Documentation Complète des Concepts TypeScript

Introduction

Ce document fournit une explication approfondie des concepts TypeScript présentés dans le diagramme, accompagnée d'exemples pratiques et d'exercices pour renforcer la compréhension.

1. Interface et Type

2. Intersection de types

3. Paramètres de fonction par défaut

4. Héritage

5. Propriété de discrimination

6. Type union

7. Attributs et méthodes de classe

8. Paramètres callback

9. Fonction générique

10. Décorateurs de propriété

11. Type unknown

12. Tableau en lecture seule

13. Type Object

14. Arrow function

15. Attribut en lecture seule

16. Test de type union

16.1 Typeof

16.2 in

16.3 Instanceof

1. Interface et Type

En TypeScript, `interface` et `type` (alias de type) sont deux concepts fondamentaux utilisés pour définir la forme des objets. Bien qu'ils partagent de nombreuses similitudes et puissent souvent être utilisés de manière interchangeable, il existe des différences clés qui déterminent quand utiliser l'un plutôt que l'autre.

1.1 Interface

Une `interface` est un moyen de définir des "contrats" de code. Elle décrit la forme qu'un objet doit avoir, spécifiant les propriétés et les méthodes qu'il doit contenir. Les interfaces sont particulièrement utiles pour la vérification de type au moment de la compilation et pour garantir que les objets respectent une certaine structure.

Caractéristiques principales des interfaces :

- **Déclaration d'objets** : Principalement utilisées pour décrire la forme des objets.

- **Implémentation par les classes** : Les classes peuvent `implement` des interfaces, garantissant qu'elles respectent le contrat défini par l'interface.
- **Extension (`extends`)** : Les interfaces peuvent étendre d'autres interfaces, permettant l'héritage de propriétés et de méthodes.
- **Fusion (`declaration merging`)** : Les interfaces avec le même nom sont automatiquement fusionnées par TypeScript. Cela signifie que si vous déclarez la même interface plusieurs fois, TypeScript combinera toutes les déclarations en une seule.

Exemple d'interface :

```
interface User {
  id: number;
  name: string;
  email?: string; // Propriété optionnelle
  greet(message: string): void;
}

const user1: User = {
  id: 1,
  name: "Alice",
  greet(message: string) {
    console.log(`${this.name} says: ${message}`);
  }
};

user1.greet("Hello!"); // Output: Alice says: Hello!

class Admin implements User {
  id: number;
  name: string;
  role: string;

  constructor(id: number, name: string, role: string) {
    this.id = id;
    this.name = name;
    this.role = role;
  }

  greet(message: string): void {
    console.log(`Admin ${this.name} (${this.role}) says: ${message}`);
  }
}

const admin1 = new Admin(2, "Bob", "SuperAdmin");
admin1.greet("Welcome!"); // Output: Admin Bob (SuperAdmin) says: Welcome!
```

1.2 Type (Alias de type)

Un `type` (ou alias de type) est un moyen de donner un nouveau nom à un type existant. Il est plus polyvalent que `interface` car il peut représenter non seulement la forme des objets, mais aussi des types primitifs, des unions, des intersections, des tuples, et bien plus encore.

Caractéristiques principales des alias de type :

- **Polyvalence** : Peut définir des types pour des primitives, des unions, des intersections, des tuples, des fonctions, et des objets.
- **Pas d'implémentation par les classes** : Les classes ne peuvent pas `implement` des alias de type.
- **Extension (via intersection)** : Les alias de type peuvent être "étendus" en utilisant des types d'intersection (`&`).
- **Pas de fusion** : Les alias de type avec le même nom ne fusionnent pas. Déclarer le même alias de type deux fois entraînera une erreur.

Exemple d'alias de type :

```
 type ID = number | string; // Alias pour un type union
type Point = { x: number; y: number; }; // Alias pour un type objet
type GreetFunction = (name: string) => string; // Alias pour un type fonction

const userId: ID = 123;
const userName: ID = "user_abc";

const p: Point = { x: 10, y: 20 };

const greet: GreetFunction = (name) => `Hello, ${name}!`;
console.log(greet("World")); // Output: Hello, World!

// Extension d'un type via intersection
type Person = { name: string; };
type Employee = Person & { employeeId: number; };

const emp1: Employee = { name: "Charlie", employeeId: 456 };
console.log(emp1); // Output: { name: 'Charlie', employeeId: 456 }
```

1.3 Quand utiliser l'un ou l'autre ?

La décision d'utiliser `interface` ou `type` dépend souvent du cas d'utilisation spécifique et des préférences personnelles. Voici quelques lignes directrices :

- **Utilisez `interface` pour définir la forme des objets et pour l'implémentation par les classes.** C'est le choix traditionnel et souvent préféré pour les structures d'objets, car il permet la fusion de déclarations, ce qui est utile pour les bibliothèques ou pour étendre des types existants.
- **Utilisez `type` pour tout le reste.** C'est-à-dire pour les types primitifs, les unions, les intersections, les tuples, les fonctions, et lorsque vous avez besoin de la flexibilité de ne pas fusionner les déclarations.

Tableau comparatif :

Caractéristique	`interface`	`type` (alias de type)
Déclaration d'objets	Oui	Oui
Implémentation par classe	Oui (`implements`)	Non
Extension	Oui (`extends`)	Oui (via types d'intersection `&`)
Fusion de déclarations	Oui	Non
Types primitifs/Unions/Intersections/Tuples	Non (seulement objets)	Oui

Exercice pour la section 'Interface et Type'

1. Créez une `interface` appelée `Product` avec les propriétés suivantes :
    - `id` (nombre)
    - `name` (chaîne de caractères)
    - `price` (nombre)
    - `description` (chaîne de caractères, optionnelle)
    - `displayInfo()` (une méthode qui ne prend pas d'arguments et ne retourne rien).
2. Créez une classe `Book` qui `implement` l'interface `Product` . La classe `Book` doit avoir une propriété supplémentaire `author` (chaîne de caractères) et

implémenter la méthode `displayInfo()` pour afficher toutes les informations du livre.

3. Créez un `type` alias appelé `StatusCode` qui peut être `200`, `404`, ou `500`.
4. Créez un `type` alias appelé `ApiResponse` qui est une union de deux types d'objets : l'un pour le succès (`{ status: StatusCode; data: any; }`) et l'autre pour l'erreur (`{ status: StatusCode; error: string; }`).
5. Écrivez une fonction `handleResponse` qui prend un argument de type `ApiResponse` et affiche un message approprié en fonction du `status` et de la présence de `data` ou `error`.

2. Intersection de types

En TypeScript, un type d'intersection combine plusieurs types en un seul. Cela signifie que le nouveau type aura toutes les propriétés de tous les types combinés. L'opérateur d'intersection est l'esperluette (`&`).

2.1 Fonctionnement de l'intersection de types

Lorsque vous combinez deux types ou plus avec l'opérateur `&`, le type résultant est un type qui possède les membres de tous les types d'entrée. C'est comme si vous preniez l'union de leurs propriétés. Si des propriétés ont le même nom mais des types différents, le type résultant pour cette propriété sera l'intersection de ces types. Si les types de propriétés sont incompatibles (par exemple, `string` et `number`), le type résultant pour cette propriété sera `never`.

Exemple 1 : Combinaison de types d'objets

```typescript
interface Person {
  name: string;
  age: number;
}

interface Employee {
  employeeId: string;
  department: string;
}

type EmployeePerson = Person & Employee;

const emp: EmployeePerson = {
  name: "Alice",
  age: 30,
  employeeId: "EMP001",
  department: "Sales"
};

console.log(emp.name);       // Alice
console.log(emp.employeeId); // EMP001
```

Dans cet exemple, `EmployeePerson` doit avoir toutes les propriétés de `Person` ET toutes les propriétés de `Employee`.

Exemple 2 : Propriétés en conflit

```typescript
interface A {
  value: string;
}

interface B {
  value: number;
}

type C = A & B;

// const obj: C = { value: "hello" }; // Erreur: Type 'string' n'est pas
assignable au type 'never'.
// const obj: C = { value: 123 };   // Erreur: Type 'number' n'est pas
assignable au type 'never'.

// Le type de 'value' dans C est 'string & number', ce qui est 'never'.
// Il est impossible de créer un objet de type C qui satisfasse les deux
conditions.
```

Lorsque des propriétés ont le même nom mais des types différents et incompatibles, leur intersection devient `never`. Cela signifie qu'aucune valeur ne peut satisfaire ce type, rendant la création d'un tel objet impossible.

Exemple 3 : Intersection avec des types primitifs ou des unions

L'intersection peut également être utilisée avec des types primitifs ou des unions, bien que les résultats puissent être moins intuitifs.

```
type StringOrNumber = string | number;
type StringAndNumber = string & number; // StringAndNumber est 'never'

// type MyType = { a: string } & string; // Erreur: Un type d'intersection ne peut pas inclure un type primitif directement comme ça.
```

L'intersection de types est principalement utile pour combiner des types d'objets ou pour ajouter des contraintes à des types existants. Elle est souvent utilisée pour créer des types plus spécifiques à partir de types plus généraux.

