리액트 스터디 2주차

20 황수연

03. 단축 평가(short-circuit evaluation) 논리 계산법(2장 이어서)

```
const dog = {
  name: '멍멍이'
};

function getName(animal) {
  if (animal) {
    return animal.name;
  }
  return undefined;
}

const name = getName();
console.log(name);
```

이런 식으로 getName 함수 안에서 if문을 통해 파라미터로 받아온 객체 animal에 대해 값이 제대로 주어졌는지 확인을 하면 조회할 수 없어서 발생하는 에러가 발생하지 않는다.

&& 연산자로 코드 단축시키기

```
const dog = {
  name: '멍멍이'
};

function getName(animal) {
  return animal && animal.name;
}

const name = getName(dog);
console.log(name); // 멍멍이
```

이 코드는 위의 코드와 똑같이 작동하는 코드이다. 이게 작동하는 이유는 A && B 연산자를 사용할 때 A가 truthy한 값이라면 B가 결과값이 되기 때문이다. 반대로, A가 falsy한 값이라면 결과는 A가 된다.

|| 연산자로 코드 단축시키기

|| 연산자는 어떤 값이 falsy하다면 대체로 사용할 값을 지정해줄 때 유용하다.

```
function getName(animal) {
  const name = animal && animal.name;
  if (!name) {
    return '이름이 없는 동물입니다';
  }
  return name;
}
```

이 함수를 아래와 같이 || 연산자를 통해 간단하게 쓸 수 있다.

```
const namelessDog = {
  name: ''
};

function getName(animal) {
  const name = animal && animal.name;
  return name || '이름이 없는 동물입니다.';
}

const name = getName(namelessDog);
console.log(name); // 이름이 없는 동물입니다.
```

A || B는 A가 truthy할 경우 결과값이 A가 되고, A가 falsy한 경우 결과는 B가 된다.

04. 함수의 기본 파라미터

```
function calculateCircleArea(r = 1) {
  return Math.PI * r * r;
}

const area = calculateCircleArea();
console.log(area); // 3.141592653589793
```

만약 매개변수가 있는 `calculateCircleArea()` 함수에서 파라미터를 받아오지 않는다면 undefined*undefined 로 숫자가 아닌 값에 곱셈을 하므로 NaN(Not A Number)라는 결과가 나온다.

따라서 매개변수의 자리에서 `r=1` 과 같이 써준다면 r 값이 주어지지 않았을 때 기본 값을 1로 사용할 수 있다.

```
const calculateCircleArea = (r = 1) => Math.PI * r * r;
```

```
const area = calculateCircleArea();
console.log(area); // 3.141592653589793
```

화살표 함수에서도 사용가능.

05. 조건문 더 스마트하게 쓰기

특정 값이 여러 값 중 하나인지 확인해야 할 때

```
function isAnimal(name) {
  const animals = ['고양이', '개', '거북이', '너구리'];
  return animals.includes(name);
}

console.log(isAnimal('개')); // true
console.log(isAnimal('노트북')); // false
```

비교해야할 때 || 를 연속하여 쓰는 것보다 배열을 만들고 배열의 `includes` 함수를 사용.

혹은 배열 선언 생략하고 화살표 함수로 작성 가능.

```
const isAnimal = name => ['고양이', '개', '거북이', '너구리'].includes(name);

console.log(isAnimal('개')); // true
console.log(isAnimal('노트북')); // false
```

값에 따라 다른 결과물을 반환해야 할 때

if문을 중첩하거나 switch case문을 사용해서 구현 가능하다.

너무 지저분할 수도 있으므로 다음과 같이 쓴다.

```
function getSound(animal) {
  const sounds = {
    개: '멍멍!',
    고양이: '야옹~',
    참새: '짹짹',
    비둘기: '구구 구 구'
  };
  return sounds[animal] || '...?';
}

console.log(getSound('개')); // 멍멍!
console.log(getSound('비둘기')); // 구구 구 구
```

객체를 사용하여 더 간단하게 구현 가능.

