타입스크립트 프로그래밍
스터디

PL 3-3 최지혜

1장. 소개

- 타입 스크립트는 *더 안전한 프로그램*을 구현하는 것을 보장
- 안전한 = 타입 안정성
- 타입 안정성 : 타입을 이용해 프로그램이 유효하지 않은 작업을 수행하는 것을 방지
  - 유효하지 않은 동작의 예
    - 숫자와 리스트 곱하기
    - 객체 리스트를 인수로 받는 함수에 문자열 리스트를 인수로 전달해 호출하기
    - 객체에 존재하지 않는 멤버 함수를 호출하기
    - 최근에 다른 곳으로 이동된 모듈 임포트하기

2장. 개요

타입스크립트 컴파일러(TSC)

- 보통의 컴파일 과정
  : 텍스트 파일(소스코드) → 추상문법트리(AST) → 바이트코드 → 실행

- 타입스크립트 컴파일러의 특별한 점
  - 바이트 코드가 아닌 자바스크립트 코드로 변환한다는 점
  - AST 를 만들어 결과 코드를 내놓기 전에 타입 검사기를 거침
    - AST : Abstract Syntax Tree
    - 타입 검사기(Type checker)

      : 코드의 타입 안전성을 검증하는 특별한 프로그램

  - TS 소스 → TS AST → 타입검사기 → JS 소스 → JS AST → JS 바이트코드 → 실행(브라우저, Node.js 등)
    - 타입검사 이후에는 개발자가 사용한 타입을 사용하지 않는다

2장. 개요

타입 시스템

- 타입스크립트의 타입 검사기가 프로그램에 타입을 할당하는 데 사용하는 규칙의 집합
  - 명시적 타입
  - 자동으로 추론되는 타입
- 타입 결정방식
  - gradually typed(컴파일 타임에 모든 타입을 알고 있으면 최상의 결과지만 반드시 모든 타입을 알고 있지 않아도 됨)
  - 그러나 모든 코드 타입을 컴파일 타임에 지정하는 것을 목표로 해야 함
- 타입 변환 여부
  - 가능하나 필요한 경우 한해 명시적으로 사용해야 함

```
3 + [1]; // TS2365: 3 과 number[] 타입에 + 연산자를 적용할 수 없음

(3).toString() + [1].toString() // 31 로 평가
```

- 타입 확인 시점
  - 컴파일 타임
  - 코드를 실행하기 전에 실수를 바로잡을 수 있음
- 에러 검출 시점
  - 컴파일 타임
  - 런타임 예외를 제외한 많은 에러를 검출할 수 있음

3장. 타입의 모든 것

- 인덱스 시그니처
  - 타입스크립트에 어떤 **객체가 여러 개의 키를 가질 수 있음**을 알려줌
  - 키 타입은 number 나 string 에 할당할 수 있는 타입이어야 함
  - 키 이름은 key 가 아니어도 됨
  - *타입 리터럴 내부에서 사용됨*

```
let a: {
    b: number;
    c? string;
    [key: number]: boolean;
};
a = { b: 1, 10: true, 20: false };
```

- cf. 자바스크립트의 계산된 프로퍼티
  - 프로퍼티 명의 제약을 완화해줌
  - 동적으로 프로퍼티명을 사용할 수 있음
  - *객체 리터럴 내부에서 사용됨*

```
let fruit = window.prompt();
const name = 'jihye';
const a = {
    [fruit]: 5,
    ['greeting ' + name]: 'Hi'
}

console.log(a.apple);
console.log(a['greeting jihye']);
```

