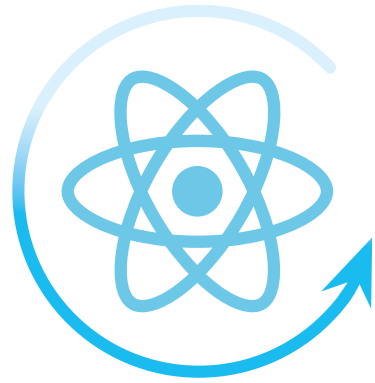
리액트, 리덕스, 주스탠드, 파이어베이스와 40여 가지 예제로 배우는
React 프론트엔드 입문자를 위한 풀 패키지

- 프론트엔드 개발자가 알아두면 좋은 지식
- 리액트를 잘하려면 알아야 하는 실습 예제 40가지
- 리덕스 툴킷, 주스탠드, 파이어베이스 주요 주변 기술



입문자를 위한 기초와 최신 테크 트리, 인기 에코시스템까지 활용한 예제로 탄탄히 배우기

리액트 잘하는 개발자 되기

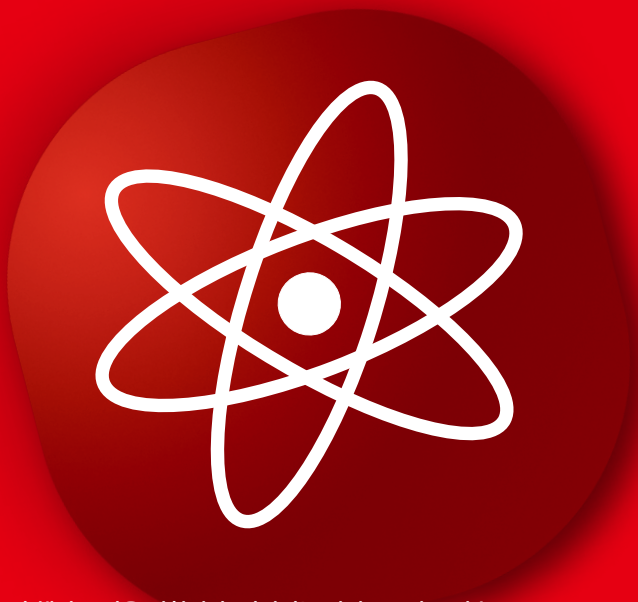
보너스 PDF 제공
리액트를 위한 자바 스크립트 핵심 문법 익히기

성낙현 지음

GOLDEN RABBIT

보너스 PDF

자바스크립트 핵심 문법



학습 목표

부록에서는 리액트를 학습하기 위해 필요한 자바스크립트의 핵심 문법을 익힙니다. 여기서는 자바스크립트 기초 문법을 한 번 공부했다고 가정하고 자바스크립트 ES5에 도입된 고차 함수의 문법과 활용 방법을 학습하고, 이후 ES6부터 시작된 모던 자바스크립트의 주요 문법과 기능을 설명하여 리액트 개발에 필요한 자바스크립트 핵심 문법을 제대로 공부하겠습니다.

문법 01 forEach( ) 함수

forEach( )는 콜백 함수의 세 번째 매개변수(반복 배열)로 전달한 배열의 각 요소에 대하여 콜백 함수를 반복해서 호출합니다. 다음 기본형을 살펴봅시다.

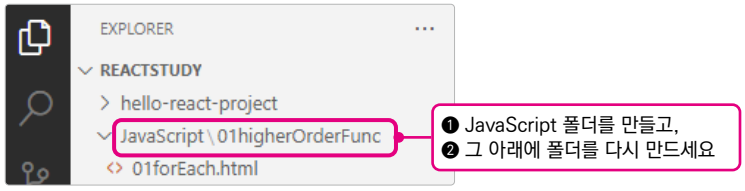
```
원본 배열 = [1, 2, 3]
원본 배열.forEach ( 콜백 함수 ( 현재값, 인덱스, 반복 배열 ) {
    배열의 각 요소에 대해 반복 호출
} );
```

기본형을 보면 원본 배열 크기가 3이므로 콜백 함수를 3번 반복 호출합니다. 반복 호출 시 반복 배열의 요소를 순차적으로 현재값을 전달하며 호출합니다. 따라서 현재값은 1→2→3으로 바뀌고, 인덱스는 0→1→2로 바뀝니다. forEach( )는 이렇게 반복 호출을 하며 반복 배열의 요소를 훑는 효과가 있습니다. 따라서 반복 배열의 요소를 살펴보거나, 반복 배열 요소를 활용할 때 유용합니다. 다만 반환하는 값이 없으므로 체이닝은 사용할 수 없습니다.


체이닝이란 함수를 .으로 연이어 호출하는 방식을 말합니다.


콜백 함수란 다른 함수에 인자로 전달되어, 특정 시점에 실행되는 함수입니다.

01 설명만 보면 조금 이해가 어려울 수 있으므로 코드를 작성하여 확인하겠습니다. ❶ 비주얼 스튜디오 코드에서 REACTSTUDY 폴더 하위에 자바스크립트 공부를 위한 JavaScript 폴더를 생성하고, ❷ 01higherOrderFunc 폴더를 또 생성하여 그 위치에 HTML 파일을 만들고 실습하겠습니다.

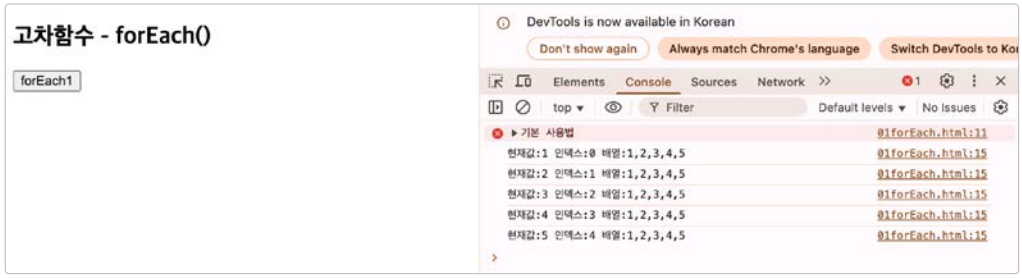


02 다음과 같이 코드를 작성하고 웹브라우저에서 열어 결과를 확인해보세요. 참고로 코드를 모두 작성한 다음에는 비주얼 스튜디오 코드 확장 프로그램인 Preview on Web Server를 이용해서

파일을 바로 실행할 수 있습니다. 코드 편집기 화면 오른쪽 클릭 후 [Launch on browser]를 클릭하면 파일을 크롬 웹브라우저로 바로 실행합니다. **가장 먼저 기본형으로 공부한 forEach( )를 공부하겠습니다.**

01higherOrderFunc/01forEach.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>forEach()</title>
  </head>
  <script>
    function forEach01() {
      console.error("기본 사용법");
      var myNumbers = [1, 2, 3, 4, 5]; // ❶
      myNumbers.forEach(function (currentValue, index, useArray) {
        // ❷
        console.log(
          "현재값:" + currentValue,
          "인덱스:" + index,
          "배열:" + useArray
        );
      });
    }
  </script>

  <body>
    <h2>고차 함수 - forEach()</h2>
    <div>
      <!-- ❸ 함수 실행 버튼 -->
      <input type="button" onclick="forEach01()" value="forEach1" />
    </div>
  </body>
</html>
```



❶ 5개의 요소를 가진 원본 배열을 선언합니다.

❷ myNumbers.forEach와 같이 체이닝으로 forEach( )를 호출했습니다. forEach( )에는 인수로 콜백 함수 function(currentValue, index, useArray){...}를 전달합니다. myNumbers에는 5개의 요소가 있으므로 콜백 함수는 5번 호출됩니다. 콜백 함수의 3개의 매개변수 currentValue에는 반복 호출 시 사용한 요소의 값, index에는 반복 호출 시 0부터 증가하는 인덱스, useArray에는 myNumbers가 전달됩니다.

```
myNumbers.forEach(function (currentValue, index, useArray) {
  console.log(
    "현재값:" + currentValue,
    "인덱스:" + index,
    "배열:" + useArray
  );
})
```

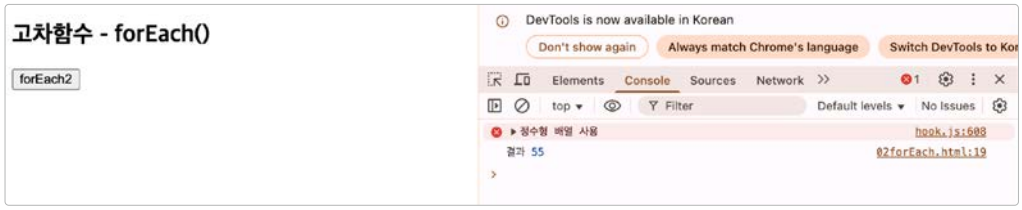
❸ [forEach1] 버튼을 누르면 forEach1( ) 함수를 실행한 결과를 콘솔에서 볼 수 있습니다.

03 이번에는 배열에 체이닝으로 forEach( )를 활용한 예를 공부하겠습니다. 배열에 체이닝으로 forEach( )를 사용하는 방식도 많이 사용하므로 알아두면 좋습니다.

01higherOrderFunc/02forEach.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>forEach()</title>
```

```
  </head>
  <script>
      function forEach02() {
      console.error("정수형 배열 사용");
      var myNumbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
      var result = 0;
      function calculationSum(myNum) {
        // ❶
        result += myNum;
      }
      myNumbers.forEach(calculationSum); // ❷
      console.log("결과", result);
    }
  </script>

  <body>
    <h2>고차 함수 - forEach()</h2>
    <div>
     <!-- ❸  -->
      <input type="button" onclick="forEach02()" value="forEach2" />
    </div>
  </body>
</html>
```



❶ 콜백 함수에 활용할 원본 배열과 forEach( )를 myNumbers.forEach(...)와 같은 방식으로 체이닝했습니다. 이때 원본 배열 myNumbers에 forEach( )를 체이닝하면 forEach( )에 전달할 콜백 함수로 알아서 원본 배열의 값을 하나씩 넘기며 반복 호출합니다.

```
myNumbers.forEach(calculationSum);
```

따라서 calculationSum( )에는 반복 호출 시에 필요한 배열 요소 myNum만 첫 번째 매개변수로 정의하면 됩니다.

```
function calculationSum(myNum) {
  result += myNum;
}
```

❷❶ 에서 미리 정의한 콜백 함수 calculationSum( )을 인수로 전달하여 forEach( )를 호출합니다. 원본 배열의 요소가 1부터 10까지 순서대로 전달되므로 55라는 결과가 출력됩니다.

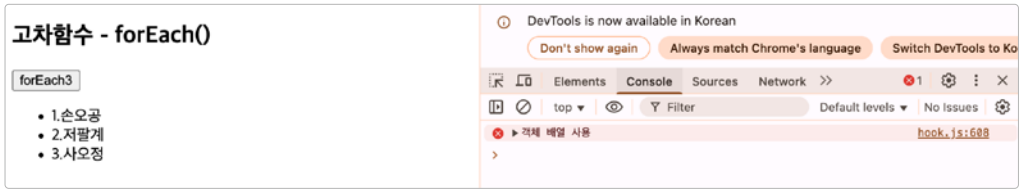
❸ [forEach2] 버튼을 누르면 forEach2( ) 함수를 실행한 결과를 콘솔에서 볼 수 있습니다. 결과를 보면 myNumbers에 있는 요소 1, 2, 3, ..., 10을 누적하여 더한 값을 result에 저장한 다음 반환한 것을 알 수 있습니다.

04 마지막으로 객체에 forEach( )를 체이닝하고, 체이닝하며 화면에 출력할 li 요소를 문자열로 만든 후 이를 innerHtml을 이용하여 화면에 적용하겠습니다.

01higherOrderFunc/03forEach.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>forEach()</title>
  </head>
  <script>
      function forEach03() {
      console.error("객체 배열 사용");
      var objArr = [
        // ❶
        { idx: 1, name: "손오공" },
        { idx: 2, name: "저팔계" },
        { idx: 3, name: "사오정" },
      ];
      var viewHtml = "";
      objArr.forEach(function (item) {
        // ❷
```

```
        viewHtml += "<li>" + item.idx + "." + item.name + "</li>";
      });
      document.getElementById("display").innerHTML = viewHtml; // ❸
    }
  </script>

  <body>
    <h2>고차 함수 - forEach()</h2>
    <div>
      <!-- ❹ -->
      <input type="button" onclick="forEach03()" value="forEach3" />
    </div>
    <ul id="display"></ul>
    <!-- ❺  -->
  </body>
</html>
```



❶ 객체 배열 objArr를 선언하고, ❷ objArr.forEach와 같이 체이닝으로 forEach( )를 사용했습니다. 원본 배열 objArr의 요소가 순서대로 콜백 함수에 전달되어 〈li〉 태그를 구성하고, 3개를 하나의 문자열로 연결합니다. 결과를 보면 〈li〉 태그를 구성하여 연결한 문자열을 출력하여 점 목록 형태로 표현되었음을 알 수 있습니다.

❸ ❷에서 연결하여 만든 〈li〉를 포함한 문자열을 ❺ 위치에 삽입합니다.

❹ 함수를 호출하기 위한 버튼입니다.

고차 함수가 뭐죠?

자바스크립트는 함수도 하나의 값처럼 다룰 수 있는 일급 객체 언어입니다. 함수를 변수에 담거나, 다른 함수의 인자로 넘기며, 함수가 또 다른 함수를 결과로 내보낼 수 있다는 뜻입니다. 이런 함수 중심의 프로그래밍 개념을 바탕으로 만든 것이 바로 고차 함수입니다.

고차 함수는 자바스크립트 표준인 ES5에서 처음 도입되었으며, 2009년 발표 이후 자바스크립트의 핵심 구성 요소로 자리 잡았습니다. 특히 배열에서 자주 사용하는 map( ), filter( ), reduce( )와 같은 메서드는 대표적인 고차 함수죠. 고차 함수를 사용하면 코드를 짧고 명확하게 작성할 수 있을 뿐만 아니라, 읽기 쉽고 고치기 편한 코드로 만들 수 있습니다. 리액트에서 이 같은 함수형 프로그래밍 개념은 핵심이라 할 수 있으므로 고차 함수에 대한 이해는 꼭 필요합니다.

문법 02 map( ) 함수

map( )은 배열의 각 요소에 대해 콜백 함수를 호출한 결과를 새로운 배열로 돌려줍니다. 원본 배열은 변경되지 않으며, 요소 개수가 같은 새로운 배열이 만들어집니다. 주로 데이터를 다른 형태로 바꿀 때 사용합니다. 다음 기본형을 살펴봅시다.

