

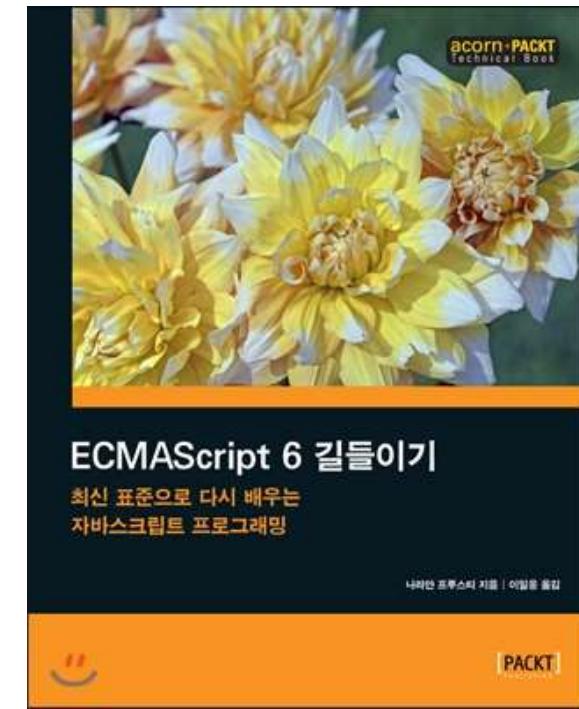
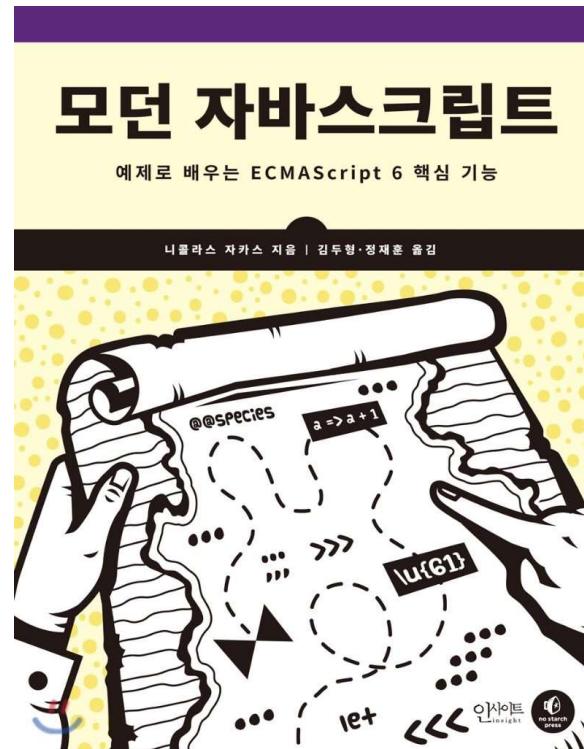
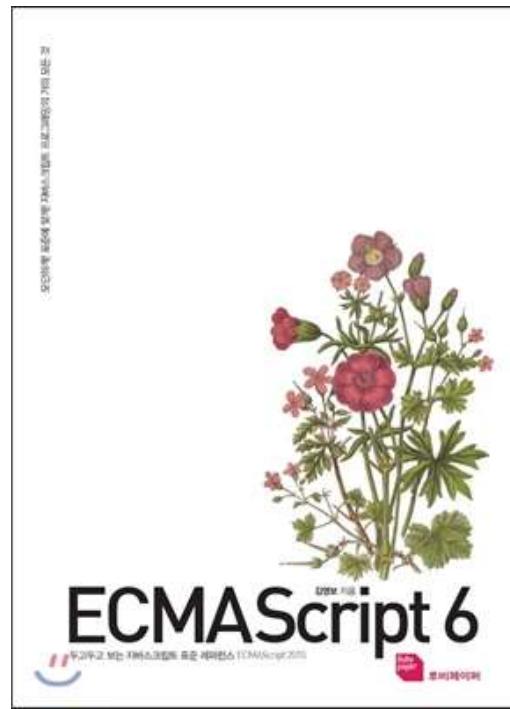
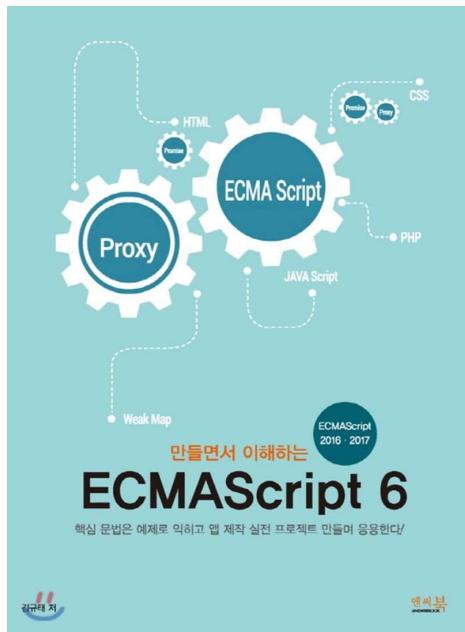
ES6 Fundamentals

Bok, Jong Soon

javaexpert@nate.com

<https://github.com/swacademy/React>

Reference Books



Introduction to ECMAScript 6

- 현재의 공식적인 최신 Version
- 현재까지 공식적으로 발표된 Version
 - ECMAScript 1, 2, 3, 5, 6
 - Version 4는 폐기되었음.
- ES 2016과 ES 2017은 ES6에 비해 큰 변화가 없는 Version
- [ECMA-262](#)



Hello ES6

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <title>Hello ES6</title>
8  </head>
9  <body>
10     <script>
11
12         let subject = 'ES6';
13         let str = `오늘의 주제는 ${subject}입니다.`;
14         console.log(str);    //오늘의 주제는 ES6입니다.
15     </script>
16 </body>
17 </html>
```

기본 문법



기본문법

■ **let & const**

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

■ *iterable* protocol & *iteration* protocol

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols

■ **for...of Statement**

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of>

■ *Template Literal*

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

■ **TypedArray**

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray

let

■ **var** 변수의 문제점

- 선언문의 생략
- 중복된 변수명 선언의 가능
- 함수 Hoisting
- 개발에 혼란
- 가독성의 떨어짐

let (Cont.)

■ **let**은 **var**와 다르게 Block에서 Scope가 설정된다.

- **var**는 함수 Block에서 Scope가 설정되지만, 그 외 Block에서는 Scope가 설정되지 않아서 변수가 공유된다.

