제너레이터와 async/await

제너레이터란?

ES6에서 도입된 제너레이터(Generator) 함수는 이터러블을 생성하는 함수이다. 제너레이터 함수를 사용하면 이터레이션 프로토콜을 준수해 이터러블을 생성하는 방식보다 간편하게 이터러블을 구현할 수 있다. 또한 제너레이터 함수는 비동기 처리에 유용하게 사용된다.

```
// 이터레이션 프로토콜을 구현하여 무한 이터러블을 생성하는 함수
const createInfinityByIteration = function () {
  let i = 0; // 자유 변수
  return {
    [Symbol.iterator]() { return this; },
    next() {
      return { value: ++i };
    }
  };
};

for (const n of createInfinityByIteration()) {
  if (n > 5) break;
  console.log(n); // 1 2 3 4 5
}

// 무한 이터러블을 생성하는 제너레이터 함수
function* createInfinityByGenerator() {
  let i = 0;
  while (true) { yield ++i; }
}

for (const n of createInfinityByGenerator()) {
  if (n > 5) break;
  console.log(n); // 1 2 3 4 5
}
```

제너레이터 함수는 일반 함수와는 다른 독특한 동작을 한다. 제너레이터 함수는 일반 함수와 같이 함수의 코드 블록을 한 번에 실행하지 않고 함수 코드 블록의 실행을 일시 중지했다가 필요한 시점에 재시작할 수 있는 특수한 함수이다.

```
function* counter() {
  console.log('첫번째 호출');
  yield 1;                    // 첫번째 호출 시에 이 지점까지 실행된
다.
  console.log('두번째 호출');
  yield 2;                    // 두번째 호출 시에 이 지점까지 실행된
다.
  console.log('세번째 호출');  // 세번째 호출 시에 이 지점까지 실행
된다.
}

const generatorObj = counter();

console.log(generatorObj.next()); // 첫번째 호출 {value: 1, d
one: false}
console.log(generatorObj.next()); // 두번째 호출 {value: 2, d
one: false}
console.log(generatorObj.next()); // 세번째 호출 {value: unde
fined, done: true}
```

일반 함수를 호출하면 return 문으로 반환값을 리턴하지만 제너레이터 함수를 호출하면 제너레이터를 반환한다. 이 **제너레이터는 이터러블(iterable)이면서 동시에 이터레이터(iterator)인 객체이다.** 다시 말해 제너레이터 함수가 생성한 제너레이터는 Symbol.iterator 메소드를 소유한 이터러블이다. 그리고 제너레이터는 next 메소드를 소유하며 next 메소드를 호출하면 value, done 프로퍼티를 갖는 이터레이터 리절트 객체를 반환하는 이터레이터이다.

```
// 제너레이터 함수 정의
function* counter() {
  for (const v of [1, 2, 3]) yield v;
  // => yield* [1, 2, 3];
}
```

```
// 제너레이터 함수를 호출하면 제너레이터를 반환한다.
let generatorObj = counter();

// 제너레이터는 이터러블이다.
console.log(Symbol.iterator in generatorObj); // true

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

generatorObj = counter();

// 제너레이터는 이터레이터이다.
console.log('next' in generatorObj); // true

console.log(generatorObj.next()); // {value: 1, done: fals
e}
console.log(generatorObj.next()); // {value: 2, done: fals
e}
console.log(generatorObj.next()); // {value: 3, done: fals
e}
console.log(generatorObj.next()); // {value: undefined, don
e: true}
```

제너레이터 함수의 정의

제너레이터 함수는 function* 키워드로 선언한다. 그리고 하나 이상의 yield 문을 포함한다.

```
// 제너레이터 함수 선언문
function* genDecFunc() {
  yield 1;
}

let generatorObj = genDecFunc();

// 제너레이터 함수 표현식
```

```
const genExpFunc = function* () {
  yield 1;
};

generatorObj = genExpFunc();

// 제너레이터 메소드
const obj = {
  * generatorObjMethod() {
    yield 1;
  }
};

generatorObj = obj.generatorObjMethod();

// 제너레이터 클래스 메소드
class MyClass {
  * generatorClsMethod() {
    yield 1;
  }
}

const myClass = new MyClass();
generatorObj = myClass.generatorClsMethod();
```

