

# **11. 배열과 객체, 좀 더 깊게 살펴보기**

---

# ECMAScript2015 + 기능 정리

---

# 매개변수 기본값

매개변수에 기본값을 지정하지 않을 때

```
function hello(name, message) {  
  console.log(`${name}님, ${message}`)  
}  
hello("도레미", "반갑습니다.") // "도레미님, 반갑습니다."  
hello("백두산")                 // "백두산님, undefined"
```

매개변수에 기본값을 지정했을 때

```
function hello(name, message = "안녕하세요?") {  
  console.log(`${name}님, ${message}`)  
}  
hello("도레미", "반갑습니다.") // "도레미님, 반갑습니다."  
hello("백두산")                 // "백두산님, 안녕하세요?"
```

# 전개구문

## 마침표 세 개(...)의 다양한 쓰임 – 함수에서

매개변수의 개수에 상관없이 매개변수를 하나의 변수로 받을 수 있다.

```
function addNum(...numbers) {  
  let sum = 0;  
  
  for (let number of numbers)  
    sum += number;  
  
  return sum;  
}  
  
console.log(addNum(1, 2));    // 3  
console.log(addNum(1, 2, 3, 4, 5));  // 15
```

# 전개구문

## 마침표 세 개(...)의 다양한 쓰임 – 배열에서

배열과 배열을 연결한다

```
let animal = ["bird", "cat"]  
let fruits = ["apple", "banana", "cherry"]  
  
animal.concat(fruits)  
[...animal, ...fruits]
```

배열의 값만 꺼내서 복사한다

```
let fruits = ["apple", "banana", "cherry"]  
let favorite = fruits  
favorite[1] = "grape"  
fruits          // ["apple", "grape", "cherry"]
```

# 객체의 프로퍼티 – 대괄호 표기법

객체의 프로퍼티에 접근할 때 일반적으로 점(.) 표기법을 사용하지만  
ECMAScript2015 이후에는 대괄호([ ])를 사용할 수 있다.

→ 대괄호 안에 입력하는 프로퍼티 이름은 문자열로 작성

```
const book = {  
  title: "Javascript",  
  pages: 500  
}  
book.published date = "2022-01"    // SyntaxError  
book["published date"] = "2022-01"  
book    // {title: 'Javascript', pages: 500, published date: '2022-01'}
```

# 객체의 프로퍼티 – 계산된 프로퍼티 이름

ECMAScript2015 이후에는 함수나 계산식을 프로퍼티 이름으로 사용할 수 있다.

```
function fn() {  
  return "result";  
}  
  
const obj = {  
  [fn()] : "함수 키"  
}  
  
obj  
// obj = {  
//   result: "함수 키"  
// }
```

```
function add(a, b) {  
  return a + b;  
}  
  
const obj = {  
  [fn()] : "함수 키",  
  [`${add(10, 20)} key`] : "계산식 키"  
}  
  
obj  
// obj = {  
//   result: "함수 키",  
//   30 key: "계산식 키"  
// }
```

# 객체의 프로퍼티 – 프로퍼티값 단축

객체를 정의할 때 변수와 프로퍼티 이름이 같다면 줄여서 사용할 수 있다

```
function makeUser(name, age) {  
  return {  
    name : name,  
    age : age  
  }  
}  
  
let user1 = makeUser("백두산",  
20)
```

```
user1    // {name: "백두산", age: 20}
```

```
function makeUser(name, age) {  
  return {  
    name,  
    age  
  }  
}  
  
let user2 = makeUser("한라산", 27)  
user2    // {name: "한라산", age: 27}
```



# 객체에서 심벌키 사용하기

## 심벌(Symbol)

- ECMAScript2015 이후에 추가된 새로운 원시 자료형
- 한 번 정의하면 값을 변경할 수 없고 유일한 값을 갖는다
- 2명 이상의 개발자가 하나의 프로그램을 개발 할 때 변수나 프로퍼티 이름을 같게 만드는 실수를 피할 수 있다.

