-bit float). TypeScript utilise ce même type `number`.

Exemple :

```
 let age: number = 30;
let price: number = 99.99;
let quantity: number = 100;

console.log(`Age: ${age}`);
console.log(`Price: ${price}`);
console.log(`Quantity: ${quantity}`);

// Opérations arithmétiques
let sum: number = age + price;
console.log(`Sum: ${sum}`); // Output: 129.99

// Tentative d'assigner une chaîne à un nombre (erreur de compilation)
// let invalidNumber: number = "hello"; // Erreur: Type 'string' n'est pas
assignable au type 'number'.
```

2.1 Le type FUNCTION

En TypeScript, le type `Function` (avec un 'F' majuscule) représente toutes les fonctions. Cependant, l'utilisation de ce type est généralement déconseillée car il ne fournit aucune information sur la signature de la fonction (ses paramètres ou son type de retour), ce qui réduit la sécurité des types. Il est préférable de typer les fonctions avec des signatures spécifiques ou des alias de type.

Exemple :

```
// Utilisation du type Function (déconseillé pour la sécurité des types)
let myGenericFunction: Function;

myGenericFunction = (a: number, b: number) => a + b;
console.log(myGenericFunction(5, 3)); // Output: 8

myGenericFunction = (name: string) => `Hello, ${name}!`;
console.log(myGenericFunction("Alice")); // Output: Hello, Alice!

// Problème: Aucune vérification de type pour les arguments ou le retour
// console.log(myGenericFunction(10)); // Pas d'erreur à la compilation, mais peut échouer à l'exécution

// Meilleure pratique: Typer la fonction avec sa signature spécifique
let addNumbers: (a: number, b: number) => number;
addNumbers = (x, y) => x + y;
console.log(addNumbers(10, 20)); // Output: 30

// Tentative d'assigner une fonction avec une signature incompatible (erreur de compilation)
// addNumbers = (name: string) => `Hello, ${name}!`; // Erreur: Type '(name: string) => string' n'est pas assignable au type '(a: number, b: number) => number'.

// Ou utiliser un alias de type pour la réutilisabilité
type GreeterFunction = (name: string) => string;
let greetUser: GreeterFunction = (user) => `Welcome, ${user}!`;
console.log(greetUser("Bob")); // Output: Welcome, Bob!
```

3.1 Surcharge de méthode et fonction

La surcharge (overloading) en TypeScript permet de définir plusieurs signatures de fonction pour une même fonction ou méthode. Cela signifie que la fonction peut être appelée avec différents types ou nombres d'arguments, et TypeScript utilisera la signature appropriée pour la vérification de type. La surcharge est implémentée en fournissant plusieurs signatures de fonction (signatures de surcharge) suivies d'une seule implémentation de fonction (signature d'implémentation).

Exemple : Surcharge de fonction

```typescript
// Signatures de surcharge
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: string, b: number): string;
function add(a: number, b: string): string;

// Implémentation de la fonction (doit être compatible avec toutes les
signatures de surcharge)
function add(a: any, b: any): any {
  if (typeof a === 'string' && typeof b === 'string') {
    return a + ' ' + b; // Concaténation de chaînes
  }
  if (typeof a === 'number' && typeof b === 'number') {
    return a + b; // Addition de nombres
  }
  if (typeof a === 'string' && typeof b === 'number') {
    return `String: $`{a}, Number: `${b}`;
  }
  if (typeof a === 'number' && typeof b === 'string') {
    return `Number: $`{a}, String: `${b}`;
  }
  throw new Error('Invalid arguments');
}

console.log(add(1, 2));         // Output: 3
console.log(add("Hello", "World")); // Output: Hello World
console.log(add("Age", 30));    // Output: String: Age, Number: 30
console.log(add(10, "items"));  // Output: Number: 10, String: items

// add(true, false); // Erreur: Aucune surcharge ne correspond à cet appel.
```

Dans cet exemple, la fonction `add` a quatre signatures de surcharge qui définissent comment elle peut être appelée. L'implémentation unique de `add` utilise des vérifications de type (`typeof`) pour déterminer la logique à exécuter en fonction des types d'arguments reçus. TypeScript utilise les signatures de surcharge pour la vérification de type au moment de la compilation, tandis que l'implémentation est ce qui est réellement exécuté au moment de l'exécution.

