Chapter 9: Interfaces

An interfaces specifies a list of fields and functions that may be expected on any class implementing the interface. Conversely, a class cannot implement an interface unless it has every field and function specified on the interface.

The primary benefit of using interfaces, is that it allows one to use objects of different types in a polymorphic way. This is because any class implementing the interface has at least those fields and functions.

Section 9.1: Extending Interface

Suppose we have an interface:

```
interface IPerson {
    name: string;
    age: number;

    breath(): void;
}
```

And we want to create more specific interface that has the same properties of the person, we can do it using the `extends` keyword:

```
interface IManager extends IPerson {
    managerId: number;

    managePeople(people: IPerson[]): void;
}
```

In addition it is possible to extend multiple interfaces.

Section 9.2: Class Interface

Declare `public` variables and methods type in the interface to define how other typescript code can interact with it.

```
interface ISampleClassInterface {
  sampleVariable: string;

  sampleMethod(): void;

  optionalVariable?: string;
}
```

Here we create a class that implements the interface.

```
class SampleClass implements ISampleClassInterface {
  public sampleVariable: string;
  private answerToLifeTheUniverseAndEverything: number;

  constructor() {
    this.sampleVariable = 'string value';
    this.answerToLifeTheUniverseAndEverything = 42;
  }

  public sampleMethod(): void {
    // do nothing
  }
  private answer(q: any): number {
    return this.answerToLifeTheUniverseAndEverything;
  }
}
```

The example shows how to create an interface `ISampleClassInterface` and a class `SampleClass` that `implements` the interface.

Section 9.3: Using Interfaces for Polymorphism

The primary reason to use interfaces to achieve polymorphism and provide developers to implement on their own way in future by implementing interface's methods.

Suppose we have an interface and three classes:

```
interface Connector{
    doConnect(): boolean;
}
```

This is connector interface. Now we will implement that for Wifi communication.

```
export class WifiConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via wifi");
        console.log("Get password");
        console.log("Lease an IP for 24 hours");
        console.log("Connected");
        return true
    }

}
```

Here we have developed our concrete class named `WifiConnector` that has its own implementation. This is now type `Connector`.

Now we are creating our `System` that has a component `Connector`. This is called dependency injection.

```
export class System {
    constructor(private connector: Connector){ #inject Connector type
        connector.doConnect()
    }
}
```

`constructor(private connector: Connector)` this line is very important here. `Connector` is an interface and must have `doConnect()`. As `Connector` is an interface this class `System` has much more flexibility. We can pass any Type which has implemented `Connector` interface. In future developer achieves more flexibility. For example, now developer want to add Bluetooth Connection module:

```
export class BluetoothConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via Bluetooth");
        console.log("Pair with PIN");
        console.log("Connected");
        return true
    }
}
```

See that Wifi and Bluetooth have its own implementation. Their own different way to connect. However, hence both have implemented Type `Connector` the are now Type `Connector`. So that we can pass any of those to `System` class as the constructor parameter. This is called polymorphism. The class `System` is now not aware of whether it is Bluetooth / Wifi even we can add another Communication module like Infrared, Bluetooth5 and whatsoever by just implementing `Connector` interface.

This is called Duck typing. `Connector` type is now dynamic as `doConnect()` is just a placeholder and developer implement this as his/her own.

if at `constructor(private connector: WifiConnector)` where `WifiConnector` is a concrete class what will happen? Then `System` class will tightly couple only with WifiConnector nothing else. Here interface solved our problem by polymorphism.

Section 9.4: Generic Interfaces

Like classes, interfaces can receive polymorphic parameters (aka Generics) too.

Declaring Generic Parameters on Interfaces

```
interface IStatus<U> {
    code: U;
}

interface IEvents<T> {
    list: T[];
    emit(event: T): void;
    getAll(): T[];
}
```

Here, you can see that our two interfaces take some generic parameters, **T** and **U**.