## 심벌을 사용해 프로퍼티 정의하기

객체를 만들면서 일부 정보를 드러내고 싶지 않을 때 심볼 사용

```
for(item in member) {  
  console.log(`${item} :  
  ${member[item]}`)  
}  
// name : Kim  
// age : 25
```



```
const id = Symbol("id")  
const tel = Symbol("telephone  
number")  
const member = {  
  name : "Kim",  
  age : 25,  
  [id] : 1235,  
  [tel] : function () {  
    alert(prompt("전화번호 : "));  
  }  
}
```

# 객체에서 심벌키 사용하기

## 심벌키에 접근하기

- 심벌키를 사용한 프로퍼티나 메서드에 접근하려면 대괄호를 사용한다.
- 심벌키를 사용한 메서드를 실행할 때는 대괄호의 오른쪽에 소괄호(())를 붙인다.

```
const id = Symbol("id")
const tel = Symbol("telephone
number")
const member = {
  name : "Kim",
  age : 25,
  [id] : 1235,
  [tel] : function () {
    alert(prompt("전화번호 : "));
  }
}
```

```
member[id]    // 1235
```

```
member[tel]() // 프롬프트 창에 전화번호를 입력하면 알림 창에
```

표시

# 전역 심벌

## Symbol.for() 메서드

키를 인수로 받고 전역 심벌 레지스트리를 뒤져서 키에 해당하는 심벌을 찾는다.

레지스트리에 키에 맞는 심벌이 있으면 해당 심벌을 반환하고, 그렇지 않으면 새로운 심벌을 만들어서 반환한다

기본형 Symbol.for(키)

```
let tel = Symbol.for("tel")    // 처음이므로 심벌 생성
let phone = Symbol.for("tel") // tel 키에 대한 심벌이 이미 있으므로 가져와서 사용
tel === phone    // true
```

# 전역 심벌

## Symbol.keyFor() 메서드

심벌값을 인수로 받아서 전역 심벌 레지스트리를 뒤져서 심벌값의 키를 찾는다.

기본형 Symbol.keyFor(심벌값)

```
let tel = Symbol.for("tel")
let phone = Symbol.for("tel")
tel === phone
Symbol.keyFor(phone) // "tel"
```

---

# 구조 분해 할당

---

# 구조 분해 할당이란

- 주어진 자료의 구조를 분해해서 변수에 할당하는 기능
- 디스트럭팅(destructuring)이라고 한다.
- 배열이나 객체는 하나의 변수에 다양한 값이 들어 있는데, 그중에서 일부만 꺼내어 다른 변수로 할당할 수 있다.

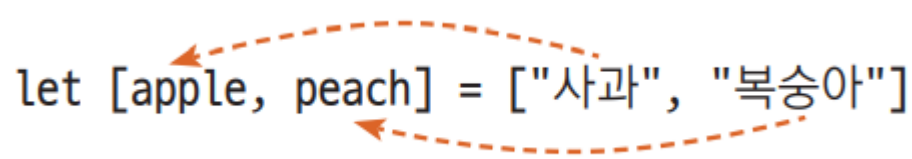
```
let [apple, peach] = ["사과", "복숭아"]
```

또는

```
const fruits = ["사과", "복숭아"]  
let [apple, peach] = fruits
```

```
apple    // "사과"  
peach    // "복숭아"
```

```
let [apple, peach] = ["사과", "복숭아"]
```



# 일부 값만 구조 분해 할당하기

구조 분해는 값의 일부만 변수에 할당할 때 더욱 편리하다.

```
let [spring, ,fall, ] = ["봄", "여름", "가을", "겨울"]  
spring      // "봄"  
fall       // "가을"
```

## 나머지 변수를 사용해 구조 분해 할당하기

- 구조 분해에서 일부 값을 변수로 지정한 후 나머지 값을 묶어서 하나의 변수로 할당할 수 있다.  
→ 나머지 값을 묶어서 만든 변수를 '나머지 변수'라고 한다.
- 나머지 변수 이름 앞에 ...를 붙이고, 나머지 변수에 할당하는 값은 마지막에 오는 값이어야 한다

```
let [teacher, ...students] = ["Kim", "Lee", "Park",  
"Choi"]  
teacher      // "Kim"  
students     // ["Lee", "Park", "Choi"]
```