```
새로운 배열 = 원본 배열.map(function(현재값, 인덱스, 배열 자체) {
  // 배열의 각 요소에 대해 반복 호출
  return 가공된_요소; // 가공된 요소를 반환
});
```

map( )은 forEach( )와 동작 방식이 유사하지만 콜백 함수의 반환값으로 새로운 배열을 만든다는 중요한 차이점이 있습니다. 생성된 배열 크기는 원본 배열과 같습니다.

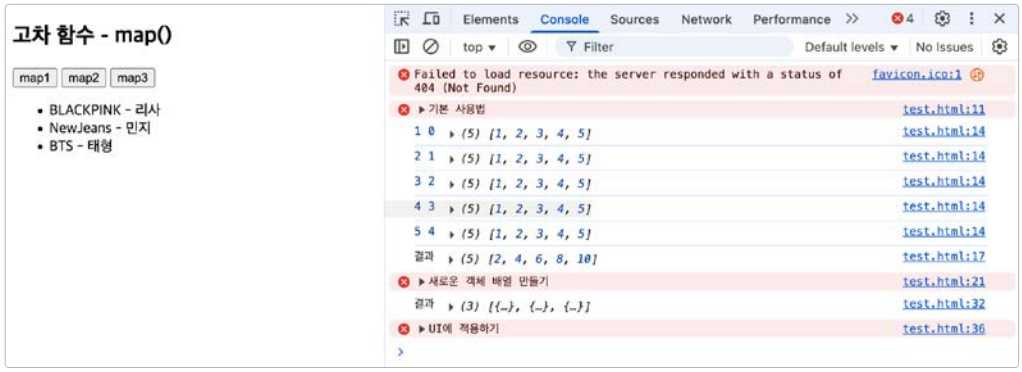
01 앞서 forEach( ) 함수를 사용했으므로 map( ) 함수는 3가지 함수를 한 파일에 모두 작성해서 실습하겠습니다. 다음 코드를 입력하고 실행해봅니다.

01higherOrderFunc/02map.html

```
<!DOCTYPE html>
<html lang="ko">
```

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>map()</title>
</head>
<script>
  function map01() {
    console.error("기본 사용법");
    var myNumbers = [1, 2, 3, 4, 5]; // ❶ 원본 배열
    var resultArr = myNumbers.map(function(currentValue, index, useArray) {
      console.log(currentValue, index, useArray);
      return currentValue * 2; // ❷ 각 요소에 x2를 한 다음 반환
    });
    console.log('결과', resultArr);
  }

  function map02() {
    console.error("새로운 객체 배열 만들기");
    var objArr = [
      { name: "아이유", salary: 1000 },
      { name: "정국", salary: 2000 },
      { name: "뉴진스", salary: 3000 }
    ];
    var resultArr = objArr.map(function(item) { // ❸ map 호출
      var returnObj = {}; // ❹ 새 객체 생성
      returnObj[item.name] = item.salary; // 키, 값 세팅
      return returnObj; // 배열 반환
    });
    console.log('결과', resultArr);
  }

  function map03() {
    console.error("UI에 적용하기");
    var objArr = [
      { name: "리사", group: "BLACKPINK" },
      { name: "민지", group: "NewJeans" },
      { name: "태형", group: "BTS" }
    ];
    var viewHtml = objArr.map(function(item) { // ❺ <li> 생성
      return '<li>' + item.group + ' - ' + item.name + '</li>';
```

```
    });
    // ❻ 배열을 하나의 문자열로 합쳐 쉼표 제거
    document.getElementById('display').innerHTML = viewHtml.join("");
  }
</script>
<body>
  <h2>고차 함수 - map()</h2>
  <div> // ❼ 함수 실행할 버튼
    <input type="button" onclick="map01()" value="map1">
    <input type="button" onclick="map02()" value="map2">
    <input type="button" onclick="map03()" value="map3">
  </div>
  <ul id="display"></ul>
</body>
</html>
```



❶ 크기가 5인 원본 배열을 통해 map( )을 호출하여 기본 동작을 확인합니다. 앞에서 사용한 forEach( )와 마찬가지로 각 요소를 반복합니다.

❷ 하지만 map( )은 반환값이 있습니다. 모든 요소에 2를 곱한 결과를 반환하여 resultArr이라는 새로운 배열을 생성합니다.

❸ 이번엔 객체 배열을 통해 map( )을 호출합니다. 매개변수가 1개이므로 배열의 각 요소인 객체가 순서대로 전달됩니다.

❹ 각 객체의 name 속성값을 키로, salary 속성값을 값으로 하는 새로운 객체를 만들어 반환합니

다. 이 객체들이 모여 새로운 배열을 구성합니다.

❺ 객체 배열을 반복하여 〈li〉 태그를 생성합니다. 각 요소의 nation과 name값을 이용해 순차적인 목록을 만듭니다.

❻ ❺에서 생성한 문자열을 본문에 삽입합니다. 여기서 join( ) 함수를 사용하지 않으면 viewHtml이 배열 형태이므로 항목 사이에 쉼표가 그대로 출력됩니다. map( ) 함수가 반환하는 값은 새 배열을 만드는 데, 배열을 그대로 출력하기 때문이죠. 따라서 join("")을 사용하여 배열 요소들이 하나의 문자열로 합쳐져 HTML 구조를 완성할 수 있도록 합니다.

❼ 함수를 실행하기 위한 버튼과 결과를 출력할 태그입니다.

문법 03 filter( ) 함수

filter( ) 함수는 배열의 각 요소에 대해 조건을 검사하여, 그 조건을 만족하는 요소만 모아 새로운 배열로 반환합니다. 따라서 원본 배열과 새로 생성된 배열의 요소 개수는 서로 다를 수 있습니다. 다음은 filter( ) 함수의 기본형입니다.

```
새로운 배열 = 원본 배열.filter(function(현재값, 인덱스, 배열 자체) {
  // 배열의 각 요소에 대해 반복 호출
  return 조건에 맞는 요소만 반환;
});
```

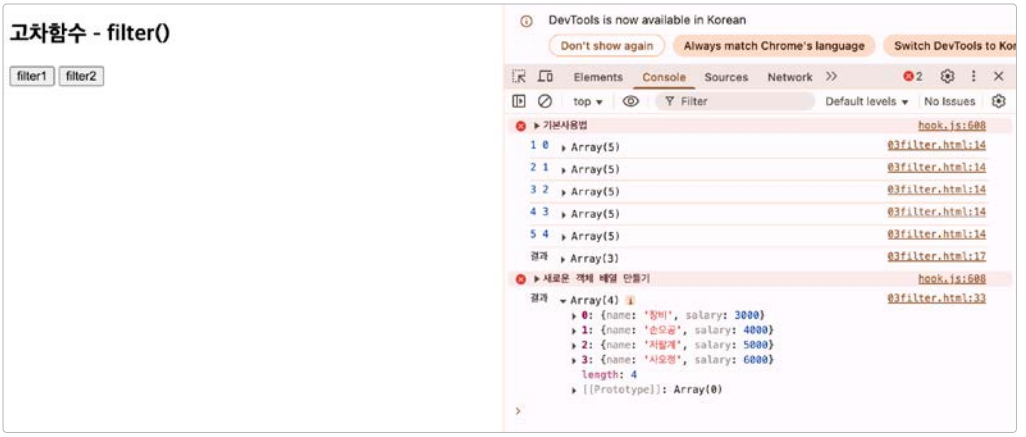
filter( )는 동작 방식은 map( )과 비슷하지만 차이점은 반환값으로 생성되는 새로운 배열 크기가 원본 배열과 다를 수 있다는 점입니다.

01 이번에는 filter( )를 사용했을 때 반환값으로 생성되는 배열 크기가 어떻게 달라지는지 다음 코드를 입력하고 실행해봅니다.

01higherOrderFunc/03filter.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>filter()</title>
</head>
<script>
  function filter01() {
    console.error("기본사용법");
    var myNumbers = [1, 2, 3, 4, 5];
    // ❶ 원본 배열을 순회
    var resultArr = myNumbers.filter(function (currentValue, index, useArray) {
      console.log(currentValue, index, useArray);
      return currentValue <= 3; // ❷ 조건을 만족(1 3)하는 값만 true
    });
    console.log('결과', resultArr);
  }

  function filter02() {
    console.error("새로운 객체 배열 만들기");
    var objArr = [
      { name: "유비",  salary: 1000 },
      { name: "관우",  salary: 2000 },
      { name: "장비",  salary: 3000 },
      { name: "손오공", salary: 4000 },
      { name: "저팔계", salary: 5000 },
      { name: "사오정", salary: 6000 }
    ];
    // ❸ salary가 3000 이상인 객체만 반환
    var resultArr = objArr.filter(function (item) {
      return item.salary >= 3000;
    });
    console.log('결과', resultArr);
  }
</script>
<body>
  <h2>고차 함수 - filter()</h2>
  <div> <!-- ❹ 각 함수를 실행할 버튼 -->
    <input type="button" onclick="filter01();" value="filter1">
    <input type="button" onclick="filter02();" value="filter2">
  </div>
</body>
</html>
```



❶ 원본 배열을 통해 filter( )를 호출하여 기본 동작을 확인합니다. 앞에서 작성한 map( )과 동일하게 5번 반복합니다.

❷ 이 조건에 만족하는 요소만 반환하여 새로운 배열을 생성합니다. 즉, 조건을 만족하는 값인 1~3까지만 반환됩니다.

❸ 객체 배열을 통해 반복하면서 salary가 3000 이상인 요소만 반환합니다. 장비~사오정까지 4개의 객체가 반환되어 새로운 객체 배열을 생성하게 됩니다.

❹ 각 함수를 실행하기 위한 버튼입니다. 순서대로 눌러보세요.

문법 04 reduce( ) 함수

reduce( )는 배열의 각 요소를 순차적으로 처리하며 하나의 누적된 결괏값을 반환합니다. 이전 요소의 반환값(누산값 Accumulator)과 현재 요소를 인수로 받아 콜백 함수가 실행되며, 다양한 형태의 계산 및 변환 작업에 유용합니다. 배열 요소의 총합, 평균 등의 작업에 사용할 수 있습니다. 특히 reduce( )는 초깃값의 유무와 콜백 함수의 반환값에 따라 결과가 달라지므로, 다른 배열 메서드보다 사용 방식이 다소 복잡할 수 있습니다. 다음은 reduce( ) 함수의 초깃값과 반환값에 따라 3가지 기본형으로 나누었습니다. 어떻게 다른지 살펴보겠습니다.

▼ 기본형 1 : 초깃값 X, 반환값 X

```
원본 배열 = [1, 2, 3]
원본 배열.reduce ( 콜백 함수 ( 누산값, 현재값, 인덱스, 배열 자체 ) {
    배열의 각 요소에 대해 반복 호출
} );
```

초깃값이 없으면 첫 번째 요소에서 원본 배열의 첫 번째 값이 누산값이 됩니다. 따라서 두 번째 값이 현재값이 됩니다. 반환값이 없으므로 두 번째 요소부터 누산값은 모두 undefined이 됩니다. 현재값은 배열 요소의 순서대로 지정됩니다. 즉, 초깃값이 없으면 원본 배열 크기 -1만큼 반복합니다.

▼ 기본형 2 : 초깃값 O, 반환값 X

```
원본 배열 = [1, 2, 3]
원본 배열.reduce ( 콜백 함수 ( 누산값, 현재값, 인덱스, 배열 자체 ) {
    배열의 각 요소에 대해 반복 호출
}, 초깃값 );
```

초깃값이 있으면 첫 번째 요소에서 누산값은 초깃값으로 할당됩니다. 따라서 원본 배열의 첫 번째 값이 현재값이 되어 원본 배열 크기만큼 반복합니다. 두 번째 요소부터의 누산값은 기본형 1과 같습니다.

▼ 기본형 3 : 초깃값 O, 반환값 O

```
원본 배열 = [1, 2, 3]
결괏값 = 원본 배열.reduce ( 콜백 함수 ( 누산값, 현재값, 인덱스, 배열 자체 ) {
    return 반환값;
}, 초깃값 );
```

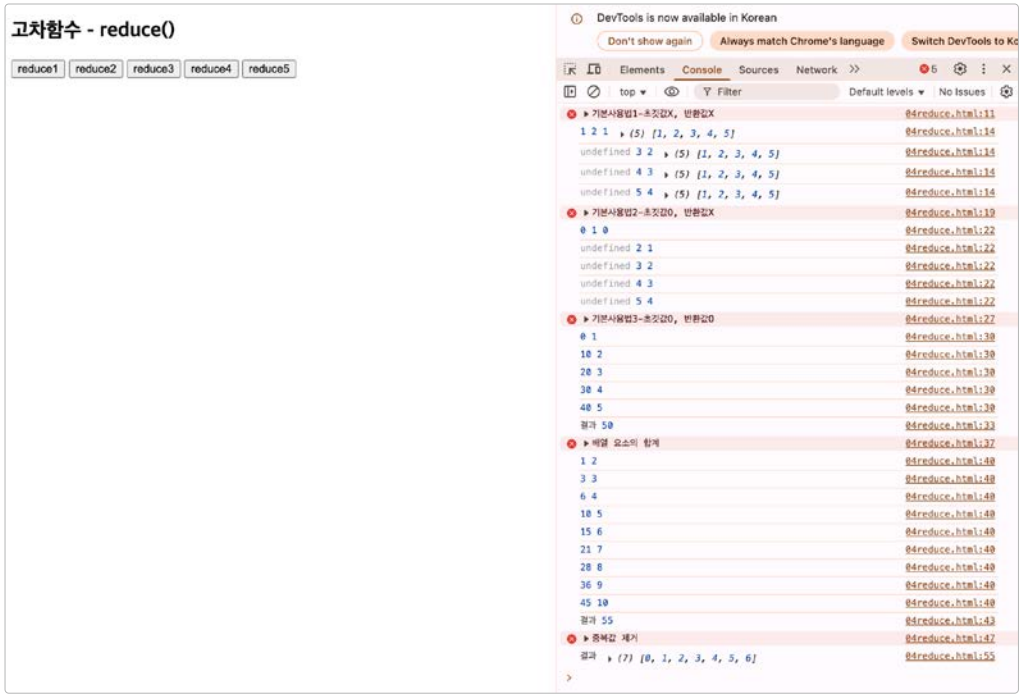
초깃값이 있으므로 원본 배열 크기만큼 반복합니다. 반환값이 있으면 이전 회차의 반환값이 다음 요소의 누산값이 됩니다. 마지막 회차에서 반환하는 값 C가 결괏값이 됩니다.

01 위 형식에서 보듯 초깃값과 반환값의 유무에 따라 반복 횟수 및 결과가 달라지게 됩니다. 다음 코드 실습을 결괏값이 어떻게 달라지는지 더 자세히 알아보겠습니다.

01higherOrderFunc/04reduce.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>reduce()</title>
</head>
<script>
  function reduce01(){
    console.error("기본사용법1-초깃값X, 반환값X");
    var myNumbers = [1,2,3,4,5];
    myNumbers.reduce(function(previousValue, currentValue, index, useArray) { // ❶
초깃값 없이 호출-prev=1·curr=2부터 시작
      console.log(previousValue, currentValue, index, useArray);
    });
  }

  function reduce02(){
    console.error("기본사용법2-초깃값0, 반환값X");
    var myNumbers = [1,2,3,4,5];
    // ❷ ❶과 같은 콜백(초깃값 지정)
    myNumbers.reduce(function(previousValue, currentValue, index) {
      console.log(previousValue, currentValue, index);
    }, 0); // ❸ 초깃값 0
  }

  function reduce03(){
    console.error("기본사용법3-초깃값0, 반환값0");
    var myNumbers = [1,2,3,4,5];
    // ❹ 초깃값·반환값 모두 사용
    var result = myNumbers.reduce(function(previousValue, currentValue) {
      console.log(previousValue, currentValue);
      return previousValue + 10; // ❺ 이전 값에 10을 더해 반환
    }, 0); // ❹ 초깃값 0
    console.log('결과', result);
```

```
  }

  function reduce04(){
    console.error("배열 요소의 합계");
    var myNumbers = [1,2,3,4,5,6,7,8,9,10];
    // ❻ 초깃값 없이 합계 계산
    var result = myNumbers.reduce(function(prevVal, currVal) {
      console.log(prevVal, currVal);
      return prevVal + currVal;  // ❼ 누적 합계
    });
    console.log('결과', result);
  }

  function reduce05(){
    console.error("중복값 제거");
    var myNumbers = [0,1,2,3,3,3,4,5,5,6,6]; // ❽ 중복 포함 배열
    var resultArr = myNumbers.reduce(function(prev, curr){ // ❾ 중복 제거 reduce
      if(prev.indexOf(curr) == -1){ // ❿ prev에 없으면 추가
        prev.push(curr);
      }
      return prev;
    }, []);
    console.log('결과', resultArr);
  }
</script>
<body>
  <h2>고차 함수 - reduce()</h2>
  <div> <!-- ⓫ 함수 실행 버튼 -->
    <input type="button" onclick="reduce01();" value="reduce1">
    <input type="button" onclick="reduce02();" value="reduce2">
    <input type="button" onclick="reduce03();" value="reduce3">
    <input type="button" onclick="reduce04();" value="reduce4">
    <input type="button" onclick="reduce05();" value="reduce5">
  </div>
</body>
</html>
```



❶ 원본 배열을 통해 reduce( )를 호출합니다. reduce( )는 4개의 매개변수를 사용할 수 있습니다. 순서대로 이전 요소의 반환값, 이번 요소의 값, 인덱스, 원본 배열입니다. 초깃값이 없으면 첫 번째 요소의 previousValue는 1, currentValue는 2가 됩니다. 따라서 전체 4회 반복됩니다. 반환값은 없으므로 두 번째 요소부터 previousValue는 undefined입니다.