4장. 함수

제네릭

- 구체타입
  - 기대하는 타입을 정확하게 알고 있고 실제 이 타입이 전달되었는지 확인할 때 유용
- 제네릭 타입 매개변수
  - 때로는 어떤 타입을 사용할지 미리 알 수 없는 상황이 있음
  - 함수, 클래스, 인터페이스, 타입별칭에 선언 가능
  - 전체 시그니처를 이용해 오버로드된 함수임을 표현하는 방법?
    - 객체 타입은 **object** 가 대표할 수 없음
    - object 타입은 객체임을 의미할 뿐 구체 타입 정보가 없기 때문
  - 타입의 **플레이스 홀더** 역할
  - 꺽쇠 괄호(<>) 이용
    - 꺽쇠 안에 여러 개의 제네릭 선언도 가능
  - 함수의 전체 시그니처와 단축 호출 시그니처에 따라 선언 방식이 조금 다름
    - 호출 시그니처 : 함수의 매개변수, 반환 타입을 모두 표현하는 방법

```
// 함수에 적용한 제네릭 예
let fiter: Filter = (array, f) => // …

// 객체타입의 경우에는 잘못된 예 (전체 시그니처로 함수 오버로드)
type Filter = {
  (array: number[], f: (item: number) => boolean): number[];
  (array: string[], f: (item: string) => boolean): string[];
  (array: object[], f: (item: object) => boolean): object[]; // (XXX)
};

// 옳은 예1 (전체 시그니처)
type Filter = {
    <T>(array: T[], f: (item: T => boolean) => T[]);
};

// 옳은 예2 (단축 시그니처)
type Filter<T> = (array: T[], f: (item: T => boolean) => T[]);
// 클래스 제네릭 예
class MyMap<K,V> {
   constructor(initialKey: K, initialValue: V) { ... }
   get(key: K): V { ... }
   set(key: K, value: V) { ... }
   merge<K1,V1>(map:MyMap<K1,V1>): MyMap<K|K1, V|V1> { ... }
   static of<K,V>(k:K, v:V): MyMap<K,V> { ... }
}

// 인터페이스 제네릭 예
interface MyMap<K,V> {
   get(key: K): V;
   set(key: K, value: V): void;
}

// 타입별칭 제네릭 예
type NumberFilter = Filter<number>;
type StringFilter = Filter<string>;
```

4장. 함수

- 제네릭이 구체화 되는 시점
  - 함수의 시그니처일 경우 : 함수를 호출할 때
  - 클래스의 경우 : 클래스를 인스턴스화 할 때
  - 타입별칭, 인터페이스의 경우 : 이들을 사용할 때
- 한정된 제네릭 타입
  - extends :고유 타입을 유지하며 공통의 프로퍼티 접근 가능
    - 이진트리 예
      a. 말단노드 : 말단 노드여부(isLeaf)
      b. 말단이 아닌 노드: 자식노드(children)
      c. 공통 : 값(value)
  - 인수의 개수 정의
    - 가변인수 함수에서 사용 가능
- 제네릭 타입 기본값
    - 특정 타입을 알 수 없는 때를 대비

```
type TreeNode = { value: string };
type LeafNode = TreeNode & { isLeaf: true };
type InnerNode
        = TreeNode & { children: [TreeNode] | [TreeNode, TreeNode]
}         // 한 개나 두 개의 자식을 가리킬 수 있음(튜플)

function mapNode<T extends TreeNode>(
  node: T, (**)
  f: (value: string) => string
): T {
  return {
    ...node,
    value: f(node.value) //(*)
  };
}

let a: TreeNode = {value: 'a'};
let b: LeafNode = {value: 'b', isLeaf: true};
let c: InnerNode = {value: 'c', children: [b]};

let a1 = mapNode(a, n => n.toUpperCase());
let b1 = mapNode(b, n => n.toUpperCase());
let c1 = mapNode(c, n => n.toUpperCase());

console.log(a1);
console.log(b1);
console.log(c1);
```

6장. 고급 타입

- 타입 간의 관계
  - 슈퍼타입 ex. number | string
  - 서브타입 ex. number
  - 슈퍼타입을 사용하는 곳에 서브타입을 사용할 수 있다
- 단순타입 아닌 경우 슈퍼/서브타입 및 할당성 판단?
  - 가변성 종류
    - 불변 : 정확히 T 를 원함
    - **공변** : 할당하려는 객체의 각 프로퍼티가 할당받는 객체의 각 프로퍼티의 서브타입임을 원함(함수 매개변수를 제외한 모든 복합 타입의 멤버의 할당성 규칙)
    - **반변** : 할당하려는 객체의 각 프로퍼티가 할당받는 객체의 각 프로퍼티의 슈퍼타입임을 원함(함수 매개변수의 할당성 규칙)
    - 양변 : 공변 또는 반변을 원함

6장. 고급 타입

객체타입은 그 멤버와 **공변**인데
타입스크립트는 어떻게 에러를 검출했을까?