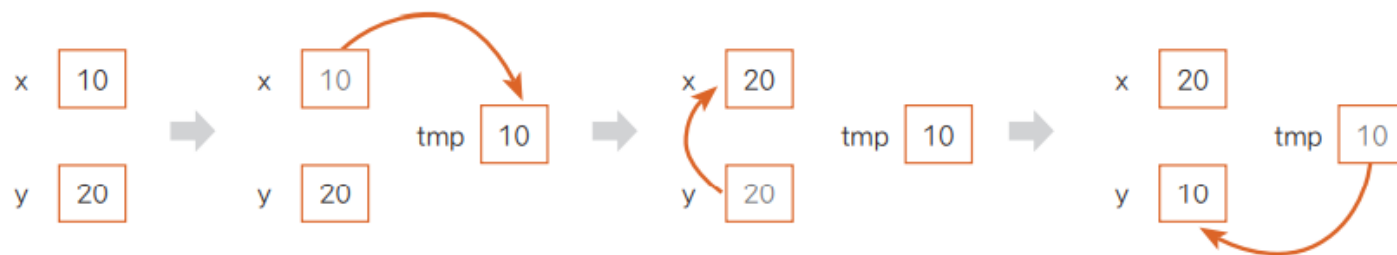
# 두 변수의 값 교환하기

구조 분해를 사용하면 두 변수의 값을 서로 교환해서 할당할 때도 편리하다

예) `let x = 10, let y = 20`

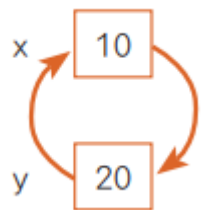
일반적인 방법

```
let tmp;  
tmp = x;  
x = y;  
y = tmp;
```



구조 분해를 사용한 방법

```
[x, y] = [y, x]
```





# 객체 구조 분해

객체를 구조 분해할 때는 프로퍼티 이름이나 메서드 이름을 변수 이름으로 사용한다.

(왜냐하면 객체에는 순서가 없기 때문에 키 이름과 변수 이름이 같아야 해당 키를 찾아서 값을 가져올 수 있기 때문)

```
const member = {  
  name : "kim",  
  age : 25  
}  
  
let {name, age} = member  
  
name    // "kim"  
age     // 25
```

```
let name, age  
{name, age} = {name : "kim", age : 25}  
  
name    // "kim"  
age     // 25
```

# 객체 구조 분해

변수의 이름을 객체의 프로퍼티 이름이 아닌 다른 이름으로 지정하려면

```
const member = {  
  name : "Kim",  
  age : 25  
}  
  
let {name : userName, age} =  
member
```

username // "Kim"

name 프로퍼티값을 userName 변수에 할당한다

# 중첩된 객체 구조 분해

변수의 이름을 객체의 프로퍼티 이름이 아닌 다른 이름으로 지정하려면

```
const student = {  
  name : "도레미",  
  score : {  
    history : 85,  
    science : 94  
  },  
  friends : ["Kim", "Lee",  
    "Park"]  
}
```

```
let {  
  name,           // student.name의 값  
  score : {  
    history,      // student.score.history의 값  
    science       // student.score.science의 값  
  },  
  friends : [f1, f2, f3] // friends 배열 구조 분해  
} = student
```

또는

```
let { name, score : { history, science }, friends : [f1, f2,  
f3] } = student
```

---

# 배열을 변형하는 메서드

---

# 배열에 같은 함수 적용하기 – map()

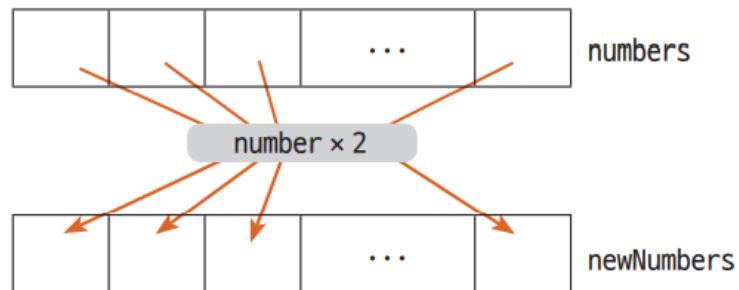
map()은 각 배열 요소에 똑같은 함수를 실행한 후 그 결과를 새로운 배열로 반환하는 메서드

