분기를 작성하는 방법 n가지

> - 이해하기 쉬운 코드 일한
> - 분기를 좀 더 이해하기 쉽도록 작성하기 위해

플랫폼FE파트

peter.cho

순서

1. 문(statement)으로 분기 작성
2. 식(expression)으로 분기 작성
3. 결론

문(statement)으로 분기 작성

1. switch 문

2. if 문

1. switch 문

```
// 하나의 조건식을 다수 케이스로 분기하고 싶을 때
switch (<조건식>) {
  case <케이스1>: return <value1>
  case <케이스2>: return <value2>
  case <케이스N>: return <valueN>
}

// 판단을 연쇄적으로 하고 싶을 때
switch (true) {
  case <판단1>: return <value1>
  case <판단2>: return <value2>
  case <판단N>: return <valueN>
}
```

하나의 조건식을 다수 케이스로 분기하는 코드 예시

```
const foo = num => {
  switch (num) {
    case 0: return 'A'
    case 1: return 'BB'
    case 2: return 'CCC'
  }
}

foo(0) // A
foo(1) // BB
foo(2) // CCC
```

판단을 연쇄적으로 하고 싶을 때 코드 예시

```
switch (true) {
  case isObject(char):
    return '{}'
  case isArray(char):
    return '[]'
  case isBoolean(char):
    return 'boolean'
  case isNull(char):
    return 'null'
  default:
    return ''
}
```

2. if 문

- 옵셔널 처리
- 여러개 사용할 경우 `else if` 사용

```
// else if
const foo = num => {
  if (num === 0) {
    return 'A'
  } else if (num === 1) {
    return 'BB'
  } else if (num === 2) {
    return 'CCC'
  }
}

// if
const foo = num => {
  if (num === 0) {
    return 'A'
  }
  if (num === 1) {
    return 'BB'
  }
  if (num === 2) {
    return 'CCC'
  }
}
```

식(expression)으로 분기 작성

1. 리터럴

2. 삼항연산자

1. 리터럴

- 목적
  - 분기를 제거하기 위해 분기만큼 데이터를 확보 가능할 때
- 이점
  - 분기 제거 및 가독성 향상
  - switch 문 대체

```
{
  <케이스1>: <value1>
  <케이스2>: <value2>
  <케이스N>: <valueN>
}
```

switch로 작성한 코드를 리터럴로 대체한 코드 예시

```
const foo = num => {
  switch (num) {
    case 0: return 'A'
    case 1: return 'BB'
    case 2: return 'CCC'
  }
}

// 리터럴
const EXAMPLE = {
  0: 'A',
  1: 'BB',
  2: 'CCC'
}

const foo = num => EXAMPLE[num]

foo(0) // A
foo(1) // BB
foo(2) // CCC
```

2. 삼항연산자

- 참과 거짓의 평가를 강제할 때
- switch 문, if 문, 리터럴은 예외 케이스를 문법적으로 처리하도록 강제하지 않음

```
// switch
const foo = num => {
  switch (num) {
    case 0: return 'A'
    case 1: return 'BB'
    case 2: return 'CCC'
    // default: return 'Z' (강제하지 않음)
  }
}

// if
const foo = num => {
  if (num === 0) {
    return 'A'
  } else if (num === 1) {
    return 'BB'
  } else if (num === 2) {
    return 'CCC'
  }
  // return 'Z' (강제하지 않음)
}

// 리터럴
const arr = ['A', 'BB', 'CCC']
const foo = num => arr[num]
```

삼항연산자 코드 예시

```
const foo = num => {
  return num === 0
    ? 'A'
    : num === 1
      ? 'BB'
      : num === 2
        ? 'CCC'
        : 'Z'
}
```

- 장점
  - 예상하지 못한 케이스를 대비할 수 있음
  - 강제성을 띠기 때문에 실수를 줄일 수 있음
- 단점
  - 식들이 서로 다른 세로선에 위치함으로 가독성 저하됨.

결론

- 이해하기 쉬운 코드를 위한 **가독성**과
- 코드 안전성을 위한 **강제성**을 고려했을 때,
- 코드별로 적재적소에 사용하는 것이 가장 이상적.

```
- 참만 필요할 때: if 문
- 참과 거짓이 필요할 때, 분기가 2개 이하일 때: 삼항연산자
- 분기가 3개 이상일 때, 조건식이 3개 이상일 때: switch 문 또는 리터럴
```

끝