Exercice pour la section 'Intersection de types'

1. Créez une interface `Loggable` avec une méthode `log(message: string): void;`.
2. Créez une interface `Savable` avec une méthode `save(): void;`.
3. Définissez un type d'intersection `PersistentData` qui combine `Loggable` et `Savable`.
4. Créez une classe `Document` qui implémente `PersistentData`. Implémentez les méthodes `log` et `save` de manière simple (par exemple, en affichant un message dans la console).
5. Instanciez `Document` et appelez ses méthodes `log` et `save`.

3. Paramètres de fonction par défaut

En TypeScript (et JavaScript ES6), les paramètres de fonction par défaut permettent d'initialiser un paramètre avec une valeur par défaut si aucune valeur n'est fournie pour cet argument lors de l'appel de la fonction, ou si la valeur fournie est `undefined`. Cela simplifie le code en évitant d'avoir à écrire des logiques conditionnelles pour assigner des valeurs par défaut à l'intérieur du corps de la fonction.

3.1 Définition des paramètres par défaut

Pour définir un paramètre par défaut, vous utilisez la syntaxe `paramName: Type = defaultValue` dans la signature de la fonction.

Exemple 1 : Paramètre numérique par défaut

```
function greet(name: string, greeting: string = "Hello"): string {
  return `${greeting}, ${name}!";
}

console.log(greet("Alice"));          // "Hello, Alice!"
console.log(greet("Bob", "Hi"));      // "Hi, Bob!"
console.log(greet("Charlie", undefined)); // "Hello, Charlie!" (undefined
déclenche la valeur par défaut)
```

Dans cet exemple, si l'argument `greeting` n'est pas fourni ou est `undefined`, il prendra la valeur `

"Hello".

Exemple 2 : Paramètre objet par défaut

```
interface Config {
  port: number;
  host: string;
}

function startServer(config: Config = { port: 3000, host: "localhost" }) {
  console.log(`Server starting on ${config.host}:${config.port}`);
}

startServer(); // Server starting on localhost:3000
startServer({ port: 8080, host: "0.0.0.0" }); // Server starting on
0.0.0.0:8080
```

3.2 Ordre des paramètres

Les paramètres par défaut doivent être placés après tous les paramètres obligatoires. Si un paramètre par défaut est suivi d'un paramètre obligatoire, TypeScript générera une erreur de compilation.

```
// function example(a: number = 10, b: string) { } // Erreur: Un paramètre
obligatoire ne peut pas suivre un paramètre facultatif.

function example(a: number, b: string = "default") { }
```

3.3 Paramètres par défaut et paramètres rest

Les paramètres par défaut peuvent être combinés avec les paramètres rest (`...`), mais les paramètres rest doivent toujours être les derniers.

```
function logMessages(prefix: string = "LOG", ...messages: string[]) {
  messages.forEach(msg => console.log(`${prefix}: ${msg}`));
}

logMessages("INFO", "User logged in", "Data fetched");
// Output:
// INFO: User logged in
// INFO: Data fetched

logMessages(undefined, "System started", "Database connected"); // Utilise le
préfixe par défaut "LOG"
// Output:
// LOG: System started
// LOG: Database connected
```

Exercice pour la section 'Paramètres de fonction par défaut'

1. Créez une fonction `calculateArea` qui prend deux paramètres numériques : `length` et `width`. `width` doit avoir une valeur par défaut de `10`.
2. Appelez `calculateArea` avec seulement `length` et vérifiez le résultat.
3. Appelez `calculateArea` avec les deux paramètres et vérifiez le résultat.
4. Créez une fonction `createUser` qui prend un `name` (chaîne obligatoire) et un objet `options` avec des propriétés `isAdmin` (booléen, par défaut `false`) et `email` (chaîne, par défaut `""`).
5. Appelez `createUser` en fournissant différentes combinaisons d'options (aucune, seulement `isAdmin`, les deux).

4. Héritage

L'héritage est un concept fondamental de la programmation orientée objet (POO) qui permet à une classe (la classe enfant ou dérivée) d'acquérir les propriétés et les méthodes d'une autre classe (la classe parent ou de base). Cela favorise la réutilisation du code, la modularité et la création d'une hiérarchie de classes qui modélise les relations

« est un(e) » (is-a). En TypeScript, l'héritage est implémenté en utilisant le mot-clé `extends`.

4.1 Concepts de base de l'héritage

- **Classe de base (Parent Class / Superclass)** : La classe dont les propriétés et méthodes sont héritées.
- **Classe dérivée (Child Class / Subclass)** : La classe qui hérite de la classe de base. Elle peut ajouter de nouvelles propriétés et méthodes, ou remplacer (override) celles de la classe de base.
- **`super()`** : Dans le constructeur d'une classe dérivée, `super()` doit être appelé avant d'accéder à `this`. Il appelle le constructeur de la classe de base.

Exemple 1 : Héritage simple

```
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  breed: string;

  constructor(name: string, breed: string) {
    super(name); // Appelle le constructeur de la classe Animal
    this.breed = breed;
  }

  bark() {
    console.log("Woof! Woof!");
  }

  // Redéfinition (override) de la méthode move
  move(distanceInMeters: number = 5) {
    console.log(`${this.name} (a ${this.breed}) is running.`);
    super.move(distanceInMeters); // Appelle la méthode move de la classe
parente
  }
}

const dog = new Dog("Buddy", "Golden Retriever");
dog.bark();        // Woof! Woof!
dog.move();        // Buddy (a Golden Retriever) is running. Buddy moved 5m.
console.log(dog.name); // Buddy
```

Dans cet exemple, `Dog` hérite de `Animal`. `Dog` a accès à la propriété `name` et à la méthode `move` de `Animal`. Il ajoute également sa propre propriété `breed` et sa

méthode `bark`. La méthode `move` est redéfinie dans `Dog` pour fournir un comportement spécifique au chien, tout en appelant la méthode `move` de la classe parente via `super.move()`.

4.2 Modificateurs d'accès et héritage

Les modificateurs d'accès (`public`, `private`, `protected`) jouent un rôle important dans l'héritage :

- **`public`** : Les membres sont accessibles de partout.
- **`private`** : Les membres sont accessibles uniquement à l'intérieur de la classe où ils sont déclarés. Ils ne sont pas accessibles par les classes dérivées.
- **`protected`** : Les membres sont accessibles à l'intérieur de la classe où ils sont déclarés et par les classes dérivées. Ils ne sont pas accessibles de l'extérieur de la hiérarchie de classes.

Exemple 2 : Modificateurs d'accès

```
class Vehicle {
  public brand: string;
  protected speed: number;
  private chassisNumber: string;

  constructor(brand: string, speed: number, chassisNumber: string) {
    this.brand = brand;
    this.speed = speed;
    this.chassisNumber = chassisNumber;
  }

  protected accelerate(amount: number) {
    this.speed += amount;
    console.log(`${this.brand} current speed: ${this.speed} km/h.`);
  }

  public getChassisNumber(): string {
    return this.chassisNumber; // Accès autorisé car à l'intérieur de la classe
  }
}

class Car extends Vehicle {
  public model: string;

  constructor(brand: string, model: string, speed: number, chassisNumber:
string) {
    super(brand, speed, chassisNumber);
    this.model = model;
  }

  drive() {
    console.log(`Driving the ${this.brand} ${this.model}.`);
    this.accelerate(50); // Accès autorisé car accelerate est protected
    // console.log(this.chassisNumber); // Erreur: chassisNumber est private
  }
}

const myCar = new Car("Toyota", "Corolla", 0, "XYZ123");
console.log(myCar.brand); // Toyota (public)
myCar.drive();
// console.log(myCar.speed); // Erreur: speed est protected
// console.log(myCar.chassisNumber); // Erreur: chassisNumber est private
console.log(myCar.getChassisNumber()); // XYZ123 (via méthode publique)
```

4.3 Héritage et interfaces

Les classes peuvent hériter d'une seule classe de base, mais elles peuvent implémenter plusieurs interfaces. Cela permet de combiner l'héritage de comportement (via la classe de base) avec l'héritage de contrat (via les interfaces).

Exemple 3 : Héritage et implémentation d'interface

```
interface Flyable {
  fly(): void;
}

class Bird extends Animal implements Flyable {
  constructor(name: string) {
    super(name);
  }

  fly() {
    console.log(`${this.name} is flying high!`);
  }
}

const eagle = new Bird("Eagle");
eagle.move(10); // Eagle moved 10m.
eagle.fly();    // Eagle is flying high!
```

Exercice pour la section 'Héritage'

1. Créez une classe de base `Shape` avec une propriété `color` (chaîne de caractères) et une méthode `getArea()` qui retourne `0` par défaut.
2. Créez une classe `Circle` qui hérite de `Shape`. Ajoutez une propriété `radius` (nombre) et redéfinissez la méthode `getArea()` pour calculer l'aire d'un cercle (`Math.PI * radius * radius`).
3. Créez une classe `Rectangle` qui hérite de `Shape`. Ajoutez des propriétés `width` et `height` (nombres) et redéfinissez `getArea()` pour calculer l'aire d'un rectangle (`width * height`).
4. Instanciez un `Circle` et un `Rectangle`, définissez leurs propriétés et affichez leurs aires et couleurs.

5. Propriété de discrimination

La propriété de discrimination (ou discriminant property) est un concept clé en TypeScript, particulièrement utile lorsqu'on travaille avec des types d'union. Elle permet à TypeScript de réduire (narrow) un type d'union à un sous-type plus spécifique en se basant sur la valeur d'une propriété commune à tous les membres de l'union. Cette propriété commune est appelée la "propriété discriminante" ou "tag".

