

46. 제너레이터와 `async/await`

2022.04.09

1. 제너레이터란?

코드 블록의 실행을 일시 중지했다가 필요한 시점에 재개할 수 있는 특수한 함수

1. 제너레이터란?

제너레이터와 일반 함수의 차이점

1. 제너레이터 함수는 함수 호출자에게 함수 실행의 제어권을 양도할 수 있다.
2. 제너레이터 함수는 함수 호출자와 함수의 상태를 주고받을 수 있다.
3. 제너레이터 함수를 호출하면 제너레이터 객체를 반환한다.

2. 제너레이터 함수의 정의

`function*` 키워드로 선언한다. 그리고 하나 이상의 `yield` 표현식을 포함한다.
위 특성을 제외하면 일반 함수를 정의하는 방법과 동일하다.

2. 제너레이터 함수의 정의

function* 키워드로 선언한다. 그리고 하나 이상의 yield 표현식을 포함한다.
위 특성을 제외하면 일반 함수를 정의하는 방법과 동일하다.

제너레이터 함수 선언문



```
function* genDecFunc() {  
  yield 1;  
}
```

2. 제너레이터 함수의 정의

function* 키워드로 선언한다. 그리고 하나 이상의 yield 표현식을 포함한다.
위 특성을 제외하면 일반 함수를 정의하는 방법과 동일하다.

제너레이터 함수 선언문

```
function* genDecFunc() {  
  yield 1;  
}
```

제너레이터 함수 표현식

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

2. 제너레이터 함수의 정의

function* 키워드로 선언한다. 그리고 하나 이상의 yield 표현식을 포함한다.
위 특성을 제외하면 일반 함수를 정의하는 방법과 동일하다.

제너레이터 함수 선언문

```
function* genDecFunc() {  
  yield 1;  
}
```

제너레이터 함수 표현식

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

제너레이터 메서드

```
const obj = {  
  * genObjMethod() {  
    yield 1;  
  }  
}
```

2. 제너레이터 함수의 정의

function* 키워드로 선언한다. 그리고 하나 이상의 yield 표현식을 포함한다.
위 특성을 제외하면 일반 함수를 정의하는 방법과 동일하다.

제너레이터 함수 선언문

```
function* genDecFunc() {  
  yield 1;  
}
```

제너레이터 함수 표현식

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

제너레이터 메서드

```
const obj = {  
  * genObjMethod() {  
    yield 1;  
  }  
}
```

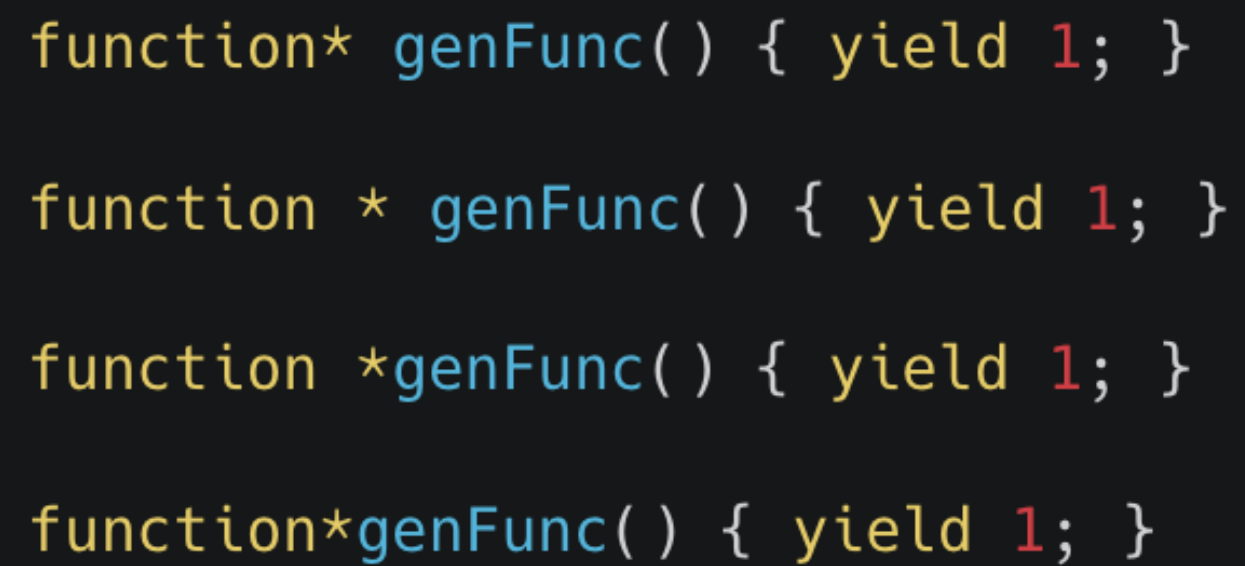
제너레이터 클래스 메서드

```
class MyClass {  
  * genClassMethod() {  
    yield 1;  
  }  
}
```


