7장 에러 처리

타입스크립트는 런타임에 발생할 수 있는 예외를 컴파일 타임에 잡을 수 있도록 최선을 다한다.

⇒ 하지만, 런타임 예외는 언젠가 발생하기 마련이다.

- 네트워크 장애, 파일시스템 장애, 입력 파싱 에러, 스택 오버플로우, 메모리 부족 에러까지 모두 막을 수는 없다.

- 타이스크립트에서 에러를 표현하고 처리하는 가장 일방적인 방식은 4가지이다.
  - null 반환
  - 예외 던지기
  - 예외 반환
  - Option 타입

7.1 Null 반환

null을 반환하는 방식.

```
{
    const prompt = (message: string) => {
        console.log(message)
        return '10-13'
    }
    const ask = () => {
        return prompt('생일이 언제에요?')
    }

    const isValid = (date: Date) => {
        return !Number.isNaN(date.getTime()) && date !== null;
    }
    const parse = (birthday: string): Date | null => {
        let date = new Date(birthday);
        if (!isValid(date)) {
            return null
        }
        return new Date(birthday);
    }

    let date = parse(ask());

    if(date) {
        console.info(`날짜는 ${date.toISOString()}`);
    }
}
```

장점

- 타입 안정성을 유지하며 에러를 처리하는 가장 간단한 방식이다.

단점

- 문제가 생긴 원인을 알 수 없다.
  - 로그를 일일히 확인하며 디버깅 해야 한다.
- 조합이 어려워진다.
  - 모든 연산에서 null을 확인해야한다.
  - 연산을 중첩하거나 연결시에 코드가 지저분해진다.

7.2 예외던지기

- 문제가 발생하면 null 대신 예외를 던지는 방식이다.

장점

- 어떤 문제인지에 따라 대처가 가능하다.
- 디버깅에 도움이되는 메타데이터도 얻을 수 있다.

RangeError 예외 던지기

```
{
    const prompt = (message: string) => {
        console.log(message)
        return 'idk'
    }
    const ask = () => {
        return prompt('생일이 언제에요?')
    }

    const isValid = (date: Date) => {
        return !Number.isNaN(date.getTime()) && date !== null;
    }
    const parse = (birthday: string): Date | null => {
        let date = new Date(birthday);
        if (!isValid(date)) {
            throw new RangeError('YYYY/MM/DD 형식에 맞춰서 입력해주세요.')
        }
        return new Date(birthday);
    }

    try {
        let date = parse(ask());

        if(date) {
            console.info(`날짜는 ${date.toISOString()}`);
        }

    } catch (e) {
        if(e instanceof RangeError) {
            console.error(e.message);
        }
    }
}
```

커스텀 에러 타입 던지기

- 어떤 문제가 생겼는지 알려줄 수 있다.

- 문제가 생긴 이유도 설명할 수 있다.

  ⇒ `try… catch` 에서 연쇄적이고 중첩된 동작을 효율적으로 만들 수 있다.

```
class InvalidDateFormatError extends RangeError {}

class DateIsInTheFutureError extends RangeError {}

// ------------------------------------------------------------
{
    const prompt = (message: string) => {
        console.log(message)
        return '2024-10-13'
    }
    const ask = () => {
        return prompt('생일이 언제에요?')
    }

    const isValid = (date: Date) => {
        return !Number.isNaN(date.getTime()) && date !== null;
    }
    const parse = (birthday: string): Date | null => {
        let date = new Date(birthday);
        if (!isValid(date)) {
            throw new InvalidDateFormatError('YYYY/MM/DD 형식에 맞춰서 입력해주세요.')
        }
        if (date.getTime() > Date.now()) {
            throw new DateIsInTheFutureError('아직 태어나지 않으셨군요!!');
        }
        return new Date(birthday);
    }

    try {
        let date = parse(ask());

        if(date) {
            console.info(`날짜는 ${date.toISOString()}`);
        }

    } catch (e) {
        if(e instanceof InvalidDateFormatError) {
            console.error(e.message);
        }
        if(e instanceof DateIsInTheFutureError) {
            console.error(e.message);
        }
    }
}
```

→ 이런 경우 어떤 에러가 던져지는지 알고 처리하기 위해서

1. 함수 이름에 명시하거나
2. 주석으로 정보를 추가하는 것이 좋다.

```
                    /**
         * @throw {InvalidDateFormatError} 사용자가 생일을 잘못 입력함
```

```
         * @throw {DateIsInTheFutureError} 사용자가 생일을 미래 날짜로 입력함
         */
```

기타 참고 자료

RangeError - JavaScript | MDN

7.2 예외반환

타입스크립트는 throws 문을 지원하지 않는다.