- 슈퍼타입에 서브타입을 전달하면
- **기대되는 타입**(`{baseURL: string, cacheSize: number | undefiend, tier: 'prod' | 'dev' | undefined, tierr: undefined}`) 각각의 프로퍼티에 대해
- **전달하는 타입**(`{baseURL: string, cacheSize: undefined, tier: undefined tierr: 'prod'}`) **의 대응 프로퍼티가 공변**하기 때문에
  - `baseURL : string` 은 `string` 과 같고
  - `cacheSize : undefined` 은 `undefined` 와 같고
  - `tier : undefined` 은 `undefined` 와 같음
- **원래는 에러를 검출하지 못해야 함**

```
type Options = {
    baseURL: string;
    cacheSize?: number;
    tier?: 'prod' | 'dev';
}
class API {
    constructor(private options: Options){ }
}
const api1 = new API({
    baseURL: 'https://api.mysite.com',
    tier: 'prod'
});

const api2 = new API({
    baseURL: 'https://api.mysite.com',
    tierr: 'prod'; // TS2345  에러 (*)
});

const api3 = new API({
        baseURL: 'https://api.mysite.com',
        tierr: 'prod'
} as Options); //  서브타입이므로  타입 어서션 가능

const badOptions = {
        baseURL: 'https://api.mysite.com',
        tierr: 'prod'
};
const api4 = new API(badOptions); //  (**)왜 에러가 검출되지 않을까?
```

6장. 고급 타입

그럼에도 불구하고 에러를 검출할 수 있는 이유는 **초과 프로퍼티 확인**을 했기 때문(*)

- 신선한 객체 리터럴 타입 T를 다른 타입 U에 할당하려는 상황에서
- *T가 U가 가지고 있지 않은 프로퍼티를 가지고 있다면 에러로 처리함*
- 신선한 객체 리터럴 타입 : 타입스크립트가 객체 리터럴로부터 추론한 타입
  - 객체 리터럴이 타입 assertion을 사용하거나 변수로 할당되면 일반 객체 타입으로 넓혀지면서 신선함이 사라짐 (**)
  - 신선함이 사라지면 타입스크립트는 초과 타입 확인을 하지 않음

```
type Options = {
    baseURL: string;
    cacheSize?: number;
    tier?: 'prod' | 'dev';
}
class API {
    constructor(private options: Options){ }
}
const api1 = new API({
    baseURL: 'https://api.mysite.com',
    tier: 'prod'
});

const api2 = new API({
    baseURL: 'https://api.mysite.com',
    tierr: 'prod'; // TS2345  에러 (*)
});

const api3 = new API({
        baseURL: 'https://api.mysite.com',
        tierr: 'prod'
} as Options); //  서브타입이므로  타입  어서션  가능

const badOptions = {
        baseURL: 'https://api.mysite.com',
        tierr: 'prod'
};
const api4 = new API(badOptions); // (**)왜 에러가 검출되지 않을까?
```

6장. 고급 타입

정제(refinement)

- 타입스크립트는 심벌 수행의 일종인 흐름 기반 타입 추론을 수행
- typeif, instanceof 뿐 아니라 if, switch, ? || 같은 제어흐름 문까지 고려하여 타입을 정제

주의 : 유니온 타입의 정제

- **A | B ⇔ A 타입이나 B 타입 둘 중 하나만 할당할 수 있다는 뜻이 아님!**
- **A | B 타입의 서브 타입도 할당 가능함**
- { a : { a: number, b: number }, b: number } 가
  A | B = { a : { a: number }, b: string | number } 의 서브타입이므로 u 에 이 타입을 할당할 수 있음

  => 상호 배타적이지 않으므로 정제가 정확히 동작하지 않음

  => 고유한 값을 가지는 별도의 프로퍼티를 이용하는 것을 권장

```
type A = {
  a: {a: number, b: string}
  b: string;
}

type B = {
  a: {a: number};
  b: number;
}

function handle(u: A | B) {
  if(typeof u.b === 'string')
      console.log('A type');
  else
      console.log('B type');
}

const u: A | B = {
   a: {a: 1, b: '1'},
   b: 1
};

handle(u); // B type (!!!)
```

6장. 고급 타입

고급 객체 타입

- 객체타입의 타입연산자
  - 키인 : 객체의 특정 프로퍼티의 타입 가져옴, **[ ]** 사용
  - keyof : 객체의 모든 키를 문자열 리터럴 타입 유니온으로 얻음

```
type APIResponse = {
   user: {
        userId: string;
        friendList: {
           count: number;
           friends: {
                firstName: string;
                lastName: string;
           }[];
        };
   };
};

type FriendList = APIResponse['user']['friendList'];
type Friend = FriendList['friends'][number];

function renderFriendList (friendList: FriendList) { ... }
```

```
type ResponseKeys = keyof APIResponse; // 'user'
type UserKeys = keyof APIResponse['user'];
                                  // 'userId' | 'fiendList';
type FriendListkeys = keyof APIResponse['user']['fiendList'];
                                  // 'count' | 'friends'
```