2. 제너레이터 함수의 정의

정의할 때 주의할 점

에스터리스크(*)의 위치는 function 키워드와 함수 이름 사이면 어디든 상관없지만, 일관성을 위해서 function 키워드 바로 뒤에 붙이는 걸 권장한다.



```
function* genFunc() { yield 1; }  
  
function * genFunc() { yield 1; }  
  
function *genFunc() { yield 1; }  
  
function*genFunc() { yield 1; }
```

2. 제너레이터 함수의 정의

정의할 때 주의할 점

제너레이터는 화살표 함수로 정의할 수 없다.
new 연산자와 함께 생성자 함수로 호출 할 수 없다.

```
const genArrowFunc = * () => {  
  yield 1;  
}; // SyntaxError: Unexpected token '*'
```

```
function* genFunc() {  
  yield 1;  
}  
  
new genFunc(); // TypeError: genFunc is not a constructor
```

3. 제너레이터 객체

제너레이터 함수를 호출하면 제너레이터 객체를 생성해 반환한다.
해당 객체는 이터러블이면서 이터레이터다.

```
function* genFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = genFunc();  
  
console.log(Symbol.iterator in generator); // true  
console.log('next' in generator); // true
```

3. 제너레이터 객체

제너레이터 객체는 이터레이터이지만 이터레이터에는 없는 `return`, `throw` 메서드를 갖는다.

3. 제너레이터 객체

제너레이터 객체는 이터레이터이지만 이터레이터에는 없는
return, throw 메서드를 갖는다.

return 메서드는 인수로 전달받은 값을 value 프로퍼티 값으로 설정한다.

```
function* genFunc() {
  try {
    yield 1;
    yield 2;
    yield 3;
  } catch(e) {
    console.error(e);
  }
}

const generator = genFunc();

console.log(generator.next()); // {value:1, done: false}
console.log(generator.return("End!")); // {value: "End!", done: true}
```

3. 제너레이터 객체

제너레이터 객체는 이터레이터이지만 이터레이터에는 없는
return, throw 메서드를 갖는다.

return 메서드는 인수로 전달받은 값을 value 프로퍼티 값으로 설정한다.

throw 메서드는 인수로 전달받은 에러를 발생시키고 undefined를 value 프로퍼티 값으로 설정한다.

```
function* genFunc() {
  try {
    yield 1;
    yield 2;
    yield 3;
  } catch(e) {
    console.error(e);
  }
}

const generator = genFunc();

console.log(generator.next()); // {value:1, done: false}
console.log(generator.return("End!")); // {value: "End!", done: true}
```