❷ ❶과 같은 함수를 통해 reduce( )를 호출합니다.

❸ 초깃값으로 0을 지정했습니다. 이때는 첫 번째 요소의 previousValue는 0, currentValue는 1이 됩니다. 따라서 전체 5회 반복합니다.

❹ 초깃값이 있으므로 전체 5회 반복하여 currentValue는 1~5까지 순서대로 할당합니다.

❺ 첫 번째 요소의 previousValue는 초깃값인 0입니다. 두 번째는 0+10, 세 번째는 10+10이 되므로 마지막 요소에서는 50을 반환합니다.

❻ 1~10까지의 요소를 가진 배열을 통해 reduce를 호출합니다. 초깃값이 없으므로 첫 번째 요소의 prevVal은 1, currVal은 2가 됩니다. 따라서 전체 9회 반복됩니다.

❼ 첫 번째 요소에서 1+2, 두 번째는 3+3, 세 번째에서 6+4와 같이 진행되므로 배열 요소의 전체를 더한 결과가 마지막 요소에서 반환됩니다. 즉 결과 55를 출력합니다.

❽ 중복값이 있는 배열을 선언합니다.

❾ 빈 배열을 초깃값으로 선언하여 prev의 첫 번째 요소의 값은 [ ]입니다.

❿ indexOf( ) 함수는 배열에서 특정 요소가 있는지 검색하여 인덱스를 반환합니다. 검색 결과가 없다면 -1을 반환합니다. 즉 prev 배열에 curr 요소가 없다면 push( ) 함수로 삽입합니다. 즉, 이미 삽입된 요소가 있다면 -1을 반환하게 되니 결과적으로 중복 요소가 제거됩니다.

⓫ 버튼을 순서대로 눌러서 함수가 제대로 실행되는지 결과를 확인해보세요.

문법 05 some( ) 함수

some( )은 배열의 요소 중 하나라도 조건을 만족하면 true를 반환하고, 반복을 종료합니다. 모든 요소가 만족하지 않으면 false를 반환하며, '하나라도 해당되면 통과'라는 조건에 사용합니다. 다음은 some( ) 함수의 기본형입니다.

```
배열 = [1,2,3]
결과 = 배열.some ( 콜백 함수 ( 현재값 ) {
    return 조건에 일치하는 요소가 하나라도 있으면 true 반환 ;
} );
※ 결과는 true / false와 같은 boolean값입니다.
```

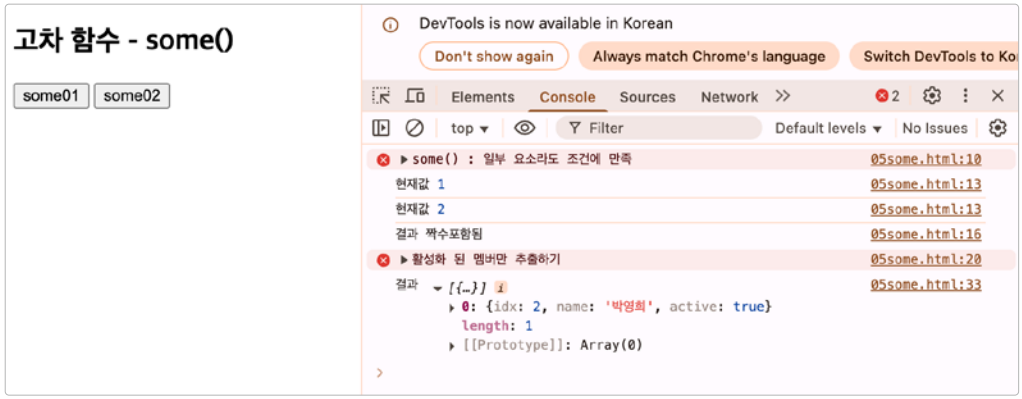
01 이번에는 some( )을 사용했을 때 반환값으로 생성되는 배열 크기가 어떻게 달라지는지 다음 코드를 입력하고 실행해봅니다.

01higherOrderFunc/05some.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>some()</title>
</head>
```

```
<script>
  function some01(){
    console.error("some() : 일부 요소라도 조건에 만족");
    var myNumbers = [1, 2, 3, 4, 5];
    var result = myNumbers.some(function(num) { // ❶ 정수 배열을 순회하며 짝수 존재
여부 확인
      console.log('현재값', num);
      return num % 2 === 0;
    });
    console.log("결과", result ? '짝수포함됨' : '짝수없음'); // ❷ true면 '짝수포함
됨', 아니면 '짝수없음' 출력
  }

  function some02(){
    console.error("활성화된 멤버만 추출하기");
    const members = [
      { idx: 1, name: "김철수", active: false },
      { idx: 2, name: "박영희", active: true },
      { idx: 3, name: "강민호", active: false }
    ];
    const isMember = members.some(function(member) { // ❸ active가 true인 멤버 존재
여부 확인
      return member.active;
    });
    if (isMember){ // ❹ 하나라도 active면 실행
      var result = members.filter(function(member){
        return member.active === true; // ❺ active가 true인 객체만 반환
      });
      console.log("결과", result); // ❻ '박영희'만 포함된 새 배열 출력
    } else { // ❼ 모두 비활성화된 경우
      console.log("모든 멤버가 비활성화되었습니다.");
    }
  }
</script>
<body>
  <h2>고차 함수 - some()</h2>
  <div>
    <input type="button" onclick="some01()" value="some01">
    <input type="button" onclick="some02()" value="some02">
```

```
  </div>
</body>
</html>
```



❶ some( )은 조건을 하나라도 만족하면 true를 반환합니다. 정수 배열을 통해 some( )을 호출하면 요소 개수만큼 반복하면서 짝수가 있는지 확인합니다. 두 번째 요소가 짝수이므로 여기에서 true를 반환하고 실행을 종료합니다.

❷ 결과는 true이므로 '짝수포함됨'이 출력됩니다.

❸ 이번에는 객체 배열을 통해 some( )을 호출해서 active가 true인 객체가 있는지 확인합니다. 두 번째 객체에서 true를 확인하고 반환한 다음 실행을 종료합니다.

❹ isMember가 true이므로 블록으로 진입하여 filter( )를 호출합니다.

❺ filter( )로 객체 배열에서 active가 true인 항목만 반환하여 새로운 배열을 생성합니다.

❻ 새로운 배열에서는 '박영희' 객체만 있습니다.

❼ 만약 모든 항목이 false라면 "모든 멤버가 비활성화되었습니다."가 출력될 겁니다.

문법 06 every( ) 함수

배열의 모든 요소가 조건을 만족해야 true를 반환합니다. 하나라도 조건을 만족하지 않으면 false를 반환하고 반복을 종료합니다. '전부 해당되어야 통과'라는 조건에 사용합니다. 다음은 every( ) 함수의 기본형입니다.

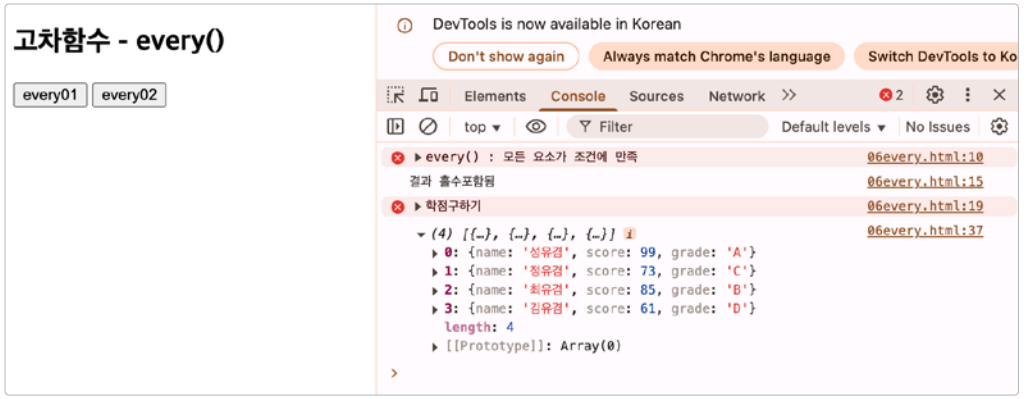
```
배열 = [1,2,3]
결과 = 배열.every ( 콜백 함수 ( 현재값 ) {
    return 모든 요소가 조건에 만족하면 true 반환 ;
} );
※ 결과는 true/false와 같은 boolean값입니다.
```

01 이번에는 every( ) 함수를 사용했을 때 조건에 따라 결과가 어떻게 달라지는지 다음 코드를 입력하고 실행해봅니다.

01higherOrderFunc/06every.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>every()</title>
</head>
<script>
  function every01(){
    console.error("every() : 모든 요소가 조건에 만족");
    var myNumbers = [1, 2, 3, 4, 5];
    var result = myNumbers.every(function(num) { // ❶ 모든 요소가 짝수인지 검사
      return num % 2 === 0;
    });
    // ❷ false이므로 '홀수포함됨' 출력
    console.log("결과", result ? '짝수만있음' : '홀수포함됨');
  }

  function every02(){
    console.error("학점구하기");
    var students = [ // ❸ 학생 점수 객체 배열
      { idx: 1, name: "성유겸", score: 99 },
```

```
      { idx: 2, name: "정유겸", score: 73 },
      { idx: 3, name: "최유겸", score: 85 },
      { idx: 4, name: "김유겸", score: 61 }
    ];
    // ❹ 모든 점수가 0점 초과인지 확인
    const isValid = students.every(function(student) {
      return student.score > 0;
    });
    if (isValid) { // ❺ 0점이 없으면 실행
      const result = students.map(function(student) {
        return { // ❻ 이름·점수·학점을 담은 새 객체 반환
          name: student.name,
          score: student.score,
          grade: getGrade(student.score) // ❽ 학점 계산 함수 호출
        };
      });
      console.log(result);
    } else { // ❼ 0점이 있으면 실행
      console.log("0점이 있으므로 결과를 볼 수 없습니다.");
    }
  }

 function getGrade(score) { // ❽ 점수 → 학점 변환
    if (score >= 90) return "A";
    if (score >= 80) return "B";
    if (score >= 70) return "C";
    if (score >= 60) return "D";
    return "F";
  }
</script>
<body>
  <h2>고차 함수 - every()</h2>
  <div>
    <input type="button" onclick="every01()" value="every01">
    <input type="button" onclick="every02()" value="every02">
  </div>
</body>
</html>
```



❶ 정수 배열을 통해 every( )를 호출해서 모든 요소가 짝수인지 판단합니다. 첫 번째 요소가 홀수이므로 false를 반환하고 실행을 종료합니다.

❷ ❶에서 false를 반환했으므로 '홀수포함됨'을 출력합니다.

❸ 네 명의 학생의 점수를 객체 배열로 선언합니다.

❹ every( )를 통해 모든 학생들의 점수가 0점을 초과하는지 확인합니다. 네 명 모두 1점 이상으로, 0점인 학생이 한 명도 없으므로 true를 반환합니다.

❺ if문 블록의 map( )을 호출합니다.

❻ 각 학생 객체를 이름, 점수, 학점(❽의 학점 계산 함수 호출)을 이용해서 새로운 객체를 만들어 반환합니다. 이 값은 새로운 배열로 만들어집니다.

❼ 만약 0점인 학생이 한 명이라도 있으면 "0점이 있으므로 결과를 볼 수 없습니다."를 실행합니다.

❽ 학점을 판단해서 반환하는 함수를 정의했습니다. ❻에서 학생의 점수를 인수로 호출하여 학점을 반환받습니다.

지금까지 고차 함수 문법을 살펴보았습니다. 고차 함수가 등장하기 전에는 배열을 다루기 위해 반복문을 직접 작성해야 했습니다. 그러나 고차 함수를 도입한 이후에는 보다 간결하고 선언적인 방식으로 배열을 처리할 수 있게 되었습니다.

고차 함수는 단순히 문법적인 편의성만 제공하는 것이 아니라, 코드의 가독성과 유지보수성을 크게 높여 줍니다. 특히 forEach, map, filter, reduce와 같은 함수들은 각 역할이 명확해서, 상황

에 맞게 잘 활용하면 코드의 품질을 높이는 데 도움이 됩니다. 다음 표는 앞서 배운 고차 함수를 정리한 겁니다. 상황에 따라 적절한 고차 함수를 선택하여 사용할 수 있도록 확인해보기 바랍니다.

▼ 고차 함수 정리표

함수명	용도	반환값	특징
forEach()	단순 반복 및 부수 효과 처리	undefined	반환값이 없고, 체이닝도 불가능
map()	요소 변환 및 가공	새로운 배열	배열의 각 요소를 가공하여 원본 배열과 크기가 같은 배열을 반환
filter()	조건에 맞는 요소 필터링	새로운 배열	조건에 만족하는 요소만 추출하여 새로운 배열을 반환. 새 배열은 원본 배열보다 크기가 작을 수 있음
reduce()	누적 계산	단일값(숫자, 객체)	누산 처리를 통한 결과 도출함. 즉 배열을 하나의 값으로 반환
some()	일부 요소 조건 만족 여부 검사	true / false	하나의 조건만 만족하면 true 반환 후 종료
every()	전체 요소 조건 만족 여부 검사	true / false	하나라도 조건에 만족하지 않으면 fasle 반환 후 종료

02.2 모던 자바스크립트(ES6)

앞서 map, filter, reduce와 같은 고차 함수를 통해 배열을 간단한 코드로 처리할 수 있는 방법을 알아봤습니다. 이제 이런 고차 함수를 포함해, 리액트 코드 전반에서 빈번히 사용되는 모던 자바스크립트 문법, 특히 ES6[ECMAScript 2015] 이후에 도입된 주요 기능을 살펴보겠습니다.

ES6는 자바스크립트에 큰 변화를 가져온 버전으로, let, const를 이용한 변수 선언 방식, 화살표 함수, 템플릿 리터럴, 구조 분해 할당 등 다양한 기능이 새롭게 추가되었습니다. 이 문법들은 React 컴포넌트를 작성할 때 매우 자주 등장하므로, 익숙해지는 것이 중요합니다. 여기서는 리액트를 배우는 데 필요한 ES6 핵심 문법을 중심으로 간결하고 실용적인 예제와 함께 소개합니다. 앞에서 배운 고차 함수와 함께, 모던 자바스크립트 문법을 학습하겠습니다.

ES6 이전의 문제점

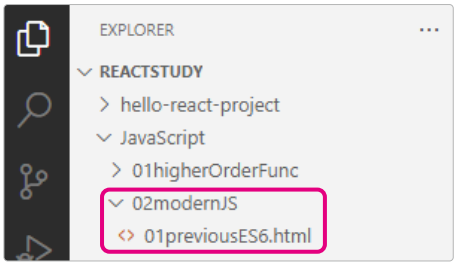
리액트를 배우기 전에 꼭 짚고 넘어가야 할 것이 있습니다. 바로 '모던 자바스크립트 문법이 왜 필요한가?'에 대한 이해입니다. ES6 이전의 자바스크립트는 유연함을 무기로 빠르게 성장했지만, 그만큼 애매하고 헷갈리는 부분도 많았습니다.

대표적인 예로 var로 선언한 변수의 범위(Scope) 문제, 암묵적인 전역 변수 생성, 변수의 중복 선언 허용, 그리고 호이스팅Hoisting으로 인한 코드의 실행 흐름을 예측하기 어려운 점 등을 들 수 있습니다.

 호이스팅(Hoisting)은 변수와 함수 선언이 코드 실행 전에 메모리에 먼저 할당되어, 선언문이 코드 최상단으로 끌어올려진 것처럼 동작하는 현상입니다. 이로 인해 변수를 선언하기 전에 접근해도 오류가 나지 않고, undefined이 출력되는 현상이 발생됩니다.

