**Generics**

Generics in TypeScript are a feature that allows you to write reusable code by passing a type as a parameter to another type, whether it's a class, interface, or function. This means that you don't always have to specify the type explicitly when writing code, as you can use Generics to work flexibly with various types without resorting to using ": any."

The main advantages of using Generics include:

1. Code Reusability: You can use the same code with different types without rewriting it.
2. Enhanced Safety: Generics help detect potential errors at compile time rather than runtime.
3. Dealing with Multiple Types: Generics allow you to work with a variety of types without specifying a particular type.

Generics can be used in TypeScript to create:

* Generic Classes.
* Generic Functions.
* Generic Interfaces.
* Generic Methods.

// A generic function that returns the input value as is

function returnType<T>(val: T): T {

return val;

}

// Usage of the generic function with different types

const numValue: number = returnType<number>(100);

const strValue: string = returnType<string>("Elzero");

const boolValue: boolean = returnType<boolean>(true);

const arrValue: number[] = returnType<number[]>([1, 2, 3, 4]);

console.log(`Number Value: ${numValue}`);

console.log(`String Value: ${strValue}`);

console.log(`Boolean Value: ${boolValue}`);

console.log(`Array Value: ${arrValue}`);

In this example:

* We define a generic function returnType<T> that takes a value of type T and returns the same value of type T.
* We demonstrate using the returnType function with various types, including numbers, strings, booleans, and arrays.
* We specify the type parameter T when calling the returnType function, allowing TypeScript to infer the correct types for the returned values.
* The function provides type safety, ensuring that the returned value has the same type as the input value.

**Generics Multiple Types**

Generics in TypeScript can handle multiple types using union types or intersection types, allowing you to create flexible and versatile code that works with a variety of data types. Let's break down these two approaches:

// Generic Function `returnTypeEx`

function returnTypeEx<T>(val: T): T {

return val;

}

console.log(returnTypeEx<number>(100)); // Returns: 100 (number)

console.log(returnTypeEx<string>("Elzero")); // Returns: "Elzero" (string)

// Arrow Function with Generics `returnTypeArrowSyntax`

const returnTypeArrowSyntax = <T>(val: T): T => val;

console.log(returnTypeArrowSyntax<number>(100)); // Returns: 100 (number)

console.log(returnTypeArrowSyntax<string>("Elzero")); // Returns: "Elzero" (string)

// Generic Function `testType`

function testType<T>(val: T): string {

return `The Value Is ${val} And Type Is ${typeof val}`;

}

console.log(testType<number>(100));

// Returns: "The Value Is 100 And Type Is number"

console.log(testType<string>("Elzero"));

// Returns: "The Value Is Elzero And Type Is string"

// Generic Function `multipleTypes`

function multipleTypes<T, S>(valueOne: T, valueTwo: S): string {

return `The First Value Is ${valueOne} And Second Value ${valueTwo}`;

}

console.log(multipleTypes<string, number>("Osama", 100));

// Returns: "The First Value Is Osama And Second Value 100"

console.log(multipleTypes<string, boolean>("Elzero", true));

// Returns: "The First Value Is Elzero And Second Value true"

In this code:

* The returnTypeEx function and the returnTypeArrowSyntax arrow function are generic functions that return the input value as is. They accept a type parameter <T> and return a value of type T. We demonstrate their usage with numbers and strings.
* The testType function is a generic function that takes a value and returns a string describing the value and its type using the typeof operator.
* The multipleTypes function is a generic function that accepts two values of potentially different types T and S and returns a string combining both values.

Each function demonstrates how generics can be used with different data types while maintaining type safety.

**Generics Classes**

Generics in classes allow you to create flexible and reusable class structures that can work with a variety of data types, enhancing code flexibility and type safety.

In this example, we'll explore generics in classes using the User class as an example:

// Generic Class `User`

class User<T = string> {

constructor(public value: T) {}

// Method that takes a message of type `T` and displays it along with the `value` property

show(msg: T): void {

console.log(`${msg} - ${this.value}`);

}

}

// Creating an instance of `User` with a specific type parameter (string)

let userOne = new User<string>("Elzero");

console.log(userOne.value); // Outputs: "Elzero"

userOne.show("Message"); // Outputs: "Message - Elzero"

// Creating an instance of `User` with a type parameter that can be a number or a string

let userTwo = new User<number | string>(100);

console.log(userTwo.value); // Outputs: 100

userTwo.show("Message"); // Outputs: "Message - 100"

In this example:

* We define a generic class User<T = string>, which allows us to create instances of the User class with different data types. The class takes a type parameter T, and by default, it is set to string if no type is provided.
* The constructor of the User class accepts an initial value of type T.
* The show method of the User class takes a message of type T and displays it along with the value property. This method demonstrates how a generic class can work with the type specified by the type parameter T.
* We create two instances of the User class: userOne with the type parameter explicitly set to string, and userTwo with the type parameter set to number | string. This showcases how the same class can be used with different data types while preserving type safety.

**Generics And Interfaces**

Generics in classes and interfaces allow us to create reusable and type-safe data structures that can work with different types, enhancing code flexibility and maintainability. In this example, we've demonstrated how the Collection class can be used with different item types while ensuring type safety.

In this example, we'll explore the use of generics in classes and interfaces to create a flexible Collection class:

// Interface definitions for Book and Game

interface Book {

itemType: string;

title: string;

isbn: number;

}

interface Game {

itemType: string;

title: string;

style: string;

price: number;

}

// Generic Class `Collection`

class Collection<T> {

public data: T[] = [];

// Method to add an item of type `T` to the collection

add(item: T): void {

this.data.push(item);

}

}

// Creating an instance of `Collection` with type parameter `Book`

let itemOne = new Collection<Book>();

itemOne.add({ itemType: "Book", title: "Atomic Habits", isbn: 150510 });

itemOne.add({ itemType: "Book", title: "Follow Your Heart", isbn: 650650 });

console.log(itemOne);

// Creating an instance of `Collection` with type parameter `Game`

let itemTwo = new Collection<Game>();

itemTwo.add({ itemType: "Game", title: "Uncharted", style: "Action", price: 150 });

console.log(itemTwo);

In this code:

* We define two interfaces, Book and Game, each representing a different type of item with specific properties.
* We create a generic class Collection<T> that allows us to create collections of items of type T. The data property is an array of items, and the add method allows us to add items of type T to the collection.
* We create two instances of the Collection class:
  + itemOne is an instance with the type parameter set to Book, so it can only store objects that conform to the Book interface.
  + itemTwo is an instance with the type parameter set to Game, so it can only store objects that conform to the Game interface.
  + We use the add method to add items of the specified types to the collections.