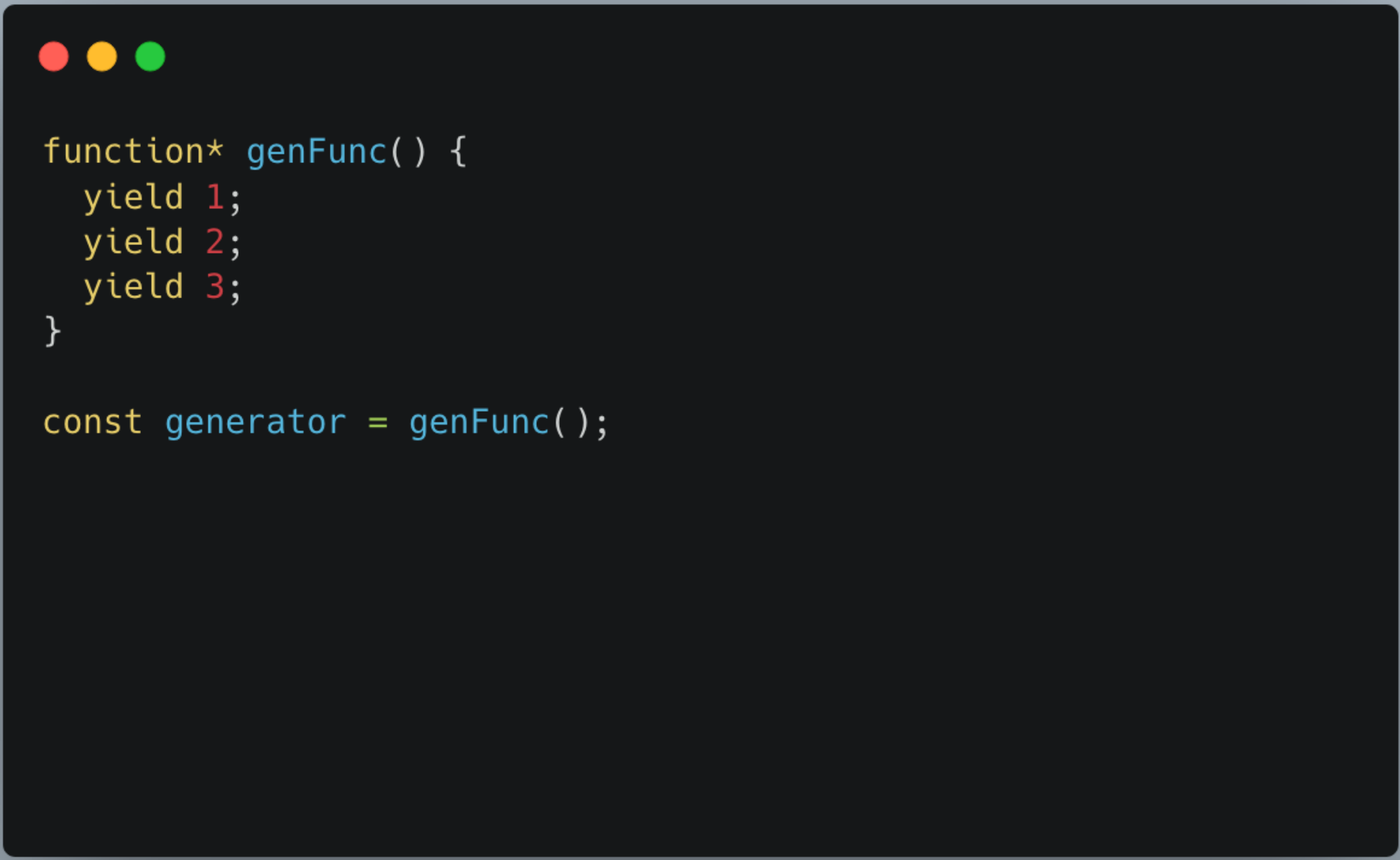
```
function* genFunc() {
  try {
    yield 1;
    yield 2;
    yield 3;
  } catch(e) {
    console.error(e);
  }
}

const generator = genFunc();

console.log(generator.next()); // {value:1, done: false}
console.log(generator.throw("Error")); // {value: undefined, done: true}
```

4. 제너레이터의 일시 중지와 재개

yield 키워드는 제너레이터 함수의 실행을 일시 중지시키거나 yield 키워드 뒤에 오는 표현식의 평가결과를 제너레이터 함수 호출자에게 반환한다.



```
function* genFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = genFunc();
```

4. 제너레이터의 일시 중지와 재개

yield 키워드는 제너레이터 함수의 실행을 일시 중지시키거나 yield 키워드 뒤에 오는 표현식의 평가결과를 제너레이터 함수 호출자에게 반환한다.



```
function* genFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = genFunc();  
console.log(generator.next()); // {value: 1, done: false}
```


