TypeScript

Generics

Content

1. Introduction
2. Generic Class
3. Generic Interface

1. Introduction

- Generics allow you to write reusable and generalized forms of functions, classes, and interfaces.

```
function getRandomNumberElement(items: number[]): number {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

```
let numbers = [1, 5, 7, 4, 2, 9];
console.log(getRandomNumberElement(numbers));
```

```
function getRandomStringElement(items: string[]): string {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

```
let colors = ['red', 'green', 'blue'];
console.log(getRandomStringElement(colors));
```

Using the any type

```
function getRandomAnyElement(items: any[]): any {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

```
let numbers = [1, 5, 7, 4, 2, 9];
let colors = ['red', 'green', 'blue'];

console.log(getRandomAnyElement(numbers));
console.log(getRandomAnyElement(colors));
```

- This solution works fine. However, it has a drawback.
- It doesn't allow you to enforce the type of the returned element. In other words, it isn't type-safe.

- Generics come to rescue

```
function getRandomElement<T>(items: T[]): T {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
```

- Calling a generic function

```
let numbers = [1, 5, 7, 4, 2, 9];
let randomEle = getRandomElement<number>(numbers);
console.log(randomEle);
```

3. Generic Class

- A generic class has a generic type parameter list in angle brackets <>

```
class className<T>{
    //...
}
```

```
class className<K,T>{
    //...
}
```

```
class Stack<T> {
    private elements: T[] = [];
    constructor(private size: number) {}
    isEmpty(): boolean {  return this.elements.length === 0;  }
    isFull(): boolean {  return this.elements.length === this.size;  }
    push(element: T): void {
        if (this.elements.length === this.size) {
            throw new Error('The stack is overflow!');
        }
        this.elements.push(element);
    }
    pop(): T {
        if (this.elements.length == 0) {
            throw new Error('The stack is empty!');
        }
        return this.elements.pop();
    }
}
```

```
let numbers = new Stack<number>(5);
```

```
function randBetween(low: number, high: number): number {
    return Math.floor(Math.random() * (high - low + 1) + low);
}
```

```
let numbers = new Stack<number>(5);

while (!numbers.isFull()) {
    let n = randBetween(1, 10);
    console.log(`Push ${n} into the stack.`)
    numbers.push(n);
}
```

4. Generic Interface

- A generic interface allows you to create an interface that can work with different types while maintaining type safety.

```
interface interfaceName<T> {
    // ...
}
```

```
interface interfaceName<U,V> {
    // ...
}
```

- Generic interfaces that describe object properties

```
interface Pair<K, V> {
    key: K;
    value: V;
}
```

```
let month: Pair<string, number> = {
    key: 'Jan',
    value: 1
};

console.log(month);
```

- Generic interfaces that describe methods

```
interface Collection<T> {
    add(o: T): void;
    remove(o: T): void;
}
```

```
class List<T> implements Collection<T>{
    private items: T[] = [];

    add(o: T): void {
        this.items.push(o);
    }
    remove(o: T): void {
        let index = this.items.indexOf(o);
        if (index > -1) {
            this.items.splice(index, 1);
        }
    }
}
```

```
let list = new List<number>();

for (let i = 0; i < 10; i++) {
    list.add(i);
}
```

THE END