Genéricos

Los genéricos en TypeScript son una herramienta para crear componentes reutilizables que pueden trabajar con múltiples tipos en lugar de uno solo. Esto permite una mejor abstracción de los componentes mientras se mantiene la seguridad de tipos. También ayuda a escribir código más flexible que puede adaptarse a diferentes tipos. Comencemos con un ejemplo básico dado aquí:

```
function printValue<T>(value: T): void {
  console.log(value);
}

printValue<number>(123);
printValue<string>("Hello");
```

<T> es un parámetro de tipo genérico que permite que esta función acepte cualquier tipo de valor. Cuando llamas a printValue<number>(123) y printValue<string>("Hello"), TypeScript trata T como number y string, respectivamente. Esto demuestra cómo los genéricos proporcionan flexibilidad sin perder los beneficios de la comprobación de tipos.

También podemos mejorar InMemoryChatResource, que definimos antes. Funciona bien ahora, pero ¿qué pasa si también necesitamos una implementación en memoria de InMemoryUserResource para IUser además de IChat? Básicamente tienen la misma funcionalidad; la única diferencia entre ellos es el tipo de recurso que manejan. Los genéricos pueden ser útiles aquí. Para ilustrarlo, definamos una clase genérica GenericsInMemoryResource y creemos instancias de chat y usuario de la misma:

```
class GenericsInMemoryResource<T extends { id: number }> extends AbstractDatabaseResource {
  private resources: T[] = [];
  constructor(resourceName: string) {
    super(resourceName);
  }
  get(id: number): T | null {
    const resource = this.resources.find((item) => item.id === id);
    return resource ? {...resource} : null;
  }
  getAll(): T[] {
    return [...this.resources];
  }
  addResource(resource: T): void {
    this.resources.push(resource);
    this.logResource(resource);
```

```
    }
}

const userInMemoryResource = new GenericsInMemoryResource<IUser>('user')
const chatInMemoryResource = new GenericsInMemoryResource<IChat>('chat')
userInMemoryResource.addResource({id: 1, name: 'Admin', email: 'admin@admin.com'});
chatInMemoryResource.addResource({id: 10, ownerId: userInMemoryResource.get(1)!.id, messages:
[]});
```

La sintaxis <T extends { id: number }> significa que T puede ser cualquier tipo, pero debe tener una propiedad id de tipo number. Esto garantiza la seguridad de tipos mientras se permite el uso de diferentes tipos de recursos. A diferencia de InMemoryChatResource, que estaba limitado a manejar objetos IChat, GenericsInMemoryResource puede manejar cualquier tipo que cumpla con la restricción. Esto elimina la necesidad de clases separadas para cada tipo de recurso, como chats o usuarios.

Las instancias como userInMemoryResource y chatInMemoryResource demuestran esto usando GenericsInMemoryResource con los tipos IUser e IChat. Los métodos como addResource y get funcionan con el tipo genérico T, adaptándose al tipo específico de recurso que se está utilizando.

La clase original InMemoryChatResource está limitada a manejar datos de chat, mientras que GenericsInMemoryResource es más flexible y escalable, capaz de manejar varios tipos de recursos. Esto lleva a un código más limpio y mantenible, ya que no necesitas crear una nueva clase para cada tipo de recurso.

Los genéricos reducen significativamente la duplicación de código al permitir que una sola clase administre múltiples tipos de datos. Este enfoque simplifica el mantenimiento y mejora la adaptabilidad de nuestra base de código.

Más ejemplos:

Ejemplo 1: Función genérica simple

Este es un ejemplo básico de una función genérica que puede trabajar con múltiples tipos.

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

console.log(identity<number>(42)); // Output: 42
console.log(identity<string>("Generics are powerful")); // Output: Generics are powerful
```

En este caso, la función identity devuelve lo mismo que recibe, sin importar el tipo de argumento. El tipo T actúa como un marcador de posición para cualquier tipo que se le pase.

Ejemplo 2: Genéricos con arrays

Este ejemplo muestra cómo usar genéricos con arrays.