```
Function(){  
    var scope 범위  
    {  
        let scope 범위  
    }  
}
```

```
var a = 100; // 변수 a 선언  
  
function f(){  
    var a = 200; // 함수 Block안에서 같은 변수명 a를 선언  
    console.log(a); // 200  
}  
console.log(a); // 100  
  
var a = 100; // 변수 a 선언  
  
if(a > 0){  
    var a = 200; // 같은 이름의 a를 선언  
    console.log(a); // 200  
}  
console.log(a); // 200
```

let (Cont.)

- **let**은 **var**와 다르게 Block에서 Scope가 설정된다.

- **var**는 함수 Block에서 Scope가 설정되지만, 그 외 Block에서는 Scope가 설정되지 않아서 변수가 공유된다.

```
let a = 100; // 변수 a 선언

if(a > 0){
    let a = 200; // 같은 이름의 a를 선언
    console.log(a); // 200
}
console.log(a); // 100
```

let (Cont.)

- **var**는 반복문 안에서 변수가 공유되는 문제가 있었는데, **let**으로 이를 개선했다.
 - 이는 반복문 안에 비동기 함수를 호출할 경우 문제가 발생할 수 있다.

```
Function(){  
    for(){  
        let scope 범위  
    }  
}
```

```
for(var i = 0; i < 10 ; i++){  
    setTimeout(function(){  
        console.log(i); // 모두 9  
    }, 100);  
}
```

let (Cont.)

- **var**는 반복문 안에서 변수가 공유되는 문제가 있었는데, **let**으로 이를 개선했다.
 - 이는 반복문 안에 비동기 함수를 호출할 경우 문제가 발생할 수 있다.

```
for(let i = 0; i < 10 ; i++){
    setTimeout(function(){
        console.log(i);    //0,1,2,3...
    }, 100);
}
```

let (Cont.)

- **let**은 같은 Scope 내에서 변수 중복 선언이 불가능하다.
 - **var**는 같은 Scope내에서 변수 중복 선언할 때 이전에 선언된 변수가 덮어씌워지지만, **let**은 이를 허용하지 않는다.
 - 변수 중복 선언시 *SyntaxError* 발생

```
function f(){
    let a = 100;
    let a = 200; //SyntaxError 발생
}
```

let (Cont.)

■ **let**은 함수 끌어올림(Hoisting)이 되지 않는다.

- **var**는 함수 Hoisting이 되어 아래의 상황에서도 Error가 발생하지 않는다.

함수선언문이 끌어올려짐

foo();

function foo(){}

```
function f(){
    console.log(a);    //Error 발생하지 않음. undefined
    var a = 100;
}
f();
```

함수 Hoisting : JavaScript가 실행될 때, 변수 선언문이나 함수 선언문을 읽기 전에 선언된 변수와 함수들을 다른 무엇보다도 먼저 읽어 Scope의 최상위에 위치시킴.

let (Cont.)

- **let**은 함수 끌어올림(Hoisting)이 되지 않는다.
 - **var**는 함수 Hoisting이 되어 아래의 상황에서도 Error가 발생하지 않는다.

```
function f(){
    console.log(a);    //Error 발생
    let a = 100;
}
f();
```

const

- 상수 선언문이 추가됨.
- 반드시 초기값 할당해야.
- 한번 선언된 값은 변경될 수 없는 불변(*Immutable*) 값이다.
- 상수명의 표기는 대체적으로 대문자만 사용
- 단어 사이에 구분을 위해 Underscore(_) 사용.
- **const**는 **let**과 같은 Scope 설정 규칙을 갖는다.
- **const**는 중복 선언과 함수 Hoisting이 되지 않는다.

```
const MY_NAME;      //SyntaxError 발생
const MY_NAME = 'Sujan';
MY_NAME = 'Smith'; //TypeError 발생
```

let 과 const 정리

	var	let	const
Scope	함수	Block	Block
Scope내 중복 선언	가능	불가능	불가능
Hoisting	일어남	일어나지 않음	일어나지 않음
값 변경	가능	가능	불가능



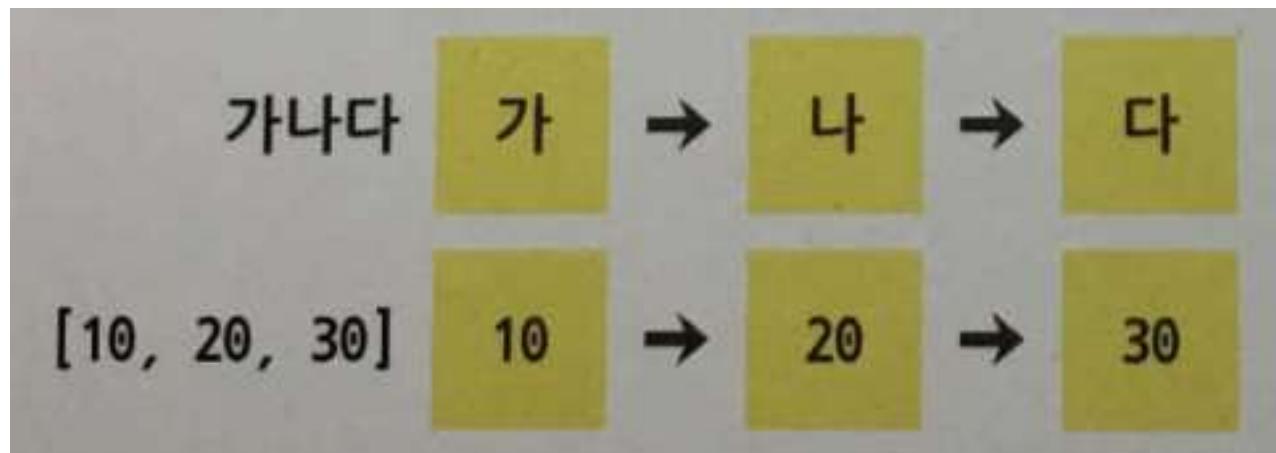
Lab. let & const



iterable Protocol and Iterable Object

■ *iterable* Protocol

- ES6에서 새로 추가된 `for...of` 문을 실행하여 반복될 때 값이 열거
- 내부적으로 `@@iterator` Method (`Symbol.iterator()`) 가 구현되어 있어야 하는 규약(Protocol)
- JavaScript 객체 중 `Array`, `String`, `Map`, `Set`, `arguments`
- `Object` 객체는 제외



iterable Protocol and Iterable Object (Cont.)

■ *iterable* Protocol

- **String** Iteration

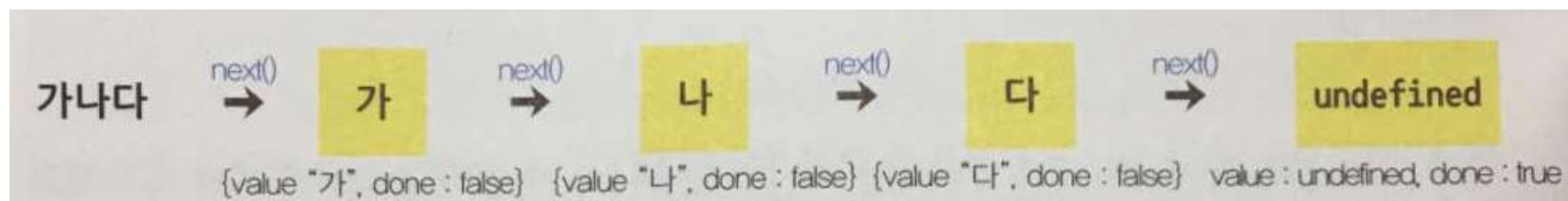
```
let str = '가나다';
for(let value of str){
    console.log(value); // '가', '나', '다'
```

- **Array** Iteration