만약 값에 따라 실행해야 하는 코드 구문이 다를 때에는 아래와 같이 객체에 함수를 넣는다.

```
function makeSound(animal) {
  const tasks = {
    개() {
      console.log('멍멍');
    },
    고양이() {
      console.log('고양이');
    },
    비둘기() {
      console.log('구구 구 구');
    }
  };
  if (!tasks[animal]) {
    console.log('...?');
    return;
  }
  tasks[animal]();
}

makeSound('개');
makeSound('비둘기');
```

06. 비구조화 할당 (구조 분해) 문법

비구조화 할당 시 기본값 설정

```
const object = { a: 1 };

function print({ a, b = 2 }) {
  console.log(a);
  console.log(b);
}

print(object);
```

원래 객체에는 b가 존재하지 않으므로 기본값이 없는 `print()` 함수를 실행시키면 `undefined` 가 출력된다.

```
const object = { a: 1 };

const { a, b = 2 } = object;

console.log(a); // 1
console.log(b); // 2
```

함수의 파라미터말고도 사용가능하다.

비구조화 할당 시 이름 바꾸기

```
const animal = {
  name: '멍멍이',
  type: '개'
};

const { name: nickname } = animal
console.log(nickname);
```

`const { name: nickname } = animal` 은 `const nickname = animal.name` 과 같은 뜻으로, 'animal 객체 안에 있는 name 을 nickname 이라고 선언하겠다' 라는 의미이다.

배열 비구조화 할당

```
const array = [1, 2];
const [one, two] = array;

console.log(one);
console.log(two);
```

배열 안에 있는 원소를 다른 이름으로 새로 선언해주고 싶을 때 사용한다.

`const [one, two = 2] = array;` 와 같이 기본값 지정 가능하다.

깊은 값 비구조화 할당

1. 비구조화 할당 문법을 두 번 쓰기

```
const deepObject = {
  state: {
    information: {
      name: 'velopert',
      languages: ['korean', 'english', 'chinese']
    }
  },
  value: 5
};

const { name, languages } = deepObject.state.information;  // 비구조화 할당
const { value } = deepObject;  // 비구조화 할당

const extracted = {
  name,
```

```
  languages,
  value
};

console.log(extracted); // {name: "velopert", languages: Array[3], value: 5}
```

위에서 extracted 객체를 선언한 코드는 다음 코드와 같다.

```
const extracted = {
  name: name,
  languages: languages,
  value: value
}
```

만약 key 이름으로 선언된 값이 존재한다면 바로 매칭시켜주는 문법이다. (ES6의 object-shorthand)

2. 한 번에 모두 추출

```
const deepObject = {
  state: {
    information: {
      name: 'velopert',
      languages: ['korean', 'english', 'chinese']
    }
  },
  value: 5
};

const {
  state: {
    information: { name, languages }
  },
  value
} = deepObject;

const extracted = {
  name,
  languages,
  value
};

console.log(extracted);
```

07. spread와 rest 문법

spread

spread 문법을 사용하면 객체 혹은 배열을 펼칠 수 있다. 기존의 것을 건드리지 않고 새로운 객체를 만들 때 유용하다. 연산자로 ... 를 사용한다.

```
const slime = {
  name: '슬라임'
};

const cuteSlime = {
  ...slime,
  attribute: 'cute'
};

const purpleCuteSlime = {
  ...cuteSlime,
  color: 'purple'
};

console.log(slime);
console.log(cuteSlime);
console.log(purpleCuteSlime);
```

`cuteSlime` 은 `slime`을, `purpleCuteSlime`은 `cuteSlime`을 그대로 가지고 온다.

spread는 배열에서도 사용 가능하다.

```
const numbers = [1, 2, 3, 4, 5];

const spreadNumbers = [...numbers, 1000, ...numbers];
console.log(spreadNumbers); // [1, 2, 3, 4, 5, 1000, 1, 2, 3, 4, 5]
```