```
function reverseArray<T>(items: T[]): T[] {
  return items.reverse();
}

const numbers = [1, 2, 3, 4];
const reversedNumbers = reverseArray<number>(numbers);
console.log(reversedNumbers); // Output: [4, 3, 2, 1]

const strings = ["apple", "banana", "cherry"];
const reversedStrings = reverseArray<string>(strings);
console.log(reversedStrings); // Output: ["cherry", "banana", "apple"]
```

La función reverseArray toma un array de cualquier tipo T y lo invierte, manteniendo la seguridad de tipos.

Ejemplo 3: Clase genérica con restricciones

Aquí se muestra una clase genérica similar a la que mencionaste, con restricciones para asegurar que los objetos tengan una propiedad id.

```
class Storage<T extends { id: number }> {
  private items: T[] = [];

  add(item: T): void {
    this.items.push(item);
  }

  getById(id: number): T | undefined {
    return this.items.find(item => item.id === id);
  }
}

interface Product {
  id: number;
```

```
  name: string;
  price: number;
}

interface User {
  id: number;
  username: string;
}

const productStorage = new Storage<Product>();
productStorage.add({ id: 1, name: "Laptop", price: 1000 });
productStorage.add({ id: 2, name: "Phone", price: 500 });

console.log(productStorage.getById(1)); // Output: { id: 1, name: 'Laptop', price: 1000 }

const userStorage = new Storage<User>();
userStorage.add({ id: 1, username: "john_doe" });

console.log(userStorage.getById(1)); // Output: { id: 1, username: 'john_doe' }
```

En este caso, la clase Storage es genérica y puede almacenar cualquier tipo de objeto que tenga una propiedad id. Esto garantiza que solo objetos con id se almacenen.

Ejemplo 4: Genéricos con múltiples parámetros de tipo

Los genéricos también pueden aceptar múltiples parámetros de tipo.

```
function pair<K, V>(key: K, value: V): [K, V] {
  return [key, value];
}

const stringNumberPair = pair<string, number>("age", 30);
console.log(stringNumberPair); // Output: ["age", 30]

const booleanStringPair = pair<boolean, string>(true, "isAdmin");
console.log(booleanStringPair); // Output: [true, "isAdmin"]
```

Aquí, la función pair acepta dos parámetros de tipo K y V, lo que permite crear tuplas con dos valores de diferentes tipos.

Ejemplo 5: Genéricos en interfaces

Puedes usar genéricos en interfaces para hacerlas más flexibles.

```
interface ApiResponse<T> {
  status: number;
  data: T;
  error?: string;
}

const userResponse: ApiResponse<User> = {
  status: 200,
  data: { id: 1, username: "john_doe" },
};

const productResponse: ApiResponse<Product> = {
  status: 200,
  data: { id: 1, name: "Laptop", price: 1000 },
};
```

La interfaz ApiResponse es genérica, lo que significa que puede adaptarse a cualquier tipo de dato como User o Product.

Ejercicios

Ejercicio 1: Implementar una función genérica de filtro

Crea una función genérica filterItems que acepte un array de cualquier tipo y un predicado (una función que devuelve un booleano) y devuelva un nuevo array con los elementos que cumplan el predicado.

```
function filterItems<T>(items: T[], predicate: (item: T) => boolean): T[] {
  return items.filter(predicate);
}

// Prueba
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = filterItems<number>(numbers, (num) => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
```

Ejercicio 2: Crear una clase genérica para manejar pilas (Stacks)

Crea una clase genérica Stack<T> que permita agregar elementos a la pila y quitarlos (LIFO - Last In, First Out).

