TypeScript

Interfaces

Content

1. Interface
2. Extend Interfaces
3. Interface vs Abstract Class

1. Interface

- An "interface" is a programming construct that defines a contract or a set of methods that a class must implement.
- An interface defines the method signatures (names, parameters, and return types) without providing the actual implementation.
- It serves as a blueprint for classes to follow, enforcing a specific structure for the methods they need to provide.
- A class that implements an interface must provide concrete implementations for all the methods declared in that interface.

- Let’s start with a simple example

```
function getFullName(person: {
    firstName: string;
    lastName: string
}) {
    return `${person.firstName} ${person.lastName}`;
}

let person = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(getFullName(person));
```

```ts
interface Person {
    firstName: string;
    lastName: string;
}
```

```ts
function getFullName(person: Person) {
    return `${person.firstName} ${person.lastName}`;
}

let john = {
    firstName: 'John',
    lastName: 'Doe'
};

console.log(getFullName(john));
```

```
let jane = {
    firstName: 'Jane',
    middleName: 'K.'
    lastName: 'Doe',
    age: 22
};
```

```
let fullName = getFullName(jane);
console.log(fullName); // Jane Doe
```

Optional properties

- An interface may have optional properties. To declare an optional property, you use the question mark (?) at the end of the property name in the declaration.

```
interface Person {
    firstName: string;
    middleName?: string;
    lastName: string;
}
```

```
function getFullName(person: Person) {
    if (person.middleName) {
        return `${person.firstName} ${person.middleName} ${person.lastName}`;
    }
    return `${person.firstName} ${person.lastName}`;
}
```

Readonly properties

If properties should be modifiable only when the object is first created, you can use the readonly keyword before the name of the property

```
interface Person {
    readonly ssn: string;
    firstName: string;
    lastName: string;
}

let person: Person;
person = {
    ssn: '171-28-0926',
    firstName: 'John',
    lastName: 'Doe'
}
```

Class Types

```
interface Json {
    toJSON(): string
}
```

```
class Person implements Json {
    constructor(private firstName: string,
        private lastName: string) {
    }
    toJson(): string {
        return JSON.stringify(this);
    }
}
```

```
let person = new Person('John', 'Doe');
console.log(person.toJson());
```

2. Extend Interface

- Suppose that you have an interface called Mailable that many classes have already implemented.

```
interface Mailable {
    send(email: string): boolean
    queue(email: string): boolean
}
```

- You want to add a new method to the Mailable interface

```
later(email: string, after: number): void
```

- You can create a new interface that extends the Mailable interface

```
interface FutureMailable extends Mailable {
    later(email: string, after: number): boolean
}
```

```
class Mail implements FutureMailable {
    later(email: string, after: number): boolean {
        console.log(`Send email to ${email} in ${after} ms.`);
        return true;
    }
    send(email: string): boolean {
        console.log(`Sent email to ${email} after ${after} ms. `);
        return true;
    }
    queue(email: string): boolean {
        console.log(`Queue an email to ${email}.`);
        return true;
    }
}
```

Extending multiple interfaces

```
interface C {
    c(): void
}

interface D extends B, C {
    d(): void
}
```

3. Interface vs Abstract Class

- Interfaces and abstract classes are both powerful tools in TypeScript for designing and organizing your code, but they serve slightly different purposes.

Aspect	Interfaces	Abstract Classes
Purpose	Define contractual structure.	Provide common functionality and structure.
Implementation	Contains only method signatures.	Can contain implemented methods and abstract methods.
Multiple Inheritance	Supports multiple interface implementation.	Supports single class inheritance.
Implementation Flexibility	No implementation code in interfaces.	Mixes implemented and abstract methods.
Extensibility	Easily extendable by adding new properties/methods.	Can provide shared methods for derived classes.
Constructors	No constructors in interfaces.	Can have constructors for initialization.
Type Checking	Ensures objects adhere to the structure.	Provides a common type and functionality.
Instantiation	Interfaces can’t be instantiated.	Abstract classes can’t be instantiated directly.
Usage	Designing contracts and structure.	Sharing functionality among related classes.

THE END