6장. 고급 타입

고급 객체 타입

- 종합성(철저검사, exhaustive checking)
  - 타입 검사기는 필요한 모든 상황을 제대로 처리했는지 검사함
  - 패턴매칭을 사용하는 언어(하스켈, 오캐멀 등)에서 차용
  - 종합성 오류 해결방법
    - 모든 경우 추가하기
    - 반환타입에 undefined 추가

```
type Weekday = 'MON' | 'TUE' | 'WED' | 'THU' | 'FRI';
type Day = Weekday | 'SAT' | 'SUN';

function getNextDay(w: Weekday): Day {
   switch(w) {
        case 'MON': return 'TUE'; // TS2366: 함수에 마무리 반환문이 없으며 반환타입은 undefined를 포함하지 않음
   }
}

// 해결 1
function getNextDay(w: Weekday): Day {
   switch(w) {
        case 'MON': return 'TUE';
        case 'TUE': return 'WED';
        ... 모든 경우 추가
   }
}

// 해결 2
function getNextDay(w: Weekday): Day | undefined {
   switch(w) {
        case 'MON': return 'TUE';
   }
}
```

6장. 고급 타입

고급 객체 타입

- 타입스크립트가 종합성 검사로 찾지 못하는 개발자의 실수를 찾아내도록 하기 위해 더 안전하게 타입을 선언할 수 있는 방법들
  - Record 타입
  - 매핑된 타입
  - 예시: **객체가 특정 키 집합을 정의하도록 강제하기**

```
type Weekday = 'MON' | 'TUE' | 'WED' | 'THU' | 'FRI';
type Day = Weekday | 'SAT' | 'SUN';

// switch 대신 객체 검색을 사용하면 상수시간 소비
let nextDay = {
   MON: 'TUE' // OK, 아래 함수에서 nextDay.TUE에 접근(*) 하려고 해야 컴파일 에러가 발생함. nextDay 을 선언할 때부터 컴파일 에러를 발생시키려면?
};

function getNextDay(w: Weekday): Day {
   return nextDay[w]; // (*)
};
```

```
// 옳은 예
type Weekday = 'MON' | 'TUE' | 'WED' | 'THU' | 'FRI';
type Day = Weekday | 'SAT' | 'SUN';

// switch 대신 객체 검색을 사용하면
let nextDay = {
   MON: 'TUE',
   TUE: 'WED',
   WED: 'THU',
   THU: 'FRI',
   FRI: 'SAT',
};

function getNextDay(w: Weekday): Day {
  return nextDay[w] as Day;
};
```

6장. 고급 타입

고급 객체 타입

- Record 타입
  - 무언가를 매핑하는 용도로 객체를 활용할 수 있음
  - 객체의 키와 값에 타입을 제공(=제한)
  - 키 타입은 string, number 의 서브타입도 가능

- cf. 인덱스 시그니처
  - 키 타입은 일반 string, number, symbol 만 가능

```
type Weekday = 'MON' | 'TUE' | 'WED' | 'THU' | 'FRI';
type Day = Weekday | 'SAT' | 'SUN';

function getNextDay(w: Weekday): Day {
   return nextDay[w]; // (*)
};

// Record 타입으로 한 주의 각 요일을 다음 요일로 매핑하기
let nextDay: Record<Weekday, Day> = {
   MON: 'TUE' // TS2739: {MON: "TUE"} 타입에는 Record<Weekday,Day> 타입 중 TUE, WED, THU, FRI 가 빠져있음
};
```

```
let a: {
   b: number;
   c? string;
   [key: number]: boolean;
};
a = { b: 1, 10: true, 20: false };
```

6장. 고급 타입

고급 객체 타입

- 매핑된 타입
  - 객체의 키와 값에 타입을 제공하는 또 다른 방법
  - 타입스크립트만의 고유한 언어 기능
  - 한 객체 당 최대 한 개의 매핑된 타입을 가질 수 있음
  - Record 타입을 구현하는 데 이용됨

```
type Record<K extends keyof any, T> = { [key in K] : T };
```

