

JavaScript

Generator – 김근형 강사

Generator

- Generator
 - ES6에서 도입된 제너레이터(Generator) 함수는 이터러블을 생성하는 함수이다.
 - 제너레이터 함수를 쓰는 이유는 다음과 같다.
 - 비동기 특성을 동기적 코드방식으로 관리 해준다.
 - 이터레이터와 이터러블을 쉽게 사용 할 수 있다.
 - 제네레이터 함수는 코루틴 특성을 가지고 있다.
 - 제네레이터 함수는 동시적인 특성을 갖고 있다.
 - 제네레이터 함수는 비동기적 특성을 갖고 있다.
 - 메모리 효율에 기여할 수 있다.

Generator

- Generator 오브젝트

- `function*`는 키워드로 이를 사용한 함수를 제너레이터 함수(Generator Function)라고 한다.
- 아래와 같이 세가지 형태로 제너레이터 함수를 작성할 수 있다.
 - `function*` 선언문
 - `function*` 표현식
 - `GeneratorFunction`
- 제너레이터 함수를 호출하면 제너레이터 오브젝트를 생성하여 반환한다.
- 생성한 제너레이터 오브젝트에 호출한 함수에서 넘겨 준 파라미터 값이 설정된다.
- 생성된 제너레이터 오브젝트는 이터레이터 오브젝트이다.
- 제너레이터 함수는 `new` 연산자를 사용할 수 없으며 사용하면 `TypeError`가 발생한다.

generator

- `function*` 선언문
 - 선언문 형태로 제너레이터 함수를 정의한다.

구분	타입	데이터(값)
형태		<code>Function* name(){} </code>
파라미터		(선택) <code>[param1, param2 ... param255]</code>
반환	<code>Function</code>	제너레이터 함수

```
function* sports(one, two){  
  console.log("함수 블록");  
  yield one + two;  
};  
console.log(typeof sports);  
  
let genObj = sports(1, 2);  
console.log(typeof genObj);
```



function
object

generator

- function* 표현식
 - 표현식 형태로 제너레이터 함수를 정의한다.

구분	타입	데이터(값)
형태		Function* [function name]()
파라미터		(선택) [param1, param2 ... param255]
반환	Function	제너레이터 함수

```
let sports = function*(one, two){  
  console.log("함수 블록");  
  yield one + two;  
};  
let genObj = sports(10, 20);  
  
console.log(genObj.next());
```



함수 블록

```
▼ Object i  
  done: false  
  value: 30  
  ► __proto__: Object
```

generator

- GeneratorFunction(): 제너레이터 함수 생성
 - 제너레이터 함수를 생성해서 반환한다.

구분	타입	데이터(값)
형태		new GeneratorFunction()
파라미터	Any	(선택) [param1, param2 ... param255], functionBody
반환	Generator	Generator Object

```
let GenConst = Object.getPrototypeOf(function*(){}).constructor;
let sports = new GenConst(
  "one", "two",
  "console.log('함수 블록'); yield one + two"
);
let genObj = sports(3, 4);

console.log(genObj.next());
```



함수 블록

```
▼ Object i
  done: false
  value: 7
  ► __proto__: Object
```

Generator

- yield : 제너레이터 함수 실행, 멈춤
 - yeild 키워드는 제너레이터 함수를 멈추게 하거나 다시 실행하는 데 사용한다.

[구문]

```
[returnValue] = yield [expression]
```

- yield 키워드의 오른쪽 표현식(expression)은 선택으로 표현식을 작성하면 이를 평가하고 평가 결과를 반환한다.
- 표현식을 작성하지 않으면 undefined를 반환한다.
- yield의 표현식 평가 결과를 왼쪽의 [returnValue]에 할당하지 않고 제너레이터 오브젝트의 next()를 호출하면 next() 파라미터 값이 [returnValue]에 설정된다.

Generator

- yield : 제너레이터 함수 실행, 멈춤
 - next()로 제너레이터 함수를 호출하면 yield 작성에 관계없이 {value:값, done:true/false} 형태로 반환된다.
 - 아래의 방법으로 value 프로퍼티 값과 done 프로퍼티 값을 설정한다.
 - yield를 수행하면 표현식 평가 결과가 value 값에 설정되고, yield를 수행하지 못하면 undefined가 설정된다.
 - yield를 수행하면 done 값에 false가 설정되고 yield를 수행하지 못하면 true가 설정된다.
 - 제너레이터 함수의 모든 yield 수행을 완료했는데, 다시 next()를 호출하면 수행할 yield가 없으므로 value값에 undefined가 설정되고 done값에 true가 설정된다.

