Advanced Type Handling in TypeScript

Interfaces

In TypeScript, interfaces define the structure of objects, specifying what properties and types those properties should have. Let's look at a basic example:

```
interface User {
  username: string;
  age: number;
}

function greetUser(user: User) {
  console.log(`Hello, ${user.username}`);
}

greetUser({
  username: 'Alice',
  age: 30,
}); // Hello, Alice
```

In this example, the `User` interface ensures that any object passed into `greetUser` has a `username` and age property.

You can also define object types using **type aliases**, which is an alternative way to describe object structures:

```
type User = {
  username: string;
  age: number;
};

function greetUser(user: User) {
  console.log(`Hello, ${user.username}`);
}

greetUser({
  username: 'Alice',
  age: 30,
}); // Hello, Alice
```

Additionally, you can define object types **inline** without creating a separate interface or type alias:

```
function greetUser(user: { username: string; age: number }) {
  console.log(`Hello, ${user.username}`);
}

greetUser({
  username: 'Alice',
  age: 30,
}); // Hello, Alice
```

Interfaces vs Type Aliases

While interfaces and type aliases are often interchangeable, there are important distinctions. A key difference is that interfaces can be extended or reopened to add more properties, whereas type aliases are fixed once declared.

Extending an Interface:

```
interface Animal {
  name: string;
}

interface Dog extends Animal {
  breed: string;
}

const myDog: Dog = {
  name: "Buddy",
  breed: "Golden Retriever",
};
```

In this example, the `Dog` interface extends the `Animal` interface, inheriting its properties and adding a new one (`breed`).

Extending a Type Alias with Intersections:

```
type Animal = {
  name: string;
};
```

```
type Dog = Animal & {
  breed: string;
};

const myDog: Dog = {
  name: "Buddy",
  breed: "Golden Retriever",
};
```

Here, we use an intersection (`&`) to combine two types into one.

Reopening Interfaces

One powerful feature of interfaces is that you can "reopen" them to add new properties:

```
interface Animal {
  name: string;
}

// Adding a new property to the existing Animal interface
interface Animal {
  legs: number;
}

const cat: Animal = {
  name: "Whiskers",
  legs: 4,
};
```

This flexibility allows you to extend existing interfaces in different parts of your code.

Type Aliases Cannot Be Reopened

Unlike interfaces, **type aliases** cannot be modified once declared:

```
type Animal = {
  name: string;
};

// Trying to redefine the same type will result in an error
type Animal = {
  legs: number;
};
```

```
// ERROR: Duplicate identifier 'Animal'.
```

For this reason, it's often recommended to use interfaces when defining object types, especially if you anticipate the need to extend them.

Function Signatures with Interfaces

Interfaces can also describe functions and their expected parameters:

```
interface Person {
  name: string;
  age: number;
  speak(phrase: string): void;
}

const person: Person = {
  name: "Alice",
  age: 30,
  speak: (phrase) => console.log(phrase),
};

person.speak("Hello, world!"); // Output: Hello, world!
```

In this example, the `Person` interface includes a method `speak`, which expects a string as an argument.

Interfaces vs Classes

You might wonder when to use an interface instead of a class. While both can define object structures, an interface only exists in TypeScript and does not affect the compiled JavaScript. It's purely for type-checking. On the other hand, classes are part of both TypeScript and JavaScript, and they get transpiled into JavaScript code.

- **Interface**: Used purely for type-checking. Does not exist in the final JavaScript output.
- **Class**: A blueprint for creating objects that exist in both TypeScript and JavaScript, and include properties, methods, and initialization logic.

Here’s an example of a class:

```
class Person {
  name: string;
  age: number;

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

  speak(phrase: string): void {
    console.log(`${this.name} says: ${phrase}`);
  }
}

const person1 = new Person("Bob", 25);
person1.speak("Good morning!"); // Output: Bob says: Good morning!
```

In this example, the class `Person` defines both the structure and the behavior (methods) of the objects we create from it.