→ 하지만, **유니온** 을 통해 비슷하게 흉내낼 수 있다.

```
const parse = (birthday: string): Date | InvalidDateFormatError | DateIsInTheFutureError => {
        let date = new Date(birthday);
        if (!isValid(date)) {
            return new InvalidDateFormatError('YYYY/MM/DD 형식에 맞춰서 입력해주세요.')
        }
        if (date.getTime() > Date.now()) {
            return new DateIsInTheFutureError('아직 태어나지 않으셨군요!!');
        }
        return new Date(birthday);
}
```

이 경우 유니온에 해당하는 상황을 처리해야 한다.

`( Date | InvalidDateFormatError | DateIsInTheFutureError )`

⇒ 그렇지 않으면 컴파일 타입에 TypeError가 발생한다.

```
class InvalidDateFormatError extends RangeError {}

class DateIsInTheFutureError extends RangeError {}

// ------------------------------------------------------------
{
    const prompt = (message: string) => {
        console.log(message)
        return '2024-10-13'
    }
    const ask = () => {
        return prompt('생일이 언제에요?')
    }

    const isValid = (date: Date) => {
        return !Number.isNaN(date.getTime()) && date !== null;
    }
    /**
     * @throw {InvalidDateFormatError} 사용자가 생일을 잘못 입력함
     * @throw {DateIsInTheFutureError} 사용자가 생일을 미래 날짜로 입력함
     */
    const parse = (birthday: string): Date | InvalidDateFormatError | DateIsInTheFutureError => {
        let date = new Date(birthday);
        if (!isValid(date)) {
            return new InvalidDateFormatError('YYYY/MM/DD 형식에 맞춰서 입력해주세요.')
```

```
        }
        if (date.getTime() > Date.now()) {
            return new DateIsInTheFutureError('아직 태어나지 않으셨군요!!');
        }
        return new Date(birthday);
    }

    try {
        let date = parse(ask());

        if(date instanceof InvalidDateFormatError) {
            console.error(date.message);
        } else if(date instanceof DateIsInTheFutureError) {
            console.error(date.message);
        } else {
            console.info(`날짜는 ${date.toISOString()}`);
        }

    } catch (e) {
        console.error(e.message);
    }
}
```

타입 시스템의 수행

1. `parse` 시그니처에서 발생 가능한 예외 나열
2. 메서드 사용자에게 어떤 에러가 발생하는지 전달
3. 메서드 사용자는 각각 에러를 모두 처리하거나 다시 던지도록 강제.

명시적으로 아래처럼 한번에 처리도 가능하다.

```
if (date instanceof Error) {
            console.error(date.message);
} else {
            console.info(`날짜는 ${date.toISOString()}`);
}
```

→ 단순하지만, 실패 유형과 추가 정보를 얻을 수 있는 길을 알려준다.

7.4 Option 타입

- 특수 목적 데이터 타입을 사용해 예외를 표현하는 방법

  — 이런 데이터 타입을 사용하지 않는 다른 코드와는 호환하지 않는다.

- 여러 연산을 연쇄적으로 수행할 수 있게 된다.

  → `Try` , `Option` , `Either`

  - 컨테이너 자체적으로 몇 가지 메서드를 제공
  - 값이 없어도 여러가지 연산을 연쇄적으로 수행
  - 값을 포함하면, 어떤 자료 구조로도 컨테이너를 구현

- 단, Option은 에러가 발생한 이유를 사용자에게 알려주지 않는다.

- 무언가가 잘못되었다는 사실만을 알린다.

배열 구조의 컨테이너

```
{
    const prompt = (message: string) => {
        console.log(message)
        return '2024-10-13'
    }
    const ask = () => {
        return prompt('생일이 언제에요?')
    }

    const isValid = (date: Date) => {
        return !Number.isNaN(date.getTime()) && date !== null;
    }

    const parse = (birthday: string): Date[] => {
        let date = new Date(birthday);
        if (!isValid(date)) {
            return []
        }
        return [date]
    }

    let date = parse(ask());

    date.map(_ => _.toISOString())
        .forEach(_ => console.info(`날짜는`, _))
}
```

- 언제든 실패할 수 있는 여러 동작을 연쇄적으로 수행시에 효과적이다.
    - **가정하기**
        a. `prompt` , `parse` 성공과 실패가 가능하다고 가정
        b. 사실은 `prompt` 도 실패할 수 있다.
        (사용자가 입력 취소하면 에러가 발생하며, 프로그램 계산을 이어 갈 수 없다.)
        c. 다른 Option을 이용하여 처리 할 수 있다.
    - `Flatten`

```
const flatten = <T>(array : T[][]) : T[] =>  {
    return Array.prototype.concat.apply([], array);
}

// ------------------------------------------------------------------
{
    const prompt = (message: string) => {
        console.log(message);
        return null;
    }
    const ask = () => {
        let result = prompt('생일이 언제에요?')
```

```
            if(result === null) {
                return []
            }
            return [result]
        }

        const isValid = (date: Date) => {
            return !Number.isNaN(date.getTime()) && date !== null;
        }

        const parse = (birthday: string): Date[] => {
            let date = new Date(birthday);
            if (!isValid(date)) {
                return []
            }
            return [date]
        }

        // ask()
        //     .map(parse)
        //     .map(date => date.toISOString()) //Property 'toISOString' does not exist on ty

        flatten(ask().map(parse))
            .map(date => date.toISOString())
            .forEach(date => console.log())
    }
```

Option and Some 사용하기

Option이란?

