

이터러블

이터레이션 프로토콜

- Iteration Protocol은 순회 가능한(iterable) 데이터 컬렉션을 만들기 위해 ECMAScript 사양에 정의하여 미리 약속된 규칙이다.
- ES6 이전의 순회 가능한 데이터 컬렉션(배열, 문자열, 유사 배열 객체)등은 통일된 규칙이 없이 각자 나름의 구조를 가지고 다양한 방법(for, for ... in, forEach)으로 순회할 수 있었다.
- ES6에서는 순회 가능한 데이터 컬렉션을 이터레이션 프로토콜을 준수하는 이터러블로 통일하여 **for ... of**, **스프레드 문법**, **배열 디스트럭처링 할당**의 대상으로 사용할 수 있도록 하였다
- 이터레이션 프로토콜에는 **이터러블 프로토콜**과 **이터레이터 프로토콜**이 있다.

이터러블 프로토콜

- Well Known Symbol인 Symbol.iterator를 프로퍼티 키로 사용한 메서드를 직접 구현하거나, 프로토타입 체인을 통해 상속 받은 Symbol.iterator 메서드를 호출하면 이터레이터 프로토콜을 준수한 이터레이터를 반환한다.
- 이러한 규약을 이터러블 프로토콜이라고 하며, 이터러블 프로토콜을 준수한 객체를 이터러블이라고 한다.

이터레이터 프로토콜

- 이터러블의 `Symbol.iterator` 메서드를 호출하면 이터레이터 프로토콜을 준수한 이터레이터를 반환한다.
- 이터레이터는 `next` 메서드를 소유하며, 이를 호출하면 이터러블을 순회하여 `value`와 `done` 프로퍼티를 갖는 이터레이터 리절트 객체를 반환한다. 이러한 규약을 이터레이터 프로토콜이라 하며, 이터레이터 프로토콜을 준수한 객체를 이터레이터라고 한다.
- 이터레이터는 이터러블의 요소를 탐색하기 위한 포인터 역할을 한다.

- **이터러블** 객체는
 - Symbol.iterator를 키로 가지는 프로퍼티(메서드)가 있다
 - 해당 메서드를 호출하면 **이터레이터**를 반환한다
- **이터레이터** 객체는
 - next 메서드를 가진다.
 - next 메서드를 호출하면 **이터레이터 리절트** 객체를 반환한다
- **이터레이터 리절트** 객체는
 - Value 값과 done 값을 가진다.

이터러블

- 이터러블 프로토콜을 준수한 객체
- 이터러블인지는 다음과 같이 확인할 수 있다.



```
const isIterable = v => v !== null && typeof v[Symbol.iterator] === 'function'
```

```
// 배열, 문자열, Map, Set 등은 이터러블이다.
```

```
isIterable([]); // true
```

```
isIterable(''); // true
```


```
isIterable(new Map()); // true
```

```
isIterable(new Set()); // true
```

```
isIterable({}); // true
```

이터러블

- 이터러블은 for ... of 문으로 순회할 수 있다
- 스프레드 문법의 대상으로 사용할 수 있다
- 배열 디스트럭처링 할당의 대상으로 사용할 수 있다



```
const array = [1, 2, 3];

for (const item of array) {
  console.log(item);
}

console.log(...array) // 1, 2, 3

const [a, ...rest] = array;
console.log(a, rest) // 1, [2, 3]
```

배열 디스트럭처링 할당

- ES2018 에서는 스프레드 연산자를 객체에서도 사용할 수 있도록 하였다.
- 이를 통해 객체의 얇은 복사나, 병합이 간단해졌다.

```
let obj1 = { foo: 'bar', x: 42 };
let obj2 = { foo: 'baz', y: 13 };

let clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }

let mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```


이터레이터

- 이터러블의 Symbol.iterator 메서드를 호출하면 이터레이터 프로토콜을 준수한 이터레이터를 반환한다.
- 이터레이터는 next 메서드를 갖는다.
 - next 메서드는 이터러블의 각 요소를 순회하기 위한 포인터의 역할을 한다.
- next 메서드를 호출하면 이터러블을 순차적으로 한 단계씩 순회하며 순회 결과를 나타내는 이터레이터 리절트 객체를 반환한다.



