[TS] 3. Function in TypeScript

by. Jbee

TypeScript에서 함수를 정의하는데 있어서 몇 가지 추가된 기능에 대해 살펴봅니다.

Table of Contents

인자와 반환값의 타입을 설정한다.

함수 또는 메소드를 정의할 때, 타입을 정의해줍니다.

ES6 code

```
function getMonthFromString(dateOfStringFormat) {
  const monthOfNumberFormat = parseInt(dateOfStringFormat
  return monthOfNumberFormat;
}
```

위 함수는 '201712'이라는 문자열을 받아서 해당하는 월을 반환하는 함수입니다. (반환값의 형식을 명시하기 위해 바로 return하지 않고 변수에 임시로 받아준 뒤 반환합니다.) 위와 같이 String 타입의 인자를 받아야 한다는 것을 명시해줘야 하기 때문에 변수명부터 굉장히 이상해집니다. 자바스크립트에는 타입이라는 것이 없기 때문에 메소드명 또는 변수명에 타입을 명시할 수 밖에 없습니다.

만약 Number 타입의 201712를 인자로 넘겨준다면 Number에는 substring 이라는 함수가 없기 때문에 에러가 발생합니다. 위 함수를 보다 안정적으로 작성하기 위해서는 다음과 같은 if 문이 필요하게 됩니다.

```
function getMonthFromString(dateOfStringFormat) {
  if (typeof dateOfStringFormat !== "string") {
    throw Error("Invalid format of parameter");
  }
  const monthOfNumberFormat = parseInt(dateOfStringFormat
  return monthOfNumberFormat;
}
```

기본적으로 함수가 수행해야하는 비즈니스 로직 외 불필요한 방어코드가 코드를 더럽히고 있습니다. 위 함수를 TypeScript 함수로 변경해보겠습니다.

```
const getMonth = (date: string): number => {
  return parseInt(date.substring(4, 6), 10);
}
```

인자와 반환값에 타입을 설정하여 메소드 이름과 변수명이 훨씬 짧아졌습니다. 그럼에도 불구하고 해당 함수가 하는 역할을 ES6로 작성했을 때보다 명확해졌으며, 불필요한 방어코드 마저 사라졌습니다.

wtfjs에서 확인하실 수 있지만 자바스크립트에서는 타입이 멋대로(사실은 매우 다양한 규칙을 기반으로) 캐스팅되는 경우가 많은데요, 이를 방지하기 위해 우리는 불필요한 방어코드를 작성해왔습니다. 타입을 지정함으로써 이러한 작업을 최소화 할 수 있습니다.

Default Parameter, Rest Parameter

해당 스펙은 ES6 표준 스펙에서도 지원하고 있는 스펙이므로 구체적은 설명은 넘어가겠습니다. 자세한 내용은 첨부하는 포스팅을 확인해주세요.

- ES6. Rest Parameter
- ES6. Default Parameter

TypeScript에서도 해당 스펙을 지원합니다.

Default parameter TypeScript code

```
const getRandomNumber = (min: number = 0, max: number = 1
    return Math.floor(Math.random() * (max - min)) + min;
}

console.log(getRandomNumber(1)); // OK!
```

위 코드에서는 인자가 넘겨지지 않았을 경우(`undefined`), 지정해준 값으로 인자를 설정합니다. 명시적으로 `null` 을 인자로 넘겨주면 default로 설정된 parameter를 무시하고 `null` 을 인자로 넘깁니다. 지정한 인자를 모두 넘기지 않으면 에러를 뱉던 TypeScript도 default parameter가 지정되어 있으면 에러를 발생시키지 않습니다.

Rest parameter TypeScript code

```
export const setSkills = (...skills: string[]): void => {
    // ...
}
```

`rest parameter`의 타입은 배열(array)이므로 인자에 해당하는 타입을 설정해줍니다.

Optional Parameter

TypeScript에서는 default parameter 없을 경우, signature에서 정의한대로 인자를 넘겨주지 않으면 에러가 발생합니다. 하지만 파라미터를 넘겨주지 않아도 되도록 설정할 수 있습니다.

```
const setSpec = (major: string, option?: string): void =>
  console.log(major);
  console.log(option);
}
setSpec("Computer Science");
// console> Computer Science
// console> undefined
```

> `?` 를 통해서 함수에 optional한 parameter를 지정할 수 있습니다.

Union Type

파라미터에 타입을 지정할 때, 두 가지 이상의 타입이 지정할 필요가 있을 수 있는데요, 그럴 때 Union type을 통해서 파라미터의 타입을 지정해줄 수 있습니다.

```
sayName(position: string | boolean | number): void {
  if (typeof position === "string") {
    console.log(`string type position`);
  } else if (typeof position === "boolean") {
    console.log(`boolean type position`)
  } else {
    console.log(`else`);
  }
}
```

위 코드에서는 `position` 이라는 파라미터가 `string` , `boolean` , `number` 세 가지의 타입일 수 있다고 signature를 지정했습니다. (예제가 송구스럽네요)

Overloading

바로 이전 Class 포스팅에서 자바와 같은 오버로딩을 지원하지 않는다고 했는데요, `optional parameter` 와 `union type` 그리고 `any` 라는 타입을 사용하면 자바에서 구현하는 것과는 조금 다르지만 오버로딩을 구현할 수 있습니다.

```
class Person {
  //..
  sayName(position: string, option?: string): void;
  sayName(position: boolean, option: string): void;
  sayName(position: string | boolean, option: any): any {
    if (typeof position === "string") {
      console.log(`string type position`);
    } else if (typeof position === "boolean") {
      console.log(`boolean type position`)
    } else {
      console.log(`else`);
    }
  }
  //..
}
```

위와 같이 동일한 메소드 명에 대해 여러 Signature를 정의할 수 있습니다. 위 코드에서는 `sayName` 이라는 메소드의 Signature가 세 개이며 마지막 메소드에서만 이를 구현하고 있습니다. 그리고 메소드 body에서는 parameter의 타입으로 분기를 하여 로직을 수행하고 있습니다.

```
const person: Person = new Person();

person.sayName("FrontEnd");
person.sayName("FrontEnd", "optional);
person.sayName(false, "option required");
// person.sayName(true); Error! (1)
// person.sayName(1); Error! (2)
```

`sayName`을 호출하게 되면, 메소드를 **호출하는 시점에서** 각 상황에 맞는 signature가 적용됩니다. `Error (1)`을 보면 `boolean` 타입이 인자로 넘어갔을 경우의 signature에 따라 `option`이 `required` 인자이므로 에러가 발생합니다. `Error (2)`는 어느 signature와도 일치하지 않으니 에러가 발생합니다. 이렇게 TypeScript에서는 여러 signature를 정의한 뒤 메소드 내에서 이를 분기하여 오버로딩을 구현할 수 있습니다.

마무리

TypeScript 좀 더 안정성 있는, 간결한, 가독성이 좋은 함수를 작성할 수 있게 되었습니다. 어떻게 보면 타이핑이 길어지는 결과처럼 보일 수 있겠지만 타입의 명시가 필요한 부분에 있어서는 오히려 코드가 더 짧아지게 되었습니다.

감사합니다.