- `Some<T>` , `None` 이 구현하게 될 인터페이스이다.
- `Option` 은 **타입**이기도 하고, **함수**이기도 하다.
  - **타입 관점** → `Some` 과 `None` 의 슈퍼타입을 의미한다.
  - **함수 관점** → Option 타입의 새 값을 만드는 기능이다.

```
interface Option<T> {} // Some<T>, None을 공유하는 인터페이스

class Some<T> implements Option<T> { //연산에 성공하여 값이 만들어짐.
    constructor(private value: T) {}
}

class None implements Option<never> {} // 연산 실패하여 값이 없음.
```

  - 배열 기반의 구현
    - `Option<T>` → `[T] | []`
    - `Some<T>` → `[T]`
    - `None` → `[]`

`flatMap` , `getOrElse` 연산 정의하기

`flatMap` : 비어있을 수도 있는 Option 연산을 연쇄적으로 수행하는 수단

`getOrElse` : Option 에서 값을 가져옴.

```
interface Option<T> {
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
  }

class Some<T> extends Option<T> {
    constructor(private value: T) {}
}

class None extends Option<never> {}
```

인터페이스에 대한 구현

```
class Some<T> implements Option<T> {
  constructor(private value: T) { }
  flatMap<U>(f: (value: T) => Option<U>): Option<U> {
    return f(this.value)
  }
  getOrElse(): T {
    return this.value
  }
}
```

1. `flatMap` 은 값을 받는 f 함수를 인자로 받는다.
2. U 타입의 값을 포함하는 Option으로 반환한다.
3. `flatMap` 은 Option을 인자로 건네고 f를 호출한다.
4. 새로운 `Option<U>` 을 반환한다.
5. getOrElse 는 Option 을 포함하는 값과 같은 T를 받는다.
    i. Option이 None이면 기본값을 반환한다.
    ii. Option이 Some이면 (존재하면) Option 안의 값을 반환한다.

```
interface Option<T> {
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
  }

  class Some<T> implements Option<T> {
    constructor(private value: T) { }
    // Some<T>를 인수로 사용된, 전달된 f로 새로운 Option을 만들어 반환한다.
    flatMap<U>(f: (value: T) => Option<U>): Option<U> {
      return f(this.value)
    }
    // Some<T>에서 호출 시, Some<T>를 반환한다.
    getOrElse(): T {
      return this.value
    }
  }

  class None implements Option<never> {
    // 계산 실패이므로 항상 None, 회복할 수 없다.
    flatMap<U>(): Option<U> {
      return this
    }
    // 항상 기본값을 반환한다.
    getOrElse<U>(value: U): U {
```

```
      return value
    }
  }
```

flatMap 호출 결과 타입

	Some	None
Some	Some	None
None	None	None

→ None은 항상 None을 반환하며, Some는 f의 호출 결과에 따라 다르다.

구체적인 타입 제공에 대한 시그니처 오버로드

```
  interface Option<T> { //시그니처 오버로드
    flatMap<U>(f: (value: T) => None): None;
    flatMap<U>(f: (value: T) => Option<U>): Option<U>
    getOrElse(value: T): T
  }

  class Some<T> implements Option<T> {
    constructor(private value: T) { }
    flatMap<U>(f: (value: T) => None): None
    flatMap<U>(f: (value: T) => Some<U>): Some<U>
    flatMap<U>(f: (value: T) => Option<U>): Option<U> {
      return f(this.value)
    }
    getOrElse(): T {
      return this.value
    }
  }

  class None implements Option<never> {
    flatMap<U>(): None {
      return this
    }
    getOrElse<U>(value: U): U {
      return value
    }
  }


//사용자가 null, undefined 전달하면 None 반환
function Option<T>(value: null | undefined): None
// 그렇지 않으면 Some<T> 반환
function Option<T>(value: T): Some<T>

// 오버로드 된 두 시그니처를 직접 계산
// 1. null, undefined의 상위 경계는 T | null | undefined이고, 간소화하면 T
// 2. None과 Some<T>의 상위 경계는 None | Some<T>이므로 Option<T>
function Option<T>(value: T): Option<T> {
  if(value == null) {
    return new None
  }
```

```
  return new Some(value)
}
```

- 성공하거나 실패할 수 있는 연산을 연달아 수행할 때, Option을 유용하게 사용할 수 있다.
    - Option 사용시 타입 안정성 제공
    - 연산이 실패할 수 있음을 사용자에게 알린다.

  → 하지만 단점도 존재한다.

    i. `None` 으로 실패를 표현하기에 무엇이 실패했는지 자세히 알리지 않는다
    ii. 다른 코드와 호환되지 않는다.