```
let array = [10, 20, 30];
for(let value of array){
    console.log(value); // 10, 20, 30
```

iterable Protocol and Iterable Object (Cont.)

- *iterator* Protocol은 *iterable* Protocol과 같이 값이 열거 되지만, `next()`를 통해서 하나씩 순차적으로 열거되어야 한다.
- 이때 열거되는 값의 형태는 객체이며 속성으로 `value`와 `done`을 갖는다.
- `value`는 실제 값이 할당
- `done`은 열거의 끝임을 알려준다.
 - 열거가 끝인 경우 `true`
 - 그렇지 않을 경우 `false`
- **Iterable** 객체 : *iterator* 규약을 따르는 객체
- 직접 구현하거나 `@@iterator` method를 통해서 전달받을 수 있다.



iterable Protocol and Iterable Object (Cont.)

- 다음 code는 배열에서 `@@iterator` method를 호출하여 `iterable` 객체를 전달받은 예이다.

```
let array = [1,2,3];
// 내장된 @@iterator Method를 호출하여 Iterator 객체를 전달받음.
let iterator = array[Symbol.iterator]();
iterator.next(); // {value:1, done:false}
iterator.next(); // {value:1, done:false}
iterator.next(); // {value:1, done:false}
iterator.next(); // {value:undefined, done:true}
```

iterable Protocol and Iterable Object (Cont.)

- 다음 code는 **Iterable** 객체를 직접 구현한 예이다.

```
let iterator = {
    i : 1,
    next : function(){
        return (this.i < 4) ? {value : this.i++, done:false} :
            {value : undefined, done:true};
    }
}
iterator.next(); // {value:1, done:false}
iterator.next(); // {value:1, done:false}
iterator.next(); // {value:1, done:false}
iterator.next(); // {value:undefined, done:true}
```

iterable Protocol and Iterable Object (Cont.)

정의	규약에 따르는 객체
iterable protocol for...of 문을 통해 열거되어야 하고, @@iterator method를 구현.	JavaScript 내장 객체 중 Array, String, Map, Set, arguments 등.
iterator protocol next() method 호출 시 순차적으로 열거되며, 열거된 값이 객체(value값, done:열거 완료 여부)이어야 한다.	iterator protocol을 따르도록 구현하거나, Iterable 객체로부터 @@iterator method를 호출하여 참조 가능.



Lab. iterable Protocol & Iterable Object



for...of Statement

- 기존에 배열이나 함수의 **arguments** 객체와 같은 Collection을 순회하는 **for...in**문이나 **forEach()** 함수와 같은 역할을 한다.
- 문자열을 한 글자 씩 잘라서 순회하거나 destructing 등이 가능하다.
- 이를 위해 **iterable** Protocol을 따라야 한다.
- 따라서, **for...of** 문으로 순회하려면 **@@iterator** method를 내장한 객체 이거나, 직접 **@@iterator** method를 구현해야 한다.
- **for...of** 문의 작성법은 아래와 같다.

```
for (variables of iterable) {
```

...

}

for...of Statement (Cont.)

- 문자열이 `@@iterator` method가 구현이 되어 있는지 확인해 보자.
- `@@iterator` Method 호출 시 `Iterable` 객체를 반환하므로 Type은 객체이어야 한다.

```
let str = 'for of Statement';
console.log(typeof str[Symbol.iterator]() === 'object');
// true
```

- 문자열이 `iterable` Protocol을 따르는 것이 확인됐으면, `for...of`문으로 순회 가능하다고 볼 수 있다.

```
let str = 'for of Statement';
for(let value of str){
    console.log(value); //f,o,r, ,o,f,s...
}
```

for...of Statement (Cont.)

- **for...in** 문은 배열 순회시 문제점을 가지고 있다.
 - 배열에 속성을 추가하는 경우 속성도 순회할 때 포함한다.

```
var array = [10,20,30];
array.add = 100;
for(var i in array){
    console.log(i); //0,1,2,add
}
```

for...of Statement (Cont.)

■ **for...in** 문은 배열 순회시 문제점을 가지고 있다.

- 배열객체의 속성명을 문자열로 알려주기 때문에 원소의 index + 1과 같은 연산할 때 문자열로 된다.

```
var array = [1,2,3];
for(var i in array){
    console.log(i + 1);    //01, 11, 21
}
```

for...of Statement (Cont.)

- **for...of**문은 이러한 문제점들을 개선하여 배열 순회시에 직관적으로 원소의 값만 전달한다.

```
let array = [10, 20, 30];
array.add = 100;

for(let value of array){
    console.log(value);    //10 ,20, 30
}
```

for...of Statement (Cont.)

Array 순회 시 문제점

for...of Statement	Array 순회시 속성을 포함하지 않고, 원소만 전달하여, for...in 문의 단점을 보완함
for...in Statement	Array 순회시 속성을 포함하여 명확하지 않음. 순회시 전달 값이 Array의 원소가 아닌 index



Lab. for...of Statement



Template Literal

- 문자열 안에 표현식을 포함시킬 수 있고, 여러 줄 작성은 허용하여 간편하게 문자열을 만들 수 있도록 해준다.
- 문자열과 다르게 따옴표 대신 역따옴표(back-tick, ``) 문자 사이에 작성
- **\$ { }**를 포함할 수 있다.
- **\$ { }** 사이에 표현식을 쓸 수 있다.
- 표현식의 결과는 문자열로 연결된다.
- *Template Literal* 앞에 함수명(Tag 표현식)이 있으면 함수를 호출한다.
- 이 때 *Template Literal*의 값이 함수에 전달되며, 함수에서 값을 조작하여 *Template Literal*을 출력할 수 있다. → *Tagged template literal*

Template Literal (Cont.)

■ 여러 줄 문자열

- 문자열을 여러 줄로 작성하려면 `\n`을 입력해야 했다.
- 또는 `+` 연산자 사용
- *Template Literal*은 `+`연산자 없이 여러 줄 작성이 가능
- 줄 바꿈시 자동으로 `\n`문자가 입력된다.

■ 일반 문자열 여러 줄 작성할 때

```
var str = "여러 줄\n 입력 테스트";
console.log(str);
```

■ 일반 문자열 여러 줄 작성할 때 Code 줄 바꿈.