4. 제너레이터의 일시 중지 và 재개

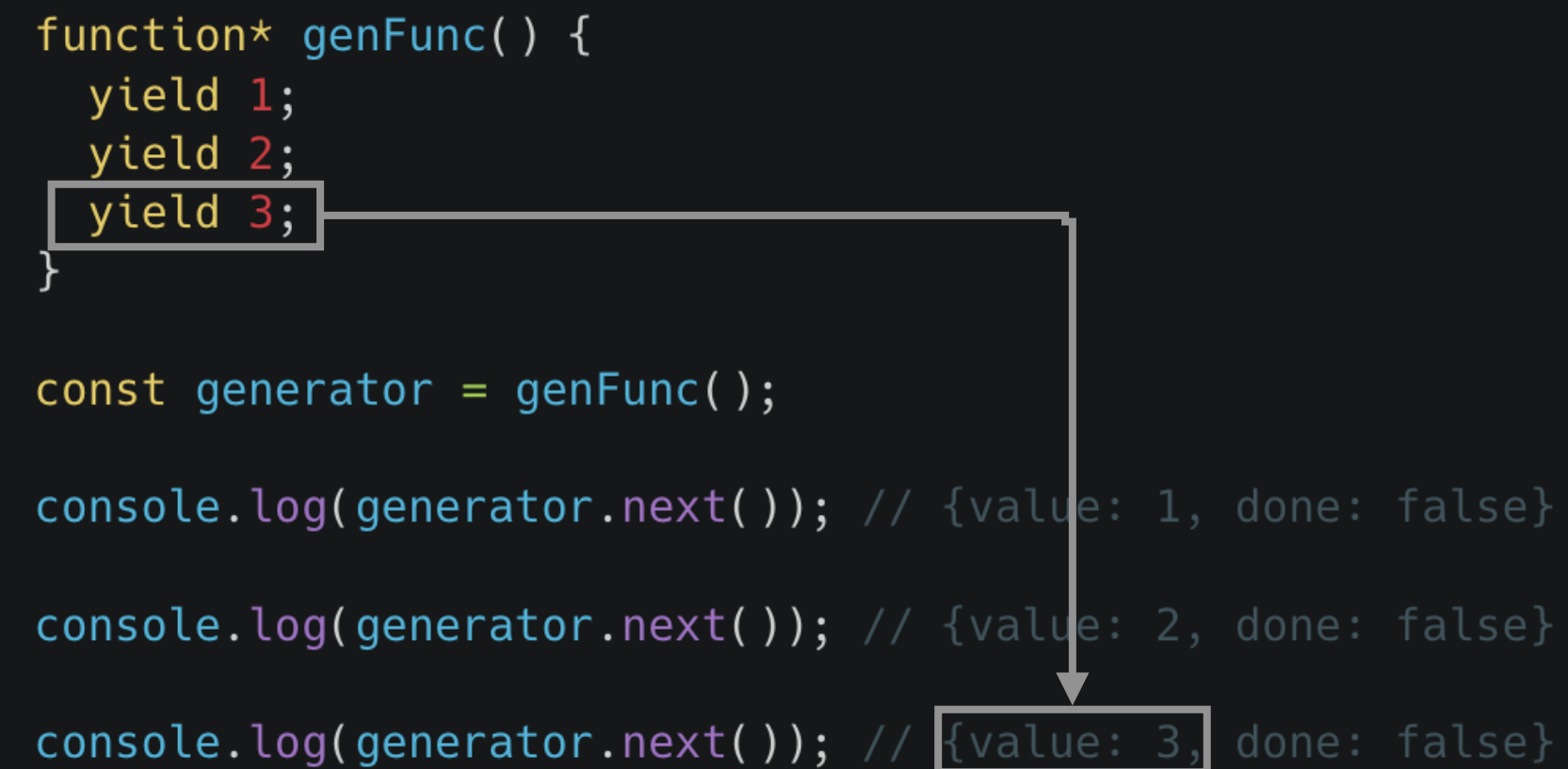
yield 키워드는 제너레이터 함수의 실행을 일시 중지시키거나 yield 키워드 뒤에 오는 표현식의 평가결과를 제너레이터 함수 호출자에게 반환한다.



```
function* genFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = genFunc();  
  
console.log(generator.next()); // {value: 1, done: false}  
console.log(generator.next()); // {value: 2, done: false}
```

4. 제너레이터의 일시 중지와 재개

yield 키워드는 제너레이터 함수의 실행을 일시 중지시키거나 yield 키워드 뒤에 오는 표현식의 평가결과를 제너레이터 함수 호출자에게 반환한다.



```
function* genFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = genFunc();  
  
console.log(generator.next()); // {value: 1, done: false}  
console.log(generator.next()); // {value: 2, done: false}  
console.log(generator.next()); // {value: 3, done: false}
```

