[TS] 4. Interface in TypeScript

by jbee

`interface`는 자바스크립트 개발자에게 친숙하지 않은 용어일꺼라고 생각됩니다. 하지만 정적 타이핑에 있어서 큰 부분을 차지하고 있는 syntax에 대해 알아봅니다.

Contents

- Interface?
- Useful Interface
- Available properties
    - Optional
    - readonly
- Interface Type
    - Function Type
    - Indexable Type
- Class interface

Interface?

`interface` 란 **객체의 껍데기 또는 설계도** 라고 할 수 있을 것 같습니다. 자바스크립트에서는 클래스도 함수도 결국 모두 객체인데요, 클래스 또는 함수의 '틀'을 정의할 때 사용할 수 있는 것이 인터페이스 입니다.

여러 함수가 특정한 시그니처를 동일하게 가져야 할 경우 또는 여러 클래스가 동일한 명세를 정의해야하는 경우 인터페이스를 통해서 정의할 수 있습니다. 인터페이스는 특정 타입으로서 사용될 수 있으며, `implements` 의 대상이 될 수 있습니다. 객체에 인터페이스를 적용하는 경우 또는 반환값 등을 설정할 때 타입으로 이용될 수 있습니다.

Useful Interface

함수에서 어떤 객체를 받아야 할 경우, 해당 type을 어떻게 정의할 수 있을까요?

```
// AjaxUtils.ts
export async function fetchBasic(param: {url: string}): P
  const response = await fetch(param.url);
  return response.json();
}
```

위와 같이 `param` 이라는 인자의 타입을 literal 형식으로 정의할 수 있습니다. 그런데 그 객체의 프로퍼티가 많아지는 경우에는 어떻게 정의할까요? (반환값 형식에 `Promise<?>` 형식으로 지정을 해줬는데요, 이는 Generics 부분에서 다룰 예정입니다.)

```
export async function fetchData(param: {baseUrl: string, 
  const {baseUrl, type, subject} = param;
  const response = await fetch(`${baseUrl}/${subject}`);
  const contentType = response.headers.get("content-type"

  if (response.ok && contentType && contentType.includes(
    return response.json();
  }
  throw Error("Invalid baseUrl or subject");
}
```

literal로 인자의 타입을 정의하다보니 함수의 signature가 너무 길어졌습니다. 그런데 여기서 `Promise<Response>` 이 부분도 정의가 필요합니다. 따라서 다음과 같이 길어집니다.

```
export async function fetchData(param: {
  baseUrl: string,
  type: string,
  subject?: string
}): Promise<{
  id: number;
  name: string;
  company: string;
}[]> {
  //...
}
```

뭔가 최대한 가독성을 좋게 하기 위해서 개행을 했지만 성공하진 못한 것 같습니다. 문제는 여기서 끝이 아닙니다. 이 함수를 호출하는 곳에서는 다음과 같은 일이 벌어집니다.

```
private async ajax(): Promise<{id: number; name: string;
  const data: {id: number; name: string; company: string;
    baseUrl: "http://localhost:3000",
    subject: "users",
    type: "application/json"
  });
  // ...do something
  return data;
}
```

이 혼란의 상황을 interface를 통해 깔끔하게 정리할 수 있습니다. 인자와 반환 값을 interface를 통해 정리해보겠습니다.

```ts
// interfaces.ts
interface Character {
  id: number;
  name: string;
  company: string;
}

export interface dataFormat {
  charaters: Character[];
}

export interface fetchDataParam {
  baseUrl: string;
  type: string;
  subject: string;
}
```

위와 같이 `interfaces.ts` 라는 파일을 생성하여 인터페이스들을 정의할 수 있습니다.

```
// AjaxUtils.ts
import { fetchDataParam, dataFormat } from "./interfaces"
export async function fetchData(param: fetchDataParam): P
  // ...
}
```

`AjaxUtils.ts` 에서는 정의한 인터페이스를 import하여 인자와 반환값에 해당 인터페이스를 통해 타입을 정의할 수 있습니다.

```
// Controller.ts
import { fetchDataParam, dataFormat } from "./interfaces"

private async ajaxCall(): Promise<dataFormat> {
  const param: fetchDataParam = {
    baseUrl: "http://localhost:3000",
    subject: "users",
    type: "application/json"
  };
  const data: dataFormat = await fetchData(param);
  console.log(data);
  return data;
}
```

호출하는 부분에서도 마찬가지로 `interfaces.ts` 파일에서 필요한 인터페이스를 import하여 타입을 지정해줍니다. 예시 코드에서 처럼 인터페이스를 정의하여 인자에 정의하던 타입들을 깔끔하게 정리할 수 있습니다. 또한 다른 파일에서 해당 함수를 정의하는 부분과 호출하는 부분이 다를 때, 하나의 인터페이스를 공유할 수 있습니다. 인터페이스를 통일시키는 것이 중요할 때 매우 유용하게 사용할 수 있습니다.

Available properties

인터페이스에도 클래스와 동일하게 optional하게 property를 지정할 수 있으며 readonly 타입으로 property를 지정할 수 있습니다.

optional property로 지정한 `amount`에 대해서는 구현하지 않아도 에러가 발생하지 않는 것을 확인할 수 있습니다.

Readonly properties

```
export interface WeatherSpec {
  readonly type: string;
  amount: number;
}
const rainfall: WeatherSpec = {
  type: "rainfall",
  amount: 24,
}
rainfall.type = "snow"; // Error!
rainfall.amout += 3; // OK!
```

에러 메시지는 다음과 같습니다. `[!] Error: Cannot assign to 'type' because it is a constant or a read-only property.`
위와 같이 interface에서 readonly로 지정한 프로퍼티에 대해서는 그 값을 바꿀 수 없습니다. 이는 변수를 사용할 때 사용하는 `const` 키워드와 동일한 역할을 수행한다고 이해할 수 있습니다.