Exemple : Surcharge de méthode

La surcharge de méthode fonctionne de la même manière que la surcharge de fonction, mais elle est appliquée aux méthodes d'une classe.

```
class Calculator {
  // Signatures de surcharge
  add(a: number, b: number): number;
  add(a: string, b: string): string;

  // Implémentation de la méthode
  add(a: any, b: any): any {
    if (typeof a === 'string' && typeof b === 'string') {
      return a + b;
    }
    if (typeof a === 'number' && typeof b === 'number') {
      return a + b;
    }
    throw new Error('Invalid arguments');
  }
}

const calc = new Calculator();
console.log(calc.add(5, 10));        // Output: 15
console.log(calc.add("First", "Last")); // Output: FirstLast
// calc.add(5, "test"); // Erreur: Aucune surcharge ne correspond à cet appel.
```

La surcharge est un moyen puissant de fournir une API flexible et typée pour les fonctions et les méthodes, permettant aux développeurs d'utiliser la même fonction avec différents types d'entrée tout en bénéficiant de la sécurité des types de TypeScript.

1.2 Boolean

Le type `boolean` en TypeScript représente une valeur logique qui peut être `true` ou `false`.

Exemple :

```
let isActive: boolean = true;
let hasPermission: boolean = false;

console.log(`Is active: ${isActive}`);
console.log(`Has permission: ${hasPermission}`);

// Opérations logiques
if (isActive && !hasPermission) {
  console.log("User is active but lacks permission.");
}

// Tentative d'assigner un nombre à un booléen (erreur de compilation)
// let isLogged: boolean = 1; // Erreur: Type 'number' n'est pas assignable au
type 'boolean'.
```

2.2 Enum

Les enums (énumérations) en TypeScript permettent de définir un ensemble de constantes nommées. Elles rendent le code plus lisible et plus maintenable en remplaçant des valeurs numériques ou textuelles "magiques" par des noms significatifs.

Exemple : Enum numérique

Par défaut, les enums sont basés sur des nombres, en commençant par 0.

```
enum Direction {
  Up,    // 0
  Down,  // 1
  Left,  // 2
  Right  // 3
}

let go: Direction = Direction.Up;
console.log(go);           // Output: 0
console.log(Direction.Left); // Output: 2

// Accéder au nom de l'enum à partir de sa valeur
console.log(Direction[0]); // Output: Up
console.log(Direction[Direction.Down]); // Output: Down
```

Vous pouvez également définir manuellement les valeurs numériques :

```
enum StatusCode {
  NotFound = 404,
  Success = 200,
  Accepted = 202,
  BadRequest = 400
}

let status: StatusCode = StatusCode.Success;
console.log(status); // Output: 200
console.log(StatusCode.NotFound); // Output: 404
```

Exemple : Enum de chaînes de caractères

Les enums de chaînes de caractères sont souvent plus lisibles et plus faciles à déboguer car ils ne sont pas transpilés en nombres.

```
enum UserRole {
  Admin = "ADMIN",
  Editor = "EDITOR",
  Viewer = "VIEWER"
}

let userRole: UserRole = UserRole.Admin;
console.log(userRole); // Output: ADMIN

function checkRole(role: UserRole) {
  if (role === UserRole.Admin) {
    console.log("Access granted: Administrator");
  } else if (role === UserRole.Editor) {
    console.log("Access granted: Editor");
  } else {
    console.log("Access granted: Viewer");
  }
}

checkRole(UserRole.Editor); // Output: Access granted: Editor
```

Exemple : Enum hétérogène (déconseillé)

Vous pouvez mélanger des valeurs numériques et de chaînes, mais c'est généralement déconseillé pour la clarté.

```
enum Mixed {
  No = 0,
  Yes = "YES",
  Maybe = 1
}

console.log(Mixed.No);    // Output: 0
console.log(Mixed.Yes);   // Output: YES
console.log(Mixed.Maybe); // Output: 1
```

Les enums sont un excellent moyen de créer des types plus expressifs et de réduire les erreurs dues à l'utilisation de valeurs arbitraires.