이런 특성은 코드의 예측 가능성과 안정성을 떨어뜨려, 협업 과정에서 버그를 유발하는 주요 원인이 되었습니다. 여기서는 ES6 이전 자바스크립트에서 자주 발생하던 문제점을 하나씩 살펴보겠습니다. 이 문제들이 실제로 어떻게 발생하는지 이해하면, 이후 학습할 let, const와 같은 ES6 문법이 왜 필요한지 자연스럽게 알 수 있습니다.

01 설명만 보면 조금 이해가 어려울 수 있으므로 코드를 작성하여 확인하겠습니다. 먼저 예제 작성을 위해 다음과 같이 JavsScript 폴더 하위에 02modernJS 폴더 생성한 다음 01previousES6.html 파일을 만드세요.



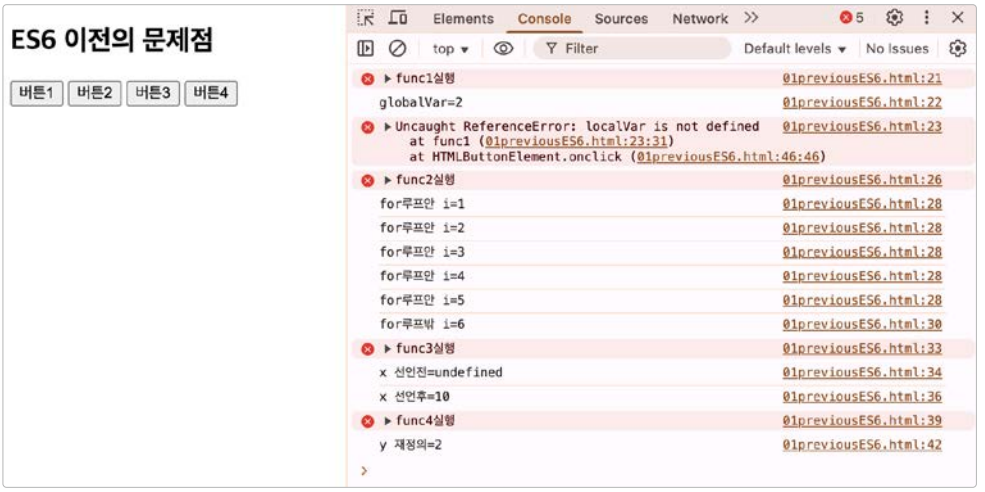
02 html 파일을 생성했다면 다음과 같이 코드를 작성하고 웹브라우저에서 열어 결과를 확인해보세요.

02modernJS/01previousES6.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>ES5 문제점 데모</title>
</head>
<body>
  <h2>ES6 이전의 문제점</h2>
  <script>
    function myFunc(){
      globalVar = 2;      // ❶ var 없이 대입 → 암묵적 전역 변수 생성(버그 원인)
      var localVar = 1;   // ❷ 함수 스코프 var → 블록 구분이 없어 누수 가능성
    }

    function func1(){
      myFunc();      // ❸ myFunc 호출 후 globalVar만 전역, localVar는 접근 불가
      console.error('func1 실행');
      console.log('globalVar = ' + globalVar); // ❹ 의도치 않은 전역 변수도 접근 가능
      console.log('localVar  = ' + localVar);  // ReferenceError 발생
    }

    function func2(){
      console.error('func2 실행');
      for (var i = 1; i <= 5; i++) { // ❺ for 루프에서도 var → 루프 밖으로 누수
        console.log('for 루프 안 i = ' + i);
      }
      console.log('for 루프 밖 i = ' + i);   // ❻  루프 밖에서도 i(=6) 접근 가능
    }

    function func3(){
      console.error('func3 실행');
      // ❼ 선언이 호이스팅되어 최상단으로 끌어올려져 undefined 출력
      console.log('x 선언 전 = ' + x);
      var x = 10;                       // ❽ 실제 대입은 여기서 이루어짐
      console.log('x 선언 후 = ' + x); //     10 출력
    }

    function func4(){
      console.error('func4 실행');
      var y = 1;   // ❾ 같은 스코프에서 var 재선언 허용 → 값 덮어쓰기
      var y = 2;   // 의도치 않은 재정의 버그 발생 가능
      console.log('y 재정의 = ' + y); //     2 출력
    }
```

```
  </script>

  <div>
    <button type="button" onclick="func1();">버튼 1: 전역 누수</button>
    <button type="button" onclick="func2();">버튼 2: 스코프 누수</button>
    <button type="button" onclick="func3();">버튼 3: 호이스팅</button>
    <button type="button" onclick="func4();">버튼 4: 재선언</button>
  </div>
</body>
</html>
```



❶ 변수 선언 시 var를 생략하면 암묵적 전역 변수로 선언됩니다.

❷ var를 통해 변수를 선언하면 해당 함수의 지역 변수가 됩니다.

❸ myFunc()를 호출하면 전역 변수인 globalVar과 지역 변수 localVar 이렇게 2개의 변수가 선언됩니다.

❹ 분명히 다른 지역에서 선언되었지만 globalVar 전역 변수이기 때문에 정상적으로 출력됩니다. localVar는 myFunc의 지역 변수이므로 출력하면 오류가 발생합니다.

❺ for문의 블록에서 선언된 변수 i는 var로 선언했기 때문에 블록 레벨이 아닌 함수 레벨의 지역 변수로 선언됩니다. 따라서 for문 내부에서 1~5까지 출력됩니다.

❻ 일반적으로 블록을 벗어나면 변수 i는 소멸되어야 합니다. 하지만 var로 선언된 변수는 함수 레벨의 지역 변수이므로 for문을 벗어나도 접근이 가능하여 6이 출력됩니다.

❼ 변수 x는 ❽에서 선언됩니다. 따라서 선언되기 전에는 오류가 발생해야 합니다. 하지만 ❽에서 선언된 변수가 호이스팅되어 끌어올려지므로 undefined가 출력됩니다.

❽ 변수 x를 선언하며, 실젯값을 대입합니다. 만약 이 부분을 주석 처리하면 ❼에서 오류가 발생합니다.

❾ 변수 y를 선언한 후 재선언을 하고 있습니다. 같은 이름의 변수를 선언하면 당연히 오류가 발생해야 하지만 var로 선언한 변수는 재선언이 가능하여 오류가 발생하지 않습니다.

문법 07 let과 const

ES6 이전에 사용하던 변수 선언 방식인 var는 변수의 중복 선언, 암묵적 전역 변수 선언, 호이스팅으로 인해 예측 불가능한 동작을 자주 일으켰습니다. 개발자에게 각종 오류를 준 원인들이었죠. 이런 문제를 극복하기 위해 ES6에서는 let과 const라는 새로운 변수 선언 키워드를 도입했습니다. 두 키워드는 var보다 더 예측 가능하고 안전한 방식으로 변수를 관리할 수 있게 해줍니다. 이제 let과 const가 var와 어떻게 다른지 더 자세히 알아보겠습니다.

let이란?

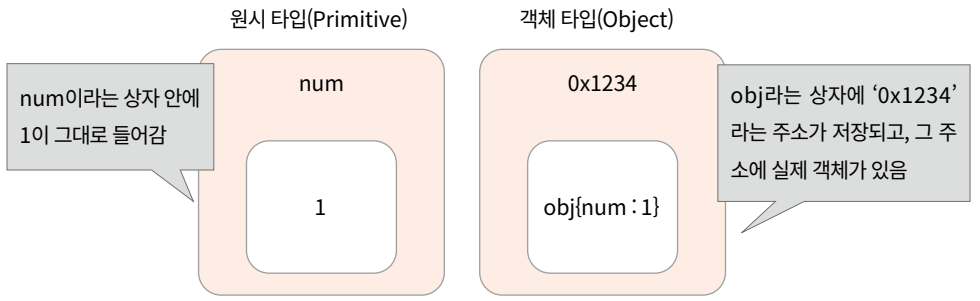
let은 블록 스코프를 가지므로 선언된 블록({}) 안에서만 유효하며, 블록 외부에서는 접근할 수 없습니다. 또한 같은 스코프 내에서 같은 이름으로 변수를 재선언할 수도 없습니다. 내부적으로는 호이스팅되지만, 선언 전에 접근하면 TDZ[Temporal Dead Zone] 때문에 참조 오류가 발생합니다. 암묵적 전역 변수 선언이 불가능합니다.

const란?

const 역시 블록 스코프를 가진다는 점에서 기본적인 특성은 let과 동일하지만 값을 변경할 수 없는 상수로 선언한다는 점에서 차이가 있습니다. const는 선언과 동시에 반드시 초기화해야 하며,

한 번 할당된 값은 변경할 수 없습니다. 다만 상수를 객체나 배열로 선언할 경우, 변수에 저장된 값은 고정되지만 내부 속성은 변경할 수 있습니다.

원시 타입(Primitive Type)과 객체 타입(Object Type) 이해하기



원시 타입은 단일값만 저장하므로 num에는 값 1이 직접 저장됩니다. 반면 객체는 여러 복잡한 값을 담는 구조로, 메모리에 객체가 저장된 다음 그 참조값(주소값)을 obj에 저장합니다.

앞의 그림에서 보듯이 const로 선언할 경우, obj 변수에 저장된 참조값 자체는 변경할 수 없습니다. 그러나 그 참조값이 가리키는 객체 내부의 속성값은 변경할 수 있습니다. 이는 참조값 자체는 그대로 유지되기 때문입니다. 따라서 객체의 내용을 수정해도 const의 제약을 위반하지 않습니다.

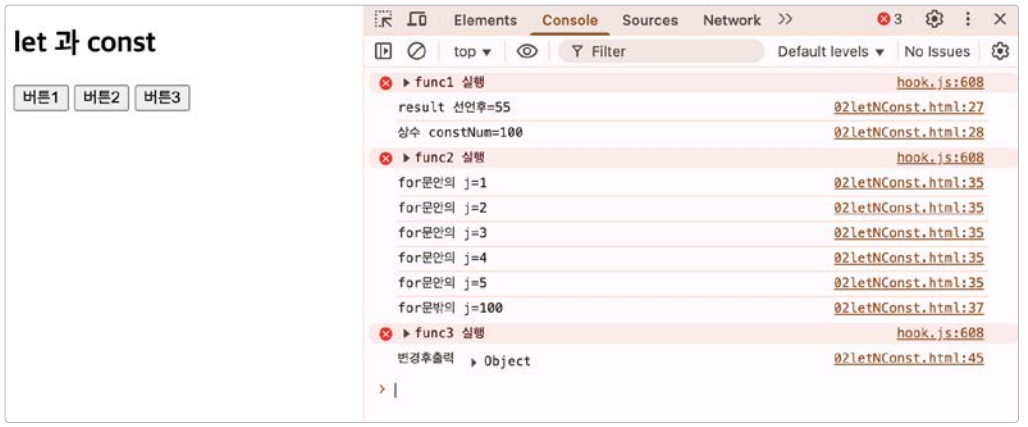
01 let과 const가 어떻게 동작하는지 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/02letNConst.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>let과 const</title>
</head>
<body>
  <h2>let과 const</h2>
  <script>
```

```
  const constNum = 100; // ❶ const 키워드로 상수 선언 및 초기화

  const sumLimit = function (start, end) { // ❷ start부터 end까지의 합을 구해 반환
    let sum = 0;
    for (let i = start; i <= end; i++) {
      sum += i;
    }
    return sum;
  };

  const func1 = function () {
    console.error('func1 실행');
    // console.log('result 선언전=' + result);
    // ❸ TDZ(Temporal Dead Zone)로 오류 → 주석 처리
    let result = sumLimit(1, 10); // ❹ 1~10까지 합을 받아 result에 저장
    console.log('result 선언후=' + result);
    console.log('상수 constNum=' + constNum);
    // constNum = 200; // ❺ const 변수 재할당 불가 → 오류 → 주석 처리
  };

  const func2 = function () {
    console.error('func2 실행');
    let j = 100; // ❻ 블록 스코프 변수 j 선언
    for (let j = 1; j <= 5; j++) { // ❼ for 루프용 j 선언, 1~5 출력
      console.log('for문 안 j=' + j);
    }
    console.log('for문 밖 j=' + j); // ❽ 외부 j는 100 그대로
    // let j = 200; // ❾ 같은 스코프 내 재선언 불가 → 오류 → 주석 처리
  };

  const func3 = function () {
    console.error('func3 실행');
    // ❿ 객체 참조를 const로 선언
    const obj = { name: '강백호', position: '포워드' };
    obj.name = '송태섭'; // ⓫ 참조는 고정, 내부 프로퍼티 변경 가능
    obj.position = '가드';
    console.log('변경 후 출력', obj); // ⓬ 변경된 객체 내용 확인
  };
</script>
```

```
  <div>
    <button type="button" onclick="func1();">버튼1</button>
    <button type="button" onclick="func2();">버튼2</button>
    <button type="button" onclick="func3();">버튼3</button>
  </div>
</body>
</html>
```



❶ const로 상수를 선언한 다음 초기화합니다. 선언한 이후에는 값을 바꿀 수 없습니다.

❷ start부터 end까지 더한 결과를 반환하는 sumLimit( ) 함수를 선언합니다.

❸ result를 선언하기 전 출력을 시도합니다. 하지만 let은 선언 전에 출력을 시도하면 오류가 발생하므로 해당 부분은 주석으로 처리했습니다.

❹ sumLimit( )을 호출한 후 1~10까지 더한 결과를 반환받아 result 변수에 저장합니다. 출력은 정상적으로 처리됩니다.

❺ 상수는 초기화 후 값을 변경할 수 없으므로 오류가 발생하므로 주석으로 처리했습니다.

❻ let으로 변수를 선언합니다.

❼ for문에서 반복을 위한 변수 j를 선언하고 5회 반복합니다. 그러면 1~5까지 출력됩니다.

❽ let은 블록 스코프를 가지므로 for문이 종료되면 j도 함께 사라집니다. 따라서 6이 아닌 100이

출력됩니다.

❾ j는 이미 선언되었으므로 재선언이 불가능합니다. 따라서 SyntaxError이 발생하므로 주석으로 처리합니다.

❿ const를 통해 객체를 선언합니다. 이 경우 객체의 참조값이 상수에 할당됩니다. 이 역시도 선언한 이후 다른 객체로 변경할 수 없습니다.

⓫ 객체의 값을 변경합니다. 이 경우 객체의 참조값을 변경하는 것이 아니라 내부의 값을 변경하는 것이므로 정상적으로 처리됩니다.

⓬ 출력하면 변경된 값을 확인할 수 있습니다.

문법 08 화살표 함수

이번에는 화살표 함수Arrow Function을 알아보겠습니다. 화살표 함수는 기존 함수에 비해 간결한 문법과 명확한 this 바인딩을 통해 코드의 가독성과 유지보수성을 크게 높여줍니다. 다음은 기존 함수와 화살표 함수의 기본형을 표로 비교한 것으로 어떤 차이점이 있는지 살펴봅시다.