제너레이터 함수의 호출과 제너레이터 객체

제너레이터 함수를 호출하면 제너레이터 함수의 코드 블록이 실행되는 것이 아니라 제너레이터 객체를 반환한다. 앞에서 살펴본 바와 같이 제너레이터 객체는 이터러블이며 동시에 이터레이터이다. 따라서 next 메소드를 호출하기 위해 Symbol.iterator 메소드로 이터레이터를 별도 생성할 필요가 없다. 아래 예제를 살펴보자.

```
// 제너레이터 함수 정의
function* counter() {
  console.log('Point 1');
  yield 1;                // 첫번째 next 메소드 호출 시 여기까지
실행된다.
```

```
  console.log('Point 2');
  yield 2;                // 두번째 next 메소드 호출 시 여기까지
실행된다.
  console.log('Point 3');
  yield 3;                // 세번째 next 메소드 호출 시 여기까지
실행된다.
  console.log('Point 4'); // 네번째 next 메소드 호출 시 여기까지
실행된다.
}

// 제너레이터 함수를 호출하면 제너레이터 객체를 반환한다.
// 제너레이터 객체는 이터러블이며 동시에 이터레이터이다.
// 따라서 Symbol.iterator 메소드로 이터레이터를 별도 생성할 필요가 없
다
const generatorObj = counter();

// 첫번째 next 메소드 호출: 첫번째 yield 문까지 실행되고 일시 중단된
다.
console.log(generatorObj.next());
// Point 1
// {value: 1, done: false}

// 두번째 next 메소드 호출: 두번째 yield 문까지 실행되고 일시 중단된
다.
console.log(generatorObj.next());
// Point 2
// {value: 2, done: false}

// 세번째 next 메소드 호출: 세번째 yield 문까지 실행되고 일시 중단된
다.
console.log(generatorObj.next());
// Point 3
// {value: 3, done: false}

// 네번째 next 메소드 호출: 제너레이터 함수 내의 모든 yield 문이 실행
되면 done 프로퍼티 값은 true가 된다.
console.log(generatorObj.next());
```

```
// Point 4
// {value: undefined, done: true}
```

제너레이터 함수가 생성한 제너레이터 객체의 next 메소드를 호출하면 처음 만나는 yield 문까지 실행되고 일시 중단(suspend)된다. 또 다시 next 메소드를 호출하면 중단된 위치에서 다시 실행(resume)이 시작하여 다음 만나는 yield 문까지 실행되고 또 다시 일시 중단된다.

```
start -> generatorObj.next() -> yield 1 -> generatorObj.nex
t() -> yield 2 -> ... -> end
```

next 메소드는 이터레이터 리절트 객체와 같이 value, done 프로퍼티를 갖는 객체를 반환한다. value 프로퍼티는 yield 문이 반환한 값이고 done 프로퍼티는 제너레이터 함수 내의 모든 yield 문이 실행되었는지를 나타내는 boolean 타입의 값이다. 마지막 yield 문까지 실행된 상태에서 next 메소드를 호출하면 done 프로퍼티 값은 true가 된다.

제너레이터의 활용

이터러블의 구현

제너레이터 함수를 사용하면 이터레이션 프로토콜을 준수해 이터러블을 생성하는 방식보다 간편하게 이터러블을 구현할 수 있다. 이터레이션 프로토콜을 준수하여 무한 피보나치 수열을 생성하는 함수를 구현해 보자.

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

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

```
}());

// infiniteFibonacci는 무한 이터러블이다.
for (const num of infiniteFibonacci) {
  if (num > 10000) break;
  console.log(num); // 1 2 3 5 8...
}
```

이터레이션 프로토콜을 보다 간단하게 처리하기 위해 제너레이터를 활용할 수 있다. 제너레이터를 활용하여 무한 피보나치 수열을 구현한 이터러블을 만들어 보자.