Summary

- **Interfaces** define object structures in TypeScript and allow for flexible, reusable types.
- **Type aliases** can also define object types but cannot be reopened like interfaces.
- When deciding between the two, prefer interfaces for objects and extendability, and use types for other use cases like unions or intersections.
- Interfaces can define both properties and function signatures, making them versatile tools for type-checking in TypeScript.

Implementing Interfaces in Classes

In TypeScript, you can use interfaces to enforce that a class contains specific properties and methods. This helps ensure that classes follow a consistent structure, especially when multiple classes need to adhere to the same blueprint.

For example, let's define an interface called `CanFormat`, which requires a `format` method:

```
interface CanFormat {
  format(): string;
}
```

```
class User implements CanFormat {
  constructor(public firstName: string, private secretCode: string) {}

  format(): string {
    return this.firstName.toUpperCase();
  }
}

// The User class now adheres to the CanFormat interface
let user1: CanFormat;
let user2: CanFormat;

user1 = new User('Alice', 'secret123');
user2 = new User('Bob', 'superSecret789');

console.log(user1.format()); // ALICE
```

In this example, the `User` class implements the `CanFormat` interface, ensuring that each instance of `User` will have a `format` method. The `format` method in this case returns the `firstName` property in uppercase.

Enforcing Interface Compliance in Arrays

You can also ensure that an array only contains objects that follow a specific interface. For instance, let's store multiple `User` objects, all of which implement the `CanFormat` interface:

```
let users: CanFormat[] = [];

users.push(user1);
users.push(user2);

users.forEach(user => {
  console.log(user.format()); // Outputs: ALICE, BOB
});
```

By specifying the type `CanFormat[]`, we ensure that all elements of the `users` array are objects that have the `format` method, as required by the interface.

Why Use Interfaces with Classes?

Interfaces offer a flexible way to define contracts between different parts of your code. This ensures that any class implementing an interface will conform to a certain structure. In contrast to

classes, interfaces are only used during development and are not compiled into the final JavaScript, making them useful purely for type-checking without adding overhead to your code.

In our example:

- The `User` class have a `format` method because it implements the `CanFormat` interface.
- This provides consistency across any class that adheres to this contract.

Summary

Using interfaces with classes helps create a clear and structured way to enforce consistent behavior across multiple objects in your TypeScript projects. This is especially useful when working with larger codebases where consistency and type safety are key.

Generics

Generics in TypeScript allow us to create reusable components that can work with various data types instead of being limited to a specific one. This approach ensures that components or functions are more flexible and type-safe. Let's dive into how generics work and why they're so useful.

Example: A Function that Adds an ID

Consider the following `addID` function. It accepts an object and returns a new object with all the original properties, plus a random `id` property.

```
const addID = (obj: object) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};

let person1 = addID({ name: 'Alice', age: 30 });
console.log(person1.id); // Random ID
console.log(person1.name); // ERROR: Property 'name' does not exist on
                                  type '{ id: number; }'
```

In this example, TypeScript throws an error when we try to access the `name` property. This is because TypeScript isn't aware of what properties the input object has. The only thing TypeScript knows about the returned object is that it contains an `id`.

Using Generics for Flexibility

To fix this, we can use generics. A generic allows us to "capture" the type of the object passed into the function, so TypeScript knows about its properties.

```
const addID = <T>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};

let person1 = addID({ name: 'Alice', age: 30 });
console.log(person1.id); // Random ID
console.log(person1.name); // Alice
```

Now, by using `<T>`, we've told TypeScript to capture the type of the object passed in. `T` will represent the type of whatever object we pass, so the return type will have both the original properties and the `id`.