5.1 Fonctionnement de la propriété de discrimination

Pour qu'une propriété serve de discriminant, elle doit remplir les conditions suivantes :

1. **Propriété commune** : La propriété doit exister dans tous les membres du type d'union.
2. **Type littéral de chaîne/numérique** : Le type de cette propriété doit être un type littéral de chaîne de caractères ou numérique unique pour chaque membre de l'union. Cela permet à TypeScript de distinguer les membres de l'union en fonction de la valeur de cette propriété.

Lorsque ces conditions sont remplies, TypeScript peut utiliser des vérifications conditionnelles (comme `if` ou `switch`) sur la propriété discriminante pour affiner le type de l'objet au sein du bloc de code correspondant.

Exemple 1 : Formes géométriques

Considérons un type d'union pour différentes formes géométriques :

```typescript
interface Circle {
  kind: "circle"; // Propriété discriminante
  radius: number;
}

interface Square {
  kind: "square"; // Propriété discriminante
  sideLength: number;
}

interface Triangle {
  kind: "triangle"; // Propriété discriminante
  base: number;
  height: number;
}

type Shape = Circle | Square | Triangle;

function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      // Ici, TypeScript sait que 'shape' est de type Circle
      return Math.PI * shape.radius ** 2;
    case "square":
      // Ici, TypeScript sait que 'shape' est de type Square
      return shape.sideLength ** 2;
    case "triangle":
      // Ici, TypeScript sait que 'shape' est de type Triangle
      return 0.5 * shape.base * shape.height;
    default:
      // Cette partie est utile pour s'assurer que tous les cas sont gérés
      // Si un nouveau type est ajouté à Shape et non géré ici, TypeScript
signalera une erreur
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}

const myCircle: Circle = { kind: "circle", radius: 10 };
const mySquare: Square = { kind: "square", sideLength: 5 };
const myTriangle: Triangle = { kind: "triangle", base: 4, height: 6 };

console.log(`Area of circle: ${getArea(myCircle)}`);     // Area of circle:
314.159...
console.log(`Area of square: ${getArea(mySquare)}`);     // Area of square: 25
console.log(`Area of triangle: ${getArea(myTriangle)}`); // Area of triangle:
12
```

Dans cet exemple, la propriété `kind` est la propriété discriminante. Sa valeur littérale (`"circle"`, `"square"`, `"triangle"`) permet à TypeScript de déterminer le type exact de `shape` à l'intérieur de chaque `case` du `switch`.

5.2 Avantages

- **Sécurité des types** : Permet à TypeScript de fournir une vérification de type rigoureuse et d'éviter les erreurs d'exécution potentielles en s'assurant que vous accédez aux propriétés correctes pour chaque sous-type.
- **Amélioration de l'autocomplétion** : Dans les éditeurs de code, l'autocomplétion sera plus précise car TypeScript connaît le type exact de l'objet.
- **Code plus lisible et maintenable** : Rend le code plus clair et plus facile à comprendre, car la logique de traitement est explicitement liée au type de l'objet.
- **Vérification d'exhaustivité** : L'utilisation de `const _exhaustiveCheck: never = shape;` dans le `default` d'un `switch` permet à TypeScript de s'assurer que tous les cas possibles du type d'union sont gérés. Si un nouveau type est ajouté à l'union `Shape` et n'est pas géré dans le `switch`, TypeScript générera une erreur de compilation, vous aidant à maintenir votre code à jour.

Exercice pour la section 'Propriété de discrimination'

1. Définissez un type d'union `Notification` qui peut être soit `EmailNotification` soit `SMSNotification`.
2. `EmailNotification` doit avoir une propriété discriminante `type: "email"`, un `subject: string`, et un `body: string`.
3. `SMSNotification` doit avoir une propriété discriminante `type: "sms"`, un `phoneNumber: string`, et un `message: string`.
4. Créez une fonction `sendNotification` qui prend un argument de type `Notification`. Utilisez une instruction `switch` sur la propriété discriminante `type` pour afficher un message différent en fonction du type de notification (par exemple, "Sending email with subject..." ou "Sending SMS to...").
5. Testez la fonction `sendNotification` avec des instances de `EmailNotification` et `SMSNotification`.

6. 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 (`|`).

6.1 Définition des types d'union

Un type d'union est formé en combinant deux ou plusieurs types avec l'opérateur `|`. La variable typée avec un type d'union peut contenir n'importe quelle valeur qui correspond à l'un des types de l'union.

Exemple 1 : Union de types primitifs

```
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'.
```

Exemple 2 : Union de types d'objets

```typescript
interface Dog {
  name: string;
  bark(): void;
}

interface Cat {
  name: string;
  meow(): void;
}

type Pet = Dog | Cat;

function makeSound(pet: Pet) {
  if ('bark' in pet) {
    pet.bark(); // TypeScript sait que 'pet' est de type Dog ici
  } else if ('meow' in pet) {
    pet.meow(); // TypeScript sait que 'pet' est de type Cat ici
  }
}

const myDog: Dog = { name: "Buddy", bark: () => console.log("Woof!") };
const myCat: Cat = { name: "Whiskers", meow: () => console.log("Meow!") };

makeSound(myDog); // Woof!
makeSound(myCat); // Meow!
```

Dans cet exemple, `Pet` peut être soit un `Dog` soit un `Cat`. La fonction `makeSound` utilise le rétrécissement de type (`'bark' in pet`) pour déterminer le type exact de `pet` à l'exécution et appeler la méthode appropriée.

6.2 Rétrécissement de type (Type Narrowing)

Lorsque vous travaillez avec des types d'union, TypeScript a besoin d'aide pour déterminer le type exact d'une variable à un moment donné. C'est là qu'intervient le rétrécissement de type. TypeScript utilise des constructions de code comme les instructions `if`, `switch`, `typeof`, `instanceof`, et les propriétés discriminantes pour affiner le type d'une variable à l'intérieur d'un bloc de code.

Exemple 3 : Rétrécissement avec `typeof`

```
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"); // MYID
printId(123);   // 246
```

Exemple 4 : Rétrécissement avec `instanceof`

```
class Car {
  drive() { console.log("Driving a car"); }
}

class Bicycle {
  pedal() { console.log("Pedaling a bicycle"); }
}

type Vehicle = Car | Bicycle;

function startVehicle(vehicle: Vehicle) {
  if (vehicle instanceof Car) {
    vehicle.drive(); // TypeScript sait que 'vehicle' est une Car ici
  } else {
    vehicle.pedal(); // TypeScript sait que 'vehicle' est une Bicycle ici
  }
}

startVehicle(new Car());     // Driving a car
startVehicle(new Bicycle()); // Pedaling a bicycle
```

Exemple 5 : Rétrécissement avec propriété discriminante (voir section 5)

```typescript
interface SuccessResponse {
  status: "success";
  data: any;
}

interface ErrorResponse {
  status: "error";
  message: string;
}

type APIResponse = SuccessResponse | ErrorResponse;

function handleResponse(response: APIResponse) {
  if (response.status === "success") {
    console.log("Data received:", response.data); // 'response' est
SuccessResponse
  } else {
    console.log("Error occurred:", response.message); // 'response' est
ErrorResponse
  }
}

handleResponse({ status: "success", data: { user: "Alice" } });
handleResponse({ status: "error", message: "Failed to fetch data" });
```

Exercice pour la section 'Type union'

1. Créez un type d'union `InputType` qui peut être `string`, `number`, ou `boolean`.
2. Écrivez une fonction `processInput` qui prend un argument de type `InputType`.
3. À l'intérieur de `processInput`, utilisez des vérifications de type (`typeof`) pour :
    - Si l'entrée est une `string`, affichez-la en majuscules.
    - Si l'entrée est un `number`, affichez le double de sa valeur.
    - Si l'entrée est un `boolean`, affichez

son inverse. 4. Testez `processInput` avec différentes valeurs de `InputType`.

7. Attributs et méthodes de classe

En TypeScript, les classes sont des modèles pour créer des objets. Elles encapsulent des données (attributs ou propriétés) et des fonctionnalités (méthodes) qui opèrent sur ces données. Comprendre comment définir et utiliser les attributs et les méthodes est fondamental pour la programmation orientée objet en TypeScript.

7.1 Attributs (Propriétés) de classe

Les attributs sont les variables qui stockent l'état d'un objet. Ils sont déclarés à l'intérieur de la classe et peuvent être initialisés directement ou via le constructeur.

Modificateurs d'accès :

TypeScript fournit des modificateurs d'accès pour contrôler la visibilité des attributs et des méthodes :

- `public` **(par défaut)** : Accessible de partout (à l'intérieur et à l'extérieur de la classe, y compris les classes dérivées).
- `private` : Accessible uniquement à l'intérieur de la classe où il est déclaré. Non accessible par les instances de la classe ou les classes dérivées.
- `protected` : Accessible à l'intérieur de la classe où il est déclaré et par les classes dérivées. Non accessible par les instances de la classe.
- `readonly` : Un attribut `readonly` peut être initialisé lors de sa déclaration ou dans le constructeur de la classe. Après cela, sa valeur ne peut plus être modifiée.