기존 함수	화살표 함수

`function 함수명(매개변수1, 매개변수2) {`
`  함수의 실행부;`
`  return 반환값;`
`}`
	`const() 함수명 = (매개변수1, 매개변수2) => {`
	`  함수의 실행부;`
	`  return 반환값;`
	`}`

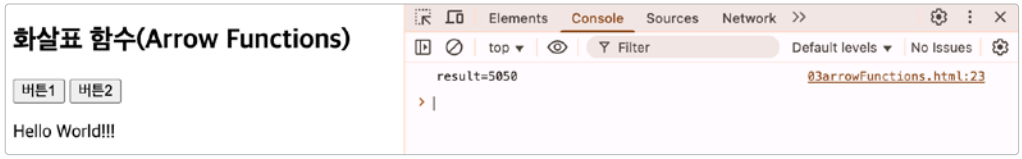
`var 함수명 = function (매개변수1, 매개변수2) {`
`  함수의 실행부;`
`  return 반환값;`
`}`
	`const() 함수명 = 매개변수 => 반환값;`

화살표 함수는 매개변수가 1개인 경우 소괄호를 생략할 수 있습니다. 반대로 매개변수가 0개 이거나 2개 이상이라면 소괄호를 반드시 써야 합니다. 또한 실행문이 한 줄뿐이라면 중괄호와 return 모두 생략할 수 있어 더욱 간결하게 코드를 작성할 수 있습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/03arrowFunctions.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>화살표 함수</title>
</head>
<body>
  <h2>화살표 함수(Arrow Functions)</h2>
  <script>
    const sumLimit = (start, end) => { // ❶ start에서 end까지의 합을 계산해 반환
      let sum = 0;
      for (var i = start; i <= end; i++) {
        sum += i;
      }
      return sum;
    };

    let func1 = () => { // ❷ sumLimit 호출용 함수(매개변수 없음 → 괄호 필요)
      let result = sumLimit(1, 100);
      console.log('result = ' + result);
    };

    // ❸ 매개변수 1개 → 괄호 생략, 한 줄 반환 → 중괄호·return 생략
    const helloStr = param => 'Hello ' + param;
    let func2 = () => { // ❹ helloStr 호출 결과를 화면에 출력
      document.getElementById('display').innerHTML =
        helloStr('World!!!');
    };
  </script>

  <div>
    <button type="button" onclick="func1();">버튼1</button>
    <button type="button" onclick="func2();">버튼2</button>
  </div>
  <p id="display"></p>
</body>
</html>
```



❶ start에서 end까지의 합을 계산해서 반환하는 화살표 함수를 정의합니다.

❷ sumLimit을 호출하기 위해 함수를 정의합니다. 매개변수가 없으므로 소괄호는 생략할 수 없습니다.

❸ 매개변수가 1개이므로 소괄호를 생략할 수 있습니다. 또한 실행 문장이 한 줄이어서 반환값 1개로, 중괄호와 return도 생략할 수 있습니다.

❹ helloStr( ) 함수를 호출한 후 반환값을 innerHTML로 삽입합니다. 웹브라우저에서 버튼을 누르면 "Hello World!!!"가 나오는 것을 확인할 수 있습니다.

문법 09 템플릿 리터럴

자바스크립트에서 문자열을 다루는 일은 웹 개발에서 가장 기본적이고 자주 사용되는 작업입니다. 화면에 텍스트를 표시하거나 HTML 요소를 조합하는 등 웹브라우저에 출력되는 대부분의 정보가 문자열로 구성되기 때문이죠.

ES6 이전에는 문자열을 따옴표로 감싸고 + 연산자로 이어붙이는 방식이 사용했습니다. 그러나 이 방식은 변수 삽입이번거롭고, 코드의 가독성도 떨어지는 한계가 있습니다. 이런 불편함을 해소하기 위해 템플릿 리터럴Template Literals이 ES6에 도입되었고, 문자열 처리 방식에 큰 변화를 가져왔습니다.

01 html 파일을 생성하고 다음 코드를 입력하여 파일을 실행해봅시다.

02modernJS/04templateLiterals.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>템플릿리터럴</title>
</head>
<body>
  <h2>템플릿 리터럴(Template literal)</h2>
  <script>
    const func1 = () => {
      console.error('func1 실행');
      // ❶ ES6 이전에는 줄바꿈을 위해 \n을 사용. 콘솔에 '\'를 표시하려고 \\ 사용
      console.log("es5의 방식\n줄바꿈엔 \\n 사용");
      // ❷ 템플릿 리터럴은 실제 줄바꿈·띄어쓰기가 그대로 반영
      console.log(`템플릿 리터럴줄바꿈과 white space 그대로    적용`);
    };

    const func2 = () => {
      console.error('func2 실행');
      let a = 5;
      let b = 10;
      // ❸ ES6 이전: 문자열과 변수를 + 연산자로 연결
      console.log("5와 10의 합은 "+ (a + b) +" 입니다.");
      // ❹ 템플릿 리터럴: ${ } 안에 표현식 삽입(표현식 인터폴레이션)
      console.log(`5와 10의 합은 ${a + b} 입니다.`);
    };

    const func3 = () => {
      console.error('func3 실행');
      const template = `템플릿 리터럴은 '작은따옴표(single quotes)'과 "큰따옴표
(double quotes)"를 혼용할 수 있다.`; // ❺ 작은따옴표와 큰따옴표를 자유롭게 혼용 가능
      console.log(template);
    };
  </script>

  <div>
    <button type="button" onclick="func1();">버튼1</button>
    <button type="button" onclick="func2();">버튼2</button>
    <button type="button" onclick="func3();">버튼3</button>
  </div>
</body>
</html>
```

❶ ES6 이전에는 텍스트 줄바꿈을 위해 \n(역슬래시 n)을 사용했습니다. 두 번째 부분에 역슬래시를 2번 쓴 이유는 콘솔에 출력하기 위함입니다.

❷ 템플릿 리터럴은 엔터키로 텍스트 자체를 줄바꿈하면 그대로 적용되어 결과가 나옵니다. 또한 띄어쓰기의 때에도 스페이스키를 사용하면 횟수만큼 적용됩니다.

❸ ES6 이전에는 문자열과 변수를 연결하기 위해 + 연산자를 사용했습니다.

❹ 템플릿 리터럴을 사용하면 ${ }로 문자열 사이에 변수를 삽입할 수 있습니다. 또한 중괄호 안에는 연산식도 추가할 수 있습니다. 이를 표현식 삽입 방법(Expression interpolation)이라고 합니다.

❺ 작은따옴표와 큰따옴표를 혼용해서 사용할 수 있습니다.

문법 10 구조 분해 할당

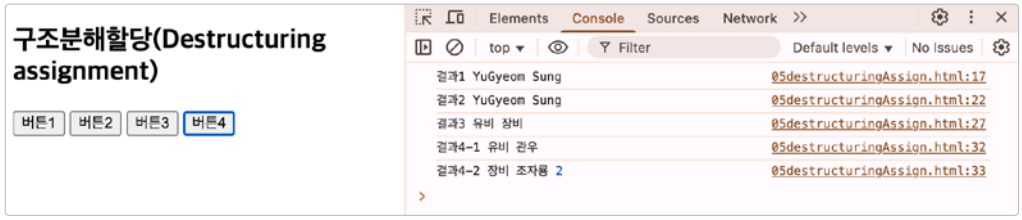
구조 분해 할당Destructuring Assignment은 객체나 배열에서 필요한 값을 손쉽게 꺼내 변수에 할당할 수 있게 해줍니다. 복잡한 데이터 구조에서도 원하는 값을 명확하고 간결하게 추출할 수 있어, 코드의 가독성과 효율성을 크게 높여주죠. 뿐만 아니라 구조 분해 할당은 변수 선언과 동시에 값을 추출할 수 있으며, 함수의 매개변수로도 활용할 수 있어 다양한 상황에서 유용하게 활용됩니다. 특히 API 응답 처리, 상태 관리, 다차원 배열 분해 등 다양한 실무 환경에서 코드의 명확성을 높이는 데 효과적입니다. 그럼 구조 분해 할당이 실제로 어떻게 동작하는지 예제를 통해 살펴보겠습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/05destructuringAssign.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>구조분해할당</title>
</head>
<body>
  <h2>구조분해할당(Destructuring assignment)</h2>
  <script>
    let func1 = () => {
      let nameArr = ["YuGyeom", "Sung"]; // ❶ 크기가 2인 배열
      let [firstName, lastName] = nameArr; // ❷ 배열 순서대로 변수에 할당
      console.log('결과1', firstName, lastName);
    }

    let func2 = () => {
      // ❸ split으로 만든 배열 → 구조 분해
      let [firstName, lastName] = "YuGyeom Sung".split(' ');
      console.log('결과2', firstName, lastName);
    }

    let func3 = () => {
      // ❹ 두 번째 요소 건너뛰기
      let [person1, , person3] = ["유비", "관우", "장비", "조자룡"];
      console.log('결과3', person1, person3);
    }

    let func4 = () => {
      // ❺ 스프레드로 나머지 묶기
      let [name1, name2, ...rest] = ["유비", "관우", "장비", "조자룡"];
      console.log('결과4-1', name1, name2);
      console.log('결과4-2', rest[0], rest[1], rest.length);  // ❻ rest는 배열
    }
  </script>

  <button type="button" onclick="func1();">버튼1</button>
```

```
  <button type="button" onclick="func2();">버튼2</button>
  <button type="button" onclick="func3();">버튼3</button>
  <button type="button" onclick="func4();">버튼4</button>
</body>
</html>
```



❶ 크기가 2인 배열을 선언했습니다.

❷ 배열을 그대로 할당하면 요소의 순서대로 첫 번째 요소는 firstName, 두 번째 요소는 lastName에 할당됩니다.

❸ split( ) 함수는 문자열을 지정한 구분자로 분리해서 배열로 반환해줍니다. 여기서는 공백을 기준으로 문자열을 분리하므로 ❶과 같은 배열이 됩니다. 따라서 ❷와 동일하게 값이 할당됩니다.

❹ 4개의 요소를 가진 배열을 할당합니다. 등호를 중심으로 왼쪽을 보면 [person1, 비어 있음, person3]와 같이 두 번째 요소가 비어 있습니다. person1에는 유비, person3에는 장비가 할당됩니다. 즉 비어 있는 부분은 관우에 해당하는 부분이므로 할당되지 않습니다.

❺ 스프레드 연산자를 사용하여 name1과 name2에는 유비, 관우가 할당됩니다. 남은 장비와 조자룡은 rest에 배열로 할당됩니다. 스프레드 연산자에 대해서는 바로 다음 예제에서 자세히 살펴보겠습니다.

❻ rest는 배열이므로 rest[0], rest[1]과 같이 인덱스로 접근하고, length로 크기를 출력할 수 있습니다.

문법 11 스프레드 연산자

배열과 객체는 자바스크립트에서 데이터를 구조화하고 다루는데 핵심적인 역할을 합니다. 여기에 스프레드 연산자Spread Operator는 데이터를 보다 유연하고 직관적으로 확장하거나 결합할 수 있습니다. 점 세 개(...)로 표현되는 이 연산자는 배열, 객체, 이터러블(iterable) 등의 요소를 펼쳐서 새로운 컬렉션을 만들도록 도와줍니다. 이를 통해 복사, 병합, 함수 호출 시 인수 전달 등 다양한 작업을 간결하게 처리할 수 있으며, 불변성을 유지하는 코드 작성에도 효과적입니다. 따라서 스프레드 연산자는 반복적인 작업을 줄이고, 코드의 가독성과 유지보수성을 높이며, 함수형 프로그래밍 스타일을 자연스럽게 적용할 수 있도록 해줍니다.


이터러블(iterable)이란 반복 가능한 요소의 컬렉션을 말하며, 기본적으로 배열은 이터러블의 일종입니다. 이 외에도 문자열, Map, Set, arguments 객체 등이 있습니다.

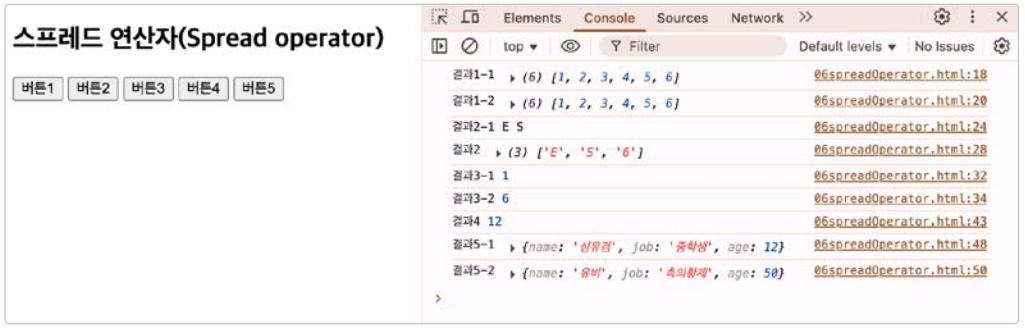
01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/06spreadOperator.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>스프레드연산자</title>
</head>
<body>
  <h2>스프레드 연산자(Spread operator)</h2>
  <script>
    let arr1 = [1, 2, 3]; // ❶ 전역 변수 arr1, arr2 선언 후 배열 할당
    let arr2 = [4, 5, 6];

    const func1 = () => {
      const new1 = arr1.concat(arr2); // ❷ ES6 이전 방식: concat으로 두 배열 병합
      console.log('결과1-1', new1);
      const new2 = [...arr1, ...arr2]; // ❸ 스프레드 연산자로 간단히 두 배열 병합
      console.log('결과1-2', new2);
    };

    const func2 = () => {
```

```
    let str1 = 'ES'; // ❹ 문자열도 인덱스로 접근 가능한 유사 배열
    console.log('결과2-1', str1[0], str1[1]);
    // str1.push('6'); // ❺ 문자열은 배열이 아니므로 push 사용 시 오류
    let strArr = [...str1]; // ❻ 문자열을 펼쳐 ['E','S'] 새 배열 생성
    strArr.push('6');
    console.log('결과2', strArr);          //     ['E','S','6'] → "ES6"
  };

  const func3 = () => {
    // ❼ 개별 인수 전달 방식(1 출력)
    let result1 = Math.min(arr1[0], arr1[1], arr1[2]);
    console.log('결과3-1', result1);
    let result2 = Math.max(...arr2); // ❽ 스프레드로 배열을 펼쳐 전달(6 출력)
    console.log('결과3-2', result2);
  };

  const func4 = () => {
    let contacts = [ // ❾ 객체 배열 선언
      { name: '철수', age: 10 },
      { name: '영희', age: 11 },
      { name: '유겸', age: 12 }
    ];
    // ❿ age값만 추출해 배열로 만든 뒤 스프레드로 펼쳐 최댓값 산출
    let maxAge = Math.max(...contacts.map(item => item.age));
    console.log('결과4', maxAge);           //     12
  };

  const func5 = () => {
    let boy = { name: '성유겸', job: '중학생' };
    boy = { ...boy, age: 12 }; // ⓫ 기존 객체를 펼치고 age 속성 추가
    console.log('결과5-1', boy);
    // ⓬ 같은 키를 포함해 스프레드 → 기존 값을 덮어쓰며 여러 속성 수정
    boy = { ...boy, name: '유비', age: 50, job: '촉의황제' };
    console.log('결과5-2', boy);
  };
</script>

<div>
  <button type="button" onclick="func1();">버튼1</button>
  <button type="button" onclick="func2();">버튼2</button>
```

```
    <button type="button" onclick="func3();">버튼3</button>
    <button type="button" onclick="func4();">버튼4</button>
    <button type="button" onclick="func5();">버튼5</button>
  </div>
</body>
</html>
```



❶ 전역 변수로 arr1, arr2를 선언한 후 배열을 할당합니다.

❷ ES6 이전의 배열을 연결하는 방식으로 concat( ) 함수를 사용합니다.

❸ 스프레드 연산자를 통해 2개의 배열을 1개로 연결합니다.

❹ 문자열을 변수에 할당합니다. 문자열은 배열과 유사하게 인덱스로 각 요소에 접근할 수 있습니다. 이를 유사 배열 객체(array-like object)라고 부릅니다.

❺ 문자열은 배열이 아니므로 push( )와 같은 배열 관련 함수를 사용하면 오류가 발생합니다.

❻ 스프레드 연산자를 문자열에 적용하면 각 문자를 개별적으로 분리하여 새로운 배열을 생성합니다. 즉 ['E', 'S']와 같은 형태가 됩니다. 따라서 push를 통해 '6'을 문자열에 추가할 수 있으므로 ES6가 출력됩니다.

❼ Math 객체에서 제공하는 min, max와 같은 함수는 요소를 개별적으로 전달해야 합니다. 따라서 전달할 요소가 많다면 Math.min(arr1[0], arr1[1]…)처럼 길어지므로 사용이 어렵습니다. arr1의 최솟값 1이 나옵니다.

❽ 스프레드 연산자를 사용하면 배열 자체를 펼쳐서 전달할 수 있으므로 요소가 많아도 편리하게

사용할 수 있습니다. arr2의 최댓값인 6이 출력됩니다.

❾ 객체 배열을 변수에 할당합니다. 여기서 contacts는 사람마다 age, name를 담은 객체 배열입니다.