```
class Stack<T> {
  private items: T[] = [];

  push(item: T): void {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }

  peek(): T | undefined {
    return this.items[this.items.length - 1];
  }
}

// Prueba
const numberStack = new Stack<number>();
numberStack.push(10);
numberStack.push(20);
console.log(numberStack.pop()); // Output: 20
console.log(numberStack.peek()); // Output: 10
```

Ejercicio 3: Implementar una clase genérica para un carrito de compras

Crea una clase genérica ShoppingCart<T> que permita agregar productos y calcular el total del carrito, asegurando que los productos tengan un precio.

```
class ShoppingCart<T extends { price: number }> {
  private items: T[] = [];

  addItem(item: T): void {
    this.items.push(item);
  }
```

```
  calculateTotal(): number {
    return this.items.reduce((total, item) => total + item.price, 0);
  }
}

// Prueba
const cart = new ShoppingCart<Product>();
cart.addItem({ id: 1, name: "Laptop", price: 1000 });
cart.addItem({ id: 2, name: "Mouse", price: 50 });
console.log(cart.calculateTotal()); // Output: 1050
```

Ejercicio 4: Extender genéricos con múltiples restricciones

Crea una función genérica mergeObjects que acepte dos objetos y los combine en uno solo. Ambos objetos deben tener una propiedad id.

```
function mergeObjects<T extends { id: number }, U extends { id: number }>(obj1: T, obj2: U): T & U {
  return { ...obj1, ...obj2 };
}

// Prueba
const obj1 = { id: 1, name: "Alice" };
const obj2 = { id: 2, age: 30 };

const mergedObj = mergeObjects(obj1, obj2);
console.log(mergedObj); // Output: { id: 2, name: 'Alice', age: 30 }
```

Promesas

En TypeScript, cuando declaras una promesa, puedes usar genéricos para indicar el tipo de datos que la promesa devolverá eventualmente. Esta indicación de tipo asegura que el valor resuelto sea consistente con las expectativas y permite que TypeScript proporcione la comprobación de tipos y la autocompletación relevantes. Declaramos una función fetchData aquí que imitará una solicitud de red con setTimeout y proporcionará el tipo para su valor de retorno utilizando el tipo Promise:

```
function fetchData(): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data Fetched"), 1000);
  });
}
```

fetchData es una función que devuelve una instancia de Promise<string>. Esto significa que la promesa, cuando se resuelva, devolverá una cadena. Dentro de la promesa, simulamos la obtención de datos con setTimeout y la resolvemos con un valor de cadena, adhiriéndose al tipo de retorno declarado string. Esta tipificación explícita del valor resuelto de la promesa asegura que cualquier consumidor de fetchData pueda esperar una cadena.

Más ejemplos:

Ejemplo 1: Promesa que devuelve un número

En este ejemplo, una promesa devuelve un número después de un retraso simulado.

```
function fetchNumber(): Promise<number> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(42), 1000);
  });
}

fetchNumber().then((result) => {
  console.log("Fetched number:", result); // Output: Fetched number: 42
});
```

En este caso, la promesa está tipada como Promise<number>, asegurando que el valor resuelto será de tipo number.

Ejemplo 2: Promesa que devuelve un objeto

Este ejemplo muestra cómo usar una promesa que resuelve un objeto con propiedades definidas.

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

```
function fetchUser(): Promise<User> {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: 1, name: "John Doe", email: "john@example.com" }), 1000);
  });
}

fetchUser().then((user) => {
  console.log("Fetched user:", user);
  // Output: Fetched user: { id: 1, name: 'John Doe', email: 'john@example.com' }
});
```

Aquí, la promesa devuelve un objeto User, garantizando que los datos resueltos tengan la estructura esperada.

Ejemplo 3: Promesa genérica que acepta cualquier tipo

Este ejemplo utiliza una función genérica que puede aceptar cualquier tipo de datos para resolver la promesa.

