Uso de Interfaces en TypeScript

Definición de Estructura de Objetos: Las interfaces permiten definir claramente qué propiedades y métodos debe tener un objeto. Esto ayuda a evitar errores comunes como el acceso a propiedades no definidas.
TypeScript.

```typescript
interface Usuario {
    nombre: string;
    edad: number;
    email: string;
}

const usuario: Usuario = {
    nombre: "Alejandro",
    edad: 30,
    email: "alejandro@example.com"
};
```

Contratos para Clases: En la Programación Orientada a Objetos (POO), las interfaces se usan para definir contratos que las clases deben seguir. Una clase puede implementar una interfaz, asegurando que cumple con las definiciones de la interfaz.

```typescript
interface Animal {
    nombre: string;
    hacerSonido(): void;
}

class Perro implements Animal {
    nombre: string;
    constructor(nombre: string) {
        this.nombre = nombre;
    }

    hacerSonido(): void {
        console.log("Guau");
    }
}
```

Polimorfismo: Las interfaces permiten la creación de código polimórfico. Puedes escribir funciones y clases que trabajen con interfaces en lugar de tipos específicos, lo que aumenta la flexibilidad y reutilización del código.

```typescript
function hacerSonidoDeAnimal(animal: Animal): void {
    animal.hacerSonido();
}

const perro = new Perro("Firulais");
hacerSonidoDeAnimal(perro); // Output: Guau
```

Rol de las Interfaces en Otros Lenguajes

En otros lenguajes de programación orientada a objetos, como Java y C#, las interfaces también juegan un rol fundamental. Aquí hay algunas similitudes y diferencias:

- **Java**: Al igual que en TypeScript, las interfaces en Java definen un contrato que las clases pueden implementar. Las clases que implementan una interfaz en Java deben proporcionar una implementación para todos los métodos definidos en la interfaz. Java utiliza interfaces para soportar la herencia múltiple, ya que una clase puede implementar múltiples interfaces, pero solo heredar de una clase.
- **C#**: En C#, las interfaces funcionan de manera similar a Java. Una interfaz define un conjunto de métodos y propiedades que una clase debe implementar. C# también permite que las interfaces definan propiedades y eventos, además de métodos. Esto promueve la flexibilidad y asegura que las clases que implementan la interfaz cumplan con un contrato específico.
- **Python**: Aunque Python no tiene interfaces en el mismo sentido estricto que Java o C#, utiliza un concepto similar conocido como "Protocolo" o "Duck Typing". Aquí, la estructura y comportamiento esperado son más importantes que la herencia formal de una interfaz. Sin embargo, Python tiene el módulo abc (Abstract Base Classes) que proporciona funcionalidad similar a las interfaces, permitiendo definir métodos que las subclases deben implementar.

Genéricos en TypeScript

Los **genéricos en TypeScript** son una poderosa característica que permite crear componentes reutilizables y flexibles. Los genéricos permiten definir tipos de una manera que no está limitada a un tipo específico, sino que se pueden adaptar a diferentes tipos según sea necesario. Esto permite escribir código más dinámico y que se pueda reutilizar de manera más efectiva.

¿Qué son los Genéricos?

Un genérico es una forma de crear una función, clase o interfaz que puede trabajar con diferentes tipos de datos mientras mantiene la seguridad de tipos. En lugar de definir un tipo fijo, los genéricos utilizan un parámetro de tipo que se sustituye por un tipo específico cuando se utiliza.

Sintaxis Básica de Genéricos

La sintaxis básica de los genéricos en TypeScript utiliza paréntesis angulares (< y >) para definir un tipo genérico.

```
function identidad<T>(valor: T): T {
    return valor;
}
```

En este ejemplo, T es un parámetro de tipo genérico. Cuando llamamos a la función identidad, T se sustituye por el tipo del argumento que se pasa:

```
let numero = identidad<number>(42); // T es sustituido por number
let texto = identidad<string>("Hola"); // T es sustituido por string
```

Ventajas de los Genéricos

1. **Reutilización de Código**: Los genéricos permiten escribir funciones o clases que funcionan con cualquier tipo de datos, lo que reduce la duplicación de código y aumenta la reutilización.
2. **Seguridad de Tipos**: Al utilizar genéricos, se mantiene la seguridad de tipos. Esto significa que se puede detectar y prevenir errores en tiempo de compilación, en lugar de en tiempo de ejecución.
3. **Flexibilidad**: Los genéricos proporcionan flexibilidad al permitir que el código funcione con diferentes tipos de datos, sin perder la capacidad de definir comportamientos específicos para esos tipos.

Uso de Genéricos con Funciones

Además del ejemplo anterior, los genéricos se pueden usar con funciones para operar con diferentes tipos de datos de manera segura:

```
function combinar<T, U>(valor1: T, valor2: U): [T, U] {
    return [valor1, valor2];
}

let combinacion = combinar<string, number>("edad", 30);
console.log(combinacion); // Output: ["edad", 30]
```

En este ejemplo, T y U son dos tipos genéricos diferentes, lo que permite combinar dos valores de distintos tipos en una tupla.

Genéricos con Clases

Las clases también pueden utilizar genéricos para trabajar con diferentes tipos:

Aquí, la clase Caja puede almacenar y devolver un valor de cualquier tipo, gracias al uso del parámetro de tipo genérico T.

```
class Caja<T> {
    contenido: T;

    constructor(contenido: T) {
        this.contenido = contenido;
    }

    obtenerContenido(): T {
        return this.contenido;
    }
}

let cajaNumero = new Caja<number>(123);
console.log(cajaNumero.obtenerContenido()); // Output: 123

let cajaTexto = new Caja<string>("Hola");
console.log(cajaTexto.obtenerContenido()); // Output: "Hola"
```

Genéricos con Interfaces

También se pueden usar genéricos con interfaces para definir estructuras de datos flexibles:

```
interface Par<T, U> {
    clave: T;
    valor: U;
}

let par: Par<string, number> = {
    clave: "edad",
    valor: 30
};
```

En este ejemplo, la interfaz Par puede ser utilizada con cualquier combinación de tipos para la clave y el valor.

Restricciones en Genéricos

A veces, es necesario restringir el tipo que un genérico puede aceptar. Esto se puede hacer utilizando la palabra clave extends para especificar que el tipo debe heredar de una cierta clase o interfaz:

```
function longitud<T extends { longitud: number }>(elemento: T): number {
    return elemento.longitud;
}

let lista = [1, 2, 3];
console.log(longitud(lista)); // Output: 3

let texto = "Hola";
console.log(longitud(texto)); // Output: 4
```

En este caso, T debe tener una propiedad longitud, lo que permite usar la función longitud con cualquier tipo que tenga esa propiedad, como arreglos y cadenas.