```
var str = "여러 줄\n";
str += " 입력 테스트";
console.log(str);
```

Template Literal (Cont.)

- *Template Literal* 여러 줄 작성

```
let str = `여러 줄  
입력 테스트`;  
console.log(str);
```

Template Literal (Cont.)

■ 보간 표현법

- 일반 문자열에 표현식을 삽입하려면 문자열을 끝맺음하고 **+**연산자로 표현식을 연결하여 작성해야 했다.
- **Template Literal**은 문자열 끝맺음없이 보간 표현법을 이용하여 보다 쉽게 작성이 가능하다.

■ 일반 문자열에 표현식 포함

```
var a = 100;  
var b = 200;  
var str = "a + b의 결과는 " + (a + b) + " 입니다.";  
console.log(str);
```

Template Literal (Cont.)

■ 보간 표현법

- 일반 문자열에 표현식을 삽입하려면 문자열을 끝맺음하고 **+**연산자로 표현식을 연결하여 작성해야 했다.
- **Template Literal**은 문자열 끝맺음없이 보간 표현법을 이용하여 보다 쉽게 작성이 가능하다.

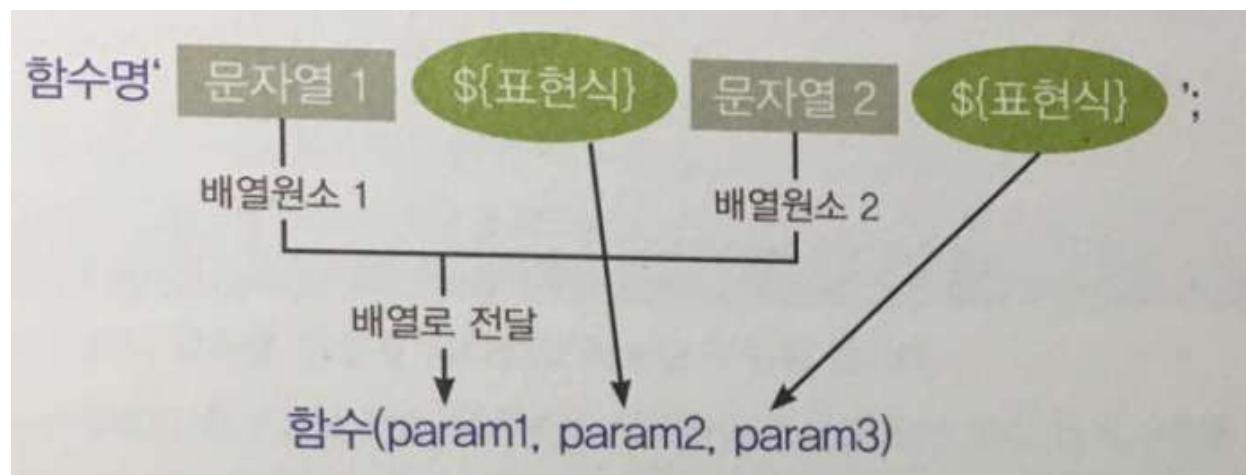
■ Template Literal에 표현식 포함

```
let a = 100;
let b = 200;
let str = `a + b의 결과는 ${a + b} 입니다.`;
console.log(str);
```

Template Literal (Cont.)

■ *Tagged Template Literal*

- 표현식(함수명)옆에 *Template Literal*이 올 경우 함수를 호출한다.
- 함수의 인수로 *Template Literal*이 전달되며, 보간 표현법이 있는 경우 보간 표현법을 앞 뒤로 나누어 문자열이 배열로 전달된다.
- 보간 표현법의 표현식의 값은 따로 인수에 전달된다.



Template Literal (Cont.)

■ *Tagged Template Literal*

- 표현식(함수명)옆에 *Template Literal*이 올 경우 함수를 호출한다.
- 함수의 인수로 *Template Literal*이 전달되며, 보간 표현법이 있는 경우 보간 표현법을 앞 뒤로 나누어 문자열이 배열로 전달된다.
- 보간 표현법의 표현식의 값은 따로 인수에 전달된다.

```
function tagged(str, a, b){  
    let bigger;  
    (a > b) ? bigger = 'A': bigger = 'B';  
  
    return str[0] + bigger + '가 더 큽니다.';  
}  
  
let a = 100;  
let b = 200;  
let str = tagged`A와 B 둘 중 ${a}, ${b}`;  
console.log(str);
```

Template Literal (Cont.)

	작성법	표현식 작성	여러 줄 작성
문자열	쌍따옴표("") 또는 홀따옴표('') 사이에 작성	+연산자를 사용하여 작성	문자열 줄 바꿈시에는 줄 바꿈 문자(\n)를 사용하여 작성하며, 코드를 여러 줄로 작성 시에는 (+)연산자를 사용하여 작성
Template Literal	역따옴표(``) 사이에 작성	보간 표현법을 사용하여 작성	문자열 줄 바꿈 또는 여러 줄 작성 시 내려쓰기하여 작성

Lab. Template Literal



TypedArray Object

- Describes an array-like view of an underlying binary data buffer.
- Binary data를 보다 빨리 접근하고 조작하도록 하기 위해 추가되었음.
- File 처리, Audio 및 Video 처리를 위한 binary data.
- Array 객체 API를 동일하게 제공하지만, **push**나 **pop** 등의 API 제공 안함.
- 버퍼(**ArrayBuffer**)와 뷰(Typed array views)로 나누임.
- 버퍼(**ArrayBuffer**)는 단순히 data chunk를 나타내는 객체이며, 스스로 읽고 쓸 수 없고 view를 통해서 저장된 data를 조작 가능.

TypedArray Object (Cont.)

■ ArrayBuffer

- 생성자 호출시 지정한 byte 크기의 buffer 생성.
- 직접적으로 data의 조작이 불가능.
- 특정 type의 view 생성자 객체를 통해 data를 읽거나 쓰기 가능