❿ map을 통해 age의 값만 추출하여 새로운 배열을 만듭니다. 이 배열을 스프레드 연산자로 펼쳐서 max() 함수의 인수로 전달합니다. 따라서 최댓값으로 가장 많은 나이인 12가 출력됩니다.

⓫ 기존 객체를 새로운 객체 내부에 펼친 후 age:12를 추가합니다. 스프레드 연산자로 속성을 쉽게 추가할 수 있습니다.

⓬ 이번에는 boy에 이미 존재하는 Key를 가진 속성을 추가했으므로 기존의 값을 수정합니다.

문법 12 for ~ of

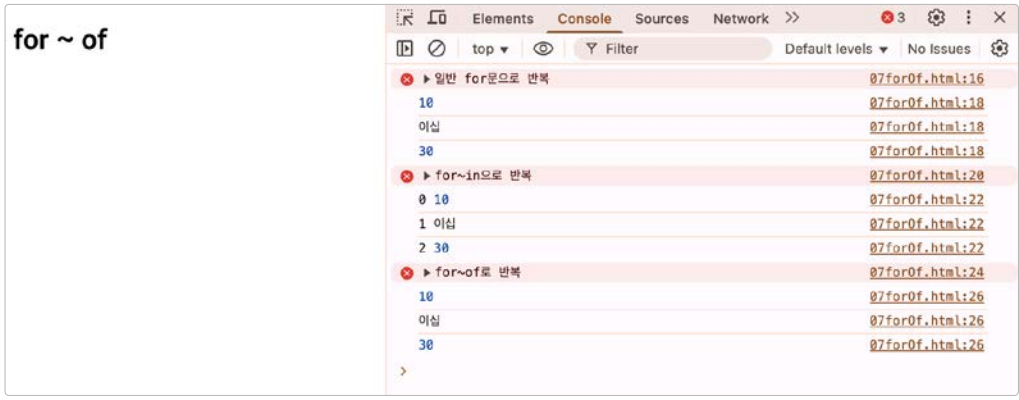
자바스크립트에서는 데이터를 반복적으로 처리하는 일이 매우 많습니다. 배열, 문자열, Set, Map 등 이터러블 객체를 순회하며 값을 추출하거나 조작하는 작업이 대표적입니다. ES6에서는 이런 반복 작업을 더욱 간결하고 직관적으로 수행하기 위해 for ~ of문을 도입했습니다. for ~ of문은 값 중심으로 반복하며, 기존의 for문이나 forEach보다 코드가 간단하고 가독성이 뛰어납니다. 한편, 기존에 사용되던 for ~ in문과 비슷하게 보이지만 반복 대상과 목적이 다릅니다. 이 부분은 예제를 통해 직접 살펴겠습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/07forOf.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>for~of문</title>
</head>
<body>
  <h2>for ~ of</h2>

  <script>
```

```
    const myArray = [10, '이십', 30];

    // ❶ 화살표 함수를 즉시 호출(IIFE) → 한 번만 실행되는 코드 블록
    (() => {
      console.error('일반 for문으로 반복');
      for (let i = 0; i < myArray.length; i++) { // ❷ 전통적 for문: 길이·인덱스 필요
        console.log(myArray[i]);
      }

      console.error('for~in으로 반복');
      for (const i in myArray) {  // ❸ for~in: 배열이면 인덱스, 객체면 키 반환
        console.log(i, myArray[i]);
      }

      console.error('for~of로 반복');
      for (const v of myArray) { // ❹ for~of: 값 자체를 반환 → 가장 간결
        console.log(v);
      }
    })(); // ❶ IIFE 끝
  </script>
</body>
</html>
```



❶ 화살표 함수를 위와 같이 괄호 형태로 감싸서 정의와 동시에 실행합니다. 이를 즉시 실행 함수Immediately Invoked Function Expression, IIFE라고 합니다. 별도의 호출 없이 딱 한 번만 실행하는 코드에 자주 쓰입니다.

```
(() => {
    이 함수의 실행부는 즉시 실행됩니다.
})();
```

❷ 일반적인 for문으로 배열을 반복하며 출력합니다. 배열의 길이를 알고 있어야 사용할 수 있으므로, 배열의 길이를 직접 확인합니다. 요소에 접근하기 위해 인덱스를 사용해야 하므로, 코드가 비교적 길고 번잡한 면이 있습니다.

❸ for ~ in문은 배열일 때 인덱스, 객체일 때 키를 반환합니다. 따라서 요소에 접근하려면 ❷ myArray[i]과 동일하게 인덱스를 한 번 더 사용해야 합니다. 만약 배열이 아닌 객체가 대상이라면 키(Key)를 반환합니다.

❹ for ~ of문은 배열 요소 값 자체를 즉시 반환합니다. 배열 크기나 인덱스가 없어도 요소에 접근할 수 있으므로 다른 for문에 비해 편리합니다.

문법 13 다양한 파라미터

자바스크립트에서 함수는 프로그램의 동작을 정의하고 구성하는 핵심 요소입니다. ES6는 함수의 유연성과 표현력을 높이기 위해 기본 매개변수Default Parameters와 나머지 매개변수Rest Parameters라는 두 가지 기능이 도입했습니다. 기본 매개변수는 인수가 전달되지 않았을 때 사용할 기본값을 함수 선언부에서 직접 지정할 수 있도록 해줍니다. 이를 통해 별도의 조건문 없이도 매개변수의 누락을 안전하게 처리할 수 있죠. 나머지 매개변수는 전달할 인수 개수가 정해지지 않았을 때, 인수 전체를 배열 형태로 수집해 하나의 변수로 다룰 수 있게 해줍니다. 덕분에 함수는 더 다양한 호출 형태를 유연하게 처리할 수 있습니다.

이 두 기능 덕분에 함수 설계는 더욱 직관적이고 견고해졌으며, 반복적인 방어 로직(값이 누락되었는지 확인하는 조건문 등을 의미) 대신 핵심 로직에 집중할 수 있게 되었습니다. 이제 기본 매개변수와 나머지 매개변수가 실제로 어떻게 활용되는지 다음 예제를 통해 살펴보겠습니다.

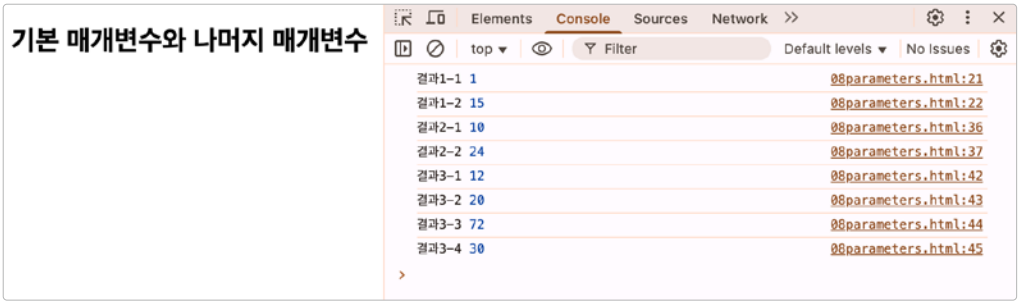
 방어 로직이란 함수 안에서 전달된 인수(또는 외부의 값)가 없는 경우나 잘못된 경우를 미리 점검하고 안전하게 처리하기 위해 넣어두는 조건문 또는 초기화 코드를 가리킵니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/08parameters.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>매개변수</title>
</head>
<body>
  <h2>기본 매개변수와 나머지 매개변수</h2>
  <script>
    // ❶ params를 나머지 매개변수로 선언(…): 전달된 모든 인수를 배열로 수집
    const func1 = (...params) => {
      let sum = 0;
      for (let item of params) { // ❷ for~of로 배열(params)을 순회하며 합산
        sum += item;
      }
      return sum;
    };
    console.log('결과1-1', func1(1)); // ❸ 인수 개수와 무관하게 호출 가능
    console.log('결과1-2', func1(1, 2, 3, 4, 5));

    // ❹ 일반 매개변수(mode) 뒤에 나머지 매개변수(params) → 반드시 마지막에 위치
    const func2 = (mode, ...params) => {
      if (mode === 'add') { // ❺ mode가 'add'면 누적 합
        let sum = 0;
        for (let item of params) sum += item;
        return sum;
      } else if (mode === 'multi') {  //  'multi'면 누적 곱
        let mul = 1;
        for (let item of params) mul *= item;
        return mul;
      }
    };
    // ❻ 첫 인수는 mode, 나머지는 params
    console.log('결과2-1', func2('add', 1, 2, 3, 4));
    console.log('결과2-2', func2('multi', 1, 2, 3, 4));

    // ❼ 기본 매개변수: 인수가 없으면 width=3, height=4 사용
```

```
    const func3 = (width = 3, height = 4) => width * height;
    console.log('결과3-1', func3()); // ❽ 인수 없음 → 기본값 3·4 사용(12)
    console.log('결과3-2', func3(5)); // ❾ width=5, height 기본값 4(20)
    console.log('결과3-3', func3(8, 9)); // ❿ width=8, height=9(72)
    console.log('결과3-4', func3(undefined, 10)); // ⓫ width 기본값 3, height=10(30)
  </script>
</body>
</html>
```



❶ params로 나머지 매개변수로 선언합니다. 선언하는 방식은 스프레드 연산자와 동일하게 …을 사용하며, 전달되는 모든 값을 한꺼번에 받아 배열로 저장합니다.

❷ 배열로 저장된 인수를 for~of문으로 순회하면서 sum에 누적해서 더해줍니다.

❸ func1은 나머지 매개변수로 정의되었습니다. 따라서 인수 개수에 상관없이 호출할 수 있습니다.

❹ 일반 매개변수와 나머지 매개변수를 함께 사용해서 함수를 정의할 수 있습니다. 단 이때는 나머지 매개변수가 반드시 제일 뒤에 위치해야 합니다. 만약 위치를 변경하면 즉시 오류가 발생합니다.

❺ mode의 값이 'add'라면 누적 합, 'mult'라면 누적 곱을 계산해서 반환합니다.

❻ 첫 번째 인수는 mode로 나머지 인수들은 params로 전달합니다.

❼ 매개변수를 선언할 때 기본값(default)을 할당합니다. 이 경우 매개변수로 전달되는 인수가 없다면 기본값을 사용하게 됩니다. 여기서는 인수가 없으면 width=3, height=4 값이 자동으로 들어갑니다.

❽ 아무 값도 전달하지 않았으므로 기본값인 3, 4가 할당되어 결과로 12가 나옵니다.

❾ 첫 번째 인수만 전달하므로 width는 5, height는 기본값이 4가 할당되어 결과로 20이 나옵니다.

❿ 모든 값을 전달하면 기본값은 무시됩니다. 따라서 8, 9가 할당되어 결과로 72가 나옵니다.

⓫ 만약 두 번째 인수만 전달하고 싶다면 첫 번째 인수를 undefined으로 설정합니다. 그러면 width는 기본값인 3, height는 10이 할당되어 결과로 30이 나옵니다.

문법 14 Array.from( )

Array.from( )은 유사 배열 객체나 반복 가능한 값을 진짜 배열로 변환하는 메서드입니다. 즉, 배열처럼 보이지만 진짜 배열이 아닌 arguments, Set, Map과 같은 객체를 대상으로 하죠. arguments, Set, Map 등은 배열이 아니므로 map( ), filter( )와 같은 고차 함수를 사용할 수 없습니다. 이때 Array.from( )을 통해 배열로 변환한 후 원하는 메서드를 적용하면 됩니다. 또한 Array.from( )은 두 번째 인수로 매핑 함수를 직접 지정할 수 있기 때문에, 배열을 만들면서 동시에 요소를 변형하는 작업도 간편하게 할 수 있습니다. 이번 예제는 다소 어려운 표현식이 많으므로 더 집중해서 학습하길 권장합니다. 그럼 시작하겠습니다.

유사 배열 객체의 생성 조건을 알아봐요

유사 배열 객체(array-like object)란 모양은 배열처럼 보이지만 실제로는 Array 인스턴스가 아닌 일반 객체입니다. 유사 배열 객체를 생성하려면 숫자 형태의 키(index)를 가진 속성과 length 속성이 존재해야 합니다. 다음을 통해 유사 배열 객체가 되는 경우와 그렇지 못한 경우를 살펴봅시다.

- **유사 배열 객체가 되는 경우**
  - { 0: “A”, 1: “B”, 2: “C”, length: 3 } ⇐ 숫자 키에 length가 포함되어 있습니다.
  - { 1: “X”, 3: “Z”, length: 4 } ⇐ 키는 연속적이지 않아도 상관없습니다.
  - { length: 3 } ⇐ length만 있습니다. 그러면 모든 값이 undefined으로 초기화됩니다. 키는 0부터 순차적으로 부여됩니다.

- “Hello” ⇐ 문자열 자체도 유사 배열 객체입니다.

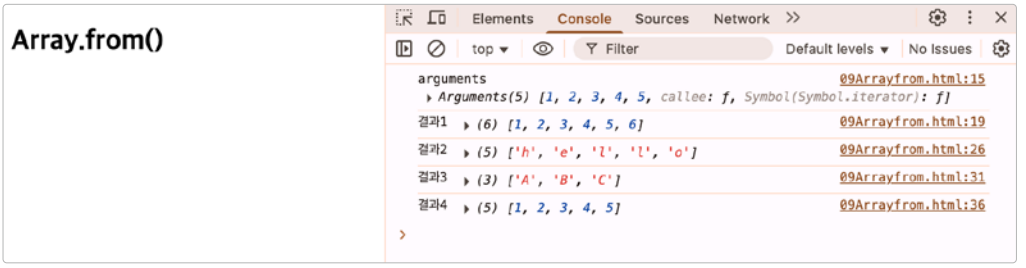
- **유사 배열 객체가 안 되는 경우**
  - { 0: “A”, 1: “B” } ⇐ length가 없습니다.
  - { a: “X”, b: “Y”, length: 2 } ⇐ 키가 숫자가 아닙니다.
  - { 0: “가”, 1: “나”, length: “two” } ⇐ length가 숫자가 아닙니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/09Arrayfrom.html

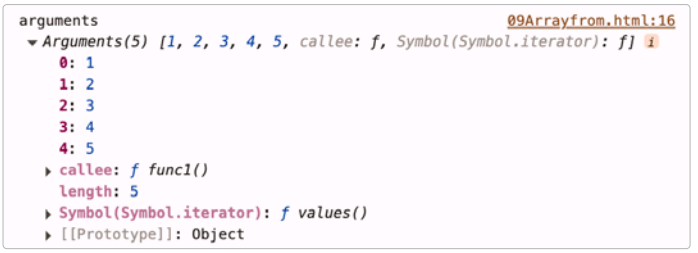
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Array.from()</title>
</head>
<body>
  <h2>Array.from()</h2>
  <script>
    /* ---------- arguments → 배열 변환 ---------- */
    (function func1() { // ❶ 매개변수 없이 정의했지만, arguments로 인수를 한 번에 받음
      console.log('arguments', arguments); // ❷ arguments는 유사 배열 객체 형태로 출력
      // arguments.push(6);  // ❸ arguments는 배열이 아니므로 push 사용 시 오류
      // ❹ Array.from으로 arguments를 진짜 배열로 변환
      let newArr = Array.from(arguments);
      newArr.push(6); // ❺ 배열이 되었으므로 push 사용 가능 → 마지막에 6 추가
      console.log('결과1', newArr);
    })(1, 2, 3, 4, 5);

  /* --------- 문자열 → 배열 변환 후 map --------- */
    ((str) => {
      // let result1 = str.map(...) // ❻ 문자열은 배열이 아니므로 map 호출 시 오류
      let strArr = Array.from(str); // ❼ 문자열을 글자 단위 배열로 변환
['H','e','l','l','o']
      let result2 = strArr.map(char => char.toLowerCase());
      console.log('결과2', result2);        //     ['h','e','l','l','o']
    })('Hello');
```

```
  /* ---------- 직접 만든 유사 배열 객체 ---------- */
    ((str) => {
      // ❽ 숫자 키와 length를 갖춘 유사 배열 객체
      const obj = { 0: 'A', 1: 'B', 2: 'C', length: 3 };
      console.log('결과3', Array.from(obj)); // ❾ Array.from으로 ['A','B','C'] 변환
    })();

  /* ---------- length와 매핑 함수 활용 ---------- */
    ((num) => {
      // ❿ length: num 객체를 전달하고, 매핑 함수로 1~num값 채워 배열 생성
      let result = Array.from({ length: num }, (_, i) => i + 1);
      console.log('결과4', result);          //     [1, 2, 3, 4, 5]
    })(5);
  </script>
</body>
</html>
```



❶ func1은 매개변수가 없는 상태로 정의되었지만 호출 시 인수를 전달하고 있습니다. 이 경우 유사 배열 객체인 arguments를 통해 인수를 한 번에 받을 수 있습니다.

❷ 콘솔을 확인하면 arguments가 배열 형식으로 출력되어 배열처럼 보일 수 있습니다. 실제로 확인하면 일반 배열과는 조금 다른 형식으로 출력되는 걸 알 수 있습니다.



❸ arguments는 진짜 배열이 아니므로 push( ) 함수 호출 시 오류가 발생합니다. 주석을 풀어서 직접 확인해보세요.