Exemple 1 : Déclaration d'attributs avec modificateurs d'accès

```
class Car {
  public brand: string; // Accessible de partout
  private _speed: number; // Accessible uniquement à l'intérieur de la classe
  protected color: string; // Accessible dans la classe et les classes dérivées
  readonly year: number; // Peut être initialisé une seule fois

  constructor(brand: string, initialSpeed: number, color: string, year: number)
{
    this.brand = brand;
    this._speed = initialSpeed;
    this.color = color;
    this.year = year;
  }

  public accelerate(amount: number) {
    this._speed += amount;
    console.log(`$`{this.brand} is accelerating. Current speed: `${this._speed}
km/h.`);
  }

  protected displayColor() {
    console.log(`The car color is ${this.color}.`);
  }

  // Getter pour accéder à la propriété privée _speed
  get speed(): number {
    return this._speed;
  }

  // Setter pour modifier la propriété privée _speed
  set speed(newSpeed: number) {
    if (newSpeed >= 0) {
      this._speed = newSpeed;
    } else {
      console.log("Speed cannot be negative.");
    }
  }
}

const myCar = new Car("Toyota", 0, "Red", 2023);
console.log(myCar.brand); // Toyota (public)
myCar.accelerate(50);
console.log(myCar.speed); // 50 (via getter)
myCar.speed = 100;
console.log(myCar.speed); // 100
// console.log(myCar._speed); // Erreur: _speed est private
// myCar.displayColor(); // Erreur: displayColor est protected
// myCar.year = 2024; // Erreur: year est readonly
```

7.2 Méthodes de classe

Les méthodes sont des fonctions définies à l'intérieur d'une classe qui effectuent des opérations sur les attributs de l'objet ou fournissent des fonctionnalités liées à l'objet. Elles peuvent également utiliser les modificateurs d'accès (`public`, `private`, `protected`).

Méthodes statiques :

Les méthodes statiques appartiennent à la classe elle-même, et non aux instances de la classe. Elles sont appelées directement sur la classe et ne peuvent pas accéder aux attributs ou méthodes d'instance (non statiques) de la classe. Elles sont souvent utilisées pour des fonctions utilitaires qui ne dépendent pas de l'état d'un objet spécifique.

Exemple 2 : Méthodes d'instance et statiques

```
class Calculator {
  // Méthode d'instance
  add(a: number, b: number): number {
    return a + b;
  }

  // Méthode statique
  static multiply(a: number, b: number): number {
    return a * b;
  }
}

const calc = new Calculator();
console.log(calc.add(5, 3)); // 8 (appel sur l'instance)

console.log(Calculator.multiply(4, 2)); // 8 (appel sur la classe)
// console.log(calc.multiply(4, 2)); // Erreur: multiply n'est pas une méthode
d'instance
```

7.3 Propriétés de paramètre dans le constructeur

TypeScript offre une syntaxe raccourcie pour déclarer et initialiser des attributs de classe directement dans le constructeur. En préfixant un paramètre du constructeur avec un modificateur d'accès (`public` , `private` , `protected` , `readonly`), TypeScript créera automatiquement un attribut de classe avec le même nom et le même modificateur d'accès, et assignera la valeur du paramètre à cet attribut.

Exemple 3 : Propriétés de paramètre

```
class Person {
  constructor(public name: string, private age: number, protected city: string)
{
    // Pas besoin de faire this.name = name; etc.
  }

  public getAge(): number {
    return this.age;
  }

  protected getCity(): string {
    return this.city;
  }
}

const person1 = new Person("Alice", 30, "Paris");
console.log(person1.name); // Alice
console.log(person1.getAge()); // 30
// console.log(person1.age); // Erreur: age est private
```

Cette syntaxe est très pratique pour réduire le code répétitif et rendre les classes plus concises.

Exercice pour la section 'Attributs et méthodes de classe'

1. Créez une classe `BankAccount` avec les attributs suivants :
   - `accountNumber` (chaîne de caractères, `readonly`)
   - `_balance` (nombre, `private`)
   - `ownerName` (chaîne de caractères, `public`)
2. Implémentez un constructeur qui initialise `accountNumber` et `ownerName`, et initialise `_balance` à `0`.
3. Ajoutez les méthodes suivantes :
   - `deposit(amount: number)` : ajoute `amount` au solde. Doit être `public`.
   - `withdraw(amount: number)` : retire `amount` du solde si suffisant. Doit être `public`.
   - `getBalance()` : retourne le solde actuel. Doit être un getter `public`.
   - `static generateAccountNumber()` : une méthode statique qui retourne une chaîne de caractères unique (par exemple, un nombre aléatoire converti en chaîne).
4. Créez une instance de `BankAccount` en utilisant la méthode statique pour générer le numéro de compte.

5. Effectuez un dépôt, puis un retrait, et affichez le solde après chaque opération.

8. Paramètres callback

En TypeScript (et JavaScript), une fonction de rappel (callback function) est une fonction passée en argument à une autre fonction, qui sera exécutée plus tard. Les callbacks sont fondamentaux pour la programmation asynchrone et pour la personnalisation du comportement des fonctions. TypeScript améliore l'utilisation des callbacks en permettant de typer précisément leurs signatures, ce qui assure la sécurité des types et une meilleure autocomplétion.

8.1 Définition et typage des callbacks

Lorsque vous définissez une fonction qui accepte un callback, il est crucial de spécifier la signature attendue du callback. Cela inclut le nombre et le type des arguments que le callback recevra, ainsi que le type de sa valeur de retour.

Exemple 1 : Callback simple

```
function processData(data: string[], callback: (item: string) => void) {
  for (const item of data) {
    callback(item); // Exécute le callback pour chaque élément
  }
}

// Utilisation de la fonction processData avec un callback
processData(["apple", "banana", "cherry"], (fruit: string) => {
  console.log(`Processing: ${fruit.toUpperCase()}`);
});
// Output:
// Processing: APPLE
// Processing: BANANA
// Processing: CHERRY

// Un callback avec une signature incorrecte générera une erreur de type
// processData(["grape"], (item: number) => console.log(item)); // Erreur: Type
'(item: number) => void' n'est pas assignable au type '(item: string) => void'.
```

Dans cet exemple, la fonction `processData` attend un `callback` qui prend un `string` en argument et ne retourne rien (`void`). TypeScript vérifiera que le callback fourni respecte cette signature.

8.2 Callbacks asynchrones

Les callbacks sont très souvent utilisés dans des opérations asynchrones, comme les requêtes réseau, les lectures de fichiers ou les temporisateurs.

Exemple 2 : Callback asynchrone avec `setTimeout`

```
 function fetchData(url: string, callback: (data: string, error?: string) =>
void) {
  console.log(`Fetching data from ${url}...`);
  setTimeout(() => {
    if (url === "success-api") {
      callback("Data from server", undefined); // Simule une réponse réussie
    } else {
      callback("", "Network error"); // Simule une erreur
    }
  }, 1000);
}

fetchData("success-api", (data, error) => {
  if (error) {
    console.error(`Error: ${error}`);
  } else {
    console.log(`Received: ${data}`);
  }
});
// Output (après 1 seconde):
// Received: Data from server

fetchData("error-api", (data, error) => {
  if (error) {
    console.error(`Error: ${error}`);
  } else {
    console.log(`Received: ${data}`);
  }
});
// Output (après 1 seconde):
// Error: Network error
```

Ici, le callback `(data: string, error?: string) => void` permet de gérer à la fois les données réussies et les erreurs potentielles, ce qui est une pratique courante dans les APIs asynchrones.

8.3 Définir des types de fonction pour les callbacks

Pour des callbacks plus complexes ou réutilisables, il est souvent préférable de définir un alias de type pour la signature de la fonction.

Exemple 3 : Alias de type pour un callback

```typescript
type TransformerCallback = (value: number) => number;

function applyTransformation(numbers: number[], transform:
TransformerCallback): number[] {
  return numbers.map(transform);
}

const result = applyTransformation([1, 2, 3], (num) => num * 2);
console.log(result); // [2, 4, 6]

const result2 = applyTransformation([10, 20, 30], (num) => num + 5);
console.log(result2); // [15, 25, 35]
```

L'utilisation d'alias de type rend le code plus lisible et facilite la réutilisation des signatures de fonction.

Exercice pour la section 'Paramètres callback'

1. Créez une fonction `filterArray` qui prend deux arguments :
   - `arr` : un tableau de nombres (`number[]`).
   - `predicate` : une fonction de rappel qui prend un nombre (`num: number`) et retourne un booléen (`boolean`).
2. La fonction `filterArray` doit retourner un nouveau tableau contenant uniquement les nombres pour lesquels le `predicate` retourne `true` .
3. Utilisez `filterArray` pour filtrer un tableau de nombres afin de ne garder que les nombres pairs.
4. Utilisez `filterArray` pour filtrer le même tableau afin de ne garder que les nombres supérieurs à 10.

9. Fonction générique

Les fonctions génériques en TypeScript permettent d'écrire des fonctions qui peuvent travailler avec une variété de types de données, sans sacrifier la sécurité des types. Elles sont particulièrement utiles lorsque vous voulez que votre fonction opère sur des données de n'importe quel type, tout en conservant la relation entre le type d'entrée et le type de sortie, ou entre différents arguments.