```
const buffer = new ArrayBuffer(16); //16Byte buffer 생성  
console.log(buffer.byteLength); //16
```

TypedArray Object (Cont.)

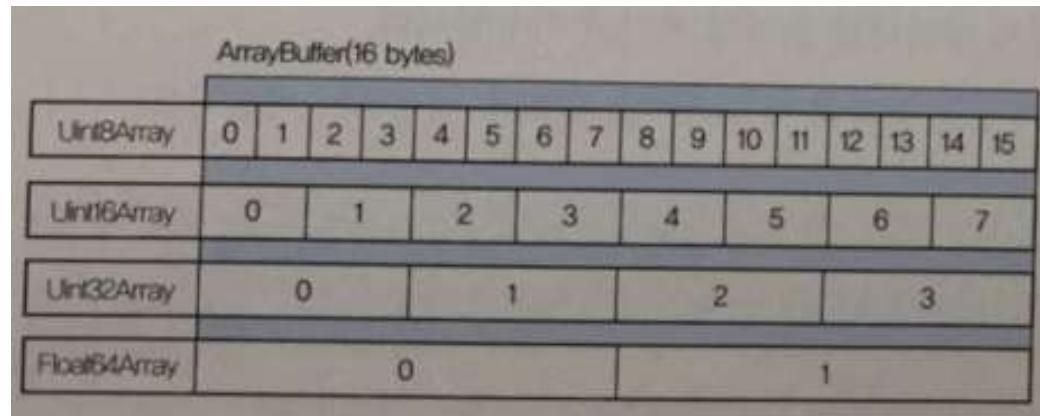
■ Typed array views

	크기(Byte)	값의 범위	설명
Int8Array	1	-128 ~ 127	8 bit 정수형
Uint8Array	1	0 ~ 255	8 bit 양의 정수형
Uint8ClamedArray	1	0 ~ 255	8 bit 양의 정수형(0 ~ 255사이의 숫자만 허용)
Int16Array	2	-32768 ~ 32767	16 bit 정수형
Uint16Array	2	0 ~ 65535	16 bit 양의 정수형
Int32Array	4	-2147483648 ~ 2147483647	32 bit 정수형
Uint32Array	4	0 ~ 4294967295	32 bit 양의 정수형
Float32Array	4	1.2×10^{-38} ~ 3.4×10^{38}	32 bit 부동소수점형
Float64Array	8	5.0×10^{-324} ~ 1.8×10^{308}	64 bit 부동소수점형

TypedArray Object (Cont.)

■ Typed array views

- Class 호출시 지정한 buffer의 byte 만큼 담을 수 있는 배열형태의 생성자 생성
- Class 이름의 bit에 따라 원소의 수가 결정
- 초기값으로 0을 지정



```
const buffer = new ArrayBuffer(16); // 16Byte buffer 생성
const view = new Uint32Array(buffer); // 양의 정수형 32bit view 선언하고 buffer 지정
console.log(view); // Uint32Array(4)[0, 0, 0, 0]
```

TypedArray Object (Cont.)

■ 공통 메소드

- `copyWithin()`
- `entries()`
- `fill()`
- `filter()`
- `find()`
- `findIndex()`
- `forEach()`
- `indexOf()`
- `join()`
- `keys()`
- `lastIndexOf()`
- `map()`
- `reduce()`
- `reduceRight()`
- `reverse()`
- `slice()`
- `some()`
- `sort()`
- `values()`

TypedArray Object (Cont.)

- 일반 배열에서 이용할 수 없는 메소드

- `concat()`
- `pop()`
- `push()`
- `shift()`
- `splice()`
- `unshift()`

- 새로 추가된 메소드

- `set()`
- `subarray()`

Lab. TypedArray



내장 객체



내장 객체

■ Generator

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator

■ 새로 추가된 Collection - Map, Set, WeakMap, WeakSet

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet

■ Symbol

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol

■ Promise

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

■ Proxy

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy

Generator

- 하나의 값만 반환하는 것이 아니라 한번에 하나씩 여러 값을 반환하는 함수
- **function***는 keyword
- 위의 keyword를 사용한 함수
- **generator()**로부터 반환된 값.
- **iterable** protocol과 **iterator** protocol을 따른다.
- 위의 2 protocol의 규약을 준수하는 객체는 **@@iterator()**와 **next()**를 구현해야 하는데, 이를 작성하기가 쉽지 않다.
- 이를 좀 더 쉽게 구현하도록 만든 함수.

```
function* gen() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
var g = gen();  
console.log(g); // "Generator { }"
```

Generator (Cont.)

- `generator()` 는 호출되면 `generator` 객체를 반환하고 동작을 정지한다.
- 이때 반환된 `generator` 객체에 `next()` 를 호출하면 `generator()` 의 구문이 실행된다.
- `yield` 표현식을 만나면 실행을 멈춘다.
- 그 때 표현식이 가리키는 값이 `next()` 가 반환하는 객체의 `value` 속성값이 되고, `done` 속성값은 `false`가 된다.
- 다시, `next()` 호출이 되면 `iterator` 객체와 같이 순환된다.

Generator (Cont.)

- *iterable* protocol / *iterator* protocol을 준수하는 **Iterable** 객체를 작성한 code와 **generator()**를 작성한 code 비교

```
<script>

let iterator = {
    i : 0,
    [Symbol.iterator] : function(){
        return this;
    },
    next : function(){
        return (this.i < 3) ? {value:this.i++, done:false} :
            {value : undefined, done:true}
    }
}
for(let value of iterator){
    console.log(value); //0, 1, 2
}

</script>
```



```
<script>

function* gen() {
    for (let i = 0; i < 3; i++) {
        yield i;
    }
}

let generator = gen();

for (let value of generator) {
    console.log(value); //0, 1, 2
}

</script>
```

Generator (Cont.)

■ `return()`

- Generator 객체의 열거를 종료

```
function* gen(){
  let i = 0;
  while(i >= 0) yield i++;
}

let generator = gen();
console.log(generator.next()); // {value: 0, done: false}
console.log(generator.next()); // {value: 1, done: false}
console.log(generator.next()); // {value: 2, done: false}
console.log(generator.return()); // {value: undefined, done: true}
console.log(generator.next()); // {value: undefined, done: true}
```

Generator (Cont.)

■ **throw()**

- Generator 객체의 열거 중 강제로 오류를 발생시킴.

