Prefer Types
That Always Represent Valid States

-Item 28-

cheol-won

```
const stateOne = {
  a: 'A',
  b: 'B',
};

const stateTwo = {
  a: 'A',
  b: 'B',
  c: 'C',
};

const stateThree = {
  a: 'A',
  b: 'B',
  d: 'D',
};
```

타입을 설계해 보자

```ts
interface State {
  a: string;
  b: string;
  c?: string;
  d?: string;
}
```

state 타입에는 a와 b속성을 반드시 가지고
c와 d속성은 선택적인 속성이다.

```ts
interface StateOne {
  a: string;
  b: string;
}

interface StateTwo {
  a: string;
  b: string;
  c: string;
}

interface StateThree {
  a: string;
  b: string;
  d: string;
}

type State = StateOne | StateTwo | StateThree;
```

정의되지 않는 상태 타입을 막을 수 있다.

"유효한 상태와 무효한 상태를 둘 다 표현하는 타입은
혼란을 초래하기 쉽고 오류를 유발하게 된다."

– effective typescript Item 28

```
interface State {
  pageText: string;
  isLoading: boolean;
  error?: string;
}
```

```
function renderPage(state: State) {
  if (state.error) {
    return `Error! Unable to load ${currentPage}: ${state.error}`;
  } else if (state.isLoading) {
    return `Loading ${currentPage}...`;
  }
  return `<h1>${currentPage}</h1>\n${state.pageText}`;
}
```

```
async function changePage(state: State, newPage: string) {
  state.isLoading = true;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const text = await response.text();
    state.isLoading = false;
    state.pageText = text;
  } catch (e) {
    state.error = '' + e;
  }
}
```

```ts
interface RequestPending {
  state: 'pending';
}
interface RequestError {
  state: 'error';
  error: string;
}
interface RequestSuccess {
  state: 'ok';
  pageText: string;
}
type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: {[page: string]: RequestState};
}
```

```
function getUrlForPage(p: string) { return ''; }
function renderPage(state: State) {
  const {currentPage} = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case 'pending':
      return `Loading ${currentPage}...`;
    case 'error':
      return `Error! Unable to load ${currentPage}: ${requestState.error}`;
    case 'ok':
      return `<h1>${currentPage}</h1>\n${requestState.pageText}`;
  }
}

async function changePage(state: State, newPage: string) {
  state.requests[newPage] = {state: 'pending'};
  state.currentPage = newPage;
  try {
    const response = await fetch(getUrlForPage(newPage));
    if (!response.ok) {
      throw new Error(`Unable to load ${newPage}: ${response.statusText}`);
    }
    const pageText = await response.text();
    state.requests[newPage] = {state: 'ok', pageText};
  } catch (e) {
    state.requests[newPage] = {state: 'error', error: '' + e};
  }
}
```

```ts
interface State {
  pageText: string;
  isLoading: boolean;
  error?: string;
}
```

```ts
function renderPage(state: State) {
  if (state.error) {
    return `Error! Unable to load ${currentPage}: ${state.error}`;
  } else if (state.isLoading) {
    return `Loading ${currentPage}...`;
  }
  return `<h1>${currentPage}</h1>\n${state.pageText}`;
}
```

```ts
interface RequestPending {
  state: 'pending';
}
interface RequestError {
  state: 'error';
  error: string;
}
interface RequestSuccess {
  state: 'ok';
  pageText: string;
}
```

```ts
function renderPage(state: State) {
  const {currentPage} = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case 'pending':
      return `Loading ${currentPage}...`;
    case 'error':
      return `Error! Unable to load ${currentPage}: ${requestState.error}`;
    case 'ok':
      return `<h1>${currentPage}</h1>\n${requestState.pageText}`;
  }
}
```

```ts
interface Vector3 {
  x: number
  y: number
  z: number
}

const getComponent = (vector: Vector3, axis: 'x' | 'y' | 'z') => {
  return vector[axis]
}

let x = 'x' // 타입은 string
let vec = {x:10, y:20, z: 30};

getComponent(vec, x) // 'string' 형식의 인수는 '"x" | "y" | "z"' 형식의 매개 변수에 할당될 수 없습니다
```

실행은 되지만 편집기에서 오류 발생