9.1 Introduction aux génériques

Sans génériques, si vous vouliez une fonction qui retourne l'identité d'un argument (c'est-à-dire, retourne l'argument tel quel), vous pourriez être tenté d'utiliser `any` :

```
function identityAny(arg: any): any {
  return arg;
}

let outputAny = identityAny("myString"); // type de outputAny est any
console.log(outputAny.length); // OK, mais pas de vérification de type
outputAny = identityAny(123); // type de outputAny est any
// console.log(outputAny.length); // Erreur à l'exécution, mais pas à la
compilation
```

Le problème avec `any` est qu'il désactive la vérification de type, ce qui peut masquer des erreurs potentielles. Les génériques résolvent ce problème en vous permettant de capturer le type de l'argument et de l'utiliser pour typer la valeur de retour, ou d'autres parties de la fonction.

9.2 Définition de fonctions génériques

Pour rendre une fonction générique, vous utilisez une variable de type, généralement `T` (pour Type), placée entre des chevrons (`< >`) après le nom de la fonction.

Exemple 1 : Fonction `identity` générique

```
function identity<T>(arg: T): T {
  return arg;
}

let outputString = identity<string>("myString"); // type de outputString est
string
console.log(outputString.length); // OK, vérification de type

let outputNumber = identity<number>(123); // type de outputNumber est number
// console.log(outputNumber.length); // Erreur à la compilation: Property
'length' does not exist on type 'number'.

// TypeScript peut souvent inférer le type, donc l'annotation explicite n'est
pas toujours nécessaire
let inferredString = identity("anotherString"); // type de inferredString est
string
let inferredNumber = identity(456); // type de inferredNumber est number
```

Dans cet exemple, `T` agit comme un espace réservé pour le type. Lorsque vous appelez `identity<string>`, `T` devient `string` pour cette invocation. TypeScript

s'assure alors que l'argument est une chaîne et que la valeur de retour est également une chaîne.

9.3 Contraintes de type génériques

Parfois, vous voulez que votre fonction générique opère sur une variété de types, mais vous avez besoin que ces types aient certaines propriétés. Par exemple, si vous voulez une fonction qui prend un tableau et retourne sa longueur, vous avez besoin que le type générique ait une propriété `length`. Vous pouvez ajouter des contraintes de type en utilisant le mot-clé `extends`.

Exemple 2 : Contrainte de type avec `length`

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // OK, car T est contraint d'avoir une propriété
length
  return arg;
}

loggingIdentity("hello"); // OK, string a une propriété length
loggingIdentity([1, 2, 3]); // OK, array a une propriété length
// loggingIdentity(3); // Erreur: Argument of type 'number' is not assignable
to parameter of type 'Lengthwise'.
```

9.4 Utilisation de plusieurs variables de type

Vous pouvez utiliser plusieurs variables de type génériques si votre fonction opère sur plusieurs types qui doivent être liés entre eux.

Exemple 3 : Fonction `zip` avec deux types génériques

```typescript
function zip<T, U>(arr1: T[], arr2: U[]): [T, U][] {
  const result: [T, U][] = [];
  const minLength = Math.min(arr1.length, arr2.length);
  for (let i = 0; i < minLength; i++) {
    result.push([arr1[i], arr2[i]]);
  }
  return result;
}

const zipped = zip([1, 2, 3], ["a", "b", "c"]);
console.log(zipped); // [[1, "a"], [2, "b"], [3, "c"]]
// Le type de zipped est (number | string)[][], ou plus précisément [number, string][]

const zippedMixed = zip([true, false], [10, 20, 30]);
console.log(zippedMixed); // [[true, 10], [false, 20]]
// Le type de zippedMixed est [boolean, number][]
```

9.5 Fonctions génériques et types de fonction

Vous pouvez également définir des types de fonction génériques.

Exemple 4 : Type de fonction générique

```typescript
type GenericIdentityFn<T> = (arg: T) => T;

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;
console.log(myIdentity(100)); // 100
// let anotherIdentity: GenericIdentityFn<string> = identity;
// console.log(anotherIdentity(true)); // Erreur: Argument of type 'boolean' is not assignable to parameter of type 'string'.
```

Les fonctions génériques sont un pilier de la programmation robuste et réutilisable en TypeScript, permettant de créer des composants qui fonctionnent de manière cohérente avec n'importe quel type de données, tout en maintenant une forte vérification de type.

Exercice pour la section 'Fonction générique'

1. Créez une fonction générique `getFirstElement<T>(arr: T[]): T | undefined` qui prend un tableau de n'importe quel type `T` et retourne le premier élément du tableau, ou `undefined` si le tableau est vide.

2. Testez `getFirstElement` avec un tableau de chaînes de caractères et un tableau de nombres.
3. Créez une interface `HasId` avec une propriété `id: number`.
4. Créez une fonction générique `findById<T extends HasId>(items: T[], id: number): T | undefined` qui prend un tableau d'objets (qui doivent avoir une propriété `id`) et un `id` numérique, et retourne l'objet correspondant ou `undefined`.
5. Testez `findById` avec un tableau d'objets `User` (où `User` a une propriété `id`) et un `id`.

10. Décorateurs de propriété

Les décorateurs de propriété sont une fonctionnalité expérimentale de TypeScript qui permet d'attacher des métadonnées ou de modifier le comportement d'une propriété de classe. Ils sont appliqués directement au-dessus de la déclaration d'une propriété.

Pour utiliser les décorateurs, vous devez activer l'option `experimentalDecorators` dans votre fichier `tsconfig.json` :

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true // Souvent utilisé avec
experimentalDecorators
  }
}
```

10.1 Fonctionnement des décorateurs de propriété

Un décorateur de propriété est une fonction qui est appelée au moment de la déclaration de la propriété. Il reçoit deux arguments :

1. `target` : Le prototype de la classe pour un membre d'instance, ou la fonction constructeur pour un membre statique.
2. `propertyKey` : Le nom de la propriété.

Contrairement aux décorateurs de méthode ou d'accesseur, les décorateurs de propriété ne reçoivent pas de descripteur de propriété. Cela signifie qu'ils ne peuvent

pas directement modifier la valeur de la propriété au moment de la décoration. Leur rôle principal est d'enregistrer des métadonnées sur la propriété, qui peuvent ensuite être lues au moment de l'exécution pour implémenter des logiques spécifiques, comme la validation, la sérialisation/désérialisation, ou l'injection de dépendances.

Cependant, un décorateur de propriété peut retourner un nouveau descripteur de propriété si la cible est un membre d'instance et que le compilateur cible est ES5 ou supérieur. Cela permet de modifier le comportement de la propriété (par exemple, en ajoutant des getters/setters personnalisés).

Exemple 1 : Décorateur de propriété simple pour les métadonnées

Ce décorateur enregistre simplement le nom de la propriété.

```
function LogProperty(target: any, propertyKey: string) {
  console.log(`Property ${propertyKey} decorated on:`, target);
}

class Product {
  @LogProperty
  name: string;

  price: number;

  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
}

const p = new Product("Laptop", 1200);
// Output (au moment de la déclaration de la classe, pas de l'instance):
// Property name decorated on: { constructor: [Function: Product] }
```

Cet exemple montre que le décorateur est exécuté au moment de la définition de la classe, pas lorsque l'instance est créée. Il peut être utilisé pour collecter des informations sur les propriétés.

Exemple 2 : Décorateur de propriété avec modification du comportement (via descripteur)

Ce décorateur va s'assurer qu'une propriété numérique ne peut pas être définie avec une valeur négative.

```
function NonNegative(target: any, propertyKey: string) {
  let value: number;

  const getter = function (this: any) {
    return value;
  };

  const setter = function (this: any, newValue: number) {
    if (newValue < 0) {
      throw new Error(`Value for ${propertyKey} cannot be negative.`);
    }
    value = newValue;
  };

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

class Item {
  @NonNegative
  price: number;

  constructor(price: number) {
    this.price = price;
  }
}

const item1 = new Item(100);
console.log(item1.price); // 100

item1.price = 50;
console.log(item1.price); // 50

try {
  item1.price = -10; // Cela déclenchera l'erreur
} catch (error: any) {
  console.error(error.message); // Output: Value for price cannot be negative.
}

const item2 = new Item(20);
console.log(item2.price); // 20
```

Dans cet exemple, le décorateur `NonNegative` remplace le getter et le setter par défaut de la propriété `price`. Le nouveau setter inclut une logique de validation qui empêche l'affectation de valeurs négatives.

10.2 Cas d'utilisation courants

Les décorateurs de propriété sont souvent utilisés pour :

- **Validation** : Ajouter des règles de validation aux propriétés (comme l'exemple `NonNegative`).
- **Sérialisation/Désérialisation** : Marquer les propriétés pour qu'elles soient incluses ou exclues lors de la conversion d'objets en JSON ou vice versa.
- **Injection de dépendances** : Marquer les propriétés où des dépendances doivent être injectées (comme dans les frameworks Angular ou NestJS).
- **Métadonnées** : Attacher des informations supplémentaires à une propriété qui peuvent être lues par d'autres parties de l'application ou des bibliothèques.