```
▶Uncaught TypeError: arguments.push is not a function 09Arrayfrom.html:17
    at func1 (09Arrayfrom.html:17:18)
    at 09Arrayfrom.html:21:7
```

❹ Array.from(arguments)로 유사 배열 객체를 진짜 배열로 변환합니다.

❺ 배열로 변환되었으므로 push( )와 같은 배열 함수를 사용할 수 있습니다. 마지막 요소로 6이 추가됩니다.

❻ 문자열도 배열처럼 인덱스로 접근할 수 있지만, 진짜 배열은 아니므로 map( )과 같은 고차 함수를 사용하면 오류가 발생합니다.

❼ 진짜 배열로 변환 후 map( ) 함수를 호출합니다. Array.from(str)로 각 요소를 소문자로 나눈 다음 새로운 배열로 반환합니다.

❽ 유사 배열 객체를 직접 생성합니다.

❾ 배열로 변환 후 출력합니다.

❿ Array.from( )은 두 번째 인수로 매핑 함수mapping function을 직접 지정할 수 있습니다. 이 매핑 함수는 map( ) 함수처럼 각 요소에 대해 반복적으로 호출되어, 새로운 배열의 요소를 생성합니다.

Array.from( )의 매핑 함수가 동작하는 원리를 알아봐요

Array.from({ length: num }, (_, i) => i + 1)는 길이 num짜리의 빈 배열 생성합니다. 그런 다음 각 인덱스마다 map( ) 함수를 호출하여 값 생성하며 이 결과를 모아 최종적으로 새로운 배열을 만듭니다. 그럼 이 코드가 어떻게 작동하는지 각 요소별로 세부적으로 살펴보겠습니다.

`{ length: num }`

- 유사 배열 객체를 생성하는 표현입니다.
- num이 2라면, 길이 2인 배열이 [undefined, undefined]과 같이 생성됩니다.

`(_, i) => i + 1`

- 매핑 함수로 map( ) 함수와 유사하게 각 요소에 대해 반복적으로 호출됩니다.

- 첫 번째, 두 번째 매개변수는 (현재값, 인덱스)인데 _를 쓴 이유는 해당 매개변수를 사용하지 않는다는 명시적 표현입니다.
- 따라서 인덱스 i만을 사용해서 인덱스가 0부터 시작해 1씩 증가하는 숫자 배열이 만들어집니다.

예를 들어 num이 5라면 결과는 [1, 2, 3, 4, 5]입니다.

문법 15 Map

자바스크립트에서는 데이터를 키와 값 이렇게 쌍으로 저장할 때 보통 Object를 사용했습니다. 그러나 Object는 문자열만 키로 쓸 수 있으며, 키가 추가된 순서가 보장되지 않는다는 한계가 있습니다. 이런 한계를 보완하기 위해 ES6에서는 Map이라는 새로운 자료구조가 도입되었습니다. Map은 객체와 달리 숫자, 함수 등 모든 타입의 값을 키로 사용할 수 있고 삽입 순서를 그대로 유지합니다. 또한 데이터 크기나 존재 여부를 쉽게 확인할 수 있는 size, has, delete와 같은 메서드를 제공합니다. 이제 Map의 사용법을 살펴보겠습니다.


'Object는 키의 순서를 보장하지 않는다'는 말은 과거에는 기술적으로 정확했지만, 현재는 '부분적으로 순서를 따르긴 하지만 완벽한 순서 보장이 필요한 경우 Map을 사용하라'는 의미로 이해하는 것이 좋습니다.

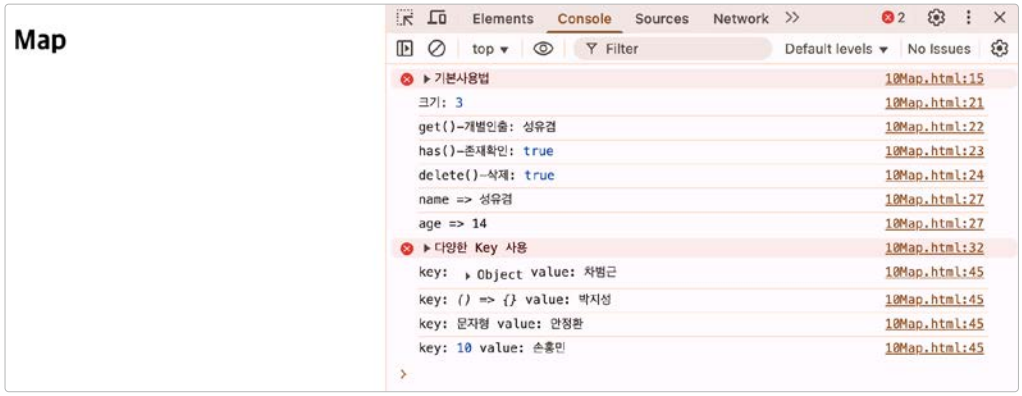
01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/10Map.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Map</title>
</head>
<body>
  <h2>Map</h2>
  <script>
    /* ------------------ 기본 사용법 ------------------ */
```

```
    (() => {
      console.error('기본사용법');
      const person = new Map();  // ❶ Map 객체 person 생성(키·값, 삽입 순서 유지)
      person.set('name', '성유겸'); // ❷ set으로 데이터 저장
      person.set('age', 14);
      person.set('hobby', '축구');

      console.log('크기:', person.size);  // ❸ size: 요소 개수 반환
      console.log('get()-개별인출:', person.get('name'));  // ❹ get(key): 값 인출
      console.log('has()-존재확인:', person.has('age'));   // has(key): 존재 여부
      console.log('delete()-삭제:', person.delete('hobby'));  // delete(key): 삭제
후 성공 여부

      for (const [key, value] of person) {  // ❺ for~of 순회, 구조 분해로 [key,
value] 접근
        console.log(`${key} => ${value}`);
      }
    })();

    /* ------------------ 다양한 Key 사용 ------------------ */
    (() => {
      console.error('다양한 Key 사용');
      const objKey = { idx: 1 };      // 객체형 Key     // ❻ 다양한 타입의 키 생성
      const funcKey = () => {};       // 함수형 Key
      const strKey = '문자형';         // 문자형 Key
      const numberKey = 10;           // 숫자형 Key

      const friends = new Map();  // ❼ 새로운 Map friends 생성 후 데이터 입력
      friends.set(objKey, '차범근');
      friends.set(funcKey, '박지성');
      friends.set(strKey, '안정환');
      friends.set(numberKey, '손흥민');

      friends.forEach((value, key) => {  // ❽ forEach로 모든 [key, value] 출력
        console.log('key:', key, 'value:', value);
        // 템플릿 리터럴 사용 시 객체 키는 [object Object]처럼 출력
        // console.log(`${key} : ${value}`);
      });
    })();
  </script>
```

```
</body>
</html>
```



❶ Map 객체 person을 생성합니다. 키key와 값value으로 저장되며, 입력한 순서대로 유지합니다.

❷ set( ) 함수로 데이터를 저장합니다.

❸ size로 Map 크기, 즉 요소 개수를 반환합니다.

❹ Map 객체가 제공하는 함수를 사용합니다.

- **get(key)** : 특정 키의 값을 반환합니다.
- **has(key)** : 해당 키가 존재하는지 확인하여 true/false를 반환합니다.
- **delete(key)** : 데이터를 삭제하고 성공 여부를 true/false로 반환합니다.

❺ for~of문으로 순회하면 [key, value] 구조 분해로 접근하여 꺼낼 수 있습니다.

❻ Map 객체는 다양한 타입의 키를 사용할 수 있습니다. 객체형, 함수형, 문자형, 숫자형 키를 생성합니다.

❼ 새로운 Map 객체 friends를 생성한 후 ❻에서 생성한 키를 통해 데이터를 입력합니다.

❽ forEach( )를 사용해서 Map에 저장된 모든 값을 키값 형태로 출력합니다.


만약 템플릿 리터럴로 출력을 시도하면 문자열로 자동 변환되어 [object]와 같이 출력되므로 주의해주세요.

문법 16 Set

Set은 중복되지 않는 유일한 값만 저장할 수 있는 자료구조입니다. 기존의 배열(Array)은 중복된 값을 허용하지만 Set은 자동으로 중복을 제거하며 삽입 순서를 유지합니다. Set은 다음과 같은 상황에서 유용하게 사용합니다.

- 배열에서 중복을 제거하고 싶을 때
- 특정 값의 존재 여부를 빠르게 확인할 때
- 유일한 요소들의 리스트를 관리하고자 할 때

이제 Set의 기본 사용법과 특징을 살펴보겠습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/11Set.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Set</title>
</head>
<body>
  <h2>Set</h2>
  <script>
    /* ---------------- 기본 사용법 ---------------- */
    (() => {
      console.error('기본사용법');
      const numbers = new Set(); // ❶ Set 객체 numbers 생성
      numbers.add(1);  // ❷ add: 정수 추가(중복 허용 안 됨)
      numbers.add(2);
      numbers.add(2); // 중복 값은 무시
      numbers.add(3);
      console.log('결과 :', numbers);  // ❸ 저장 결과 : {1, 2, 3} → 크기는 3
      console.log('크기:', numbers.size);
      numbers.forEach((value) => { // ❹ forEach로 값 순차 출력
        console.log('value:', value);
      });
    })();
```

```
    /* ---------------- 다양한 메서드 ---------------- */
    (() => {
      console.error('다양한 함수');
      const frontEnd = new Set(['html', 'css', 'js', 'react']); // 초기화된 Set
      // ❺ delete: 요소 삭제
      console.log('delete()-개별삭제:', frontEnd.delete('css'));
      console.log('has()-존재확인:', frontEnd.has('css'));   // has: 존재 여부
      console.log('clear()-전체삭제:', frontEnd.clear());    // clear: 모두 삭제
    })();

    /* ---------------- 개별 요소 접근 ---------------- */
    (() => {
      console.error('개별 요소 접근');
      const dragonBall = new Set(['손오공', '베지터', '프리저']); // ❻ 새로운 Set 생성
      const data1 = dragonBall.values().next().value; // ❼ iterator로 첫 요소 추출
      console.log('요소1', data1);
      const dragonArr = Array.from(dragonBall); // ❽ Array.from: 배열로 변환
      console.log('요소2,3', dragonArr[1], dragonArr[2]); // 인덱스로 간편 접근
    })();
  </script>
</body>
</html>
```

Set

❶ 값만 저장하고 중복은 자동으로 제거하는 Set 객체 numbers를 생성합니다.

❷ add( ) 함수로 정수를 추가합니다. Set은 중복을 허용하지 않으므로 두 번째로 입력되는 2는 저장되지 않습니다.

❸ 결과에는 1,2,3만 저장되며, 크기 3을 출력합니다.

❹ forEach로 Set에 저장된 값을 순서대로 하나씩 출력합니다.

❺ Set 객체가 제공하는 함수를 사용합니다. 먼저 새로운 Set 객체 frontEnd를 생성한 후 초기화합니다.

- **delete(value)** : 데이터를 삭제한 후 성공 여부를 true/false로 반환합니다.
- **has(value)** : Set에 해당 데이터가 존재하는지 확인하여 true/false를 반환합니다.
- **clear(value)** : 모든 요소를 삭제합니다.

❻ 새로운 Set 객체 {'손오공','베지터','프리저'}를 생성합니다.

❼ Set은 반복 가능한 객체이지만, 배열처럼 인덱스를 사용해 요소에 직접 접근할 수는 없습니다. 따라서 개별 요소를 가져오려면 .values( ) 메서드를 사용해 반복자를 얻은 뒤 .next( ).value처럼 다소 번거로운 방식을 사용해야 합니다.

❽ 앞에서 학습한 Array.from( )을 사용하면 배열로 변환할 수 있으므로, 인덱스를 통해 간단히 개별 요소에 접근할 수 있습니다.

문법 17 프라미스

리액트를 제대로 활용하려면 비동기 처리에 대한 이해가 필요합니다. 그 핵심에는 프라미스Promise가 있죠. 프라미스는 비동기 작업의 '결과'를 표현하는 객체로, 복잡한 비동기 흐름을 보다 직선적이고 읽기 쉬운 코드로 바꿔주어 코드의 가독성과 유지보수성을 크게 향상시켜줍니다.

예전에는 콜백 함수를 반복 중첩하여 사용하다 보니 코드가 복잡해지는 콜백 지옥callback hell 문제가 있었습니다. 프라미스는 이런 문제를 해결하기 위해 등장했는데요, 특히 리액트에서는 API 호출, 데이터 페칭, 상태 업데이트 등 다양한 비동기 작업에서 프라미스의 활용이 매우 중요합니다. 프라미스는 다음과 같은 세 가지 상태를 가집니다.

- **대기(Pending)** : 아직 결과가 결정되지 않은 상태
- **이행(Fulfilled)** : 작업이 성공적으로 완료된 상태(resolve)
- **거부(Rejected)** : 작업이 실패한 상태(reject)

이제 프라미스의 동작 방식과 활용 방법을 알아보겠습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/12promise.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Promise</title>
</head>
<body>
  <h2>Promise</h2>
  <script>
    /* ---------------------- 첫 번째 프라미스 ---------------------- */
    // ❶ 콜백에 resolve/reject 자동 전달
    const myPromise1 = new Promise((resolve, reject) => {
      const success = true;  // ❷ 작업 성공 여부(테스트용 플래그)
      if (success) { // ❸ 성공 → resolve, 실패 → reject
        resolve('작업 성공');
      } else {
        reject('작업 실패');
      }
    });

    const func1 = () => { // ❹ myPromise1 실행
      myPromise1
        .then(result => {  // ❺ 이행(Fulfilled) 시 결과 출력
          console.log(result);
        })
        .catch(error => {  // ❻ 거부(Rejected) 시 에러 출력
          console.error(error);
        });
    };
```

```
    /* ---------------------- 두 번째 프라미스 ---------------------- */
    function myPromise2() {  // ❼ 3초 지연 후 resolve
      return new Promise((resolve) => {
        setTimeout(() => {
          resolve('3초 후 완료된 작업');
        }, 3000);
      });
    }

    const func2 = () => {  // ❽ myPromise2 실행
      myPromise2().then(result => {
        console.log(result);
      });
    };
  </script>

  <div>
    <button type="button" onclick="func1();">버튼1</button>
    <button type="button" onclick="func2();">버튼2</button>
  </div>
</body>
</html>
```

❶ 첫 번째 프로미스 객체를 생성합니다. 콜백 함수에는 두 개의 매개변수 resolve와 reject가 자동으로 전달되며, 각각 비동기 작업이 성공하거나 실패했을 때 호출합니다.

❷ 작업의 성공 여부를 true로 설정하고 테스트합니다(false로 설정을 변경하고 테스트해보세요).

❸ 성공인 경우 resolve를 호출하고, 실패인 경우 reject를 호출합니다.