4. 제너레이터의 일시 중지와 재개

yield 키워드는 제너레이터 함수의 실행을 일시 중지시키거나 yield 키워드 뒤에 오는 표현식의 평가결과를 제너레이터 함수 호출자에게 반환한다.

```
function* genFunc() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = genFunc();  
  
console.log(generator.next()); // {value: 1, done: false}  
console.log(generator.next()); // {value: 2, done: false}  
console.log(generator.next()); // {value: 3, done: false}  
console.log(generator.next()); // {value: undefined, done: true}
```

5. 제너레이터의 활용

1. 이터러블의 구현

이터레이션 프로토콜을 이용한 구현 방법

```
const infiniteFibonacci = (function() {
  let [pre, cur] = [0, 1];

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

for (const num of infiniteFibonacci) {
  if (num > 10000) break;
  console.log(num); // 1 2 3 5 8 ... 6765
}
```

제너레이터를 이용한 구현 방법

```
const infiniteFibonacci = (function* () {
  let [pre, cur] = [0, 1];

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

for (const num of infiniteFibonacci) {
  if (num > 10000) break;
  console.log(num); // 1 2 3 5 8 ... 6765
}
```

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

1. async 함수가 호출된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

2. 제너레이터 객체를 생성한다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

3. onResolved를 반환한다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};
```

```
(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

4. onResolved 함수를 즉시 호출한다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

5. 제너레이터 객체의 next 메서드를 호출한다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

6. 첫번째 yield문까지 실행된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑥
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

7. 아직 제너레이터 함수가 끝까지 실행되지 않아
fetch 함수가 반환한 Response 객체를
onResolved 함수의 인수로 전달하면서 재귀호출을 한다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

8. Response 객체를 next 메서드의 인수로 전달하면서 next 메서드를 다시 호출한다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

9. next 메서드로 전달된 Response 객체는 response 변수에 할당된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

10. 그 다음 두 번째 yield 문까지 실행된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑥
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

11. 아직 제너레이터가 끝나지 않아서 response.json()의 반환값이 onResolved 함수의 인자로 간다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

12. next 메서드가 호출된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

13. next 메서드로 전달된 인자는 todo 변수에 할당된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

14. fetchTodo 함수가 끝까지 실행된다.

5. 제너레이터의 활용

2. 비동기 처리

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

const async = generatorFunc => {
  const generator = generatorFunc(); // ②

  const onResolved = arg => {
    const result = generator.next(arg); // ⑤

    return result.done
      ? result.value // ⑨
      : result.value.then(res => onResolved(res)); // ⑦
  };

  return onResolved; // ③
};

(async(function* fetchTodo() { // ①
  const url = "https://jsonplaceholder.typicode.com/todos/1";

  const response = yield fetch(url); // ⑥
  const todo = yield response.json(); // ⑧
  console.log(todo);
})()); // ④
```

15. 제너레이터 객체가 종료된다.

6. `async/await`

1. `async` 함수


`async` 함수는 `async` 키워드를 사용해 정의하며 언제나 프로미스를 반환한다.

6. async/await

1. async 함수

async 함수는 async 키워드를 사용해 정의하며 언제나 프로미스를 반환한다.

async 함수 선언문



```
async function foo(n) { return n; }  
foo(1).then(v => console.log(v)); // 1
```

6. async/await

1. async 함수

async 함수는 `async` 키워드를 사용해 정의하며 언제나 프로미스를 반환한다.

async 함수 선언문

```
async function foo(n) { return n; }  
foo(1).then(v => console.log(v)); // 1
```

async 함수 표현식

```
const bar = async function(n) { return n; }  
bar(2).then(v => console.log(v)); // 2
```

6. async/await

1. async 함수

async 함수는 async 키워드를 사용해 정의하며 언제나 프로미스를 반환한다.

async 함수 선언문

```
async function foo(n) { return n; }  
foo(1).then(v => console.log(v)); // 1
```

async 함수 표현식

```
const bar = async function(n) { return n; }  
bar(2).then(v => console.log(v)); // 2
```

async 화살표 함수