1.3 String

Le type `string` en TypeScript représente des séquences de caractères. Les chaînes de caractères sont utilisées pour stocker du texte.

Exemple :

```
let greeting: string = "Hello, TypeScript!";
let userName: string = 'Alice';
let multiLine: string = `This is a
multi-line string.`;

console.log(greeting);
console.log(userName);
console.log(multiLine);

// Concaténation de chaînes
let fullName: string = userName + " Smith";
console.log(fullName); // Output: Alice Smith

// Longueur de la chaîne
console.log(greeting.length); // Output: 18

// Méthodes de chaîne
console.log(greeting.toUpperCase()); // Output: HELLO, TYPESCRIPT!
console.log(greeting.includes("Type")); // Output: true

// Tentative d'assigner un nombre à une chaîne (erreur de compilation)
// let invalidString: string = 123; // Erreur: Type 'number' n'est pas
assignable au type 'string'.
```

2.3 Type union

Les types d'union en TypeScript permettent de déclarer qu'une variable peut être de l'un de plusieurs types possibles. C'est une fonctionnalité très puissante pour la flexibilité des types, permettant à une variable de contenir des valeurs de différents types à différents moments, tout en conservant la sécurité des types au moment de la compilation. L'opérateur d'union est la barre verticale (`|`).

Exemple :

```
type StringOrNumber = string | number;

let id: StringOrNumber;
id = "abc-123"; // Valide
id = 123;       // Valide
// id = true;    // Erreur: Type 'boolean' n'est pas assignable au type
'StringOrNumber'.

function printId(id: string | number) {
  if (typeof id === "string") {
    console.log(id.toUpperCase()); // TypeScript sait que 'id' est une chaîne
ici
  } else {
    console.log(id * 2); // TypeScript sait que 'id' est un nombre ici
  }
}

printId("myid"); // Output: MYID
printId(123);   // Output: 246
```

2.4 Type never

Le type `never` représente le type des valeurs qui n'arrivent jamais. Il est utilisé pour indiquer que la fonction ne retournera jamais (par exemple, elle lève une exception ou contient une boucle infinie) ou que la variable ne peut jamais avoir une certaine valeur.

Exemple : Fonction qui ne retourne jamais

```typescript
function error(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {
    // Boucle infinie
  }
}

// Utilisation
try {
  error("Something went wrong!");
} catch (e) {
  console.log(e.message); // Output: Something went wrong!
}

// La fonction suivante utilise 'never' pour garantir l'exhaustivité dans un switch
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    default:
      // Si un nouveau type est ajouté à Shape et n'est pas géré ici,
      // TypeScript générera une erreur car 'shape' ne peut pas être de type 'never'.
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}

const myCircle: Circle = { kind: "circle", radius: 10 };
console.log(getArea(myCircle)); // Output: 314.159...
```

2.5 Type object

Le type `object` (avec un 'o' minuscule) en TypeScript représente toute valeur qui n'est pas un type primitif (string, number, boolean, symbol, null, undefined, bigint). Il est plus strict que le type `Object` (avec un 'O' majuscule) et plus général que `{}`.

Exemple :

```typescript
let myObject: object;

myObject = {};           // Valide
myObject = [];           // Valide
myObject = new Date();   // Valide
myObject = () => {};     // Valide (les fonctions sont des objets en JS)

// myObject = 123;       // Erreur: Type 'number' n'est pas assignable au type
'object'.
// myObject = "hello";   // Erreur: Type 'string' n'est pas assignable au type
'object'.

function processGenericObject(obj: object) {
  console.log("Processing a generic object.");
  // Vous ne pouvez pas accéder aux propriétés spécifiques sans affinement de
type
  // console.log(obj.length); // Erreur de compilation

  if (Array.isArray(obj)) {
    console.log(`It's an array with length: ${obj.length}`);
  } else if (typeof obj === 'function') {
    console.log(`It's a function named: ${obj.name}`);
  }
}

processGenericObject({ key: "value" });
processGenericObject([1, 2, 3]);
processGenericObject(function myFunc() {});
```

2.6 Alias ou type personnalisé

Les alias de type (type aliases) en TypeScript permettent de donner un nouveau nom à un type existant. Ils sont créés avec le mot-clé `type` et sont très utiles pour rendre le code plus lisible et pour définir des types complexes ou des unions de types.