```
function fetchData<T>(data: T): Promise<T> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(data), 1000);
  });
}

fetchData<string>("Hello World").then((result) => {
  console.log("Fetched string:", result); // Output: Fetched string: Hello World
});

fetchData<number>(123).then((result) => {
  console.log("Fetched number:", result); // Output: Fetched number: 123
});
```

La función fetchData acepta un tipo genérico T, lo que permite que se resuelva con cualquier tipo que se pase como argumento.

Ejemplo 4: Promesas anidadas con genéricos

Este ejemplo muestra cómo trabajar con promesas anidadas, donde cada paso tiene un tipo definido.

```
function fetchUserById(id: number): Promise<User> {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id, name: "John Doe", email: "john@example.com" }), 1000);
  });
}

function fetchUserData(user: User): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(`User Data: ${user.name}, ${user.email}`), 1000);
  });
}

fetchUserById(1)
  .then((user) => fetchUserData(user))
  .then((userData) => {
    console.log(userData);
    // Output: User Data: John Doe, john@example.com
  });
```

En este caso, cada promesa devuelve un valor de tipo específico, y se puede encadenar con la seguridad de que los tipos son correctos.

Ejercicios

Ejercicio 1: Función genérica de búsqueda

Crea una función genérica llamada findItem que acepte un array de cualquier tipo T y un predicado (una función que determina si un elemento cumple una condición). La función debe devolver el primer elemento que cumpla con el predicado o null si no se encuentra.

Ejercicio 2: Genéricos con restricciones

Crea una clase genérica KeyValueStore<T> que almacene pares de clave-valor, donde la clave sea una cadena y el valor sea del tipo genérico T. Implementa métodos para agregar, obtener y eliminar valores del almacén. Asegúrate de que el almacén solo acepte valores que tengan una propiedad id.

Ejercicio 3: Promesas con genéricos

Define una función genérica fetchData<T> que acepte una URL y devuelva una promesa que resuelva con un dato de tipo T. Debes simular una solicitud HTTP que devuelva el dato correcto en función del tipo proporcionado.

Ejercicio 4: Función genérica de combinación

Crea una función genérica llamada merge<T, U> que acepte dos objetos de diferentes tipos y los combine en uno solo. La función debe devolver un objeto que contenga todas las propiedades de ambos objetos.

Ejercicio 5: Genéricos con múltiples parámetros de tipo

Escribe una función genérica llamada mapTwoArrays<T, U> que acepte dos arrays de tipos diferentes y una función de mapeo. La función debe devolver un nuevo array que contenga los resultados de aplicar la función de mapeo a los elementos de ambos arrays.

Ejercicio 6: Genéricos con interfaces

Define una interfaz genérica ApiResponse<T> que represente una respuesta de una API. La interfaz debe tener una propiedad status de tipo número y una propiedad data de tipo T. Crea otra interfaz PaginatedResponse<T> que extienda ApiResponse<T> y agregue propiedades para manejar paginación.

Ejercicio 7: Clase genérica de caché

Crea una clase genérica Cache<T> que permita almacenar y recuperar valores en memoria. La clase debe tener métodos para agregar un valor con una clave de tipo cadena, obtener el valor por clave y eliminar un valor por clave. Asegúrate de que el tipo T sea opcionalmente restringido a objetos que tengan una propiedad id.

Ejercicio 8: Clase genérica con restricciones múltiples

Crea una clase genérica SortableCollection<T> que acepte solo objetos que tengan propiedades name y age. Implementa un método sortByAge que devuelva los objetos ordenados por edad y otro método sortByName que los ordene por nombre.

Ejercicio 9: Promesas con genéricos y manejo de errores

Escribe una función genérica fetchWithRetry<T> que acepte una URL y un número máximo de reintentos. La función debe devolver una promesa de tipo T y reintentar la solicitud si falla, hasta alcanzar el número máximo de reintentos.

Ejercicio 10: Implementar una pila (Stack) genérica

Crea una clase genérica Stack<T> que funcione como una pila (LIFO). La clase debe tener métodos para agregar elementos (push), eliminar el último elemento agregado (pop) y obtener el último elemento sin eliminarlo (peek).