기존의 numbers는 건드리지 않으면서 새로운 spreadNumbers 배열에 numbers 가 가지고 있는 내용을 모두 넣고 1000 이라는 항목을 추가하였다. 위처럼 여러번 사용 가능하다.

rest

rest는 객체, 배열, 그리고 함수의 파라미터에서 사용 가능

객체, 배열에서 사용할 때는 비구조화 할당 문법과 함께 사용된다

1. 객체에서의 rest

```
const purpleCuteSlime = {
  name: '슬라임',
  attribute: 'cute',
  color: 'purple'
};
```

```
const { color, ...rest } = purpleCuteSlime;
console.log(color);
console.log(rest);
```

rest 안에는 color 값을 제외한 나머지 값들이 들어있다. 이때 추출한 값의 이름이 꼭 rest일 필요는 없다.

2. 배열에서의 rest

사용법은 객체에서의 사용법과 같다. 그러나 아래와 같이 하는 것은 불가능하다.

```
const numbers = [0, 1, 2, 3, 4, 5, 6];

const [..rest, last] = numbers;
```

3. 함수의 파라미터에서의 rest

함수의 파라미터가 몇 개가 될 지 모르는 상황에서 유용하다.

```
function sum(...rest) {
  return rest.reduce((acc, current) => acc + current, 0);
}

const result = sum(1, 2, 3, 4, 5, 6);
console.log(result); // 21
```

이렇게 하면 파라미터로 몇 개가 넘어와도 함수를 제대로 작동시킬 수 있다.

함수 인자와 spread

*파라미터: 함수에서 값을 읽을 때의 그 값들 / 인자: 함수에서 값을 넣어줄 때의 값

함수 파라티머와 rest를 사용한 것과 반대의 역할이다.

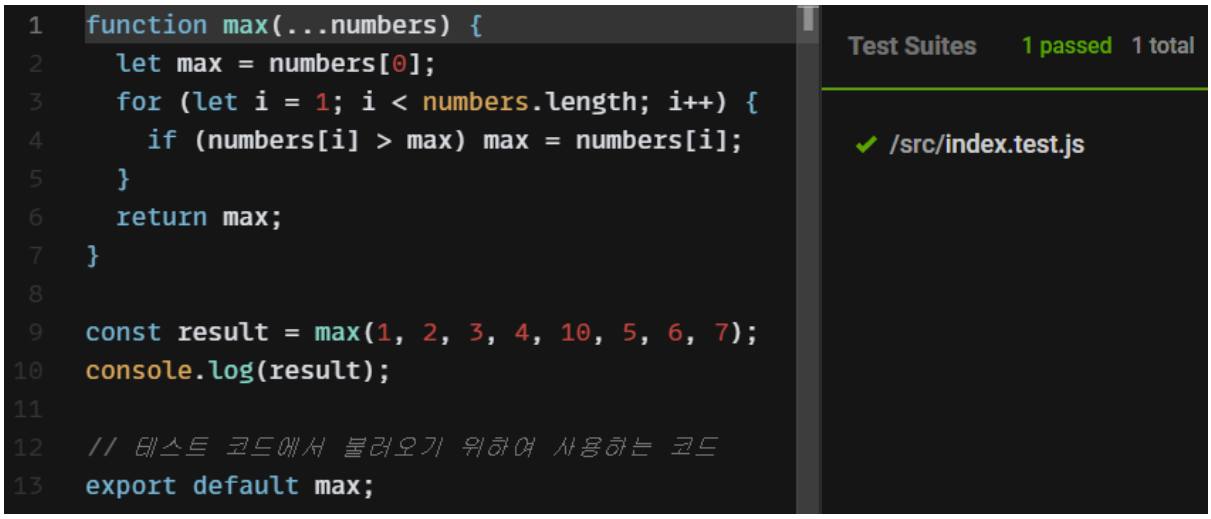
```
function sum(...rest) {
  return rest.reduce((acc, current) => acc + current, 0);
}
```

```
const numbers = [1, 2, 3, 4, 5, 6];
const result = sum(...numbers);
console.log(result);
```

spread를 사용하지 않으면 항목을 일일이 넣어주어야 한다.