Exemple :

```ts
type UserId = string | number;

let user1: UserId = "abc-123";
let user2: UserId = 456;

// let user3: UserId = true; // Erreur: Type 'boolean' n'est pas assignable au type 'UserId'.

function greetUser(id: UserId) {
  if (typeof id === "string") {
    console.log(`Hello, user with string ID: ${id}`);
  } else {
    console.log(`Hello, user with numeric ID: ${id}`);
  }
}

greetUser("def-789"); // Output: Hello, user with string ID: def-789
greetUser(789);       // Output: Hello, user with numeric ID: 789

// Alias pour un objet complexe
type Point = {
  x: number;
  y: number;
};

let p: Point = { x: 10, y: 20 };
console.log(`Point coordinates: ($`{p.x}, `${p.y})`);

// Alias pour une fonction
type MathOperation = (a: number, b: number) => number;

const add: MathOperation = (x, y) => x + y;
const subtract: MathOperation = (x, y) => x - y;

console.log(add(5, 3));      // Output: 8
console.log(subtract(10, 4)); // Output: 6
```

2.7 Type objet anonyme

Un type objet anonyme est un type d'objet défini directement en ligne, sans lui donner de nom via une interface ou un alias de type. Il est utile pour des objets simples qui ne sont utilisés qu'une seule fois ou dans un contexte très local.

Exemple :

```typescript
// Définition d'un objet avec un type anonyme
let person: { name: string; age: number; };

person = { name: "Alice", age: 30 };
console.log(person.name); // Output: Alice

// Utilisation comme paramètre de fonction
function printCoordinates(pt: { x: number; y: number; }) {
  console.log(`The coordinate's x value is ${pt.x}`);
  console.log(`The coordinate's y value is ${pt.y}`);
}

printCoordinates({ x: 3, y: 7 });
// Output:
// The coordinate's x value is 3
// The coordinate's y value is 7

// Retour d'un objet anonyme
function createPoint(x: number, y: number): { x: number; y: number; } {
  return { x, y };
}

const myPoint = createPoint(10, 20);
console.log(myPoint.x); // Output: 10
```

Bien que pratiques pour des cas simples, les types objets anonymes peuvent rendre le code moins lisible et plus difficile à maintenir pour des structures plus complexes ou réutilisées. Dans ces cas, il est préférable d'utiliser des interfaces ou des alias de type.

3.3 Type de retour de fonction

En TypeScript, vous pouvez spécifier le type de la valeur qu'une fonction est censée retourner. Cela améliore la sécurité des types et aide à la détection précoce des erreurs. Le type de retour est placé après la liste des paramètres, séparé par un côlon.

Exemple :

```typescript
// Fonction retournant un nombre
function add(a: number, b: number): number {
  return a + b;
}

console.log(add(5, 3)); // Output: 8

// Fonction retournant une chaîne de caractères
function greet(name: string): string {
  return `Hello, ${name}!`;
}

console.log(greet("Alice")); // Output: Hello, Alice!

// Fonction retournant un booléen
function isEven(num: number): boolean {
  return num % 2 === 0;
}

console.log(isEven(4)); // Output: true
console.log(isEven(7)); // Output: false

// Fonction retournant un objet
interface Point {
  x: number;
  y: number;
}

function createPoint(x: number, y: number): Point {
  return { x, y };
}

const myPoint = createPoint(10, 20);
console.log(myPoint); // Output: { x: 10, y: 20 }

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

logMessage("This is a log message."); // Output: This is a log message.

// Fonction qui peut retourner undefined (implicitement ou explicitement)
function findElement(arr: string[], element: string): string | undefined {
  const found = arr.find(item => item === element);
  return found; // Peut être undefined si non trouvé
}

const names = ["Alice", "Bob", "Charlie"];
console.log(findElement(names, "Bob"));    // Output: Bob
console.log(findElement(names, "David")); // Output: undefined
```

Spécifier le type de retour est une bonne pratique car cela rend le code plus clair et permet à TypeScript de vous aider à attraper les erreurs où la fonction ne retourne pas le type attendu.