기본형 `map(함수(값))`

(예) numbers 배열의 각 요소에 2를 곱해서 새로운 배열을 만든다면

```
let numbers = [1, 2, 3, 4, 5]
let newNumbers = numbers.map(number => number * 2);
```

newNumbers // [2, 4, 6, 8, 10]



# 배열에 같은 함수 적용하기 – map()

map()은 각 배열 요소에 똑같은 함수를 실행한 후 그 결과를 새로운 배열로 반환하는 메서드

```
기본형 map(함수(값, 인덱스)) // 요소 반환  
map(함수(값, 인덱스, 배열))
```

```
let numbers = [1, 2, 3, 4, 5]  
let newNumbers2 = numbers.map((number, index) => index +  
(number * 3))
```

```
newNumbers2 // [3, 7, 11, 15, 19]
```

# 특정 조건으로 골라내기 – filter()

특정 조건에 맞는 요소만 골라내는 메서드

```
기본형 filter(함수(값))  
filter(함수(값, 인덱스))  
filter(함수(값, 인덱스, 배열))
```

(예) 여러 점수가 저장된 scores 배열에서 85점 이상만 골라서 highScores 배열을 만들려면

```
let scores = [90, 35, 64, 88, 45, 92]  
highScores = scores.filter(score => score >= 85) // [90, 88,  
92]
```

(예) scores 배열에서 85점 이상의 값을 찾으면서 인덱스도 같이 알고 싶다면

```
let scores = [90, 35, 64, 88, 45, 92]  
let highScores2 = scores.filter((score, index) => {  
  if(score >= 85) {  
    console.log(`index : ${index}, score : ${score}`);  
    return score;  
  }  
});
```

# 값 누적하기 – reduce()

- 배열 요소에 차례로 함수를 실행하면서 결과값을 하나로 누적하는 메서드
- 초깃값은 생략할 수 있다. 생략하면 배열의 첫번째 값을 초깃값으로 사용

**기본형** `reduce(함수(누산기, 현재값, 인덱스, 원래 배열), 초깃값)`

- ① 현재값 = 초깃값 + 첫번째 배열 요소 실행한 값
- ② 현재값 = 기존 현재값 + 두번째 배열 요소 실행한 값
- ③ ...

예) numbers 배열에 있는 요소를 차례대로 더해서 result에 저장하려면

```
let numbers = [1, 2, 3, 4, 5]
let result = numbers.reduce((total, current) => total + current, 0);
result      // 15
```

표 11-1 reduce() 메서드의 진행 순서

	total	current	result
첫 번째 실행	0	1	①
두 번째 실행	1	2	③
세 번째 실행	3	3	⑥
네 번째 실행	6	4	⑩
다섯 번째 실행	10	5	15



---

# Map과 Set

---

# 맵과 셋이 등장한 이유

- 자바스크립트에서 여러 값을 하나의 변수로 묶어서 처리하기 위해 배열이나 객체 사용
- 배열과 객체에서 해야 할 일이 점점 많아지면서 배열과 객체로는 부족하다고 생각했던 부분을 보완해 맵과 셋이 도입되었다.

- 객체에서 '키'에는 문자열만 사용할 수 없다.  
→ 맵에서는 키에 모든 값을 사용할 수 있다.
- 객체에는 여러 정보를 담을 수 있지만 프로퍼티 간에 순서가 없다.  
→ 맵과 셋에는 순서가 있다.
- for 문과 같은 반복문을 사용해서 객체의 프로퍼티를 반복할 수 없다.  
→ 맵과 셋에서는 for... of 같은 반복문을 사용할 수 있다.
- 객체에는 프로퍼티의 개수를 알려 주는 프로퍼티가 없다.  
→ 맵과 셋에는 별도의 프로퍼티가 있다.
- 맵과 셋은 배열이나 객체보다 많은 메서드를 가지고 있다.