퀴즈

: 함수에 n개의 숫자들이 파라미터로 주어졌을 때, 그 중 가장 큰 값을 알아내세요.



08. scope의 이해

*scope: 변수 혹은 함수를 선언할 때 해당 변수, 함수가 유효한 범위.

*종류: global scope(전역), function scope(함수), block scope(블록)

예시를 통한 scope의 이해

`const`와 `let` 으로 선언한 값은 block scope로 선언된다. if문 같은 블록 내에서 새로운 변수, 상수를 선언한다면 해당 블록 내부에서만 사용 가능. 블록 밖의 범위에서 똑같은 이름의 변수/상수가 있어도 영향을 주지 않음. (const, let)

그러나 `var` 를 사용하게 되면 function scope로 선언되므로 블록 내부에서 선언한 값이 블록 밖의 같은 이름의 변수에도 영향을 미친다.

Hoisting 이해하기

Hoisting: 자바스크립트에서 아직 선언되지 않은 함수/변수를 끌어올려서 사용할 수 있는 작동 방식

1. 함수의 hoisting

```
myFunction();

function myFunction() {
  console.log('hello world!');
}
```

이 코드는 myFunction 함수를 선언하기 전에, myFunction 함수를 호출했지만 그럼에도 불구하고 정상적으로 작동한다. 이유는 자바스크립트 엔진이 위 코드를 해석하는 과정에서 다음과 같이 받아들이기 때문이다. 이러한 현상이 hoisting이다.

```
function myFunction() {
  console.log('hello world!');
}

myFunction();
```

2. 변수 hoisting

- `var` 쓰는 경우

```
console.log(number);
var number = 2;
```

이 경우 `undefined` 출력됨. 엔진이 `number`라는 변수가 선언→ `console`에 찍고→값이 정해졌다고 해석하기 때문.

- `const`, `let` 쓰는 경우

hoisting 발생하지 않고 에러가 발생한다.

** Hoisting을 일부러 할 필요는 없지만 hoisting이 발생하는 코드는 이해가 어렵기 때문에 유지보수도 힘들고 의도치 않은 결과물이 나타날 수 있기 때문에 방지하는 것이 좋다. `var` 대신 `const`, `let` 을 위주로 사용하기.

3장. 자바스크립트에서 비동기 처리 다루기

- 동기적 처리(Synchronous)

작업이 끝날 때까지 기다리는 동안 중지 상태가 되기 때문에 다른 작업을 할 수 없다. 작업이 끝나야 그 다음 예정된 작업을 할 수 있다.

- 비동기적 처리(Asynchronous)

흐름이 멈추지 않기 때문에 동시에 여러가지 작업을 처리할 수도 있고, 기다리는 과정에서 다른 함수도 호출할 수 있다.

- `setTimeout` 함수를 통해 함수를 비동기 형태로 전환해 줄 수 있다. 첫번째 파라미터에 넣는 함수를 두번째 파라미터에 넣은 시간(ms)이 흐른 후에 호출한다. 두번째 파라미터에 0을 넣으면 함수가 바로 실행된다. 0ms 이후에 실행한다는 의미이지만 실제로는 4ms 이후에 실행된다.(딜레이때문)
- 만약 `setTimeout` 함수를 통해 불러온 함수가 끝난 다음에 어떤 작업을 처리하고 싶다면 콜백 함수를 파라미터로 전달해준다.

```
function work(callback) {
  setTimeout(() => {
    const start = Date.now();
    for (let i = 0; i < 1000000000; i++) {}
    const end = Date.now();
    console.log(end - start + 'ms');
    callback();
  }, 0);
}

console.log('작업 시작!');
work(() => {
  console.log('작업이 끝났어요!')
});
console.log('다음 작업');
```

- 주로 비동기적으로 처리하는 작업: Ajax Web API 요청, 파일 읽기, 암호화/복호화, 작업 예약

01. Promise

Promise: 비동기 작업을 조금 더 편하게 처리할 수 있도록 ES6에 도입된 기능. 콜백 함수로 처리하면 비동기 처리 작업이 많아질수록 코드가 쉽게 난잡해지기 때문에 사용.(Callback Hell - 콜백지옥)

