타입 단언 보다 타입 선언을 사용하기

READIAN SOLUTION

기본 개념

```ts
interface Person {name: string};

const alice: Person = {name: 'Alice'};
const bob = {name: 'Bob'} as Person;
const bob = <Person>{name: 'Bob'};
```

- **`: Type` 타입 선언**

  변수에 '타입 선언' 을 붙여서 그 값이 선언된 타입임을 명시

  할당된 값에 대한 **타입 검사**

  잉여 속성 체크 동작 O

- **`as Type` 타입 단언**

  ts 가 추론한 타입이 있더라도 Person 타입으로 간주한다.

  타입 체커에게 **오류 무시 명령**

  잉여 속성 체크 동작 X

- **(안정성 체크) 타입 선언 > 타입 단언**

READIAN SOLUTION

화살표 함수의 return type 명시법

- **arrow function 의 Return 타입 선언**

  arrow function 의 타입 선언은 추론된 타입이 모호할 때가 있다.

```
const people = ['alice', 'bob', 'jan'].map(name => ({name}));
// Person[] 타입을 원했지만 {name: string;}[]
```

  {name }에 타입 단언을 사용하면 문제가 해결되는 것으로 보이지만 런타임에 문제가 발생한다.

  단언문을 쓰지 않고 다음과 같이 arrow function 내부에서 타입과 함께 변수를 선언하는 것이 직관적

```
const people = ['alice', 'bob', 'jan'].map(name => {
    const person: Person = {name};
    return person
});
```

  변수 대신 화살표 함수의 **반환 타입**을 선언하여 간결하게 작성

```
const people = ["alice", "bob", "jan"].map((name): Person => ({ name }));
```

READIAN SOLUTION

화살표 함수의 return type 명시법

- **arrow function 의 Return 타입 선언**

  여기서 `(name)` 의 소괄호를 통해 Return 타입이 `Person` 이라고 명시한다.

  `(name: Person)` 로 작성시 `name` 의 type 이 `Person` 이라고 명시 / Return 타입이 없음

  최종적으로는 원하는 타입을 직접 명시하고 ts 가 할당문의 유효성을 검사하게 한다.

```
const people: Person[] = ["alice", "bob", "jan"].map((name): Person => ({ name }));
```

  체이닝 시작에서부터 명명된 타입을 가져야 한다. => 정확한 곳에 오류가 표시

타입 단언문을 사용하는 경우

- **추론한 타입보다 더 정확한 타입**

```
const divEl = document.querySelector('#myButton').addEventListener('click', e=> {
    e.currentTarget // 타입은 EventTarget
    const button = e.currentTarget as HTMLButtonElement;
})
```

DOM 엘리먼트에 대해서는 ts 보다 더 정확히 알고 있다.

```
const divEl = document.querySelector('#myButton').addEventListener('click', e=> {
    e.currentTarget // 타입은 EventTarget
    const button = e.currentTarget as HTMLButtonElement;
})
```

ts 는 DOM 에 접근 X => `document.querySelector('#myButton')` 이 버튼 element 인지 인식 X

우리는 ts 가 알지 못하는 정보를 가지고 있기에 여기서는 타입 단언문을 사용하는 것이 타당

READIAN SOLUTION

타입 단언문을 사용하는 경우

- **Go To Definition**

```
querySelector<E extends Element = Element>(selectors: string): E | null;
```

```
const divEl = document.querySelector<HTMLButtonElement>('#myButton').addEventListener('click', e=> {
    e.currentTarget // 타입은 EventTarget
    const button = e.currentTarget;
})
```

DOM 타입에 대해서는 item 55 에서 자세히 다룸

READIAN SOLUTION

타입 단언문을 사용하는 경우

- **null 이 아님을 단언하는 경우**

  또한 자주 쓰이는 `!` 문법을 사용하여 null 이 아님을 단언

```
const elNull = document.getElementById('foo');
const el = document.getElementById('foo')!;
```

  변수의 접두사로 쓰인 `!` 는 boolean 의 부정문

  그러나 접미사로 쓰이는 `!` 는 값이 `null` 이 아니라는 단언문으로 해석

  단언문은 컴파일 과정 중 제거되므로 ts 는 알지 못하지만 그 값이 null 이 아니라고 확신할 수 있을 때 사용

  확신이 없을 경우 null 체크 조건문 사용

단언문의 특징

- ## 임의의 타입간 변환 불가

A 가 B 의 부분 집합인 경우 타입 변환이 가능하다.

`HTMLElement` 는 `HTMLButtonElement | null` 의 서브타입이기에 타입 단언이 동작

`HTMLElement` 는 `HTMLTarget` 의 서브타입이기에 역시 동작

`Person` 은 {} 의 서브타입이므로 동작

`Person` 과 `HTMLElement` 는 서로 서브타입이 아니기에 변환이 불가능

```
interface Person {
  name: string;
}
const body = document.body;
const el = body as Person;
```

이 오류를 해결하려면 unknown 타입을 사용 (모든 타입은 unknown 의 서브타입)

```
const el = body as unknown as Person;
```

READIAN SOLUTION