Constraints with Generics

While using generics, we might run into issues where anything can be passed into the function. For instance, if we pass a string instead of an object, TypeScript won't throw an error until we try to access object properties that don't exist.

```
let invalidInput = addID('Hello'); // No error at this point
console.log(invalidInput.id); // Random ID
console.log(invalidInput.name); // ERROR: Property 'name' does not exist on type
                                    '"Hello" & { id: number; }'
```

To prevent this, we can constrain our generic type to ensure that only objects are allowed:

```
const addID = <T extends object>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};
let person1 = addID({ name: 'Alice', age: 30 });
let invalidInput = addID('Hello'); // ERROR: Argument of type 'string' is not
                                       assignable to parameter of type 'object'
```

This ensures that only objects can be passed into the `addID` function.

Further Restricting Types

We can further narrow down the type constraint by specifying that the input object must have a specific property, such as `name`.

```
const addID = <T extends { name: string }>(obj: T) => {
  let id = Math.floor(Math.random() * 1000);
  return { ...obj, id };
};

let person1 = addID({ name: 'Alice', age: 30 }); // Valid
let invalidInput = addID({ age: 30 }); // ERROR: Argument must have a 'name'
                                          property
```

Generics with Interfaces

Generics also work well with interfaces. Imagine we have an interface `Person` but the type of one of its properties is unknown. We can use generics to specify the type later.

```
interface Person<T> {
  name: string;
  age: number;
  documents: T;
}

// Use the Person interface, specifying that documents is an array of strings
const person1: Person<string[]> = {
  name: 'John',
  age: 48,
  documents: ['passport', 'bank statement'],
};

// Use the Person interface, specifying that documents is a single string
const person2: Person<string> = {
  name: 'Delia',
  age: 46,
  documents: 'passport',
};
```

In this example, the `documents` property can take on different types, depending on how we define it when we implement the `Person` interface.

Generic Functions with Multiple Types

Generics can also be used to create functions that operate over multiple types. This is especially useful when the relationship between the input and output types needs to be maintained. Here's an example:

```
const mergeObjects = <T, U>(obj1: T, obj2: U) => {
  return { ...obj1, ...obj2 };
};

let merged = mergeObjects({ name: 'Alice' }, { age: 30 });
console.log(merged); // { name: 'Alice', age: 30 }
```

In this case, `mergeObjects` takes two objects with potentially different types, and returns a new object that combines the properties of both.

Type Safety with Generics

One of the main benefits of using generics is maintaining type safety. For instance, when using the `any` type, TypeScript won't enforce type checks:

```
function logLength(value: any) {
  console.log(value.length); // No error, but may cause runtime issues
}

logLength(123); // Undefined length property
```

Using generics with constraints, we can restrict the function to values that have a `length` property:

```
interface HasLength {
  length: number;
}

function logLength<T extends HasLength>(value: T) {
  console.log(value.length);
}
```

```
logLength('Hello'); // 5
logLength([1, 2, 3]); // 3
logLength(123); // ERROR: Type 'number' does not have a 'length' property
```

Summary

Generics in TypeScript allow for:

- Flexibility and type safety in reusable components.
- Working with a variety of types while still ensuring TypeScript's type-checking mechanisms.
- Creating relationships between input and output types for more complex functions.
- Using constraints to narrow down the accepted types for added safety.

By utilizing generics, you can write code that is reusable, scalable, and type-safe.

Literal Types in TypeScript

In TypeScript, literal types allow us to **specify exact values for variables**, rather than just general types like `string`, `number`, or `boolean`. This provides greater type safety by restricting the values a variable can hold. Literal types can be either string literals, numeric literals, or even boolean literals.

String Literal Types

A common use case is defining a variable that can only take on a few predefined string values. For example, when dealing with a limited set of options like color choices, we can define a union of string literal types:

```
let favouriteColor: 'red' | 'blue' | 'green' | 'yellow';

favouriteColor = 'blue'; // OK
favouriteColor = 'crimson'; // ERROR: Type '"crimson"' is not assignable to type
                                '"red" | "blue" | "green" | "yellow"'
```

Here, the variable `favouriteColor` is restricted to four possible values: `'red'`, `'blue'`, `'green'`, or 'yellow'. Any assignment outside these values will trigger a TypeScript error.