Promise 만들기

```
const myPromise = new Promise((resolve, reject) => {
  // 구현..
})
```

promise는 성공할 수도, 실패할 수도 있음. 성공할 때에는 resolve를, 실패할 때는 reject를 호출해줌. `.catch` 를 통해 실패 시 수행할 작업 설정 가능. 작업이 끝난 후 다른 작업을 해야

할 때에는 promise 뒤에 `.them(...)`을 붙여서 사용하면 된다.

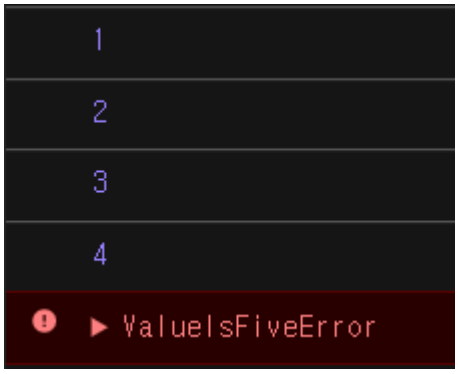
```
function increaseAndPrint(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const value = n + 1;
      if (value === 5) {
        const error = new Error();
        error.name = 'ValueIsFiveError';
        reject(error);
        return;
      }
      console.log(value);
      resolve(value);
    }, 1000);
  });
}

increaseAndPrint(0).then((n) => {
  console.log('result: ', n);
})
// 1
// result: n
```

만약 then 내부에 넣은 함수에서 또 promise 를 리턴하게 되면 연달아서 사용할 수 있다.

```
function increaseAndPrint(n) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const value = n + 1;
      if (value === 5) {
        const error = new Error();
        error.name = 'ValueIsFiveError';
        reject(error);
        return;
      }
      console.log(value);
      resolve(value);
    }, 1000);
  });
}

increaseAndPrint(0)
  .then(increaseAndPrint)
  .then(increaseAndPrint)
  .then(increaseAndPrint)
  .then(increaseAndPrint)
  .then(increaseAndPrint)
  .catch(e => {
    console.error(e);
  });
```



02. async/await

함수 선언 시 앞부분에 `async` 키워드를 붙인다. 그리고 promise의 앞부분에 `await` 을 넣어주면 해당 promise가 끝날 때까지 기다렸다가 다음 작업을 수행한다.

```
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function process() {
  console.log('안녕하세요!');
  await sleep(1000); // 1초쉬고
  console.log('반갑습니다!');
}

process().then(() => {
  console.log('작업이 끝났어요!');
});
```

`sleep` 이라는 함수에서 파라미터로 넣어준 시간만큼 기다리는 promise를 만들고, 이를 `process` 함수에서 사용했다. 함수에서 `async`를 사용하면 그 함수는 결과값으로 promise를 반환하게 되어 `.then` 을 사용할 수 있다.

- 에러를 발생시킬 때에는 `throw`를 사용하고, 잡아낼 때에는 try/catch 문을 사용한다.

```
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function makeError() {
  await sleep(1000);
  const error = new Error();
  throw error;
}

async function process() {
  try {
```

```
    await makeError();
  } catch (e) {
    console.error(e);
  }
}

process();  // Error
```

- `Promise.all` : `await`을 여러번 쓰면 순서대로 실행된다. 동시에 작업을 시작하고 싶다면 `Promise.all`을 사용한다. 이때 등록한 프로미스 중 하나라도 실패하면 모든게 실패한 것으로 간주된다. 배열 비구조화 할당을 사용하면 각 결과값을 추출해 조회할 수 있다.

```
function sleep(ms) {
  return new Promise((resolve) => setTimeout(resolve, ms));
}
const getDog = async () => {
  await sleep(1000);
  return "멍멍이";
};
const getRabbit = async () => {
  await sleep(500);
  return "토끼";
};
const getTurtle = async () => {
  await sleep(3000);
  return "거북이";
};
async function process() {
  const [dog, rabbit, turtle] = await Promise.all([
    getDog(),
    getRabbit(),
    getTurtle()
  ]);
  console.log(dog);
  console.log(rabbit);
  console.log(turtle);
}

process();
```