# 맵

- '키'와 '값'이 하나의 쌍으로 이루어졌고 여러 개의 프로퍼티를 가지고 있는 자료 형태 (객체와 비슷)
- 맵의 프로퍼티키는 문자열뿐만 아니라 모든 자료형을 사용할 수 있다.  
객체나 함수도 사용할 수 있다.
- 맵의 프로퍼티는 순서대로 접근하고 처리할 수 있다.

```
기본형  new Map()    // Map 객체를 만듭니다.  
        set(키, 값)  // Map 객체에 프로퍼티를 추가합니다.
```

```
let bag = new Map()           // Map 객체의 인스턴스인 bag을 만든다.  
bag.set("color", "red")      // {"color" => "red"}
```

# 맵

Map 객체를 만들면서 프로퍼티를 지정할 수도 있다

```
기본형 new Map()([
    [키1, 값1],
    [키2, 값2],
    :
]);
```

```
let myCup = new Map ([
    ["color", "white"],
    ["haveHandle", true],
    ["material", "ceramic"],
    ["capacity", "300ml"]
])
```

myCup

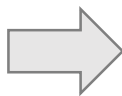
```
// {"color" => "white", "haveHandle" => true, "material" => "ceramic", "capacity" =>
"300ml"}
```

# 맵 체이닝

```
let bag = new Map()  
bag.set("color", "red")
```

← 앞에서 만들었던 bag

```
bag.set("type", "mini")  
bag.set("purpose", "daily")
```



```
bag.set("type", "mini").set("purpose",  
"daily")
```

맵의 메서드는 맵을 반환한다.

bag.set("type", "mini").

bag.set("purpose", "daily")

# 맵의 프로퍼티와 메서드

**기본형**

<code>size</code>	// 맵 요소의 개수를 알려 주는 프로퍼티
<code>set(키, 값)</code>	// 프로퍼티 추가를 추가합니다.
<code>get(키)</code>	// 해당 키의 값을 반환합니다.
<code>has(키)</code>	// 해당 키가 맵에 있는지 체크하고 true 또는 false로 반환합니다.
<code>delete(키)</code>	// 해당 키가 있는 프로퍼티를 삭제합니다.
<code>clear()</code>	// 맵의 모든 요소를 삭제합니다.

```
bag.size
bag.get("color")    // "red"
bag.has("color")    // true
bag.delete("type")  // true
bag.delete("name")  // false
bag                // {"color" => "red", "purpose" => "daily"}
bag.clear()
bag                // {}
```

# 맵의 프로퍼티와 메서드

맵에서 키와 값을 가져오는 메서드

```
keys()      // 각 요소의 키를 모아서 반환합니다.  
values()    // 각 요소의 값을 모아서 반환합니다.  
entries()   // [키, 값] 형태로 모든 요소를 반환합니다.
```

```
let myCup = new Map ([  
  ["color", "white"],  
  ["haveHandle", true],  
  ["material", "ceramic"],  
  ["capacity", "300ml"]  
])  
myCup.keys()  
// MapIterator {"color", "haveHandle", "material", "capacity"}
```

- 맵은 순서를 가지고 있는 객체 → 이터러블 객체
- 키나 값을 가져오는 메서드는 이터러블 객체를 반환한다.

```
for(let key of myCup.keys()) {  
  console.log(key)  
}
```

# 셋

- 배열은 키 없이 여러 개의 값을 모아 놓은 것으로, 값이 중복되어도 상관없다.
- 셋은 키 없이 여러 개의 값을 모아 놓은 것은 배열과 같지만, 값이 중복되지 않는다.

Set 객체의 인스턴스를 만든 후 값을 추가한다.