3.4 Find

La méthode `find()` renvoie la **valeur** du premier élément trouvé dans le tableau qui satisfait la fonction de test fournie. Sinon, elle renvoie `undefined`. Elle ne modifie pas le tableau original.

Exemple :

```
 const numbers = [5, 12, 8, 130, 44];

const foundNumber = numbers.find(num => num > 10);
console.log(foundNumber); // Output: 12 (le premier nombre supérieur à 10)

const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
  { id: 3, name: "Charlie" }
];

const userBob = users.find(user => user.name === "Bob");
console.log(userBob); // Output: { id: 2, name: "Bob" }

const userDavid = users.find(user => user.name === "David");
console.log(userDavid); // Output: undefined (non trouvé)

// Trouver un élément qui n'existe pas
const noMatch = numbers.find(num => num > 200);
console.log(noMatch); // Output: undefined
```

2.8 Type objet imbriqué

Un type objet imbriqué fait référence à un objet qui contient d'autres objets comme propriétés. En TypeScript, vous pouvez définir la structure de ces objets imbriqués en utilisant des interfaces ou des alias de type.

Exemple :

```typescript
interface Address {
  street: string;
  city: string;
  zipCode: string;
}

interface UserProfile {
  id: number;
  name: string;
  contact: {
    email: string;
    phone?: string; // Propriété optionnelle
  };
  address: Address; // Utilisation d'une interface imbriquée
}

const user: UserProfile = {
  id: 1,
  name: "Alice Smith",
  contact: {
    email: "alice@example.com",
    phone: "123-456-7890"
  },
  address: {
    street: "123 Main St",
    city: "Anytown",
    zipCode: "12345"
  }
};

console.log(user.name);           // Output: Alice Smith
console.log(user.contact.email);  // Output: alice@example.com
console.log(user.address.city);   // Output: Anytown

// Accès à une propriété optionnelle
if (user.contact.phone) {
  console.log(user.contact.phone); // Output: 123-456-7890
}

// Un autre exemple avec un type anonyme imbriqué
interface Order {
  orderId: string;
  customer: {
    name: string;
    customerId: number;
  };
  items: { productId: number; quantity: number; }[]; // Tableau d'objets
anonymes
}

const order: Order = {
  orderId: "ORD-001",
  customer: {
    name: "Bob Johnson",
    customerId: 101
  },
  items: [
    { productId: 1001, quantity: 2 },
    { productId: 1002, quantity: 1 }
  ]
};
```

```
console.log(order.customer.name); // Output: Bob Johnson
console.log(order.items[0].productId); // Output: 1001
```

Les types objets imbriqués sont essentiels pour modéliser des structures de données complexes et hiérarchiques, courantes dans les applications réelles. Ils permettent de maintenir une forte typisation et une bonne organisation du code.

3.5 Fn sans retour : void - undefined

En TypeScript, le type de retour `void` est utilisé pour les fonctions qui ne retournent aucune valeur. Cela signifie que la fonction exécute une action mais ne produit pas de résultat qui pourrait être utilisé dans une expression. Le type `undefined` est différent; il représente la valeur `undefined` elle-même, et une fonction peut explicitement retourner `undefined`.

Exemple : Fonction avec retour `void`

```
function logMessage(message: string): void {
  console.log(message);
}

logMessage("Hello, world!"); // Output: Hello, world!

// Tenter d'assigner le résultat d'une fonction void à une variable est une
erreur (si strictNullChecks est activé)
// let result: string = logMessage("Test"); // Erreur: Type 'void' n'est pas
assignable au type 'string'.
```

Exemple : Fonction avec retour `undefined`

Une fonction peut explicitement retourner `undefined`. Cela est souvent utilisé lorsque la fonction peut ou non trouver une valeur, ou si elle a un effet de bord mais que son "résultat" est intentionnellement `undefined`.

```typescript
function findFirstEven(numbers: number[]): number | undefined {
  for (const num of numbers) {
    if (num % 2 === 0) {
      return num; // Retourne le premier nombre pair trouvé
    }
  }
  return undefined; // Si aucun nombre pair n'est trouvé
}

console.log(findFirstEven([1, 3, 5, 7, 9])); // Output: undefined
console.log(findFirstEven([1, 2, 3, 4, 5])); // Output: 2

// Une fonction qui ne retourne rien implicitement retourne 'undefined'
function doSomething(): undefined {
  // Pas de 'return' explicite, ou 'return undefined;'
  console.log("Doing something...");
  return undefined;
}

let resultOfDoSomething = doSomething();
console.log(resultOfDoSomething); // Output: Doing something...
                                  // Output: undefined
```