```
function* gen(){
  let i = 0;
  while(i >= 0) {
    try{
      yield i++;
    }catch(e){
      console.log(e);
    }
  }
}

let generator = gen();
console.log(generator.next()); // {value : 0, done : false}
console.log(generator.next()); // {value : 1, done : false}
console.log(generator.next()); // {value : 2, done : false}
console.log(generator.throw()); // undefined 출력후, {value : 3, done : false}
console.log(generator.next()); // {value : 4, done : false}
```

Generator (Cont.)

	iteration	iteration 종료 또는 오류 발생
generator	yield 표현식을 사용	iteration을 종료시키는 return()와 오류를 발생시키는 throw()를 제공
iterator	@@iterator() 구현	별도로 제공하는 method가 없어서 직접 구현해야 함.

Lab. Generator Object



새로 추가된 Collection - Map

- key / value 쌍(pair), 항목(entries)으로 이루어진 Collection.
- 기존에도 key와 value로 이루어진 Collection 객체가 이미 존재했었다.
- 하지만, Map은 몇 가지 불편한 사항들을 개선했다.

Map (Cont.)

■ Object와 Map의 차이점

- Object는 추가된 속성의 수를 정확히 알기 어렵다.
- Map은 size 속성으로 추가된 항목의 수를 알 수 있다.

`map.size`

- Object는 속성 추가 시 내장 속성과 중복으로 사용하지 않도록 주의해야.

```
var obj = {}  
obj.toString();      //"[object Object]"
```

```
obj.toString = function() {};
```

```
obj.toString();    //undefined, 내장 속성이 덮어 씌워짐
```

- Map은 이를 방지하기 위해 set으로 값을 저장하고 get으로 읽어온다.

```
map.set(key,value);  
map.get(key);      //내장 속성과 충돌할 염려가 없다.
```

- Object는 iterable Protocol을 따르지 않지만 Map은 따른다.

- Object는 for...of 사용하지 못하지만, Map은 사용 가능

Map (Cont.)

■ Map Properties

- **size** : Map에 추가된 항목 수

Map (Cont.)

■ Map Methods

- **set(key, value)** : Map에 새로운 항목 추가하고 Instance 반환
- **get(key)** : key를 갖는 항목의 value 값 반환
- **clear()** : Map의 항목 모두 삭제
- **delete(key)** : key를 갖는 항목만 삭제, **true**(존재할경우) / **false**(존재하지 않을 경우)
- **entries()** : 추가된 항목 열거할 수 있도록 **iterator** 객체 반환
- **forEach(callbackFn)** : Map에 추가된 항목 순회
- **has(key)** : **true**(key를 갖는 항목 존재)/**false**(존재하지 않을 경우) 반환
- **keys()** : key들을 열거할 수 있도록 **iterator** 객체 반환
- **values()** : value들을 열거할 수 있는 **iterator** 객체 반환
- **[@@iterator]()** : 항목들을 열거할 수 있도록 **iterator** 객체 반환, **entries()**와 동일

Map (Cont.)

■ set(key, value)

- Key는 항목을 구분하는 역할
- 객체와 달리 모든 type 사용 가능

```
let obj = {};
let f = function() {};
let map = new Map();

map.set(obj, 100);
console.log(map.size);    //1

map.set(f, 200);
console.log(map.size);    //2
```

Map (Cont.)

■ **set(key, value)**

- 호출 뒤에 **Map** instance를 반환하기 때문에 다음과 같은 구문의 사용이 가능.

```
map.set('a', 100).set('b', 200);
```

Map (Cont.)

■ get(key)

- 추가된 항목 중 key 인자와 일치하는 key를 갖는 항목의 value 반환.

```
let obj = {};
let map = new Map();

map.set(obj, 100);
console.log(map.get(obj)); // 100
```

Map (Cont.)

■ clear()

- 추가된 모든 항목 삭제

```
let map = new Map();

map.set('a', 100).set('b', 200);
console.log(map.size);    // 2

map.clear();
console.log(map.size);    // 0
```

Map (Cont.)

■ **delete(key)**

- 인자 key와 일치하는 항목 삭제

```
let map = new Map();
map.set('a', 100).set('b', 200);

map.delete('b');
console.log(map.get('b'))); //undefined
```

Map (Cont.)

■ entries()

- Map의 항목을 열거할 수 있는 iterator 객체 반환
- iterator 객체에 next() 호출 시 반환되는 객체의 value 속성값은 Map의 항목을 원소로 하는 배열([key, value])가 된다.

```
let map = new Map();
map.set('a', 100).set('b', 200);

let mapIter = map.entries();
console.log(mapIter.next()); // {value: Array(2), done: false}
console.log(mapIter.next()); // {value: Array(2), done: false}
console.log(mapIter.next()); // {value: undefined, done: true}
```

Map (Cont.)

■ forEach(callbackFn)

- Map 항목 순회.
- 인수인 Callback 함수로 value와 key 그리고 Map을 전달
- **주의할 점은 전달 순서가 value, key, map 순서라는 점.**

```
let map = new Map();
map.set('a', 100).set('b', 200);

map.forEach(function(value, key){
    console.log(value, key); //100 "a", 200 "b"
});
```

Map (Cont.)

■ has(key)

- Map 항목에 인자 key와 일치하는 항목의 유무 확인한 후 결과를 true, false로 알려줌.

```
let obj = {};
let map = new Map();

map.set(obj, 100);
map.set({a : 100}, 200);

console.log(map.has(obj));    // true
console.log(map.has({a:100})); // false, 별도로 Map 생성됨.
```

Map (Cont.)

■ keys()

- Map 항목 전체의 key를 열거 가능한 *iterator* 객체로 반환

```
let map = new Map();

map.set('a', 100).set('b', 200);

let mapIter = map.keys();
console.log(mapIter.next());
console.log(mapIter.next());
console.log(mapIter.next());
```

Map (Cont.)

■ values()

- Map 항목 전체의 value를 열거 가능한 *iterator* 객체로 반환

```
let map = new Map();

map.set('a', 100).set('b', 200);

let mapIter = map.values();
console.log(mapIter.next());
console.log(mapIter.next());
console.log(mapIter.next());
```

Map (Cont.)

■ [@@iterator] ()

- **entries()** 와 동일하게 **Map**의 항목을 열거할 수 있는 **iterator** 객체 반환.
- **iterator** 객체에 **next()** 호출 시 반환되는 객체의 value 속성 값은 map의 항목을 원소로 하는 배열([key, value])가 된다.

```
let map = new Map();

map.set('a', 100).set('b', 200);

let mapIter = map[Symbol.iterator]();
console.log(mapIter.next());    // {value: Array(2), done: false}
console.log(mapIter.next());    // { value: Array(2), done: false }
console.log(mapIter.next());    // {value: undefined, done: true}
```

새로 추가된 Collection - Set

- **Map**과 다르게 value들로 구성된 Collection이다.
- 물론, **Array**도 value들로만 구성된 Collection 이지만, 차이가 있다.
- **Set**은 **Array**처럼 **index**로 값을 읽을 수 없고, 열거를 통해서만 값을 얻을 수 있다.
- **Set**은 중복된 값을 저장하지 않는다.
- Syntax

```
let set = new Set( iterable );
```

Set (Cont.)

- Set은 중복된 값을 저장하지 않는다.