Exercice pour la section 'Décorateurs de propriété'

1. Créez un décorateur de propriété `Required` qui, lorsqu'il est appliqué à une propriété de type `string`, s'assure que la propriété n'est pas vide ou `null`/`undefined` lors de l'initialisation de l'objet ou de l'affectation.
    - Si la valeur est invalide, lancez une erreur.
2. Créez une classe `User` avec une propriété `username: string` décorée avec `@Required`.
3. Testez la création d'un `User` avec un `username` valide et un `username` vide ou `null` pour voir l'erreur.

11. Type unknown

Le type `unknown` a été introduit en TypeScript 3.0 comme une alternative plus sûre au type `any`. Alors que `any` permet d'ignorer complètement la vérification de type (ce qui peut masquer des erreurs), `unknown` exige que vous effectuiez une vérification de type ou un affinement de type avant de pouvoir interagir avec la valeur. Cela rend `unknown` très utile pour les situations où vous ne connaissez pas le type d'une valeur à l'avance, mais où vous voulez garantir la sécurité des types.

11.1 `unknown` vs `any`

La principale différence entre `unknown` et `any` réside dans le fait que `unknown` est un type sûr. Vous ne pouvez pas effectuer d'opérations arbitraires sur une valeur de type `unknown` sans d'abord affiner son type. Avec `any`, toutes les opérations sont

autorisées, ce qui peut entraîner des erreurs d'exécution non détectées au moment de la compilation.

Exemple 1 : Comparaison `unknown` et `any`

```
 let value: unknown;
value = "hello";
value = 123;
value = true;

let valueAny: any;
valueAny = "hello";
valueAny = 123;

// Opérations avec unknown (nécessite un affinement de type)
if (typeof value === "string") {
  console.log(value.toUpperCase()); // OK, value est affiné à string
}
// console.log(value.toFixed(2)); // Erreur: Object is of type 'unknown'.

// Opérations avec any (aucune vérification de type)
console.log(valueAny.toUpperCase()); // OK à la compilation, peut échouer à
l'exécution si valueAny n'est pas une chaîne
console.log(valueAny.toFixed(2));    // OK à la compilation, peut échouer à
l'exécution si valueAny n'est pas un nombre
```

11.2 Affinement de type avec `unknown`

Pour travailler avec une valeur de type `unknown`, vous devez utiliser des techniques d'affinement de type (type narrowing) pour prouver à TypeScript quel est le type réel de la valeur. Les méthodes courantes d'affinement de type incluent :

- `typeof` (pour les types primitifs)
- `instanceof` (pour les instances de classes)
- Assertions de type (`as Type`)
- Vérifications de propriétés (`'prop' in obj`)
- Fonctions de garde de type (type guards)

Exemple 2 : Affinement de type avec `typeof` et `instanceof`

```
function processUnknown(input: unknown) {
  if (typeof input === "string") {
    console.log(`String: ${input.length}`);
  } else if (typeof input === "number") {
    console.log(`Number: ${input * 2}`);
  } else if (input instanceof Date) {
    console.log(`Date: ${input.toDateString()}`);
  } else {
    console.log("Unknown type");
  }
}

processUnknown("TypeScript"); // String: 10
processUnknown(42);           // Number: 84
processUnknown(new Date()); // Date: Thu Jul 31 2025
processUnknown({});           // Unknown type
```

Exemple 3 : Assertion de type (à utiliser avec prudence)

L'assertion de type (`as Type`) indique à TypeScript que vous savez mieux que lui quel est le type d'une valeur. Elle ne fait aucune vérification au moment de l'exécution et doit être utilisée uniquement lorsque vous êtes absolument certain du type.

```
let data: unknown = "This is a string";

// Utilisation de l'assertion de type
const len = (data as string).length;
console.log(len); // 16

let numData: unknown = 123.456;
const fixedNum = (numData as number).toFixed(2);
console.log(fixedNum); // "123.46"

// Si l'assertion est incorrecte, cela peut entraîner des erreurs d'exécution
let badData: unknown = true;
// const badLen = (badData as string).length; // Aucune erreur à la
compilation, mais échouera à l'exécution
// console.log(badLen);
```

11.3 Cas d'utilisation de `unknown`

`unknown` est idéal pour les situations où vous recevez des données de sources externes (par exemple, API, saisie utilisateur) dont le type n'est pas garanti. Il vous force à valider le type avant d'utiliser les données, ce qui rend votre code plus robuste.

- **Parsing JSON** : Lorsque vous parsez une chaîne JSON, le résultat est souvent de type `any` ou `unknown` .
- **Gestion des erreurs** : Le type des erreurs dans les blocs `catch` est `unknown` par défaut en TypeScript 4.4+.

```
try {
  // Code qui pourrait lancer une erreur
  throw new Error("Something went wrong");
} catch (err: unknown) {
  if (err instanceof Error) {
    console.error(`Caught an Error: ${err.message}`);
  } else if (typeof err === "string") {
    console.error(`Caught a string error: ${err}`);
  } else {
    console.error("Caught an unknown error");
  }
}
```

En résumé, `unknown` est un type puissant qui vous aide à écrire du code plus sûr et plus prévisible en vous obligeant à gérer explicitement les types de données inconnus.

Exercice pour la section 'Type unknown'

1. Créez une fonction `safeParseJSON(jsonString: string): unknown` qui utilise `JSON.parse` pour parser une chaîne JSON. Gérez les erreurs de parsing en retournant `undefined` si le parsing échoue.
2. Créez une fonction `processParsedData(data: unknown)`.
3. À l'intérieur de `processParsedData`, utilisez des affinements de type pour :
   - Si `data` est un objet avec une propriété `name` de type `string`, affichez `Hello, [name]!`.
   - Si `data` est un tableau de nombres, affichez la somme des nombres.
   - Pour tout autre type, affichez `Data format not recognized.`.
4. Testez `safeParseJSON` et `processParsedData` avec :
   - Une chaîne JSON valide représentant un objet avec `name`.
   - Une chaîne JSON valide représentant un tableau de nombres.
   - Une chaîne JSON invalide.
   - Une chaîne JSON valide représentant un autre type (par exemple, un booléen).

12. Tableau en lecture seule

En TypeScript, il est souvent souhaitable de s'assurer que certaines structures de données ne peuvent pas être modifiées après leur création. Pour les tableaux, TypeScript offre le type `ReadonlyArray<T>` ou la syntaxe raccourcie `readonly T[]` pour créer des tableaux en lecture seule. Cela signifie que vous ne pouvez pas ajouter, supprimer ou modifier des éléments du tableau une fois qu'il a été initialisé. Cette fonctionnalité est cruciale pour l'immuabilité des données, ce qui peut prévenir des bugs et rendre le code plus prévisible, en particulier dans les applications où l'état est géré de manière immuable (par exemple, dans Redux).

12.1 Définition d'un tableau en lecture seule

Vous pouvez définir un tableau en lecture seule de deux manières :

1. **`ReadonlyArray<T>`** : Un type générique qui indique que le tableau est en lecture seule.
2. **`readonly T[]`** : Une syntaxe plus concise, équivalente à `ReadonlyArray<T>`.

Exemple 1 : Déclaration et utilisation

```
const numbers: ReadonlyArray<number> = [1, 2, 3];
// numbers.push(4); // Erreur: Property 'push' does not exist on type 'readonly
number[]'.
// numbers[0] = 10; // Erreur: Index signature in type 'readonly number[]' only
permits reading.

const names: readonly string[] = ["Alice", "Bob", "Charlie"];
// names.pop(); // Erreur: Property 'pop' does not exist on type 'readonly
string[]'.
// names[1] = "Bobby"; // Erreur: Index signature in type 'readonly string[]'
only permits reading.

console.log(numbers); // [1, 2, 3]
console.log(names);   // ["Alice", "Bob", "Charlie"]
```

Vous pouvez lire les éléments d'un tableau en lecture seule, mais vous ne pouvez pas les modifier.

12.2 Différence avec `const`

Il est important de noter la différence entre un tableau `const` et un tableau `readonly` :

- `const` : Empêche la réaffectation de la variable elle-même, mais n'empêche pas la modification du contenu du tableau.
- `readonly` : Empêche la modification du contenu du tableau, mais la variable peut être réaffectée si elle n'est pas aussi `const`.

Exemple 2 : `const` vs `readonly`

```
const mutableArray = [1, 2, 3];
mutableArray.push(4); // OK
console.log(mutableArray); // [1, 2, 3, 4]
// mutableArray = [5, 6]; // Erreur: Cannot assign to 'mutableArray' because it
is a constant.

let readOnlyArray: readonly number[] = [1, 2, 3];
// readOnlyArray.push(4); // Erreur: Property 'push' does not exist on type
'readonly number[]'.
readOnlyArray = [4, 5, 6]; // OK, car 'readOnlyArray' n'est pas 'const'
console.log(readOnlyArray); // [4, 5, 6]
```

Pour avoir un tableau dont la référence et le contenu sont immuables, vous devez combiner `const` et `readonly` :

```
const immutableArray: readonly number[] = [1, 2, 3];
// immutableArray.push(4); // Erreur
// immutableArray = [4, 5, 6]; // Erreur
```

12.3 Conversion vers un tableau en lecture seule

Vous pouvez convertir un tableau mutable en un tableau en lecture seule. Cependant, la conversion inverse (d'un tableau `readonly` vers un tableau mutable) nécessite une assertion de type, car cela pourrait potentiellement violer la sécurité des types.