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

  while (true) {
    [pre, cur] = [cur, pre + cur];
    yield cur;
  }
}());

// infiniteFibonacci는 무한 이터러블이다.
for (const num of infiniteFibonacci) {
  if (num > 10000) break;
  console.log(num);
}
```

제너레이터 함수에 최대값을 인수를 전달해보자.

```
// 무한 이터러블을 생성하는 제너레이터 함수
const createInfiniteFibByGen = function* (max) {
  let [prev, curr] = [0, 1];

  while (true) {
    [prev, curr] = [curr, prev + curr];
    if (curr >= max) return; // 제너레이터 함수 종료
```

```
    yield curr;
  }
};

for (const num of createInfiniteFibByGen(10000)) {
  console.log(num);
}
```

이터레이터의 next 메소드와 다르게 제너레이터 객체의 next 메소드에는 인수를 전달할 수도 있다. 이를 통해 제너레이터 객체에 데이터를 전달할 수 있다.

```
function* gen(n) {
  let res;
  res = yield n;    // n: 0 ⇐ gen 함수에 전달한 인수

  console.log(res); // res: 1 ⇐ 두번째 next 호출 시 전달한 데이터
  res = yield res;

  console.log(res); // res: 2 ⇐ 세번째 next 호출 시 전달한 데이터
  res = yield res;

  console.log(res); // res: 3 ⇐ 네번째 next 호출 시 전달한 데이터
  return res;
}
const generatorObj = gen(0);

console.log(generatorObj.next());  // 제너레이터 함수 시작
console.log(generatorObj.next(1)); // 제너레이터 객체에 1 전달
console.log(generatorObj.next(2)); // 제너레이터 객체에 2 전달
console.log(generatorObj.next(3)); // 제너레이터 객체에 3 전달
/*
{ value: 0, done: false }
{ value: 1, done: false }
{ value: 2, done: false }
```

```
{ value: 3, done: true }
*/
```

이터레이터의 next 메소드는 이터러블의 데이터를 꺼내 온다. 이에 반해 제너레이터의 next 메소드에 인수를 전달하면 제너레이터 객체에 데이터를 밀어 넣는다. 제너레이터의 이런 특성은 동시성 프로그래밍을 가능하게 한다.

비동기 처리

제너레이터를 사용해 비동기 처리를 동기 처리처럼 구현할 수 있다. 다시 말해 비동기 처리 함수가 처리 결과를 반환하도록 구현할 수 있다.

```
const fetch = require('node-fetch');

function getUser(genObj, username) {
  fetch(`https://api.github.com/users/${username}`)
    .then(res => res.json())
    // ① 제너레이터 객체에 비동기 처리 결과를 전달한다.
    .then(user => genObj.next(user.name));
}

// 제너레이터 객체 생성
const g = (function* () {
  let user;
  // ② 비동기 처리 함수가 결과를 반환한다.
  // 비동기 처리의 순서가 보장된다.
  user = yield getUser(g, 'jeresig');
  console.log(user); // John Resig

  user = yield getUser(g, 'ahejlsberg');
  console.log(user); // Anders Hejlsberg

  user = yield getUser(g, 'ungmo2');
  console.log(user); // Ungmo Lee
}());
```

```
// 제너레이터 함수 시작
g.next();
```

① 비동기 처리가 완료되면 next 메소드를 통해 제너레이터 객체에 비동기 처리 결과를 전달한다.

② 제너레이터 객체에 전달된 비동기 처리 결과는 user 변수에 할당된다.

제너레이터을 통해 비동기 처리를 동기 처리처럼 구현할 수 있으나 코드는 장황해졌다. 따라서 좀 더 간편하게 비동기 처리를 구현할 수 있는 async/await가 ES7에서 도입되었다.

위 예제를 async/await 구현해 보자.

```
const fetch = require('node-fetch');

// Promise를 반환하는 함수 정의
function getUser(username) {
  return fetch(`https://api.github.com/users/${username}`)
    .then(res => res.json())
    .then(user => user.name);
}

async function getUserAll() {
  let user;
  user = await getUser('jeresig');
  console.log(user);

  user = await getUser('ahejlsberg');
  console.log(user);

  user = await getUser('ungmo2');
  console.log(user);
}

getUserAll();
```