Numeric Literal Types

Similarly, we can use numeric literals to enforce specific number values. This can be useful in scenarios where only certain numeric values are valid, such as in configurations or settings:

```
let diceRoll: 1 | 2 | 3 | 4 | 5;

diceRoll = 4; // OK
diceRoll = 6; // ERROR: Type '6' is not assignable to type '1 | 2 | 3 | 4 | 5'
```

In this example, the `diceRoll` variable is restricted to the numbers 1 through 5. Assigning any number outside this range results in a type error.

Boolean Literal Types

You can also define boolean literals, though their use is less common since a boolean can naturally only be `true` or `false`:

```
let isComplete: true | false;

isComplete = true;  // OK
isComplete = false; // OK
isComplete = 'yes'; // ERROR: Type '"yes"' is not assignable to type 'true | false'
```

This provides strict control over the boolean state, especially in cases where the status must be a defined binary value.

Combining Literal Types with Other Types

Literal types are not limited to basic values; you can combine them with other types for more complex scenarios. For instance, combining **literal types with union types** allows for broader flexibility while still restricting the range of values.

```
type Shape = 'circle' | 'square' | 'rectangle';
type Size = 'small' | 'medium' | 'large';

let shape: Shape;
let size: Size;
```

```
shape = 'circle';  // OK
size = 'medium';   // OK
shape = 'triangle'; // ERROR: Type '"triangle"' is not assignable to type 'Shape'.
```

Use Cases for Literal Types

1. Enums: While TypeScript has an enum feature, literal types can sometimes provide a more lightweight alternative, especially when the enum values are simple strings or numbers.

```
type Direction = 'north' | 'south' | 'east' | 'west';

let travelDirection: Direction = 'north';
```

2. Function Parameters: Literal types are often used to restrict the input values for function parameters, ensuring that only specific values are passed to the function.

```
function move(direction: 'left' | 'right' | 'up' | 'down') {
  console.log(`Moving ${direction}`);
}

move('left'); // OK
move('forward'); // ERROR: Argument of type '"forward"' is not assignable to
                    parameter of type '"left" | "right" | "up" | "down"'
```

3. Configuration Options: Literal types are useful for defining configuration options or settings, where only certain predefined values are allowed.

```
type Mode = 'dark' | 'light' | 'auto';

function setMode(mode: Mode) {
  console.log(`Setting mode to ${mode}`);
}

setMode('dark'); // OK
setMode('blue'); // ERROR: Type '"blue"' is not assignable to type 'Mode'.
```

Literal Types with Type Aliases

To make your code cleaner, you can use type aliases with literal types, which gives a name to your literal union:

```
type Color = 'red' | 'green' | 'blue';
type Status = 'active' | 'inactive' | 'pending';

let currentColor: Color;
let userStatus: Status;

currentColor = 'red';    // OK
userStatus = 'active';   // OK
userStatus = 'deleted';  // ERROR: Type '"deleted"' is not assignable to type
                            'Status'.
```

By defining `Color` and `Status` as aliases, you can reuse these types in multiple places, making your code more maintainable and readable.

Literal Inference with `const`

When you use the `const` keyword to declare a variable, TypeScript automatically infers the literal type of that variable. This prevents it from being reassigned to a different value later.

```
const buttonState = 'pressed';

buttonState = 'released'; // ERROR: Type '"released"' is not assignable to type
                             '"pressed"'.
```

Because `buttonState` was declared as a constant with the value 'pressed', it cannot be changed to any other string value later in the code.

Summary

Literal types in TypeScript allow you to constrain variables to specific, predefined values, enhancing type safety and preventing invalid assignments. Whether you're working with string, number, or boolean literals, they provide a useful mechanism for enforcing stricter type checking. These types are commonly used in situations where only a limited set of values is valid, such as in enums, configuration options, or function parameters..