- `Promise.race`: 여러 프로미스를 등록하여 실행이 가장 빨리 끝난 하나의 결과값을 가져온다. 등록된 다른 promise가 먼저 성공하기 전에 가장 먼저 끝난 promise가 실패하면 이를 실패로 간주한다.

```
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

const getDog = async () => {
  await sleep(1000);
```

```
  return '멍멍이';
};

const getRabbit = async () => {
  await sleep(500);
  return '토끼';
};
const getTurtle = async () => {
  await sleep(3000);
  return '거북이';
};

async function process() {
  const first = await Promise.race([
    getDog(),
    getRabbit(),
    getTurtle()
  ]);
  console.log(first);
}

process();  // '토끼'
```

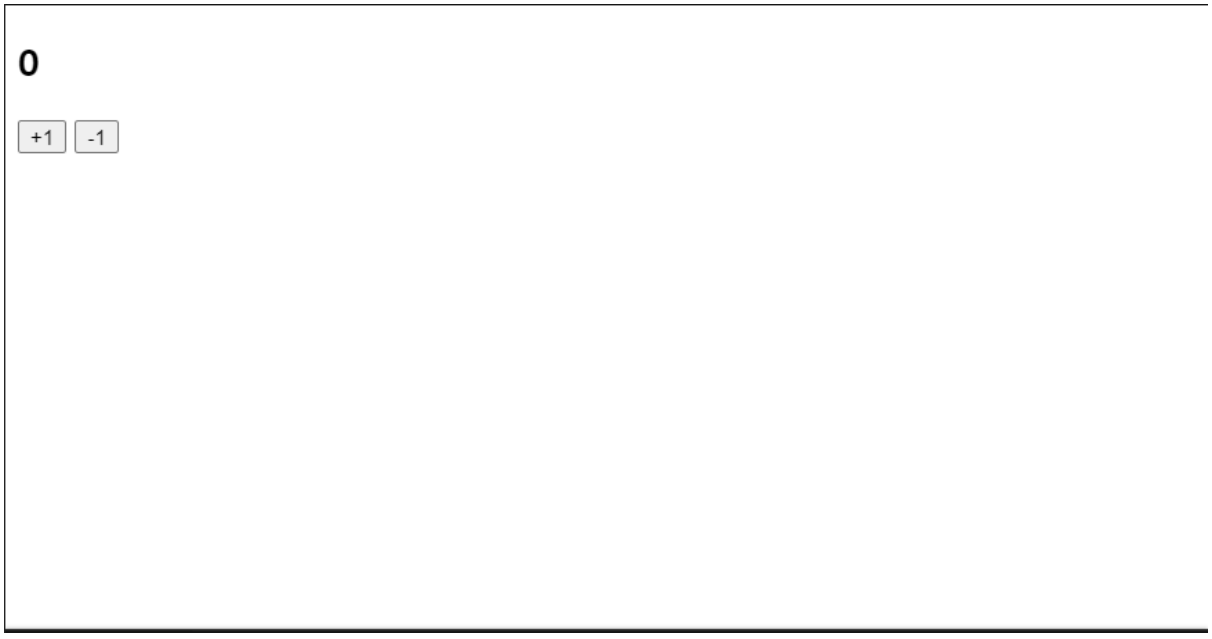
4장. HTML 과 Javascript 연동하기

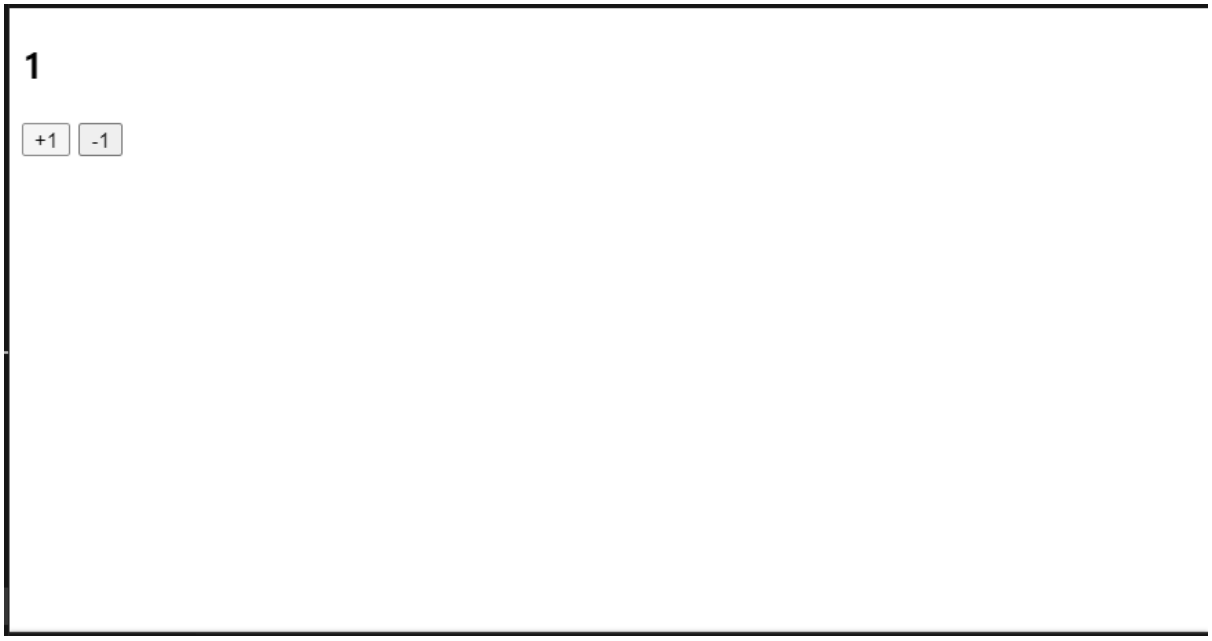
HTML을 사용하면 우리가 보여주고 싶은 UI를 브라우저에서 보여줄 수 있다. 사용자와의 상호작용에 따라 동적으로 UI를 업데이트하고 싶다면 Javascript를 연동해줘야 한다.

01. 카운터

버튼 클릭시 숫자가 올라가거나 내려가는 카운터 만들어보기.