  - Record 타입보다 강력함
    - 키인 타입과 조합하면 키 이름별로 매핑할 수 있는 값 타입을 제한 가능

```
type Weekday = 'MON' | 'TUE' | 'WED' | 'THU' | 'FRI';
type Day = Weekday | 'SAT' | 'SUN';

function getNextDay(w: Weekday): Day {
   return nextDay[w]; // (*)
};

// 매핑된 타입
let nextDay: { [K in Weekday]: Day } = {
   MON: 'TUE' // TS2739: {MON: "TUE"} 타입에는 { MON:Weekday; TUE:Weekday; WED:Weekday; THU:Weekday; FRI:Weekday; } 타입이 정의한 프로퍼티 중 TUE, WED, THU, FRI 가 빠져있음
};
```

```
type Account = {
   id: number;
   isEmployee: boolean;
   notes: string[];
};

type OptionalAccount = {[K in keyof Account]?: Account[K];};

type NullableAccount = {[K in keyof Account]: Account[K] | null;};

type ReadonlyAccount = {readonly [K in keyof Account]: Account[K];};

type Account2 = {-readonly [K in keyof ReadonlyAccount]: Account[K]};

type Account3 = {[K in keyof OptionalAccount]-?: Account[K];};
```

6장. 고급 타입 추가자료

타입스크립트가 제공하는 내장 매핑 타입
(=유틸리티 타입)

- Record<Keys,Values>
- Partial<Object> : Object의 모든 필드를 선택형으로
- Required<Object> : Object의 모든 필드를 필수형으로
- Pick<Object,Keys> : 주어진 Keys 에 대응하는 Object의 서브타입
- Readonly<Object> : Object의 모든 필드를 읽기전용으로
- +) Omit<Object,Keys> 특정 속성만 제거한 타입

6장. 고급 타입

고급 객체 타입

- 컴패니언 객체 패턴
  - 스칼라에서 유래
  - 같은 이름을 공유하는 객체와 클래스를 쌍으로 연결
  - 타입과 객체가 의미상 관련되어 있고 이 객체가 타입을 활용하는 유틸리티 메서드를 제공하는 경우 유용
  - 타입과 값 정보를 한 개의 이름으로 그룹화 하고 다른 파일에서 호출자가 한 번에 임포트할 수 있음

```
type Currency = {
   unit: 'EUR' | 'GBP' | 'JPY' | 'USD';
   value: number;
};

let Currency = {
   DEFAULT: 'USD',
   from(): Currency {
        return { unit, value };
   }
};
```

```
import { Currency } from './Currency';
let amountDue: Currency = {
   unit: 'JPY',
   value: 83733.10
};

let otherAmountDue = Currency.from(330, 'EUR');
```

6장. 고급 타입

고급 함수 타입들

- 튜플 타입 추론 개선
  - 타입스크립트는 튜플 선언에 관대함
    - 튜플 길이, 어떤 위치에 어떤 타입인지는 무시

```
let a = [1, true]; // (number | boolean)[] 으로 추론
```

  - 타입어서션이나 as const 를 이용해서 타입좁히기, 읽기전용 한정자를 적용하지 않고 (배열이 아닌) 튜플 타입으로 추론되기 위한 방법?
  - 나머지 매개변수의 타입을 추론하는 기법 이용

```
let a = [1, true] as [number, boolean];
let b = [1, true] as const;

function tuple<T extends unknown[]>(...ts: T): T {
   return ts;
}

let c = tuple(1, true); // [number, boolean] 으로 추론
```

6장. 고급 타입

고급 함수 타입들

- 사용자 정의 타입 안전장치
  - 타입 정제의 한계

```
function parseInput(input: string | number) {
   let formattedInput: string;
   if(typeof input === 'string') formattedInput =
input.toUpperCase();
}
```

```
function isString(a: unknown): boolean {
   return typeof a === 'string';
}

console.log(isString( 'a')); // true
console.log(isString([ 7])); // false

function parseInput(input: string | number) {
   let formattedInput: string;
   if(isString(input)) formattedInput = input.toUpperCase();
// Property 'toUpperCase' does not exist on type 'number'
}
```

```
function isString(a: unknown): a is string {
   return typeof a === 'string';
}

console.log(isString( 'a')); // true
console.log(isString([ 7])); // false

function parseInput(input: string | number) {
   let formattedInput: string;
   if(isString(input)) formattedInput = input.toUpperCase();
}
```

6장. 고급 타입

조건부 타입

- 타입스크립트가 제공하는 독특한 기능
- U 와 V타입에 의존하는 T 타입을 선언
  - U가 V와 같거나 서브타입이면 T 를 A에 할당
  - V가 U와 같거나 서브타입이면 T 를 B에 할당
- 분배적 조건부
  - 타입 조건은 분배법칙을 따름

```
type IsString<T> = T extends string ? true : false;

type A = IsString<string>; // true 타입
type B = IsString<number>; // false 타입
```