```
<script>

    let set = new Set([1, 2, 3, 1, 2, 3]);
    console.log(set); //Set(3) {1, 2, 3}
    set.add(2);

</script>
```

Set (Cont.)

- **Set** properties
 - **size** : 개수 반환

Set (Cont.)

■ Set Methods

- **add(value)** : Set에 새로운 항목을 추가하고 Set instance를 반환
- **clear()** : Set의 항목 모두 삭제
- **delete(value)** : value를 갖는 항목만 삭제, true(존재할경우) / false(존재하지 않을 경우)
- **entries()** : 추가된 항목 열거할 수 있도록 iterator 객체 반환
- **forEach(callbackFn)** : Set에 추가된 항목 순회
- **has(value)** : true(value를 갖는 항목 존재)/false(존재하지 않을 경우) 반환
- **keys()** : key들을 열거할 수 있도록 iterator 객체 반환
- **values()** : value들을 열거할 수 있는 iterator 객체 반환
- **[@@iterator]()** : 항목들을 열거할 수 있도록 iterator 객체 반환, entries()와 동일

Set (Cont.)

■ add(value)

- 인자 value를 순서대로 Set 항목에 추가한 뒤 Set instance 반환

```
<script>

    let set = new Set();
    set.add(100);
    set.add(200);

    for(let value of set){
        console.log(value);      //100, 200
    }

</script>
```

Set (Cont.)

■ clear()

- 추가된 모든 항목 삭제

```
<script>

    let set = new Set();
    set.add(100);
    set.add(200);
    console.log(set.size);    // 2

    set.clear();
    console.log(set.size);    // 0

</script>
```

Set (Cont.)

■ **delete (value)**

- value 인자와 일치하는 Set 항목 삭제

```
<script>

    let obj = {};
    let set = new Set();
    set.add(obj);
    set.add(100);
    console.log(set.size);    // 2

    set.delete(obj);
    console.log(set.size);    // 1

</script>
```

Set (Cont.)

■ entries()

- Set의 항목을 열거할 수 있는 *iterator* 객체 반환
- *iterator* 객체의 항목은 Set 항목을 [value, value]의 형태의 배열이 된다.

```
<script>

    let set = new Set('abcabc');
    let setIter = set.entries();

    for(let value of setIter){
        console.log(value); // ['a', 'a'], ['b', 'b'], ['c', 'c']
    }

</script>
```

Set (Cont.)

■ `forEach(callbackFn)`

- Set 항목 순회.
- 인수인 Callback 함수로 value와 key 그리고 Set을 전달
- 유의할 점은 value와 key 2개 모두 Set 항목의 value가 할당되어 있고, 전달 순서가 value, key, map 순서라는 점.

```
<script>

    let set = new Set('abab');
    set.forEach(function(value, key){
        console.log(value, key); //a a, b b
    });

</script>
```

Set (Cont.)

■ has (value)

- Set 항목에 인자 value와 일치하는 항목의 유무 확인한 후 결과를 true, false로 알려줌.

```
<script>

    let obj = {};
    let set = new Set();
    set.add(obj);

    console.log(set.has(obj));    //true

</script>
```

Set (Cont.)

■ `keys()`, `values()`, `[@@iterator]()`

- Set 항목을 열거 가능한 `iterator` 객체로 반환

```
<script>

let set = new Set('abab');

//keys
let keys = set.keys();
for(let value of keys){
    console.log(value); //a, b
}

//values
let values = set.values();
for (let value of values) {
    console.log(value); //a, b
}

//@@iterator
let setIter = set[Symbol.iterator]();
for (let value of setIter) {
    console.log(value); //a, b
}

</script>
```



Lab. Map, Set



새로 추가된 Collection - **WeakMap**

- **Map**과 같이 key와 value 쌍으로 이루어진 항목을 갖는 Collection.
- **Map**과 기능이 거의 동일.
- 다른 점은, **Map** 항목 key는 type 제한이 없는데, **WeakMap** 항목 key는 참조 Type(Reference Type)만 가능.
- **WeakMap** 항목 key는 열거되거나 조회될 수 없다.
- **Map**과 대부분의 API는 동일하지만, 열거 관련 method와 목록 수 조회 속성은 없다.
- Syntax

```
let weakMap = new WeakMap( [iterable] );
```

WeakMap (Cont.)

■ WeakMap Methods

- **set (key, value)** : WeakMap에 새로운 항목 추가하고 Instance 반환
- **get (key)** : key를 갖는 항목의 value 값 반환
- **delete (key)** : key를 갖는 항목만 삭제, **true**(존재할경우) / **false**(존재하지 않을 경우)
- **has (key)** : **true**(key를 갖는 항목 존재)/**false**(존재하지 않을 경우) 반환

WeakMap (Cont.)

■ set(key, value)

- Map의 set() 와 동일하지만, 항목 key가 반드시 Reference Type이어야 한다.

```
let obj = {};
let weakMap = new WeakMap();
weakMap.set(obj, 100); // key로 참조 탐색 사용
console.log(weakMap.get(obj)); //100
```

Lab. WeakMap



새로 추가된 Collection - **WeakSet**

- **Set**과 같이 value로만 이루어진 항목을 갖는 Collection.
- **Set**과 기능이 거의 동일.
- 다른 점은, **Set**은 value의 type 제한이 없는데, **WeakSet** value는 참조 Type(Reference Type)만 가능.
- **WeakSet** value는 열거되거나 조회될 수 없다.
- **Set**과 대부분의 API는 동일하지만, 열거 관련 method와 목록 수 조회 속성은 없다.
- Syntax

```
let weakSet = new WeakSet( [iterable] );
```

WeakSet (Cont.)

■ WeakSet Methods

- **add (value)** : WeakSet에 새로운 항목 추가하고 Instance 반환
- **delete (value)** : 인자와 같은 value를 갖는 항목 삭제, **true**(존재할경우) / **false**(존재하지 않을 경우)
- **has (value)** : **true**(key를 갖는 항목 존재)/**false**(존재하지 않을 경우) 반환

WeakSet (Cont.)

■ add(value)

- Set의 add() 와 동일하지만, 항목 value가 반드시 Reference Type이어야 한다.