Generator

- yield : 제너레이터 함수 실행, 멈춤

```
function* sports(one){  
  let two = yield one;  
  let param = yield two + one;  
  yield param + one;  
}  
let generatorObj = sports(10);  
  
console.log(generatorObj.next());  
console.log(generatorObj.next());  
console.log(generatorObj.next(20));
```



```
▼ Object ⓘ  
  done: false  
  value: 10  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: NaN  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 30  
  ▶ __proto__: Object
```

Generator

- yield : 제너레이터 함수 실행, 멈춤

```
function* sports(one){  
  yield one;  
  let check = 10;  
}  
let genObj = sports(10);  
  
console.log(genObj.next());  
console.log(genObj.next());
```



```
▼ Object ⓘ  
  done: false  
  value: 10  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: true  
  value: undefined  
  ▶ __proto__: Object
```

Generator

- `next()` : `yield` 단위로 실행
 - 제너레이터 함수에서 `yield` 단위로 실행한다.

구분	타입	데이터(값)
형태		<code>Generator.prototype.next()</code>
파라미터	Any	(선택) 제너레이터 함수에 넘겨 줄 값
반환	object	제너레이터 함수에서 반환한 값

- `next()`를 호출하면 `yield`를 기준으로 이전 `yield`의 다음 줄부터 `yield`까지 수행한다.
- 제너레이터 함수에 `yield`가 다수 작성되어 있으면, `yield` 수만큼 `next()`를 작성해야 제너레이터 함수 전체를 실행하게 된다.
- 파라미터는 제너레이터 함수가 멈춘 `yield`의 왼쪽 변수에 설정한다.

Generator

- next() : yield 단위로 실행

```
let gen = function*(value){  
  value = value + 10;  
  yield ++value;  
  value = value + 7;  
  yield ++value;  
};  
let genObj = gen(1);  
  
console.log(genObj.next());  
console.log(genObj.next());  
console.log(genObj.next());
```



```
▼ Object i  
  done: false  
  value: 12  
  ▶ __proto__: Object  
  
▼ Object i  
  done: false  
  value: 20  
  ▶ __proto__: Object  
  
▼ Object i  
  done: true  
  value: undefined  
  ▶ __proto__: Object
```

Generator

- next() : yield 단위로 실행

```
let gen = function*(value){  
  return ++value;  
}  
let genObj = gen(1);  
console.log(genObj.next());
```



```
▼ Object ⓘ  
  done: true  
  value: 2  
  ► __proto__: Object
```

Generator

- next() : yield 단위로 실행

```
let gen = function*(param){  
  let one = param + 1;  
  yield one;  
  var two = 2;  
  yield one + two;  
};  
let genObj = gen(10);  
  
console.log(genObj.next());  
console.log(genObj.next());
```



```
▼ Object ⓘ  
  done: false  
  value: 11  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 13  
  ▶ __proto__: Object
```

Generator

- next() : yield 단위로 실행

```
let one, two;  
let gen = function*(){  
  one = yield;  
  two = yield one + 10;  
}  
let genObj = gen();  
  
console.log(genObj.next());  
console.log(genObj.next(12));  
console.log(genObj.next(34));
```



```
▼ Object ⓘ  
  done: false  
  value: undefined  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 22  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: true  
  value: undefined  
  ▶ __proto__: Object
```

Generator

- next() 활용

- 시나리오

- 청구 금액을 계산하는 제너레이터 함수의 할인 금액을 계산하는 일반 함수를 정의한다.
 - 청구 금액 계산 제너레이터 함수는 수량과 단가를 파라미터로 받아 금액을 계산한다.
 - 계산한 금액을 yield로 반환한다.
 - 할인 금액 함수를 호출하면서 yield로 반환된 값을 파라미터 값으로 넘겨준다.
 - 파라미터의 금액에 따라 할인 금액을 계산하여 반환한다.
 - 청구 금액 계산 제너레이터 함수를 호출하면서 할인 금액을 파라미터로 넘겨준다.
 - 합계 금액에서 할인 금액을 빼서 청구 금액을 계산한다.
 - 계산된 청구 금액을 반환한다.

Generator

○ next() 활용

```
let getAmount = function*(qty, price){
  let amount = Math.floor(qty * price);
  let discount = yield amount;
  return amount - discount;
};
let getDiscount = function(amount){
  return amount > 1000 ? amount * 0.2 : amount * 0.1;
};
let amountObj = getAmount(10, 60);

let result = amountObj.next();
console.log(result);

let dcAmount = getDiscount(result.value);
console.log(dcAmount);

console.log(amountObj.next(dcAmount));
```



```
▼ Object ⓘ
  done: false
  value: 600
  ▶ __proto__: Object
60
▼ Object ⓘ
  done: true
  value: 540
  ▶ __proto__: Object
```

Generator

○ next() 의 다양한 형태

```
let gen = function*(value) {  
  let count = 0;  
  while (value){  
    yield ++count;  
  }  
};  
let genObj = gen(true);  
  
console.log(genObj.next());  
console.log(genObj.next());
```



```
▼ Object ⓘ  
  done: false  
  value: 1  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 2  
  ▶ __proto__: Object
```

Generator

○ next() 의 다양한 형태

```
let gen = function*(){  
  return yield yield yield;  
}  
let genObj = gen();  
  
console.log(genObj.next());  
console.log(genObj.next(10));  
  
console.log(genObj.next(20));  
console.log(genObj.next(30));
```



```
▼ Object ⓘ  
  done: false  
  value: undefined  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 10  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 20  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: true  
  value: 30  
  ▶ __proto__: Object
```