6장. 고급 타입

내장 조건부 타입

- Exclude
- Extract
- NonNull
- ReturnType
- InstanceType

```ts
type A = number | string;
type B = string;
type C = Exclude<A,B>; // number

type A = number | string;
type B = string;
type C = Extract<A,B>; // string

type A = { a?: number | null };
type B = NonNullable<A['a']>; // number

type F = (a: number) => string;
type C = ReturnType<F>; // string

type A = {new(): B};
type B = {b: number}
type C = InstanceType<A>; // {b: number}
```

6장. 고급 타입

Assertion

- 타입 Assertion
  - 어떤 타입은 그 타입의 슈퍼타입 또는 서브타입이라고 assertion 할 수 있음
  - 서로 관련 없는 타입끼리는 assertion 할 수 없음
  - as 키워드 이용
  - cf. as const
    - 자동 추론된 타입에서 더 넓히지 않음을 알림
    - readonly 도 자동으로 적용됨
- Nonnull Assertion
  - 항상 null 이 아니라고 타입스크립트에 알려줌
  - **너무 많은 assertion은 리팩토링의 징후**일 수 있음
    - union 이용하여 리팩토링 가능

```
function formatInput(input: string) { ... }
function getUserInput(): string | number { ... }

let input = getUserInput();
formatInput(input as string);

let a = {x: 3}; // {x: number} 타입으로 추론
let b: {x: 3}; // {x: 3} 타입으로 추론
let c = {x: 3} as const; // {readonly x: 3}
타입으로 추론
```

7장. 에러처리

- 타입스크립트는 런타임에 발생할 수 있는 예외를 컴파일 타임에 잡을 수 있도록 최선을 다함
- 이런 목표 하에 강력한 정적, 기호적 분석을 수행하는 풍부한 타입 시스템을 도입함
- 그러나 어떤 언어를 사용하든 런타임 예외는 발생함
  - 네트워크 장애, 파일 시스템 장애, 사용자 입력 파싱 에러, 스택 오버플로, 메모리 부족 에러 등
- 타입 스크립트에서 에러를 표현하고 처리하는 4가지 패턴
  - null 반환
  - 예외 던지기
  - 예외 반환
  - Option 타입

7장 - 1. null 반환

- 타입 안전성을 유지하면서 에러를 처리하는 가장 간단한 방법
- 문제가 생긴 원인을 알 수 없다는 단점

```
function ask() {
      return prompt('When is your birthday?');
}
function parse(birthday: string): Date | null {
      let date = new Date(birthday);
      if(!isValid(date) return null;
      return date;
}
function isValid(date: Date): boolean {
      return Object.prototype.toString.call(date) ===
      '[object Date]' && !Number.isNaN(date.getTime());
}

let date = parse(ask());
if(date) console.info('Date is', date.toISOString());
else console.error('Error parsing date for some
reason');
```

7장 - 2. 예외 던지기

- 커스텀 에러를 사용하면 어떤 문제가 생겼는지 알려줄 수 있을 뿐 아니라 문제가 생긴 이유도 설명할 수 있음
- 그런데 메서드사용자가 사전에 어떤 에러 타입이 있는지 알아야 한다(try~catch 문에서 분기 가능)는 단점

```
class InvalidDateFormatError extends RangeError { }
class DateIsInTheFutureFormatError extends RangeError { }
function ask() {
    return prompt('When is your birthday?');
}
function parse(birthday: string): Date {
    let date = new Date(birthday);
    if(!isValid(date)
        throw new InvalidDateFormatError('Enter a date in the form YYYY/MM/DD');
    if(date.getTime() > Date.now())
        throw new DateIsInTheFutureFormatError('are you a timelord?');
    return date;
}
function isValid(date: Date): boolean {
    return Object.prototype.toString.call(date) === '[object Date]' &&
!Number.isNaN(date.getTime());
}

try {
    let date = parse(ask());
    console.info('Date is', date.toISOString());
} catch(e) {
    if(e instanceof InvalidDateFormatError) console.error(e.message);
    else if(e instanceof DateIsInTheFutureFormatError) console.info(e.message);
    else throw e;
}
```

7장 - 3. 예외 반환

- 타입스크립트는 throws 문을 지원하지 않음
- 그러나 유니온 타입을 이용해 비슷하게 흉내낼 수 있음

```
class InvalidDateFormatError extends RangeError { }
class DateIsInTheFutureFormatError extends RangeError { }
function ask() {
    return prompt('When is your birthday?');
}
function parse(birthday: string): Date | InvalidDateFormatError |
DateIsInTheFutureFormatError {
    let date = new Date(birthday);
    if(!isValid(date)
        return new InvalidDateFormatError('Enter a date in the form
YYYY/MM/DD');
    if(date.getTime() > Date.now())
        return new DateIsInTheFutureFormatError('are you a timelord?');
    return date;
}
function isValid(date: Date): boolean {
    return Object.prototype.toString.call(date) === '[object Date]' &&
!Number.isNaN(date.getTime());
}

let result = parse(ask());
if(result instance of InvalidDateFormatError)    console.error(result.message);
else if(result instance of DateIsInTheFutureFormatError)
console.info(result.message);
else console.info(Date is', date.toISOString());
```