기본형    `new Set()`  
          `new Set(배열)`

기본형    `add(값)`

```
let numSet1 = new Set()
numSet1.add("one")    // {"one"}
numSet1.add("two")    // {"one",
                       "two"}
```



체이닝해서 작성하면?

```
let numSet1 = new Set().add("one").add("two")
```



# 셋

배열을 인수로 받아서 셋으로 만들 수도 있다.

```
let numSet2 = new Set([1, 2, 3])  
numSet2    // {1, 2, 3}  
let numSet3 = new Set([1, 2, 1, 3, 1, 5])  
numSet3    // {1, 2, 3, 5}
```

중복 값이 있는 배열을 받아도 중복값을 모두 제거하고 셋을 만든다

# 셋의 프로퍼티와 메서드

**기본형**

size	// 셋 요소의 개수를 반환합니다.
add(값)	// 셋에 값을 추가합니다.
has(값)	// 셋에 해당 값이 있는지 체크합니다.
delete(값)	// 셋에서 해당 값을 삭제합니다.
clear()	// 셋을 비웁니다.

예) 강의실에 출석 체크하기 위해 입장하는 학생 이름을 저장할 경우  
→ 학생이 잠시 나갔다가 다시 강의실에 들어와도 학생 이름을 2번 저장할 필요가 없으므로  
이 경우에는 배열보다 셋이 적합하다.

```
let students = new Set();
students.add("도레미")
students.add("백두산")
students.add("도레미")
students      // {"도레미", "백두산"}
```

# 셋의 프로퍼티와 메서드

```
기본형 keys()      // 셋에 있는 모든 값을 반환합니다.  
        values()   // 셋에 있는 모든 값을 반환합니다.  
        entries()  // [값, 값] 형식으로 모든 값을 반환합니다.
```

keys(), values(), entries() 메서드는 이터러블 객체를 반환한다.

```
students.keys()      // {'도레미', '백두산'}  
students.values()    // {'도레미', '백두산'}  
students.entries()   // {'도레미' => '도레미', '백두산' => '백두산'  
'}'
```

---

# 이터레이터와 제너레이터

---

# 이터러블 객체

- 이터러블 객체에서 이터러블(iterable)이란, '순서대로 처리할 수 있다'는 뜻
- 예를 들어, 배열은 인덱스와 값을 가지고 있으므로 인덱스 0부터 차례대로 값을 가져와서 처리할 수 있기 때문에 이터러블 객체이다.
- 문자열과 배열, 맵, 셋이 이터러블 객체이다

이터러블 객체에서는 다음과 같은 기능을 사용할 수 있다.

- for...of 반복문
- 전개 연산자(...)
- 구조 분해 할당

# 이터러블 객체

(예) 문자열

```
let hi = "hello"

// for...of로 반복
for(let ch of hi){
  console.log(ch)
}

// 전개 연산자 사용
let chArray = [...hi]
chArray // ["h", "e", "l", "l", "o"]

// 구조 분해 할당 사용
let [ch1, ch2] = hi
ch1 // "h"
ch2 // "e"
```

일반 객체는 이터러블하지 않다.

객체 안에 많은 자료를 저장하고 처리해야 하기 때  
문에 전개 연산자나 구조 분해 할당을 사용하거나  
for...of문으로 순회하는 것이 편리하다

→ 일반 객체를 이터러블하게 만들어서 사용한다  
(제너레이터 함수 사용)

# 이터러블 객체

콘솔 창에서 배열을 만든 후, 배열의 프로퍼티, 메서드를 확인해 본다.

```
▶ toLocaleString: f toLocaleString()  
▶ toString: f toString()  
▶ unshift: f unshift()  
▶ values: f values()  
▶ Symbol(Symbol.iterator): f values()  
▶ Symbol(Symbol.unscopables): {copyWithin: true, entries: true, fill: true, fin  
▶ [[Prototype]]: Object  
>
```

Symbol(Symbol.iterator): f values()



이터러블 객체에는 Symbol.iterator 메서드가 포함되어 있다.

Symbol.iterator 메서드를 실행하면 Iterator 객체가 반환된다.