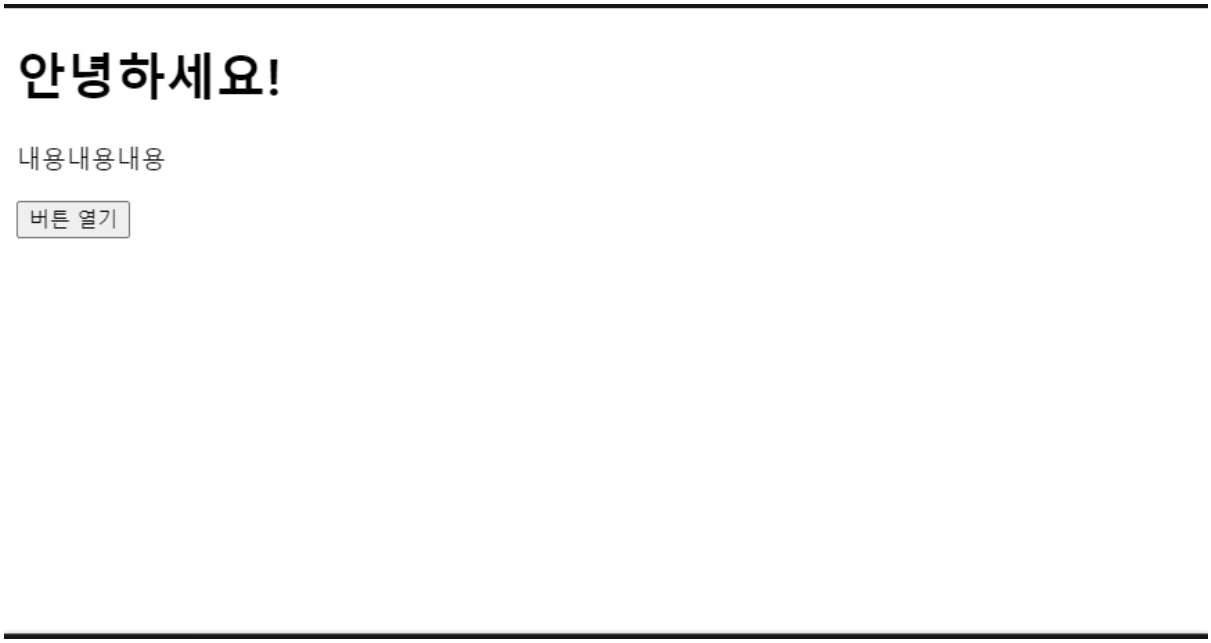
UI 만들기(HTML) → DOM 선택하기 → 이벤트 설정하기



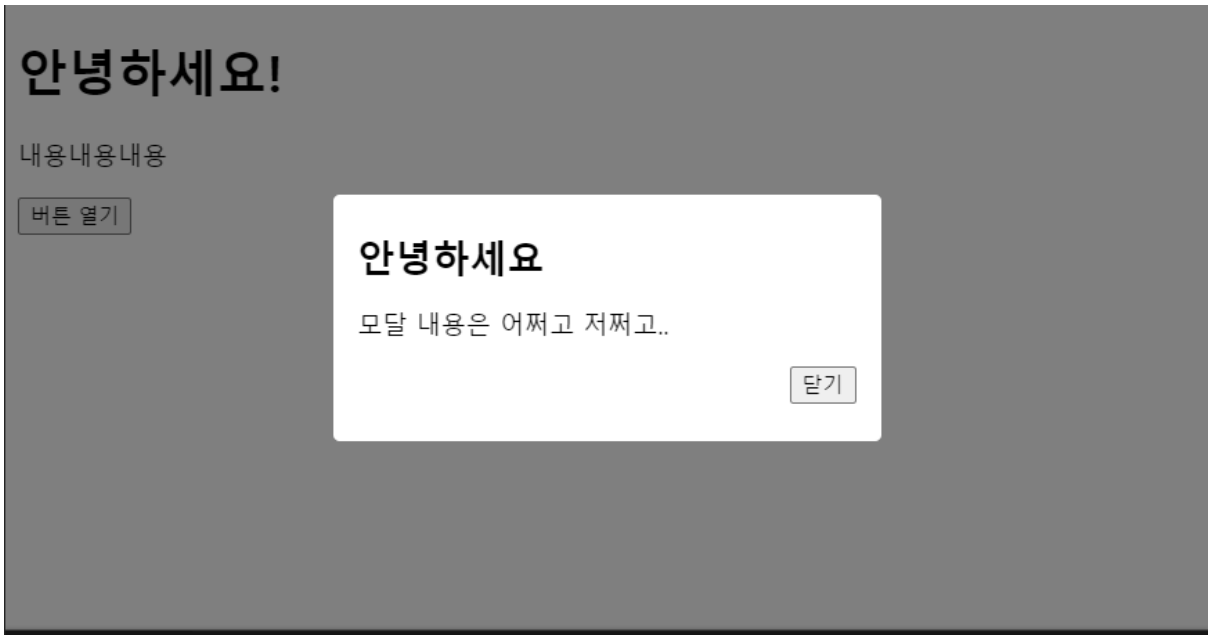


02. 모달

모달이란, 기존의 내용을 가리고 나타나는 메시지박스같은 형태의 UI이다.



초기 화면



'버튼 열기'라는 버튼을 누르면 모달이 뜬다. 닫기를 누르면 작게 열린 창이 사라진다.