```
const array = [1, 2, 3];

const iterator = array[Symbol.iterator]();

console.log(iterator.next()); // {value: 1, done: false};
console.log(iterator.next()); // {value: 2, done: false};
console.log(iterator.next()); // {value: 3, done: false};
console.log(iterator.next()); // {value: undefined, done: true};
```

for ... in VS for ... of

- for ... in
 - 객체의 프로토타입 체인 상에 존재하는 모든 프로토타입의 프로퍼티 중에서 프로퍼티 어트리뷰트 `[[Enumerable]]`의 값이 `true`인 프로퍼티를 순회하며 열거한다.
 - 키가 `Symbol`인 프로퍼티는 열거하지 않는다.
- for ... of
 - 내부적으로 이터레이터의 `next` 메서드를 호출하며 반환하는 리절트 객체의 `value` 프로퍼티 값을 `for ... of` 문의 변수에 할당한다.

유사 배열 객체

- 유사 배열 객체는 마치 배열처럼 인덱스로 프로퍼티 값에 접근할 수 있고 length 프로퍼티를 갖는 객체를 말한다
- 유사 배열 객체는 iterable이 아니다. 따라서 for ... of문으로 순회할 수 없다.
- 단 arguments, NodeList, HTMLCollection은 유사 배열 객체이면서 이터러블이다.
- Array.from 메서드를 사용하여 유사 배열 객체 또는 이터러블을 배열로 변환할 수 있다.

이터레이션 프로토콜의 필요성

- ES6 이전에는 순회 가능한 데이터 컬렉션은 통일된 규약 없이 각자 나름의 규칙을 가지고 다양한 방법으로 순회할 수 있었다.
- 이렇게 다양한 데이터 공급자가 각자의 순회 방식을 갖는다면 데이터 소비자는 다양한 데이터 공급자의 순회 방식을 모두 지원해야 한다.
- 하지만 데이터 공급자가 이터레이션 프로토콜을 준수하도록 규정한다면 소비자는 이터레이션 프로토콜만 지원하도록 구현하면 된다.
- 이처럼 이터레이션 프로토콜은 데이터 공급자와 소비자를 연결하는 인터페이스 역할을 한다.

사용자 이터러블 구현

```
const fibonacci = {
  // Symbol.iterator 메서드를 구현하여 이터러블 프로토콜을 준수한다.
  [Symbol.iterator]() {
    let [pre, cur] = [0, 1]
    const max = 10;

    return {
      // Symbol.iterator 메서드는 next 메서드를 소유한 이터레이터를 반환해야 하고
      // next 메서드는 이터레이터 리절트 객체를 반환해야 한다
      next() {
        [pre, cur] = [cur, pre + cur];

        // Iterator Result Object
        return {value: cur, done: cur >= max}
      }
    }
  },
}

for (const num of fibonacci) {
  console.log(num); // 1 2 3 5 8
}
```

이터러블을 생성하는 함수

```
const fibonacciFunc = function (max) {  
  let [pre, cur] = [0, 1]  
  
  return {  
    [Symbol.iterator]() {  
      return {  
        next() {  
          [pre, cur] = [cur, pre + cur];  
  
          return {value: cur, done: cur >= max}  
        }  
      }  
    }  
  }  
}  
  
for (const num of fibonacci(10)) {  
  console.log(num); // 1 2 3 5 8  
}
```

이터러블이면서 이터레이터인 개체를 생성하는 함수

```
const fibonacciFunc = function (max) {  
  let [pre, cur] = [0, 1]  
  
  return {  
    [Symbol.iterator]() { return this; },  
  
    next() {  
      [pre, cur] = [cur, pre + cur];  
      return {value: cur, done: cur >= max}  
    }  
  }  
}  
  
let iter = fibonacciFunc(10);  
  
// iter는 이터러블이다.  
for (const num of iter) {  
  console.log(num); // 1 2 3 5 8  
}  
  
let iter = fibonacciFunc(10);  
  
// iter는 이터레이터이다.  
console.log(iter.next()); // {value: 1, done: false}  
console.log(iter.next()); // {value: 2, done: false}  
console.log(iter.next()); // {value: 3, done: false}  
console.log(iter.next()); // {value: 5, done: false}
```

무한 이터러블과 지연 평가

```
const fibonacciFunc = function() {  
  let [pre, cur] = [0, 1]  
  
  return {  
    [Symbol.iterator]() { return this; },  
  
    next() {  
      [pre, cur] = [cur, pre + cur];  
      return {value: cur}  
    }  
  }  
}  
  
let [f1, f2, f3] = fibonacciFunc();  
console.log(f1, f2, f3); // 1 2 3
```

- 지연 평가를 통해 데이터를 생성한다. 즉, 데이터가 필요한 시점 이전까지는 데이터를 생성하지 않다가 데이터가 필요한 시점이 되면 그때야 비로소 데이터를 생성한다.
- 이렇게 지연 평가를 사용하면 불필요한 데이터를 미리 생성하지 않아서 메모리를 소비하지 않고, 무한도 표현할 수 있다는 장점이 있다.