Дженерики (Generics) в TypeScript — это мощный инструмент, который позволяет нам писать обобщённый и гибкий код, работающий с разными типами данных, сохраняя типовую безопасность. Они помогают избежать дублирования кода, обеспечивая возможность работы с различными типами при сохранении строгой системы типов.

Давайте разберём дженерики подробно.

**1. Что такое дженерики и зачем они нужны**

Дженерики позволяют определять функции, классы, интерфейсы или типы данных таким образом, чтобы они могли работать с любым (или ограниченным) типом данных, переданным в них. Это помогает избегать использования типа any, который отключает проверку типов, увеличивая вероятность ошибок.

**Преимущества дженериков:**

Типовая безопасность: TypeScript может вывести тип данных и предотвращать ошибки несоответствия типов.

Повторное использование: Код можно переиспользовать для работы с разными типами данных.

Уменьшение дублирования: Нет необходимости писать одинаковый код для каждого типа данных.

Например, вместо того чтобы писать разные функции для обработки number[] и string[], можно использовать дженерик.

**2. Дженерики в функциях**

Функции с дженериками позволяют указать тип переменной, который будет использован в этой функции.

Пример базовой функции с дженериком:

// Определяем функцию, которая принимает массив любого типа

function identity(value: T): T {

  return value;

}

// Использование

const result1 = identity(42); // Явно указываем тип T как number

const result2 = identity("Hello"); // Тип T автоматически выводится как string

console.log(result1); // 42

console.log(result2); // Hello

T — переменная типа (Type variable), это название условного типа, который будет определён во время вызова функции.

Тип выводится автоматически (во втором вызове identity).

**3. Дженерики с массивами**

Массивы любого типа можно обработать с помощью дженериков.

Пример:

function getArray(items: T[]): T[] {

  return items;

}

const numberArray = getArray([1, 2, 3]); // Массив чисел

const stringArray = getArray(["a", "b", "c"]); // Массив строк

console.log(numberArray);

console.log(stringArray);

**4. Ограничения типов c extends**

Мы можем ограничить тип переменной, чтобы дженерик принимал только определённые типы.

Пример:

// T должен быть либо объектом, содержащим свойство length

function logLength(item: T): void {

  console.log(item.length);

}

logLength("Hello, TypeScript!"); // Работает (string имеет свойство length)

logLength([1, 2, 3]); // Работает (массив также имеет length)

// logLength(42); // Ошибка, так как число не имеет length

Ключевое слово extends вводит ограничения на тип. В данном случае T должен быть типом, который включает свойство length.

**5. Дженерики в интерфейсах**

Интерфейсы также могут быть обобщёнными, чтобы описывать структуры, которые работают с разными типами данных.

Пример:

interface ItemContainer {

  items: T[];

  addItem(item: T): void;

  getItem(index: number): T;

}

const stringContainer: ItemContainer = {

  items: [],

  addItem(item: string) {

    this.items.push(item);

  },

  getItem(index: number) {

    return this.items[index];

  },

};

stringContainer.addItem("TypeScript");

console.log(stringContainer.getItem(0)); // TypeScript

Здесь интерфейс ItemContainer описывает контейнер для любого типа данных, а конкретный тип указывается (например, stringContainer работает со строками).

**6. Классы с дженериками**

Классы также могут быть обобщёнными.

Пример:

class DataStorage {

  private data: T[] = [];

  addItem(item: T): void {

    this.data.push(item);

  }

  removeItem(item: T): void {

    this.data = this.data.filter(dataItem => dataItem !== item);

  }

  getItems(): T[] {

    return [...this.data];

  }

}

const textStorage = new DataStorage();

textStorage.addItem("Hello");

textStorage.addItem("World");

textStorage.removeItem("Hello");

console.log(textStorage.getItems()); // ["World"]

Класс DataStorage может работать с любым типом данных. В данном случае он управляет строками.

**7. Дженерики в типах**

TypeScript позволяет создавать обобщённые типы с помощью ключевого слова type.

Пример:

type Pair = {

  first: T;

  second: U;

};

const pair: Pair = {

  first: "Hello",

  second: 42,

};

console.log(pair);

Дженерики здесь используются для создания обобщённой пары (например, строка + число).

8. Дефолтные типы для дженериков

Если пользователь не указал тип явно, можно задавать значения по умолчанию.

Пример:

function identity(value: T): T {

  return value;

}

const result = identity(); // По умолчанию T = string

console.log(result); // Пустая строка

**9. Множественные дженерики**

Можно использовать несколько переменных с типами.

Пример:

function merge(obj1: T, obj2: U): T & U {

  return Object.assign({}, obj1, obj2);

}

const mergedObject = merge({ name: "Alice" }, { age: 25 });

console.log(mergedObject); // { name: "Alice", age: 25 }

Здесь дженерики T и U описывают типы двух объектов, которые объединяются.

10. Примеры из реальной жизни

Дженерики уже встроены в стандартные коллекции TypeScript, такие как Array, Promise и др.

Пример:

const stringArray: Array = ["a", "b", "c"]; // Вместо Array указываем Array

const numberPromise: Promise = new Promise((resolve) => resolve(42));

Используя встроенные дженерики, можно избежать ошибок при работе с коллекциями или асинхронным кодом.

**Вывод**

Дженерики — это ключевой инструмент TypeScript, который делает код:

более гибким;

более выразительным;

с минимальными дублированиями.

Это особенно полезно в библиотечных функциях, инструментах или больших проектах, где необходимо работать с неизвестными типами.