7장 - 4. Option 타입

- 특수목적 데이터타입
  - Option, Try, Either
- 에러가 발생할 수 있는 계산에 여러 연산을 연쇄적으로 수행할 수 있음
- 호환성 측면 단점
- **어떤 특정 값을 반환하는 대신 값을 포함하거나 포함하지 않을 수도 있는 컨테이너를 반환한다는 개념**
- 값을 포함할 수 있는 어떤 자료구조로도 컨테이너를 구현 가능함

```
function parse(birthday: string): Date[] {
    let date = new Date(birthday);

    if(!isValid(date) return[];
    return [date];
}

function isValid(date: string): boolean {
    // …
}

let date = parse(ask());
date.map(_ => _.toISOString())
    .forEach(_ => console.log('Date is ' + _));
```

8장. 비동기 프로그래밍, 동시성과 병렬성

- CPU 를 많이 소비하는 작업을 별도의 스레드에서 수행
  - 콜백, 프로미스, 스트림 등 다양한 비동기 API를 사용
- 자바스크립트는 비동기 작업을 처리할 때 위력을 발휘
  - 스레드 하나로 비동기 작업을 처리
  - ⇒ 이벤트 루프
  - 멀티스레드 기반 프로그래밍에서 공통적으로 나타나는 문제점 해결
- 비동기 프로그래밍은 코드를 이해하기 어려움
- 타입스크립트는 비동기 프로그램을 더 잘 이해할 수 있는 도구를 제공
  - **타입을 이용하여 비동기 작업 추적 가능**

8장. 비동기 프로그래밍, 동시성과 병렬성

자바스크립트 VM 이 동시성을 흉내내는 방법

- 메인 자바스크립트 스레드는 서버요청, 타이머, 파일 접근 등의 네이티브 비동기 API를 호출함
- 네이티브 비동기 API 를 호출한 이후에 다시 메인 스레드로 제어가 반환되며 아무 일 없었던 것 처럼 코드를 계속 실행함
- 비동기 작업이 완료되면 플랫폼은 태스크를 이벤트 큐에 추가함. 각 스레드가 자신만의 큐를 가지고 있으며 이를 이용해 비동기 연산 결과를 메인 스레드로 전달함. 태스크에는 호출 자체와 관련한 메타 정보 일부와 메인 스레드와 연결된 콜백함수의 참조가 들어있음
- 메인스레드의 콜 스택이 비면 플랫폼은 이벤트 큐에 남아있는 태스크가 있는지 확인함. 대기 중인 태스크가 있으면 플랫폼은 그 태스크를 실행함. 이 때 함수 호출이 일어나며 제어는 메인 스레드 함수로 반환됨. 함수 호출이 끝나고 콜 스택이 다시 비면 플랫폼은 다시 기다리는 태스크가 있는지 이벤트 큐에서 확인함. 콜 스택과 이벤트 큐가 모두 비고 모든 비동기 네이티브 API 호출이 완료될 때까지 이 과정을 반복함
- 참고 : https://www.youtube.com/watch?v=8aGhZQkoFbQ

8장 비동기 프로그래밍, 동시성과 병렬성 추가자료

(비동기함수의) **return, await, return await** 비교

- async 함수는 반환 값을 프로미스로 감싸서 반환함(또는 직접 프로미스 반환도 가능)
- return : 프라미스 리젝트 처리 못함. 호출한 곳으로 작업중인 상태의 프라미스를 그대로 넘김 (올려줌)
- await : 프라미스 처리하고 끝
- return await : 프라미스 처리하고 결과를 호출한 곳으로 리턴함

10장. namespace, module

모듈이란?