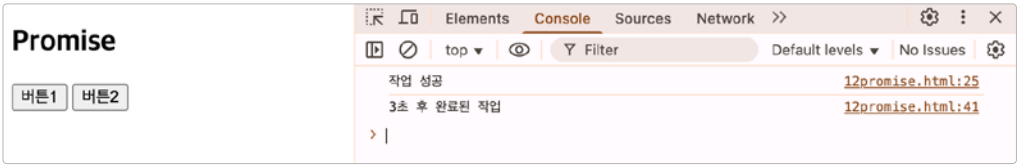
❹ func1() 버튼을 누르면 myPromise1을 실행됩니다.

❺ 작업에 성공하면 resolve가 호출되어 프라미스는 이행(Fulfilled) 상태가 됩니다. 이때 resolve에 전달된 값은 then 블록의 매개변수로 전달됩니다. 따라서 콘솔에 '작업 성공'을 출력합니다.

❻ 작업에 실패하면 reject가 호출되어 프라미스는 거부(Rejected) 상태가 됩니다. 이때 reject에 전달된 오류 정보는 catch 블록의 매개변수로 전달됩니다. 이때는 '작업 실패'가 콘솔에 출력됩니다.

❼ 두 번째 프로미스는 setTimeout( ) 함수를 사용해 3초 후 작업이 완료되는 형태로 정의되어 있습니다. 이처럼 약간의 지연을 주는 방식은, 실제 서버와 통신할 때 발생하는 응답 지연(딜레이)을 흉내 낼 때 자주 쓰는 패턴입니다.

❽ myPromise2를 실행하면 3초 후 resolve가 호출되며, 이 결과는 then 블록으로 전달되어 콘솔에 출력됩니다.



문법 18 async/await

async/await는 자바스크립트에서 프라미스를 더욱 쉽고 효과적으로 다루기 위한 '문법적 설탕syntactic sugar'입니다. 복잡한 .then( )과 .catch( )로 이어지는 프라미스 체인 대신, 마치 동기 코드처럼 비동기 작업을 순서대로 작성할 수 있습니다. 이를 통해 코드의 가독성을 높이고, try-catch문을 활용하여 비동기 작업 중 발생하는 오류를 보다 명확하게 처리할 수 있습니다.

리액트에서는 API 호출이나 데이터 처리처럼 비동기 작업이 자주 등장하기 때문에, async/await 문법을 익혀두면 더욱 효율적으로 컴포넌트를 구성할 수 있습니다. 이번에는 async/await을 사용하여 비동기 작업을 순차적으로 처리하는 방법을 살펴봅니다. 다음은 사용자 로그인을 위한 간단한 시나리오로, async/await를 사용해 1초씩 지연되는 작업을 '순차적으로 처리하는 방법'을 보여줍니다.

- **로그인 요청** : 서버에 사용자 아이디를 보내 로그인합니다.
- **회원 정보 조회** : 로그인 성공하면 해당 사용자의 회원 정보를 서버에서 조회합니다.
- **환영 메시지 출력** : 조회된 회원 정보를 바탕으로 환영 메시지를 콘솔에 출력합니다.

여기서 중요한 점은 각 작업이 비동기로 처리되지만, 순차적으로 실행되어야 한다는 겁니다. 로그인이 되지도 않았는데 환영 메시지가 출력되는 건 논리적으로 맞지 않기 때문입니다. 이제 async와 await를 사용하여 이 비동기적인 순차 처리 과정을 어떻게 구현하는지 예제 코드를 통해 자세

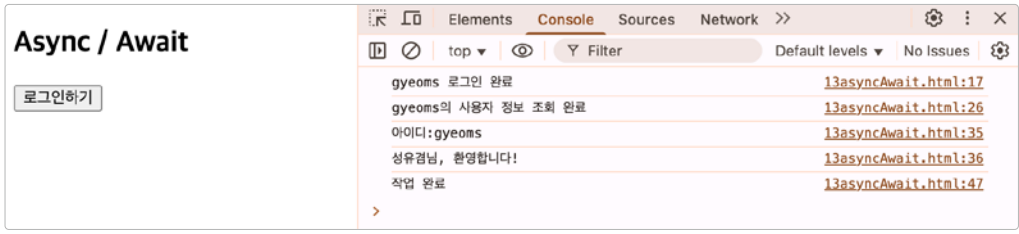
히 살펴보겠습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

App.js02modernJS/13asyncAwait.htmlx

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Async/Await</title>
</head>
<body>
  <h2>Async / Await</h2>
  <script>
    /* ------------------------------- 로그인 요청 ------------------------------ */
    function requestLogin(userid) {  // ❶ 1초 지연 후 로그인 완료를 시뮬레이션
      return new Promise((resolve) => {
        setTimeout(() => {
          console.log(`${userid} 로그인 완료`);
          resolve(userid);  // ❷ 성공 시 아이디를 resolve로 전달
        }, 1000);
      });
    }

    /* ------------------------------ 사용자 정보 조회 ------------------------- */
    function getUserInfo(userid) { // ❸ 아이디로 사용자 정보 조회(1초 지연)
      return new Promise((resolve) => {
        setTimeout(() => {
          console.log(`${userid}의 사용자 정보 조회 완료`);
          resolve({ userid, name: '성유겸' });  // ❹ 조회 결과 객체 반환
        }, 1000);
      });
    }

    /* ------------------------------ 환영 메시지 출력 ------------------------- */
    function welcomeHome(responseInfo) { // ❺ 조회된 사용자 정보를 받아 환영 출력
      return new Promise((resolve) => {
        setTimeout(() => {
          // ❻ 아이디·이름을 콘솔에 표시
          console.log(`아이디: ${responseInfo.userid}`);
```

```
        console.log(`${responseInfo.name}님, 환영합니다!`);
        resolve();
      }, 1000);
    });
  }

  /* ------------------------- 순차 실행 컨트롤러 ------------------------- */
  async function processMain() { // ❼ async 함수: 비동기 작업을 순차 실행
    try { // ❽ await로 각 단계가 끝날 때까지 대기
      const my_id   = await requestLogin('gyeoms');
      const my_info = await getUserInfo(my_id);
      await welcomeHome(my_info);
      console.log('작업 완료');
    } catch (error) {  // ❾ 오류 발생 시 처리
      console.error('오류 발생:', error);
    }
  }
</script>

<!-- ❿ 버튼 클릭 시 전체 프로세스 실행 -->
<div>
  <button type="button" onclick="processMain();">로그인하기</button>
</div>
</body>
</html>
```



❶ 로그인을 요청하는 함수입니다. Promise 객체를 반환하며, 내부 콜백 함수에서는 로그인을 처리하는 로직이 들어 있습니다. 처리 시간은 1초가 걸린다고 가정합니다.

❷ 로그인이 성공하면 resolve( ) 함수를 호출하여, 로그인에 사용된 아이디를 반환합니다.

❸ 전달받은 아이디를 이용하여 사용자 정보를 조회하는 함수입니다. 마찬가지로 1초 뒤 Promise

객체를 반환합니다.

❹ 조회가 완료되면 아이디와 이름을 포함한 객체를 반환합니다.

❺ 앞서 조회한 사용자 정보를 객체 형태로 받아 환영 메시지를 출력하기 위한 함수입니다.

❻ 사용자 정보에서 아이디와 이름을 파싱한 후 환영하는 메시지와 함께 콘솔에 출력합니다.

❼ 앞서 정의한 세 개의 비동기 함수를 순차적으로 실행하기 위한 함수입니다. processMain( ) 함수 선언 앞에 async를 사용하여 비동기 함수로 정의했습니다.

❽ 각 작업을 실행할 때는 await를 사용하여, 해당 비동기 작업이 완료될 때까지 기다린 후 다음 작업을 실행합니다. 이를 통해 비동기 작업 간의 순서를 제어할 수 있으며, 로그인 → 정보 조회 → 환영 출력 순서로 깔끔하게 이어집니다.

❾ 실행 중 오류가 발생할 경우, catch 블록이 실행되어 오류 메시지를 출력하게 됩니다. try-catch문을 통해 예외 상황을 안정적으로 처리할 수 있습니다.

❿ 로그인하기 버튼을 누르면 위 과정이 차례로 실행되고, 마지막에 '작업 완료'가 표시됩니다.

이처럼 async/await을 사용하면 비동기 작업을 마치 동기 작업처럼 순차적으로 처리할 수 있습니다. 이를 통해 코드의 흐름을 직관적으로 이해할 수 있으며, 가독성이 높아질 뿐만 아니라 유지보수 또한 훨씬 수월해집니다.

ES6의 객체 리터럴 문법 개선 방법을 알아봐요

ES6부터는 객체를 더욱 간결하게 정의할 수 있는 편리한 문법을 제공합니다. 변수와 속성 이름이 같은 경우, ':'의 오른쪽 값 즉, 속성값을 명시적으로 작성하는 것을 생략할 수 있습니다. 덕분에 객체를 선언할 때 중복 입력을 줄여 코드가 짧고 읽기 쉬워집니다. 어떻게 다른지 다음 코드 예를 통해 확인해봅시다.

기존 방식	ES6의 방식

`const user1 = {`
`  id: id,`
`  name: name`
`};`
	`const user2 = {`
	`  id,`
	`  name`
	`};`

❹ 항목에서 사용자 정보를 반환하는 코드에서 사용되었습니다.

문법 19 fetch

앞서 자바스크립트에서 비동기 처리를 다루기 위한 프라미스와 async/await 문법을 살펴보았습니다. 이제 실제로 데이터를 주고받는 상황에서 활용하는 방법을 알아볼 차례입니다. 웹 애플리케이션은 사용자와 서버 사이에서 끊임없이 데이터를 주고받습니다. 예를 들어 로그인 요청이나, 외부 API로부터 정보를 불러오기, 사용자 입력을 서버에 저장하는 작업 등은 모두 서버와의 비동기 통신을 통해 이루어집니다.

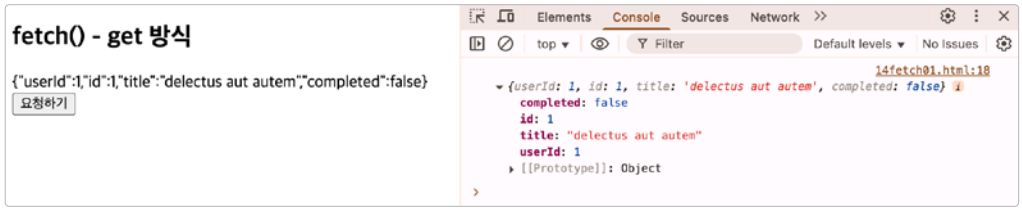
이번 예제에서는 자바스크립트에서 가장 널리 쓰이는 HTTP 요청 도구인 Fetch API에 대해 살펴보겠습니다. fetch() 함수는 간결한 문법으로 네트워크 요청을 처리하며, 프라미스 기반이기 때문에 async/await과도 자연스럽게 어우러집니다. 실습에는 외부 서버와의 통신을 위해 REST API 서비스인 JSONPlaceholder 사이트를 사용합니다. 이 사이트는 테스트 및 프로토타입 제작을 위한 가상의 REST API를 무료로 제공하므로, 실제 백엔드 서버 없이도 데이터를 가져오거나 생성, 수정, 삭제하는 API 요청을 자유롭게 테스트할 수 있습니다. 먼저 GET 방식의 요청을 처리하는 예제를 살펴보겠습니다.

01 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/14fetch01.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>fetch API</title>
</head>
<body>
  <h2>fetch() - GET 방식</h2>
  <script>
    // ❶ 비동기 통신 함수 : async 덕분에 내부에서 await 사용 가능
    async function reqeustData() {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/
todos/1'); // ❷ GET 요청을 보내고 응답이 올 때까지 대기
        const data = await response.json();  // ❸ 응답 본문을 JSON으로 파싱
        console.log(data);
        // document.getElementById('callBackMsg1').innerText = data;
```

```
// ❹ 객체 그대로 넣으면 [object Object]로 표시됨
        document.getElementById('callBackMsg').innerText = JSON.stringify(data);
// ❺ JSON 객체를 문자열로 변환 후 화면에 출력
      } catch (error) {  // ❻ 통신·파싱 오류가 발생하면 여기서 처리
        console.error('fetch 통신 오류:', error);
      }
    }
  </script>

  <div id="callBackMsg"></div>
  <div>
    <button onclick="reqeustData();">요청하기</button>
  </div>
</body>
</html>
```



❶ 비동기 통신을 처리하는 함수 async function reqeustData()를 정의합니다. async를 붙이면 함수 안에서 await를 사용할 수 있어 코드 흐름을 동기 형태로 표현할 수 있습니다.

❷ fetch() 함수를 사용해 외부 서버로 GET 요청을 보냅니다. await이 있으므로 요청 처리가 완료될 때까지 기다린 후 반환값을 받습니다.

❸ await를 붙여 응답받은 데이터를 JSON 형식으로 파싱한 후 콘솔에 출력하여 확인합니다.

❹ JSON 데이터를 객체 그대로 innerText에 넣을 경우 문자열이 아니므로 [object Object]로 표시됩니다.

❺ JSON.stringify() 함수로 JSON 객체를 문자열로 변환한 후, HTML 엘리먼트에 표시하여 사람이 읽을 수 있는 형태로 만들어줍니다.

❻ fetch 요청 중 오류가 발생하면 try-catch 블록이 실행됩니다. 오류가 나면 'fetch 통신 오류:'

와 함께 상세 메시지가 콘솔에 보이며, 보통 네트워크 문제나 응답 파싱 실패 등이 이유일 수 있습니다.

02 다음은 POST 방식의 요청을 처리하는 예제를 보겠습니다. 파일을 생성하고 다음과 같이 코드를 작성하여 실습하겠습니다.

02modernJS/14fetch02.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>fetch API</title>
</head>
<body>
  <h2>fetch() - POST 방식</h2>
  <script>
    // ❶ POST 요청을 실행하는 함수. 버튼 클릭 시 호출
    const reqeustData = () => {
      fetch('https://jsonplaceholder.typicode.com/posts', {
        method: 'POST', // ❷ 옵션 객체의 method를 'POST'로 지정
        body: JSON.stringify({ // ❸ 전송 데이터를 객체로 작성 후 문자열화
          title: 'React',
          body: '최고의 프론트엔드 라이브러리',
          userId: 1,
        }),
        headers: { // ❹ JSON 데이터를 전송한다는 사실을 명시
          'Content-type': 'application/json; charset=UTF-8',
        },
      })
      .then((response) => response.json()) // ❺ 응답을 JSON으로 변환(비동기)
      .then((json) => { // ❻ 변환된 JSON을 받아 처리
        console.log(json);
        document.getElementById('callBackMsg').innerText =
          JSON.stringify(json);
      });
    };
  </script>

  <div id="callBackMsg"></div>
```

```
  <div>
    <button onclick="reqeustData();">요청하기</button>
  </div>
</body>
</html>
```

❶ reqeustData( ) 함수는 POST 요청을 실행하기 위해 정의한 함수입니다. 실행을 위해 [요청하기] 버튼을 누르면 실행됩니다.

❷ fetch( ) 함수의 두 번째 인자로 전달된 객체에서 method 속성을 'POST'로 지정해 새로운 데이터를 서버에 등록합니다.

❸ 서버로 전송할 데이터를 JSON 형식의 객체로 작성한 뒤, JSON.stringify( )를 사용하여 문자열로 변환합니다. 서버는 일반적으로 문자열 형태의 JSON 데이터를 받기 때문에 이 과정이 필요합니다.

❹ 요청 헤더에서 Content-type을 'application/json; charset=UTF-8'로 설정합니다. 이를 통해 서버에 JSON 형식의 데이터를 전송합니다. 사실을 알려줍니다.

❺ 서버로 POST 요청을 보낸 뒤, 첫 번째 then( ) 절에서는 응답을 받아 JSON 형식으로 변환합니다. 이때 response.json( )은 비동기 작업이므로, 다음 then( ) 절로 결과를 반환합니다.

❻ 두 번째 then( ) 절에서는 앞에서 반환된 JSON 데이터를 매개변수로 받아 처리합니다. 콘솔에 출력하여 응답 내용을 확인하고, 동시에 이를 HTML 문서 내에 표시합니다.