Generic Interfaces - OOP Perspective

This lesson introduces generic types and discusses them from the perspective of Object-Oriented Programming.

WE'LL COVER THE FOLLOWING

Interfaces in Object-Oriented Programming

Interfaces in TypeScript are very flexible and have multiple uses. One way of looking at them is from the perspective of Object-Oriented Programming (OOP). In this context, an interface defines a list of methods that a class needs to implement in order to conform to the interface.

OOP interfaces are often used to fulfill the **Dependency Inversion Principle** from SOLID design principles. A class (e.g., an Angular component) might have dependencies (e.g., an Angular service). Dependencies are things that you need to provide to the constructor of the class to create an instance of it. The principle says that it's much better to have interfaces than concrete classes as dependencies.

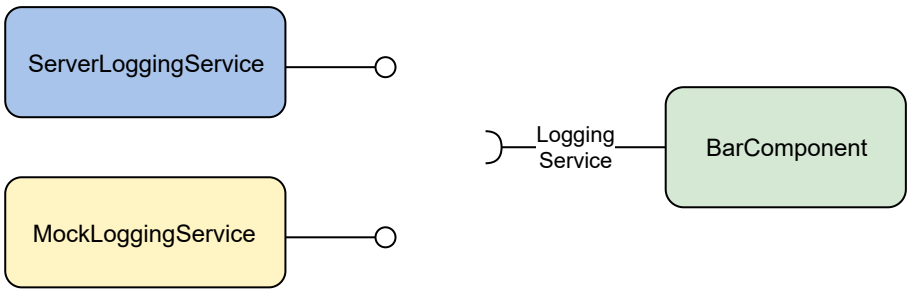
If the class requires a concrete class (e.g., a `ServerLoggingService` class), it's difficult to replace the dependency. For example, unit testing such a class is much harder since you have to provide an instance of this concrete class.

```
interface LoggingService {
    log(message: string): void;
}

class ServerLoggingService implements LoggingService {
    log(message: string): void {
        // send logs to the server
```

```
    }
}

class FooComponent {
    constructor(private loggingService: ServerLoggingService) {}
}

const foo = new FooComponent(new ServerLoggingService());
```

On the other hand, if the class only requires an interface (e.g., a `LoggingService` interface), you might provide any implementation of that interface; for example, a mock implementation for testing.

```
class BarComponent {
    constructor(private loggingService: LoggingService) {}
}

class MockLoggingService implements LoggingService {
    log(message: string): void {
        // don't do anything
    }
}

const bar = new BarComponent(new ServerLoggingService());
const testedBar = new BarComponent(new MockLoggingService());
```



In general, the OOP part of interfaces is their ability to express some *required behavior*, as opposed to *data* in the functional paradigm. This style of writing code might seem familiar to some Angular developers.

Generic OOP interfaces - `Observer`

An interface can specify a list of methods that must be provided by the implementation. A **generic interface** is very similar with the exception that some of the methods can be generic.

Let's look at the `Observer` interface from the RxJS library.

```
interface Observer<T> {
  closed?: boolean;
  next: (value: T) => void;
  error: (err: any) => void;
  complete: () => void;
}
```

The `Observer` interface represents an entity that can *observe* a stream of values. When the stream emits a value, the `next` method is called. When the stream encounters an error, the `error` method is called. Finally, when the stream completes, the `complete` method is called.

Stream of values is an abstract concept. We know that it represents some sequence of values, but we don't really care about the type of these values. That's why `Observer` is a generic interface. Its generic type argument, `T`, represents the type of values emitted by the stream. As mentioned previously, the `next` method is called for every emitted value. The value is passed as an argument to the `next`. We already know that `T` is the type of the value, so we can use it to type the `value` parameter of the `next` method.

Generic OOP interfaces - `ReadonlyArray`

Another interesting example is the `ReadonlyArray` interface that comes predefined with TypeScript.

```
interface ReadonlyArray<T> {
    // Skipped most of the methods for brevity.
    slice(start?: number, end?: number): T[];
    map<U>(callbackfn: (value: T, index: number, array: readonly T[]) => U
, thisArg?: any): U[];
    reduce<U>(callbackfn: (previousValue: U, currentValue: T, currentIndex
: number, array: readonly T[]) => U, initialValue: U): U;
    readonly [n: number]: T;
}
```

The `ReadonlyArray` is an interface that represents an immutable array. If you use `ReadonlyArray` to type the parameter of some function, you guarantee that the array passed to this function will never be modified. It's a compile-time guarantee, whereas at runtime, a regular array will be used

guarantee, whereas at runtime, a regular array will be used.

`ReadonlyArray` achieves this by only specifying a non-mutating subset of regular array methods. For example, it doesn't specify `push` or `pop`. What's more, its indexer is marked as `readonly`.

The interface is generic for the same reason `Observer` was generic; we don't care about the type of values contained in the array. We mark the type of the values as `T` and can later use the types in the method declarations. For example, the `slice` method returns an array of `T` values where a slice of the array will have the same value type as the source array.

Interestingly, a generic interface can have actual generic methods, see `map` as an example. Feel free to compare it with the `map` function explained in previous lessons. Here, `map` takes only one type argument, `U`, which represents the type of elements of the resulting array. The type of `callbackFn` is a function type that accepts a `value` of type `T` (the type argument of the interface) and returns a `U` (the type argument of the method).

```
interface ReadonlyArray<T> {

    map<U>(
        callbackfn: (value: T) => U,
        thisArg?: any
    ): U[];

}
```

The next lesson looks at generic interfaces from a different angle.