```
const baz = async n => n;  
baz(3).then(v => console.log(v)); // 3
```


6. async/await

1. async 함수

async 함수는 async 키워드를 사용해 정의하며 언제나 프로미스를 반환한다.

async 함수 선언문

```
async function foo(n) { return n; }  
foo(1).then(v => console.log(v)); // 1
```

async 함수 표현식

```
const bar = async function(n) { return n; }  
bar(2).then(v => console.log(v)); // 2
```

async 화살표 함수

```
const baz = async n => n;  
baz(3).then(v => console.log(v)); // 3
```

async 메서드

```
const obj = {  
  async foo(n) { return n; }  
};  
obj.foo(4).then(v => console.log(v)); // 4
```

6. async/await

1. async 함수

async 함수는 async 키워드를 사용해 정의하며 언제나 프로미스를 반환한다.

async 함수 선언문

```
async function foo(n) { return n; }  
foo(1).then(v => console.log(v)); // 1
```

async 함수 표현식

```
const bar = async function(n) { return n; }  
bar(2).then(v => console.log(v)); // 2
```

async 화살표 함수

```
const baz = async n => n;  
baz(3).then(v => console.log(v)); // 3
```

async 메서드

```
const obj = {  
  async foo(n) { return n; }  
};  
obj.foo(4).then(v => console.log(v)); // 4
```

async 클래스 메서드

```
class MyClass {  
  async bar(n) { return n; }  
}  
const myClass = new MyClass();  
myClass.foo(5).then(v => console.log(v)); // 5
```

6. async/await

1. async 함수 - 주의할 점

클래스의 constructor 메서드는 async 메서드가 될 수 없다.



```
class MyClass {  
  async constructor() {}  
  // Syntax Error: Class constructor may not be any async method  
}  
  
const myClass = new MyClass();
```

6. async/await

1. async 함수 - 주의할 점

클래스의 constructor 메서드는 async 메서드가 될 수 없다.



```
class MyClass {  
  async constructor() {}  
  // Syntax Error: Class constructor may not be any async method  
}  
  
const myClass = new MyClass();
```

클래스의 constructor 메서드는 인스턴스를 반환해야하지만,
async 함수는 언제나 프로미스를 반환하기 때문이다.

6. async/await

2. await 키워드

await 키워드는 프로미스가 settled 상태가 될 때까지 대기하다가 settled 상태가 되면 프로미스가 resolve한 처리 결과를 반환한다.



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

const getGithubUserName = async id => {
  const res = await fetch(`https://api.github.com/users/${id}`); // ①
  const { name } = await res.json(); // ②
  console.log(name); // Ungmo Lee
};

getGithubUserName("ungmo2");
```

6. async/await

2. await 키워드

await 키워드는 프로미스가 settled 상태가 될 때까지 대기하다가 settled 상태가 되면 프로미스가 resolve한 처리 결과를 반환한다.

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

const getGithubUserName = async id => {
  const res = await fetch(`https://api.github.com/users/${id}`); // ①
  const { name } = await res.json(); // ②
  console.log(name); // Ungmo Lee
};

getGithubUserName("ungmo2");
```

1. HTTP 요청에 대한 서버의 응답이 도착해서 fetch 함수가 반환한 프로미스가 settled 상태가 될 때까지 대기한다.

6. async/await

2. await 키워드

await 키워드는 프로미스가 settled 상태가 될 때까지 대기하다가 settled 상태가 되면 프로미스가 resolve한 처리 결과를 반환한다.

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

const getGithubUserName = async id => {
  const res = await fetch(`https://api.github.com/users/${id}`); // ①
  const { name } = await res.json(); // ②
  console.log(name); // Ungmo Lee
};

getGithubUserName("ungmo2");
```


1. HTTP 요청에 대한 서버의 응답이 도착해서 fetch 함수가 반환한 프로미스가 settled 상태가 될 때까지 대기한다.

await 키워드는 반드시 async 함수 내에서 사용해야하고,
또한 프로미스 앞에서 사용해야한다.