Implementing Generic Interfaces

We will create a simple class in order to implements the interface **IEvents**.

```
class State<T> implements IEvents<T> {

    list: T[];

    constructor() {
        this.list = [];
    }

    emit(event: T): void {
        this.list.push(event);
    }

    getAll(): T[] {
        return this.list;
    }

}
```

Let's create some instances of our **State** class.

In our example, the `State` class will handle a generic status by using `IStatus<T>`. In this way, the interface

`IEvent<T>` will also handle a `IStatus<T>`.

```
const s = new State<IStatus<number>>();

// The 'code' property is expected to be a number, so:
s.emit({ code: 200 }); // works
s.emit({ code: '500' }); // type error

s.getAll().forEach(event => console.log(event.code));
```

Here our `State` class is typed as `IStatus<number>`.

```
const s2 = new State<IStatus<Code>>();

//We are able to emit code as the type Code
s2.emit({ code: { message: 'OK', status: 200 } });

s2.getAll().map(event => event.code).forEach(event => {
    console.log(event.message);
    console.log(event.status);
});
```

Our `State` class is typed as `IStatus<Code>`. In this way, we are able to pass more complex type to our emit method.

As you can see, generic interfaces can be a very useful tool for statically typed code.

Section 9.5: Add functions or properties to an existing interface

Let's suppose we have a reference to the `JQuery` type definition and we want to extend it to have additional functions from a plugin we included and which doesn't have an official type definition. We can easily extend it by declaring functions added by plugin in a separate interface declaration with the same `JQuery` name:

```
interface JQuery {
  pluginFunctionThatDoesNothing(): void;

  // create chainable function
  manipulateDOM(HTMLElement): JQuery;
}
```

The compiler will merge all declarations with the same name into one - see declaration merging for more details.

Section 9.6: Implicit Implementation And Object Shape

TypeScript supports interfaces, but the compiler outputs JavaScript, which doesn't. Therefore, interfaces are effectively lost in the compile step. This is why type checking on interfaces relies on the *shape* of the object - meaning whether the object supports the fields and functions on the interface - and not on whether the interface is actually implemented or not.

```
interface IKickable {
  kick(distance: number): void;
}
class Ball {
  kick(distance: number): void {
    console.log("Kicked", distance, "meters!");
  }
}
let kickable: IKickable = new Ball();
kickable.kick(40);
```

So even if Ball doesn't explicitly implement IKickable, a Ball instance may be assigned to (and manipulated as) an IKickable, even when the type is specified.

Section 9.7: Using Interfaces to Enforce Types

One of the core benefits of TypeScript is that it enforces data types of values that you are passing around your code to help prevent mistakes.

Let's say you're making a pet dating application.

You have this simple function that checks if two pets are compatible with each other...

```
checkCompatible(petOne, petTwo) {
  if (petOne.species === petTwo.species &&
      Math.abs(petOne.age - petTwo.age) <= 5) {
    return true;
  }
}
```

This is completely functional code, but it would be far too easy for someone, especially other people working on this application who didn't write this function, to be unaware that they are supposed to pass it objects with 'species' and 'age' properties. They may mistakenly try `checkCompatible(petOne.species, petTwo.species)` and then be left to figure out the errors thrown when the function tries to access petOne.species.species or petOne.species.age!

One way we can prevent this from happening is to specify the properties we want on the pet parameters:

```
checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age: number}) {
    //...
}
```

In this case, TypeScript will make sure everything passed to the function has 'species' and 'age' properties (it is okay if they have additional properties), but this is a bit of an unwieldy solution, even with only two properties specified. With interfaces, there is a better way!

First we define our interface:

```
interface Pet {
  species: string;
  age: number;
  //We can add more properties if we choose.
}
```

Now all we have to do is specify the type of our parameters as our new interface, like so...

```
checkCompatible(petOne: Pet, petTwo: Pet) {
  //...
}
```

... and TypeScript will make sure that the parameters passed to our function contain the properties specified in the Pet interface!