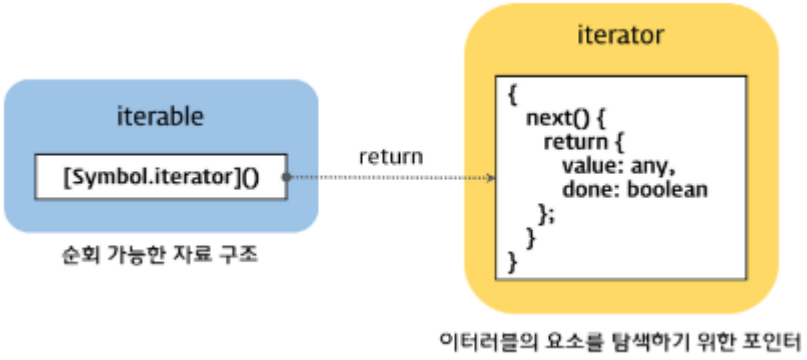
이터레이터 프로토콜

ES6에서 도입된 이터레이션 프로토콜은 순회 가능한 데이터 컬렉션을 만들기 위해 ECMAScript사양에 정의하여 미리 약속한 규칙이다.

- 이터러블의 Symbol.iterator 메서드를 호출하면 이터레이터 프로토콜을 준수한 이터레이터를 반환한다.
- 이터레이터는 next메서드를 소유하며 **next메서드를 호출하면 이터러블을 순회하며 value와 done프로퍼티를 갖는 이터레이터 리절트 객체를 반환**한다.
- 이터레이터는 이터러블의 요소를 탐색하기 위한 포인터 역할을 한다.




```
// 배열은 이터러블 프로토콜을 준수한 이터러블이다.
const array = [1, 2, 3];

// Symbol.iterator 메서드는 이터레이터를 반환한다. 이터레이터는 next 메서드를 갖는다.
const iterator = array[Symbol.iterator]();

// next 메서드를 호출하면 이터러블을 순회하며 순회 결과를 나타내는 이터레이터 리절트 객체를
// 반환한다. 이터레이터 리절트 객체는 value와 done 프로퍼티를 갖는 객체다.
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 ... of문

이터러블을 순회하면서 이터러블 요소를 변수에 할당한다.

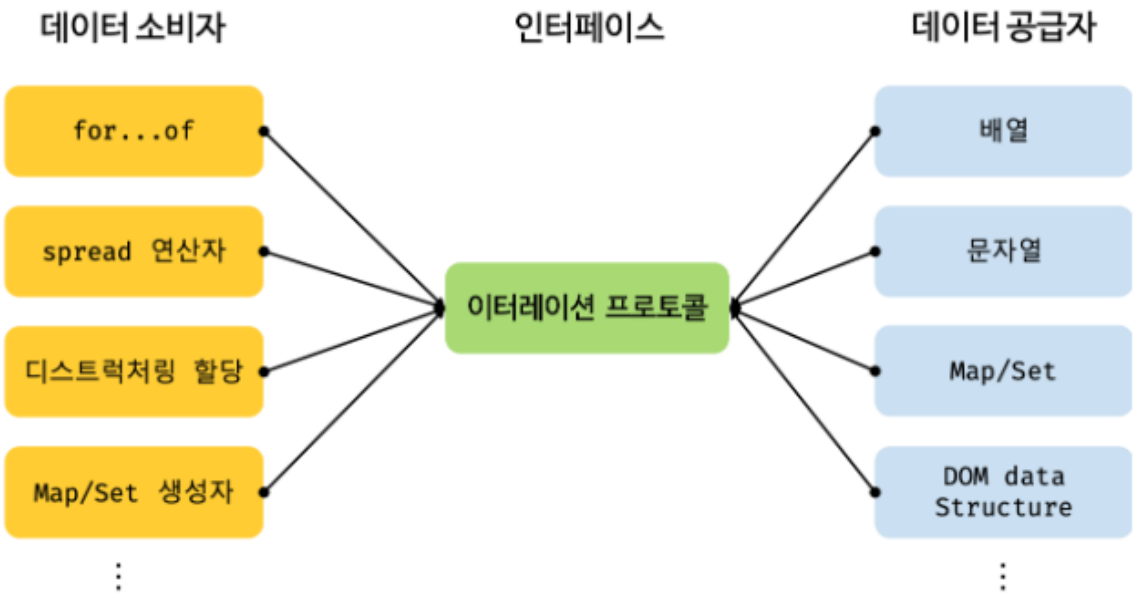
```
for (const item of [1, 2, 3]) {
  // item 변수에 순차적으로 1, 2, 3이 할당된다.
  console.log(item); // 1 2 3
}