Function Type

인터페이스의 프로퍼티로 함수의 시그니처를 정의할 수 있습니다. 반환하는 형식 또는 그 값이 다르지만 시그니처를 통일시켜야 하는 경우가 존재할 수 있습니다. 그럴 경우 다음과 같이 interface를 설계하여 함수를 구현할 수 있습니다.

```
interface TimeFunc {
  (hour: number, minutes: number): any;
}

const buildTimeStamp: TimeFunc = (hour, minutes): number =
  if (minutes < 10) {
    return Number(`${hour}0${minutes}`);
  }
  return  Number(`${hour}${minutes}`);
}

const buildTimeText: TimeFunc = (hour, minutes): string =>
  if (minutes < 10) {
    return `${hour}시 0${minutes}분`;
  }
  return `${hour}시 ${minutes}분`;
}

buildTimeStamp(12, 33); //1233
buildTimeText(12, 33); //12시 33분
```

반환 타입을 제외하고 동일한 형식의 함수를 정의했습니다. 이것은 특정 콜백 함수를 받는 함수를 구현할 때 그 유용성이 더 빛을 발합니다.

```
const buildTime = (timeText: string, cb: TimeFunc) => {
  const hour: number = Number(timeText.split(":")[0]);
  const minutes: number = Number(timeText.split(":")[1]);

  return cb(hour, minutes);
}
console.log(buildTime("12:33", buildTimeStamp)); //1233
console.log(buildTime("12:33", buildTimeText)); //12시 33분
```

콜백 함수를 인자로 받을 때 해당하는 시그니처가 통일되어야 하는 부분을 인터페이스를 통해 해결할 수 있습니다.

Indexable Types

자바스크립트에서 다음과 같은 코드는 매우 자연스럽습니다.

```
const obj = {
  first: 1,
  second: 2,
};
Object.keys(obj).forEach(key => console.log(obj[key]));
```

즉, 객체의 프로퍼티에 접근할 때, 동적으로 생성된 `key` 를 통해 객체의 프로퍼티에 `[]` 표기법으로 접근하는 경우입니다. 하지만 이 코드는 타입스크립트에서 동작하지 않습니다.

`[!] Element implicitly has an 'any' type because type '{ first: number; second: number; }' has no index signature.` 이란 에러를 발생시킵니다. 왜냐하면 정의한 `obj` 라는 객체에 index signature가 없기 때문입니다. 따라서 이 에러는 다음과 같이 해결할 수 있습니다.

```
interface Indexable {
  [key: string]: any;
};
const obj: Indexable = {
  first: 1,
  second: 2,
};

Object.keys(obj).forEach((key: string) => obj[key]);
```

`Indexable` 이란 이름의 인터페이스를 정의해준 다음, `string` 타입의 key 에 `any` 타입을 지정해줍니다. 이 인터페이스를 통해서 객체를 생성하면 `[]` 표기법을 통해 객체의 프로퍼티에 접근할 수 있습니다.

Class interface

인터페이스를 클래스에서도 사용할 수 있습니다. 상위 클래스를 `extends` 라는 키워드로 상속하듯이 `implements` 라는 키워드로 인터페이스를 구현할 수 있습니다. 클래스는 인터페이스를 `implements` 하면서 인터페이스에 명세되어 있는 기능들을 구현해야 하는 의무를 갖게 됩니다.

인터페이스에서는 클래스의 프로퍼티(필드 멤버), 메소드 등을 정의할 수 있습니다. 또한 optional 한 명세 또한 정의할 수 있습니다.

```
interface Movable {
  velocity: number;
  move(time: number): Position;
  startPos?: Position;
}

class BMWCar implements Movable {

}
```

여기까지 입력했을 때 나타나는 에러 메세지는 다음과 같습니다. `[!] Class 'Car' incorrectly implements interface 'Movable'.` . 특정 인터페이스를 구현한 클래스는 인터페이스에 정의된 명세를 구현해야 합니다.

즉 여기서는 `velocity` 라는 프로퍼티를 포함해야 하며, `move` 라는 메소드를 구현해야만 합니다. 이 `BMWCar` 클래스는 생성할 때 다음과 같이 생성할 수 있습니다.

```
class BMWCar implements Movable {
  velocity: number;
  constructor(velocity) {
    this.velocity = velocity;
  }

  move(time: number): Position {
    //... do something
  }
}

const bmw: Movable = new BMWCar();
```

`BMWCar` 라는 타입 말고도 해당 클래스에서 구현한 인터페이스인 `Movable` 타입으로 지정할 수 있습니다.

Public Property

TypeScript Official Document에 `Interfaces describe the public side of the class, rather than both the public and private side.` 이런 말이 나옵니다. `인터페이스`를 통해 구현해야 함을 명시하는 메소드는 `private` 접근 제어자로 정의될 메소드가 아니라 `public` 접근 제어자로 정의될 메소드이어야 한다고 합니다.

즉, 인터페이스를 통해 명세를 정의할 때는 `private` 속성말고 `public` 속성에 대해 정의합니다.

마무리

인터페이스를 통해 보다 세밀한 구조 설계와 추상화가 가능해졌습니다. 해당 포스팅 외 다른 타입스크립트 포스팅은 여기에서 보실 수 있으며 예제에 사용된 코드는 여기에서 확인하실 수 있습니다.
감사합니다.

3. Interface in TypeScript end