6. async/await

2. await 키워드

await 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개한다.



```
async function foo() {  
  const a = await new Promise(resolve => setTimeout(() => resolve(1), 3000));  
  const b = await new Promise(resolve => setTimeout(() => resolve(2), 2000));  
  const c = await new Promise(resolve => setTimeout(() => resolve(3), 1000));  
  
  console.log([a, b, c]);  
}  
  
foo(); // 약 6초가 소요된다.
```


6. async/await

2. await 키워드

await 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개한다.

```
async function foo() {  
  const a = await new Promise(resolve => setTimeout(() => resolve(1), 3000));  
  const b = await new Promise(resolve => setTimeout(() => resolve(2), 2000));  
  const c = await new Promise(resolve => setTimeout(() => resolve(3), 1000));  
  
  console.log([a, b, c]);  
}  
  
foo(); // 약 6초가 소요된다.
```

3초동안 대기 (누적: 3초)

6. async/await

2. await 키워드

await 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개한다.

```
async function foo() {  
  const a = await new Promise(resolve => setTimeout(() => resolve(1), 3000));  
  const b = await new Promise(resolve => setTimeout(() => resolve(2), 2000));  
  const c = await new Promise(resolve => setTimeout(() => resolve(3), 1000));  
  
  console.log([a, b, c]);  
}  
  
foo(); // 약 6초가 소요된다.
```

2초동안 대기 (누적: 5초)

6. async/await

2. await 키워드

await 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개한다.

```
async function foo() {  
  const a = await new Promise(resolve => setTimeout(() => resolve(1), 3000));  
  const b = await new Promise(resolve => setTimeout(() => resolve(2), 2000));  
  const c = await new Promise(resolve => setTimeout(() => resolve(3), 1000));  
  
  console.log([a, b, c]);  
}  
  
foo(); // 약 6초가 소요된다.
```

1초동안 대기 (누적: 6초)

6. async/await

2. await 키워드

await 키워드는 다음 실행을 일시 중지시켰다가 프로미스가 settled 상태가 되면 다시 재개한다.

```
async function foo() {  
  const a = await new Promise(resolve => setTimeout(() => resolve(1), 3000));  
  const b = await new Promise(resolve => setTimeout(() => resolve(2), 2000));  
  const c = await new Promise(resolve => setTimeout(() => resolve(3), 1000));  
  
  console.log([a, b, c]);  
}  
  
foo(); // 약 6초가 소요된다.
```

1초동안 대기 (누적: 6초)

위 작업은 약 6초가 소요된다.

6. async/await

2. await 키워드

순서대로 실행될 필요가 없는 비동기 작업은 다음과 같이 처리할 수 있다.

```
async function foo() {  
  const res = await Promise.all([  
    new Promise(resolve => setTimeout(() => resolve(1), 3000)),  
    new Promise(resolve => setTimeout(() => resolve(2), 2000)),  
    new Promise(resolve => setTimeout(() => resolve(3), 1000))  
  ]);  
  console.log(res);  
}  
  
foo(); // 약 3초가 소요된다.
```

6. async/await

3. 에러 처리

비동기 처리를 위한 콜백 패턴의 단점 중 가장 심각한 것은 에러 처리가 곤란하다는 것이다.

에러는 호출자 방향으로 전파된다.

하지만, 비동기 함수의 콜백 함수를 호출한 것은 비동기 함수가 아니기 때문에 try ... catch문을 사용해 에러를 캐치할 수 없다.



```
try {
  setTimeout(() => { throw new Error("Error!"); }, 1000);
} catch(e) {
  console.error("Error: ", e);
}
```

6. async/await

3. 에러 처리

프로미스를 반환하는 비동기 함수는 명시적으로 호출할 수 있기 때문에 호출자가 명확하다.
그래서 try ... catch 문을 사용할 수 있다.

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

const foo = async () => {
  try {
    const wrongUrl = 'https://wrong.url';

    const response = await fetch(wrongUrl);
    const data = await response.json();
    console.log(data);
  } catch(e) {
    console.error(e); // TypeError: Failed to fetch
  }
}
```

6. async/await

3. 에러 처리

Promise.prototype.catch 후속 처리 메서드를 사용해 에러를 캐치할 수도 있다.



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

const foo = async () => {
  const wrongUrl = 'https://wrong.url';
  const response = await fetch(wrongUrl);
  const data = await response.json();
  return data;
};

foo().then(console.log).catch(console.error); // TypeError: Failed to fetch
```