Exemple 3 : Conversion

```
const mutableNumbers: number[] = [1, 2, 3];
const readOnlyNumbers: readonly number[] = mutableNumbers; // OK, conversion
implicite

// readOnlyNumbers.push(4); // Erreur

// Conversion inverse (nécessite une assertion de type)
const backToMutable: number[] = readOnlyNumbers as number[];
backToMutable.push(4); // OK, mais soyez prudent!
console.log(backToMutable); // [1, 2, 3, 4]
console.log(readOnlyNumbers); // [1, 2, 3, 4] (car ils partagent la même
référence sous-jacente)
```

Exercice pour la section 'Tableau en lecture seule'

1. Déclarez un tableau `fruits` de type `readonly string[]` et initialisez-le avec quelques noms de fruits.
2. Tentez d'ajouter un nouveau fruit à ce tableau en utilisant `push()` et observez l'erreur de compilation.
3. Tentez de modifier un élément existant du tableau par son index et observez l'erreur de compilation.
4. Créez une fonction `printFruits(items: readonly string[])` qui prend un tableau de chaînes en lecture seule et affiche chaque fruit. Assurez-vous que la fonction ne tente pas de modifier le tableau.
5. Appelez `printFruits` avec votre tableau `fruits`.

13. Type Object

En TypeScript, le type `object` (avec un 'o' minuscule) est un type non-primitif qui représente toute valeur qui n'est pas un type primitif. Les types primitifs en JavaScript sont `string`, `number`, `boolean`, `symbol`, `null`, `undefined`, et `bigint`. Par conséquent, le type `object` englobe tout ce qui est un objet au sens JavaScript, y compris les objets littéraux, les tableaux, les fonctions, les dates, les expressions régulières, etc.

13.1 `object` vs `Object` vs `{}`

Il est important de distinguer `object` (minuscule) de `Object` (majuscule) et de l'objet littéral vide `{}`.

- **`object` (minuscule)** : Représente tout type non-primitif. C'est le type le plus général pour les objets.

  ```
  typescript let obj1: object; obj1 = {}; obj1 = []; obj1 = () => {}; obj1 = new Date(); // obj1 = 123; // Erreur: Type 'number' n'est pas assignable au type 'object'. // obj1 = "hello"; // Erreur: Type 'string' n'est pas assignable au type 'object'.
  ```

- **`Object` (majuscule)** : Représente le type de toutes les instances de `Object` en JavaScript. Il inclut tous les types primitifs et non-primitifs. C'est un type très large et son utilisation est souvent déconseillée car il offre peu de sécurité de type.

  `typescript let obj2: Object; obj2 = {}; obj2 = []; obj2 = () => {}; obj2 = 123; // OK obj2 = "hello"; // OK`

- **`{}` (objet littéral vide)** : Représente un objet qui n'a aucune propriété propre. C'est un type plus strict que `object`.

  `typescript let obj3: {}; obj3 = {}; // obj3 = { a: 1 }; // Erreur si strictNullChecks est activé et que l'objet a des propriétés // obj3 = 123; // Erreur`

En général, `object` est le type préféré lorsque vous voulez indiquer qu'une variable doit être un objet non-primitif, mais que vous ne connaissez pas sa structure spécifique. Si vous avez besoin d'un type plus spécifique, utilisez des interfaces ou des alias de type.

13.2 Utilisation et limitations du type `object`

Le type `object` est utile lorsque vous manipulez des objets de manière générique, sans vous soucier de leurs propriétés spécifiques. Cependant, comme il est très général, vous ne pouvez pas accéder directement aux propriétés ou méthodes spécifiques d'un objet de type `object` sans un affinement de type.

Exemple 1 : Utilisation générique

```
function displayType(arg: object) {
  console.log(`Type of argument: ${typeof arg}`);
}

displayType({});
displayType([]);
displayType(new Date());
displayType(() => {});
```

Exemple 2 : Limitations et affinement de type

```
function processObject(data: object) {
  // console.log(data.length); // Erreur: Property 'length' does not exist on
type 'object'.

  if (Array.isArray(data)) {
    console.log(`Array length: ${data.length}`); // OK, data est affiné à Array
  } else if (typeof data === 'function') {
    console.log(`Function name: ${data.name}`); // OK, data est affiné à
Function
  } else if (data instanceof Date) {
    console.log(`Date year: ${data.getFullYear()}`); // OK, data est affiné à
Date
  }
}

processObject([1, 2, 3]);
processObject(() => {});
processObject(new Date());
```

Exercice pour la section 'Type Object'

1. Créez une fonction `inspectValue` qui accepte un argument de type `object`.
2. À l'intérieur de la fonction, utilisez des affinements de type pour :
   - Si l'argument est un tableau, affichez sa longueur.
   - Si l'argument est une fonction, affichez son nom.
   - Si l'argument est un objet littéral (c'est-à-dire, ni un tableau ni une fonction ni une date, etc.), affichez les clés de l'objet.
   - Pour tout autre type d'objet, affichez `Unhandled object type`.
3. Testez `inspectValue` avec :
   - Un tableau de chaînes.
   - Une fonction anonyme.
   - Un objet littéral simple `{ a: 1, b: 'hello' }`.
   - Une instance de `Date`.

14. Arrow function

Les fonctions fléchées (Arrow Functions), introduites en ECMAScript 2015 (ES6) et pleinement prises en charge par TypeScript, offrent une syntaxe plus concise pour écrire des expressions de fonction. Elles sont particulièrement utiles pour les fonctions

anonymes et les callbacks. Au-delà de la syntaxe, leur principale caractéristique est la façon dont elles gèrent le contexte `this`.

14.1 Syntaxe des fonctions fléchées

La syntaxe de base d'une fonction fléchée est `(param1, param2, ...) => { statements }`.

- **Sans paramètres** : `() => { ... }`
- **Un seul paramètre** : `param => { ... }` (les parenthèses autour du paramètre sont facultatives)
- **Plusieurs paramètres** : `(param1, param2) => { ... }`
- **Corps de fonction sur une seule ligne (retour implicite)** : Si le corps de la fonction est une seule expression, vous pouvez omettre les accolades et le mot-clé `return`. La valeur de l'expression sera implicitement retournée.

Exemple 1 : Syntaxe de base

```
// Fonction traditionnelle
const addTraditional = function(a: number, b: number): number {
  return a + b;
};
console.log(addTraditional(2, 3)); // 5

// Fonction fléchée avec corps explicite
const addArrowExplicit = (a: number, b: number): number => {
  return a + b;
};
console.log(addArrowExplicit(2, 3)); // 5

// Fonction fléchée avec retour implicite (corps sur une seule ligne)
const addArrowImplicit = (a: number, b: number): number => a + b;
console.log(addArrowImplicit(2, 3)); // 5

// Fonction fléchée sans paramètres
const sayHello = () => "Hello!";
console.log(sayHello()); // Hello!

// Fonction fléchée avec un seul paramètre
const square = (num: number) => num * num;
console.log(square(4)); // 16
```

14.2 Le contexte `this`

C'est la différence la plus significative entre les fonctions fléchées et les fonctions traditionnelles. Les fonctions traditionnelles ont leur propre `this` contextuel, qui dépend de la façon dont la fonction est appelée. Les fonctions fléchées, en revanche, n'ont pas leur propre `this`. Elles capturent le `this` du contexte lexical (le `this` de la portée englobante) au moment de leur définition.

Exemple 2 : `this` dans les fonctions traditionnelles vs fonctions fléchées

```
class Greeter {
  message: string = "Hello";

  greetTraditional() {
    setTimeout(function() {
      // Dans une fonction traditionnelle, 'this' fait référence à l'objet
global (window) ou est undefined en mode strict
      // console.log(this.message); // Erreur ou undefined
    }, 100);
  }

  greetArrow() {
    setTimeout(() => {
      // Dans une fonction fléchée, 'this' capture le 'this' de la classe
Greeter
      console.log(this.message); // Hello
    }, 100);
  }
}

const greeter = new Greeter();
greeter.greetArrow();
```

Cette caractéristique des fonctions fléchées résout le problème courant de la perte du contexte `this` dans les callbacks et les gestionnaires d'événements, rendant le code plus prévisible et plus facile à écrire.

14.3 Autres considérations

- **Pas d'objet `arguments`** : Les fonctions fléchées n'ont pas leur propre objet `arguments`. Si vous avez besoin d'accéder aux arguments passés à la fonction, vous devez utiliser les paramètres rest (`...args`).
- **Ne peuvent pas être utilisées comme constructeurs** : Les fonctions fléchées ne peuvent pas être utilisées avec le mot-clé `new`. Elles n'ont pas de propriété `prototype`.

- **Pas de `yield`** : Les fonctions fléchées ne peuvent pas être utilisées comme générateurs (elles ne peuvent pas contenir le mot-clé `yield`).

Les fonctions fléchées sont devenues la norme pour de nombreux cas d'utilisation en JavaScript et TypeScript en raison de leur concision et de leur gestion prévisible de `this`. Cependant, il est important de comprendre leurs différences avec les fonctions traditionnelles pour les utiliser correctement.