- 애플리케이션의 크기가 커지면 언젠가는 파일을 여러 개로 분리해야 함
  - 이 때 **분리된 파일 각각**을 모듈이라고 부름
- 모듈은 대개
  - *클래스 하나*, 혹은
  - 특정 목적을 가진 *복수의 함수*로 구성된 라이브러리 하나

10장. namespace, module

자바스크립트 모듈의 역사

- 처음에는 모듈 시스템을 지원하지 않아 모든 것을 전역 네임스페이스에 정의함
  - 사용할 수 있는 변수명이 금세 고갈되어 충돌 발생
- 이런 문제를 해결하고자 하는 노력들이 있었지만 여러가지 한계점 있음
  - AMD, CommonJS, UMD 등
- ES2015 에서 표준 모듈 시스템 등재됨
  - 특수한 지시자 export, import
- TSC 의 빌드시스템 덕분에 다양한 환경에 맞게 모듈을 컴파일 할 수 있음
  - **값 뿐 아니라 타입과 인터페이스도 익스포트 가능**함

10장. namespace, module

모듈의 핵심 기능(일반 스크립트와 차이점)

- 엄격모드로 실행됨
- 모듈 레벨 스코프
  - 모듈 내부에서 정의한 변수나 함수는 다른 스크립트에서 접근할 수 없음
- 단 한 번만 평가됨
  - 동일한 모듈이 여러 곳에서 사용되더라도 모듈은 최초 호출 시 단 한 번만 실행됨
  - 실행 후 결과는 이 모듈을 가져가려는 모든 모듈에 내보내짐
  - 모듈 *설정(configuration)*을 쉽게 할 수 있음. 최초로 실행되는 모듈의 객체 프로퍼티를 원하는 대로 설정하면 다른 모듈에서 이 설정을 그대로 사용할 수 있기 때문
- import.meta
  - 현재 모듈에 대한 정보를 제공해줌
- 모듈 최상위 레벨의 this는 undefined
  - 일반 스크립트의 `this`는 전역 객체

10장. namespace, module

- 프로그램 구현의 **캡슐화**
  - 함수의 동작을 캡슐화
  - 객체와 리스트 같은 자료구조로 데이터를 캡슐화
  - 함수와 데이터를 클래스로 묶거나 네임스페이스로 구분된 유틸리티 형태로 별도의 데이터베이스나 저장소에 보관
  - 클래스나 유틸리티를 패키지로 묶어서 NPM 으로 발행
- **모듈**을 이해하려면..
  - TSC 가 모듈을 해석하는 방법(10장)
  - 빌드 시스템이 모듈을 해석하는 방법(12장)
  - 모듈이 실제로 런타임에 응용프로그램으로 로드되는 방법 (런타임 로더, 12장)

10장. namespace, module

10.2. import, export

- 예시

```
// a.ts
export function foo() {}
export function bar() {}

// b.ts
import { foo, bar } from './a';
foo();
export let result = bar();
```

```
// c.ts
export default function
meow(loudness: number) {}

// d.ts
import meow from './c';
meow(11);
```

```
// e.ts
import * as a from './a';
a.foo();
a.bar();

// f.ts
export * from './a';
export { result } from './b';
export meow from './c';
```

10장. namespace, module

10.2. import, export - 동적임포트

- 응용 프로그램이 커지면서 첫 렌더링 시간이 점점 길어지는 문제
- 네트워크 병목이 생기기 쉬운 프론트엔드 응용프로그램에서 많이 발생
- 코드 분할(splitting)로 해결 가능
  - 자바스크립트 파일을 여러 개 생성하여 나누어 저장하면 병렬 로딩 가능
- 게으른 로딩으로도 해결 가능
  - 필요할 때만 코드를 로딩
- **LABjs 의 게으른 로딩 → 동적임포트 공식화**
  - tsconfig.json의 compilerOptions 에서 {“module”: “esnext” } 설정
  - 타입 안전성을 유지하며 동적임포트하는 방법
    - import 에 문자열 리터럴을 직접 제공
    - import 에 표현식을 전달하고 모듈의 시그니처를 직접 명시

```
import { locale } from ‘./locales/locale-us’;


async function main() {
   let userLocale = await getUserLocale();
   let path = `./locales/local-${userLocale}`;
   let localeUS: typeof locale
              = await import(path);
}
```

10장. namespace, module

10.2. import, export - 모듈 모드 vs. 스크립트 모드

- 타입스크립트가 타입스크립트 파일을 파싱하는 모드
  - 모듈 모드
  - 스크립트 모드
  - 동작 기준
    - 파일이 import, export 포함하면 모듈 모드로, 그렇지 않으면 스크립트 모드로 동작
- **실무에서는 대부분 모듈 모드를 사용**

10.3. 네임스페이스

- 타입스크립트가 제공하는 또 다른 캡슐화 방법
- **실무에서는 대부분 모듈을 사용**

출처

타입스크립트 프로그래밍

모던 자바스크립트 튜토리얼(ko.javascript.info)