# Iterator 객체

콘솔 창에서 배열을 만든 후, 배열의 프로퍼티, 메서드를 확인해 본다.

```
▶ toLocaleString: f toLocaleString()  
▶ toString: f toString()  
▶ unshift: f unshift()  
▶ values: f values()  
▶ Symbol(Symbol.iterator): f values()  
▶ Symbol(Symbol.unscopables): {copyWithin: true, entries: true, fill: true, find: true, findIndex: true, findLast: true, findLastIndex: true, flat: true, flatMap: true, from: true, fromEntries: true, fromIterables: true, fromKeys: true, fromValues: true, groupBy: true, include: true, join: true, keys: true, lastIndexOf: true, map: true, mapValues: true, maxBy: true, maxOf: true, minBy: true, minOf: true, of: true, omit: true, omitBy: true, orderBy: true, pad: true, padEnd: true, padStart: true, partition: true, pick: true, pickBy: true, reduce: true, reduceFrom: true, reduceRight: true, reduceRightFrom: true, repeat: true, repeatElement: true, sample: true, sampleSize: true, shuffle: true, slice: true, some: true, someFrom: true, sort: true, sortBy: true, splice: true, split: true, splitFrom: true, sum: true, sumBy: true, sumOf: true, take: true, takeFrom: true, takeLast: true, takeLastFrom: true, takeWhile: true, takeWhileFrom: true, times: true, timesFrom: true, timesLast: true, timesLastFrom: true, timesWhile: true, timesWhileFrom: true, toInteger: true, toIntegerFrom: true, toIntegerLast: true, toIntegerLastFrom: true, toIntegerWhile: true, toIntegerWhileFrom: true, toLocaleString: true, toString: true, unshift: true, values: true, Symbol(Symbol.iterator): true, Symbol(Symbol.unscopables): true, [[Prototype]]: Object
```

Symbol(Symbol.iterator): f values()

이터러블 객체에는 Symbol.iterator 메서드가 포함되어 있다. → '이터러블 프로토콜' 이라고 한다

Symbol.iterator 메서드를 실행하면 Iterator 객체가 반환된다.

```
let arr = [1, 2, 3, 4, 5]  
let it =  
arr[Symbol.iterator]()
```



# Iterator 객체

이터레이터 객체란 객체의 요소를 순서대로 꺼낼 수 있는 객체

next() 메서드가 있기 때문에 가능

next() 메서드는 value와 done을 반환한다. value – 다음 값, done – 이터레이터 객체가 끝났는지 여부

```
let arr = [1, 2, 3, 4, 5]
let it = arr[Symbol.iterator]()
it.next()    // {value: 1, done: false}
it.next()    // {value: 2, done: false}
it.next()    // {value: 3, done: false}
it.next()    // {value: 4, done: false}
it.next()    // {value: 5, done: false}
it.next()    // {value: undefined, done: true}
```

# 제너레이터 함수

- 일반 객체를 이터러블하게 만들기 위해 사용하는 함수
- 일반 함수와 구별하기 위해 function 다음에 \*기호를 붙여서 작성하고 함수 안에 return 문 대신 yield 문을 사용한다.

## 함수 정의하기

```
기본형  function* 함수명() {  
    :  
    yield  
}
```

```
function* gen() {  
    yield 1;  
    yield 2;  
    yield 3;  
}
```

## 객체 만들기

```
기본형  객체명 = 함수명()
```

```
let g1 = gen()  
  
g1.next()    // {value: 1, done:  
false}  
  
g1.next()    // {value: 2, done:  
false}  
  
g1.next()    // {value: 3, done:  
false}  
  
g1.next()    // {value: undefined,
```

# 제너레이터 함수

## 객체 만들기

기본형    객체명 = 함수명()

next() 메서드 사용

```
let g1 = gen()

g1.next()    // {value: 1, done:
false}

g1.next()    // {value: 2, done:
false}

g1.next()    // {value: 3, done:
false}

g1.next()    // {value: undefined,
```

for...of 사용

```
let g2 = gen()

for(let i of g2) console.log(i)
```

```
let g3 = gen()

g3.next()    // {value: 1, done: false}

for(let i of g3) console.log(i)
```