// 유사 배열 객체는 이터러블이 아니기 때문에 for...of 문으로 순회할 수
없다
const arrayLike = {
  0: 1,
  1: 2,
  2: 3,
  length: 3
};

// Array.from은 유사 배열 객체 또는 이터러블을 배열로 변환한다
const arr = Array.from(arrayLike);
console.log(arr); // [1, 2, 3]

for (const i of arrayLike) {
  console.log(i); // 1 2 3
}
```

이터레이션 프로토콜은 데이터 소비자가 효율적으로 다양한 데이터 공급자를 사용할 수 있도록 **데이터 소비자와 데이터 공급자를 연결하는 인터페이스의 역할**을 한다



이터러블은 데이터 소비자와 데이터 소스를 연결하는 인터페이스

배열 디스트럭처링 할당

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

// ES6 배열 디스트럭처링 할당
// 변수 one, two, three를 선언하고 배열 arr을 디스트럭처링하여 할당한다.
// 이때 할당 기준은 배열의 인덱스다.
const [one, two, three] = arr;

console.log(one, two, three); // 1 2 3
```

```
// 피보나치 수열을 구현한 사용자 정의 이터러블
const fibonacci = function (max) {
    let [pre, cur] = [0, 1]; // 배열 디스트럭처링 할당

    // Symbol.iterator 메서드와 next 메서드를 소유한 이터러블이면서 이터레이터인 객체를 반환
```

```
    return {
      // Symbol.iterator 메서드를 구현하여 이터러블 프로토콜을 준수
한다.
      [Symbol.iterator]() { return this; }
        // next 메서드는 이터레이터 리절트 객체를 반환해야 한다.
      next() {
        [pre, cur] = [cur, pre + cur]; //pre에는 cur값을 cur
에는 pre+cur값을 할당
        // 이터레이터 리절트 객체를 반환한다.
        return { value: cur, done: cur >= max };
      }
    }
  }
}

// iter는 이터러블이면서 이터레이터다.
let iter = fibonacci(10);

// iter는 이터러블이므로 for...of 문으로 순회할 수 있다.
for (const num of iter) {
  console.log(num); // 1 2 3 5 8
}

// iter는 이터러블이면서 이터레이터다
iter = fibonacci(10);
// iter는 이터레이터이므로 이터레이션 리절트 객체를 반환하는 next 메서
드를 소유한다.
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 }
console.log(iter.next()); // { value: 8, done: false }
console.log(iter.next()); // { value: 13, done: true }
```

▼ 참고

무한 이터러블과 지연 평가

무한 수열 구현

```
// 무한 이터러블을 생성하는 함수
const fibonacciFunc2 = function () {
  let [pre, cur] = [0, 1];

  return {
    [Symbol.iterator]() { return this; },
    next() {
      [pre, cur] = [cur, pre + cur];
      // 무한을 구현해야 하므로 done 프로퍼티를 생략한다.
      return { value: cur };
    }
  }
};

// fibonacciFunc 함수는 무한 이터러블을 생성한다.
for (const num of fibonacciFunc2()) {
  if (num > 10000) break;
  console.log(num); // 1 2 3 5 8...4181 6765
}

// 배열 디스트럭처링 할당을 통해 무한 이터러블에서 3개의 요소만 취득한다.
const [f1, f2, f3] = fibonacciFunc2();
console.log(f1, f2, f3); // 1 2 3
```