Différence clé :

- `void` signifie que la fonction ne produit *pas* de valeur observable. Vous ne pouvez pas utiliser le résultat de la fonction.
- `undefined` signifie que la fonction retourne explicitement la valeur `undefined`. Vous pouvez assigner ce `undefined` à une variable.

Dans la pratique, pour les fonctions qui n'ont pas de valeur de retour significative, `void` est le type de retour le plus couramment utilisé et le plus approprié.

2.9 Array

Le type `Array` en TypeScript représente un tableau, qui est une collection ordonnée d'éléments. TypeScript permet de typer les tableaux de manière stricte, en spécifiant le type des éléments qu'ils contiennent.

Exemple :

```typescript
// Déclaration d'un tableau de nombres
let numbers: number[] = [1, 2, 3, 4, 5];

// Autre syntaxe pour un tableau (générique)
let names: Array<string> = ["Alice", "Bob", "Charlie"];

console.log(numbers); // Output: [1, 2, 3, 4, 5]
console.log(names);   // Output: ["Alice", "Bob", "Charlie"]

// Accès aux éléments
console.log(numbers[0]); // Output: 1
console.log(names[1]);   // Output: Bob

// Longueur du tableau
console.log(numbers.length); // Output: 5

// Ajout d'éléments
numbers.push(6);
console.log(numbers); // Output: [1, 2, 3, 4, 5, 6]

// Tentative d'ajouter un élément de type incorrect (erreur de compilation)
// numbers.push("seven"); // Erreur: Argument of type 'string' is not
assignable to parameter of type 'number'.

// Tableau de types mixtes (en utilisant un type union)
let mixedArray: (string | number)[] = [1, "hello", 2, "world"];
console.log(mixedArray); // Output: [1, "hello", 2, "world"]

// Tableau d'objets
interface User {
  id: number;
  name: string;
}

let users: User[] = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

console.log(users[0].name); // Output: Alice
```

3.6 Fn avec retour

Une fonction avec retour est une fonction qui produit une valeur après son exécution. Le type de cette valeur de retour est explicitement défini dans la signature de la fonction en TypeScript, ce qui garantit la sécurité des types et aide à la prévisibilité du code.

Exemple : Fonction retournant un nombre

```
function multiply(a: number, b: number): number {
  return a * b;
}

let product: number = multiply(4, 5);
console.log(product); // Output: 20

// Utilisation directe dans une expression
console.log(multiply(2, 7)); // Output: 14
```

Exemple : Fonction retournant une chaîne de caractères

```
function getFullName(firstName: string, lastName: string): string {
  return `$`{firstName} `${lastName}`;
}

let fullName: string = getFullName("John", "Doe");
console.log(fullName); // Output: John Doe
```

Exemple : Fonction retournant un booléen

```
function isAdult(age: number): boolean {
  return age >= 18;
}

console.log(isAdult(20)); // Output: true
console.log(isAdult(16)); // Output: false
```

Exemple : Fonction retournant un objet ou une interface

```
interface User {
  id: number;
  name: string;
}

function createUser(id: number, name: string): User {
  return { id, name };
}

let newUser: User = createUser(1, "Alice");
console.log(newUser); // Output: { id: 1, name: "Alice" }
console.log(newUser.name); // Output: Alice
```

Exemple : Fonction retournant un tableau

```
 function getEvenNumbers(numbers: number[]): number[] {
   return numbers.filter(num => num % 2 === 0);
}

let originalNumbers = [1, 2, 3, 4, 5, 6];
let evenNumbers = getEvenNumbers(originalNumbers);
console.log(evenNumbers); // Output: [2, 4, 6]
```

Ces exemples illustrent comment les fonctions peuvent retourner différents types de données, et comment TypeScript enforce ces types de retour pour améliorer la robustesse et la clarté du code.