Generator

○ next() 의 다양한 형태

```
let gen = function*(){  
  return [yield yield];  
};  
let genObj = gen();  
  
console.log(genObj.next());  
console.log(genObj.next(10));  
console.log(genObj.next(20));
```



```
▼ Object ⓘ  
  done: false  
  value: undefined  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 10  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: true  
  ▼ value: Array(1)  
    0: 20  
    length: 1  
    ▶ __proto__: Array(0)  
  ▶ __proto__: Object
```

Generator

○ next() 의 다양한 형태

```
let gen = function*(start){  
  let value = start;  
  while (true){  
    yield ++value;  
  }  
};  
  
for (var count of gen(10)){  
  console.log(count);  
  if (count > 12){  
    break;  
  }  
};
```



11
12
13

Generator

- `return()` : 이터레이터 종료
 - 제너레이터 함수의 이터레이터를 종료한다.

구분	타입	데이터(값)
형태		<code>Generator.prototype.return()</code>
파라미터	Any	(선택) 제너레이터 함수에 넘겨 줄 값
반환	Any	<code>Return()</code> 의 파라미터 값

Generator

- `return()` : 이터레이터 종료

```
let gen = function*(start){
  let count = start;
  while(true){
    yield ++count;
  }
}
let genObj = gen(10);

console.log(genObj.next());
console.log(genObj.return(77));
console.log(genObj.next(55));
```



```
▼ Object ⓘ
  done: false
  value: 11
  ▶ __proto__: Object

▼ Object ⓘ
  done: true
  value: 77
  ▶ __proto__: Object

▼ Object ⓘ
  done: true
  value: undefined
  ▶ __proto__: Object
```

Generator

- throw() : 에러 발생
 - Error를 발생시킨다.

구분	타입	데이터(값)
형태		Generator.prototype.throw()
파라미터	Any	(선택) 에러 메시지
반환	Object	{value : 에러 메시지, done : true}

Generator

○ throw() : 에러 발생

```
let gen = function*(){
  try {
    yield 10;
  } catch (message) {
    yield message;
  }
  yield 20;
}
let genObj = gen();

console.log(genObj.next());
console.log(genObj.throw("에러 발생"));
console.log(genObj.next());
```



```
▼ Object i
  done: false
  value: 10
  ▶ __proto__: Object

▼ Object i
  done: false
  value: "에러 발생"
  ▶ __proto__: Object

▼ Object i
  done: false
  value: 20
  ▶ __proto__: Object
```

Generator

- throw() : 에러 발생

```
let gen = function*(){  
  throw "에러 발생";  
  yield 20;  
};  
let genObj = gen();  
  
try {  
  let result = genObj.next();  
} catch (error) {  
  console.log(error);  
}  
console.log(genObj.next());
```



에러 발생

```
▼ Object ⓘ  
  done: true  
  value: undefined  
  ► __proto__: Object
```

Generator

- yield* 키워드

[구문]

```
[yield* [[expression]]
```

- 표현식에 이터러블 오브젝트를 작성하는 점이 다르다.
- next()를 호출할 때마다 이터러블 오브젝트를 하나씩 실행하며, 결과 값을 yield의 반환 값으로 사용한다.
- 또한 표현식에 제너레이터 함수를 작성할 수 있다.
- 표현식으로 호출된 함수에 다수의 yield가 있으면 호출된 함수의 yield를 전부 처리한 후 yield* 아래에 작성한 코드를 실행한다.

Generator

○ yield* 키워드

```
function* gen() {  
  yield* [10, 20];  
}  
let genObj = gen();  
  
console.log(genObj.next());  
console.log(genObj.next());  
console.log(genObj.next(77));
```



```
▼ Object ⓘ  
  done: false  
  value: 10  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 20  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: true  
  value: undefined  
  ▶ __proto__: Object
```

Generator

○ yield* 키워드

```
let plusGen = function*(value) {  
  yield value + 5;  
  yield value + 10;  
};  
let gen = function*(value) {  
  yield* plusGen(value);  
  yield value + 20;  
};  
let genObj = gen(10);  
  
console.log("1:", genObj.next());  
console.log("2:", genObj.next());  
console.log("3:", genObj.next());
```



```
1: ▼ Object ⓘ  
   done: false  
   value: 15  
   ▶ __proto__: Object  
2: ▼ Object ⓘ  
   done: false  
   value: 20  
   ▶ __proto__: Object  
3: ▼ Object ⓘ  
   done: false  
   value: 30  
   ▶ __proto__: Object
```

Generator

○ yield* 키워드

```
let gen = function*(value) {  
  yield value;  
  yield* gen(value + 10);  
}  
let genObj = gen(1);  
  
console.log(genObj.next());  
console.log(genObj.next());  
console.log(genObj.next());
```



```
▼ Object ⓘ  
  done: false  
  value: 1  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 11  
  ▶ __proto__: Object  
▼ Object ⓘ  
  done: false  
  value: 21  
  ▶ __proto__: Object
```