Exercice pour la section 'Arrow function'

1. Convertissez la fonction traditionnelle suivante en une fonction fléchée avec retour implicite : `typescript function multiply(a: number, b: number): number { return a * b; }`
2. Créez une classe `Timer` avec une propriété `seconds: number` initialisée à `0`.
3. Ajoutez une méthode `start()` à la classe `Timer` qui utilise `setInterval` pour incrémenter `seconds` chaque seconde et afficher la valeur. Assurez-vous que `this.seconds` est correctement référencé à l'intérieur du callback de `setInterval` en utilisant une fonction fléchée.
4. Instanciez `Timer` et appelez `start()`. Laissez-le s'exécuter pendant quelques secondes.

15. Attribut en lecture seule

En TypeScript, le mot-clé `readonly` peut être utilisé pour marquer une propriété de classe comme étant en lecture seule. Cela signifie que la propriété peut être initialisée soit lors de sa déclaration, soit dans le constructeur de la classe, mais qu'elle ne peut plus être modifiée par la suite. Cette fonctionnalité est essentielle pour créer des objets immuables et garantir que certaines données ne sont pas accidentellement modifiées après leur création.

15.1 Définition d'un attribut `readonly`

Pour déclarer une propriété en lecture seule, vous placez le mot-clé `readonly` avant le nom de la propriété.

Exemple 1 : Propriété `readonly` dans une classe

```typescript
class UserProfile {
  readonly id: string;
  readonly username: string;
  email: string; // Propriété mutable

  constructor(id: string, username: string, email: string) {
    this.id = id; // Initialisation dans le constructeur
    this.username = username;
    this.email = email;
  }

  updateEmail(newEmail: string) {
    this.email = newEmail; // OK, email est mutable
  }

  // setUsername(newUsername: string) {
  //   this.username = newUsername; // Erreur: Cannot assign to 'username'
because it is a read-only property.
  // }
}

const user = new UserProfile("user-123", "john_doe", "john@example.com");
console.log(user.id);       // user-123
console.log(user.username); // john_doe

user.email = "john.doe@new.com"; // OK
console.log(user.email);    // john.doe@new.com

// user.id = "new-id"; // Erreur: Cannot assign to 'id' because it is a read-
only property.
```

Dans cet exemple, `id` et `username` sont des propriétés en lecture seule. Elles sont définies une fois dans le constructeur et ne peuvent plus être modifiées par la suite. `email`, en revanche, est une propriété normale et peut être mise à jour.

15.2 `readonly` vs `const`

Il est important de ne pas confondre `readonly` avec `const` :

- **`const`** : Utilisé pour les variables. Il empêche la réaffectation de la variable elle-même. La valeur à laquelle la variable fait référence peut être mutable si c'est un objet. ```typescript const MY_CONSTANT = 10; // MY_CONSTANT ne peut pas être réassigné // MY_CONSTANT = 20; // Erreur

  const MY_OBJECT = { value: 1 }; MY_OBJECT.value = 2; // OK, le contenu de l'objet est mutable // MY_OBJECT = { value: 3 }; // Erreur ```

- **`readonly`** : Utilisé pour les propriétés de classe. Il empêche la modification de la valeur de la propriété après son initialisation (dans la déclaration ou le

constructeur).

15.3 `readonly` et les interfaces/types

Le mot-clé `readonly` peut également être utilisé dans les interfaces et les alias de type pour indiquer que certaines propriétés ne doivent pas être modifiables.

Exemple 2 : `readonly` dans une interface

```
interface Point {
  readonly x: number;
  readonly y: number;
}

const p1: Point = { x: 10, y: 20 };
// p1.x = 30; // Erreur: Cannot assign to 'x' because it is a read-only
property.

// Vous pouvez créer un nouvel objet si vous voulez des valeurs différentes
const p2: Point = { x: 5, y: 15 };
```

15.4 Avantages de `readonly`

- **Immuabilité** : Favorise la création d'objets immuables, ce qui peut simplifier la gestion de l'état et réduire les bugs liés aux modifications inattendues des données.
- **Prévisibilité** : Rend le code plus facile à comprendre et à raisonner, car vous savez que certaines propriétés ne changeront jamais après l'initialisation.
- **Sécurité des types** : TypeScript applique cette contrainte au moment de la compilation, vous aidant à détecter les erreurs tôt dans le cycle de développement.

L'utilisation de `readonly` est une bonne pratique pour les propriétés qui ne sont pas censées changer après la construction d'un objet, contribuant à un code plus robuste et maintenable.

Exercice pour la section 'Attribut en lecture seule'

1. Créez une classe `Configuration` avec les propriétés suivantes :
    - `apiUrl` (chaîne de caractères, `readonly`)
    - `timeout` (nombre, `readonly`)

- `debugMode` (booléen, mutable)

2. Implémentez un constructeur qui initialise toutes ces propriétés.
3. Créez une instance de `Configuration`.
4. Tentez de modifier `apiUrl` et `timeout` après l'initialisation et observez les erreurs de compilation.
5. Modifiez `debugMode` et vérifiez que cela fonctionne.

16. Test de type union

Lorsque vous travaillez avec des types d'union en TypeScript, il est souvent nécessaire de déterminer le type exact d'une variable à un moment donné pour pouvoir effectuer des opérations spécifiques à ce type. Ce processus est appelé "rétrécissement de type" (type narrowing). TypeScript fournit plusieurs opérateurs et constructions de code pour vous aider à affiner le type d'une variable au sein d'un bloc de code.

16.1 Typeof

L'opérateur `typeof` est un moyen courant de rétrécir les types d'union basés sur des types primitifs. Il retourne une chaîne de caractères indiquant le type de l'opérande non évalué. TypeScript est capable d'utiliser le résultat de `typeof` pour affiner le type d'une variable.

Valeurs de retour de `typeof` :

- `"string"`
- `"number"`
- `"boolean"`
- `"undefined"`
- `"symbol"`
- `"bigint"`
- `"object"` (pour les objets, tableaux, `null`)
- `"function"`

Exemple 1 : Utilisation de `typeof` pour rétrécir un type union

```
function printValue(value: string | number | boolean) {
  if (typeof value === "string") {
    console.log(`Ceci est une chaîne: ${value.toUpperCase()}`);
  } else if (typeof value === "number") {
    console.log(`Ceci est un nombre: ${value * 10}`);
  } else {
    console.log(`Ceci est un booléen: ${!value}`);
  }
}

printValue("hello"); // Ceci est une chaîne: HELLO
printValue(123);     // Ceci est un nombre: 1230
printValue(true);    // Ceci est un booléen: false
```

16.2 in

L'opérateur `in` est utilisé pour vérifier si une propriété existe sur un objet. C'est un moyen efficace de rétrécir les types d'union lorsque les membres de l'union sont des types d'objets qui diffèrent par la présence de certaines propriétés.

Exemple 2 : Utilisation de `in` pour rétrécir un type union d'objets

```
interface Car {
  drive(): void;
  brand: string;
}

interface Bicycle {
  pedal(): void;
  gears: number;
}

type Vehicle = Car | Bicycle;

function startVehicle(vehicle: Vehicle) {
  if ("drive" in vehicle) {
    // TypeScript sait que 'vehicle' est de type Car ici
    vehicle.drive();
    console.log(`Brand: ${vehicle.brand}`);
  } else {
    // TypeScript sait que 'vehicle' est de type Bicycle ici
    vehicle.pedal();
    console.log(`Gears: ${vehicle.gears}`);
  }
}

const myCar: Car = { drive: () => console.log("Vroom!"), brand: "Tesla" };
const myBike: Bicycle = { pedal: () => console.log("Pedal, pedal!"), gears: 21
};

startVehicle(myCar);  // Vroom! Brand: Tesla
startVehicle(myBike); // Pedal, pedal! Gears: 21
```

16.3 Instanceof

L'opérateur `instanceof` est utilisé pour vérifier si un objet est une instance d'une classe spécifique. Il est particulièrement utile pour rétrécir les types d'union lorsque les membres de l'union sont des instances de classes différentes.

Exemple 3 : Utilisation de `instanceof` pour rétrécir un type union de classes

```
 class Dog {
  bark() { console.log("Woof!"); }
}

class Cat {
  meow() { console.log("Meow!"); }
}

type Animal = Dog | Cat;

function makeAnimalSound(animal: Animal) {
  if (animal instanceof Dog) {
    // TypeScript sait que 'animal' est de type Dog ici
    animal.bark();
  } else {
    // TypeScript sait que 'animal' est de type Cat ici
    animal.meow();
  }
}

const buddy = new Dog();
const whiskers = new Cat();

makeAnimalSound(buddy);    // Woof!
makeAnimalSound(whiskers); // Meow!
```

Ces opérateurs (`typeof`, `in`, `instanceof`) sont des outils essentiels pour gérer la flexibilité des types d'union tout en maintenant la sécurité des types en TypeScript. Ils permettent d'écrire du code plus robuste et plus prévisible.

Exercice pour la section 'Test de type union'

1. Créez un type d'union `MixedData` qui peut être `string`, `number`, ou une instance de la classe `CustomObject`.
2. Définissez la classe `CustomObject` avec une propriété `value: string` et une méthode `printValue(): void`.
3. Écrivez une fonction `handleMixedData(data: MixedData)`.

4. À l'intérieur de `handleMixedData`, utilisez `typeof` pour gérer les `string` et `number`.
5. Utilisez `instanceof` pour gérer les instances de `CustomObject` et appelez leur méthode `printValue()`.
6. Testez `handleMixedData` avec un exemple de chaque type.