```
let obj = {};
let weakSet = new WeakSet();
weakSet.add(obj);
console.log(weakSet.has(obj));    //true
```



Lab. WeakSet



새로 추가된 Collection – Map, Set

항목	항목 조회	
Object	key와 value로 이뤄지며, 내장 속성이 덮어씌워지므로 주의 필요	별도로 제공하는 method가 없고, iterable protocol을 따르지 않기 때문에 for...of문으로 열거할 수 없다.
Map	key와 value로 이뤄지며, 항목 추가하는 method 별도 제공하므로 내장 속성이 덮어씌워질 염려 없음.	항목을 열거할 수 있도록 entries()제공하며, iterable protocol을 따르므로 for...of문으로 열거할 수 있다.
Set	value로만 이뤄지며, 항목 추가하는 method 별도 제공하므로 내장 속성이 덮어씌워질 염려 없음.	항목을 열거할 수 있도록 entries()제공하며, iterable protocol을 따르므로 for...of문으로 열거할 수 있다.

새로 추가된 Collection – WeakMap, WeakSet

	Map	Set
Collection	항목이 key와 value로 이루어지며, key의 유형으로 primitive type과 reference type 모두 가능.	항목이 value로 이루어지며, value의 유형으로 primitive type과 reference type 모두 가능.
WeakCollection	항목이 key와 value로 이루어지며, key의 유형으로 reference type만 가능.	항목이 value로 이루어지며, value의 유형으로 reference type만 가능.

Symbol

- 새로 추가된 Primitive Type.
- 객체의 속성으로 사용.
- 객체의 속성으로 **Symbol**을 사용하는 이유는 내장 속성과의 충돌을 피하기 위함이다.
- 객체에 **Symbol**로 추가한 속성은 **for...in** 반복문에 나타나지 않는다.
- 비 공개 객체 멤버를 만들기 위한 방법
- Syntax

```
let symbol = Symbol(description); //new 사용 못함.
```

- **description** 인자는 **Symbol**을 구분하지 못한다.
 - 이유는 **Symbol**은 함수 호출할 때마다 새로운 **Symbol**을 생성하기 때문.

Symbol (Cont.)

```
<script>

    console.log(Symbol("foo") !== Symbol("foo")); //true
    const foo = Symbol();
    const bar = Symbol();
    console.log(typeof foo === "symbol");           //true
    console.log(typeof bar === "symbol");           //true
    let obj = {}
    obj[foo] = "foo"
    obj[bar] = "bar"
    console.log(JSON.stringify(obj)); // {}
    console.log(Object.keys(obj)); // []
    console.log(Object.getOwnPropertyNames(obj)); // []
    console.log(Object.getOwnPropertySymbols(obj)); // [ Symbol(), Symbol() ]

</script>
```

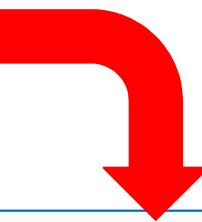
Symbol (Cont.)

- Symbol은 객체에 속성 추가할 때 내장 속성과의 충돌을 피할 수 있다.

```
<script>

let arr = [1, 2, 3];
console.log(arr.length);    // 3
arr.length = 100;
console.log(arr.length);    // 100

</script>
```



```
<script>

let arr = [1, 2, 3];
const length = Symbol('length');
arr[length] = 100;           // Array에 Symbol을 속성으로 추가
console.log(arr[length]);   // 100
console.log(arr.length);    // 3

</script>
```

Symbol (Cont.)

- 객체에 **Symbol**로 추가한 속성은 **for...in** 반복문에서 나타나지 않는다.

```
<script>

let array = [1, 2, 3];
array.prop = 100;
for(let i in array){
    console.log(i); //0, 1, 2, prop
}

</script>
```



```
<script>

let array = [1, 2, 3];
let prop = Symbol('prop');
array[prop] = 100;
for(let i in array){
    console.log(i); //0, 1, 2
}

</script>
```

Symbol (Cont.)

■ **Symbol.for()**

- Symbol 공유하기
- 언제든지 접근할 수 있는 전역 Symbol 저장소 제공.

```
let uid = Symbol.for('uid');
let obj = {};

obj[uid] = '12345';
console.log(obj[uid]);    //12345
console.log(uid);         //Symbol(uid)

let uid2 = Symbol.for('uid');
console.log(obj[uid2]);   //12345
console.log(uid === uid2); //true
```

Symbol (Cont.)

■ Well-known Symbols(표준|상용 Symbol)

- Spec에서 @@ 형태로 작성된 것
- @@는 Symbol 대신 사용한 것임.
- `Symbol.hasInstance` → `@@hasInstance`
- `Symbol.isConcatSpreadable` → `@@isConcatSpreadable`
- `Symbol.iterator` → `@@iterator`
- `Symbol.match` → `@@match`
- `Symbol.replace` → `@@replace`
- `Symbol.search` → `@@search`
- `Symbol.species` → `@@species`
- `Symbol.split` → `@@split`
- `Symbol.toPrimitive` → `@@toPrimitive`
- `Symbol.toStringTag` → `@@toStringTag`
- `Symbol.unscopables` → `@@unscopables`



Lab. Symbol

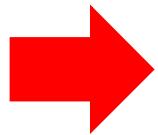


Promise

- 자연 함수와 비동기 연산을 제어를 위한 Class.
- ES5까지 비동기식 언어의 특성상 자연 함수 또는 비동기 연산이 종료되기 전에 아래쪽 구문이 실행되기 때문에 결과값을 활용하기 쉽지 않고, code 가독성이 떨어지는 단점 발생

```
<script>
  var cnt = 0;
  setTimeout(function(){
    cnt++;
  }, 1000);

  console.log(cnt); //0
  //자연 함수안의 Context보다 먼저 실행되기 때문.
</script>
```



```
<script>
  var cnt = 0;
  setTimeout(function(){
    receiveCount(++cnt);
  }, 1000);

  function receiveCount($cnt){
    cnt = $cnt;
    console.log(cnt); //1
  }
</script>
```

Promise (Cont.)

- **Promise**는 자연 함수 또는 비동기 연산을 내부에서 처리 후 이행 여부에 따라 결과 또는 실패 원인만 전달
- Method chain을 통해 가독성을 높였다.

```
<script>
    let cnt = 0;

    let promise = new Promise(function(resolve, reject){
        setTimeout(function(){ // 자연 함수를 Promise 내부에서 관리
            cnt++;
            resolve(cnt); // 결과값을 전달
        }, 1000);
    });

    promise.then(function($cnt){
        cnt = $cnt;
        console.log(cnt); // 1
    });
</script>
```

Promise (Cont.)

■ Promise 실행 과정

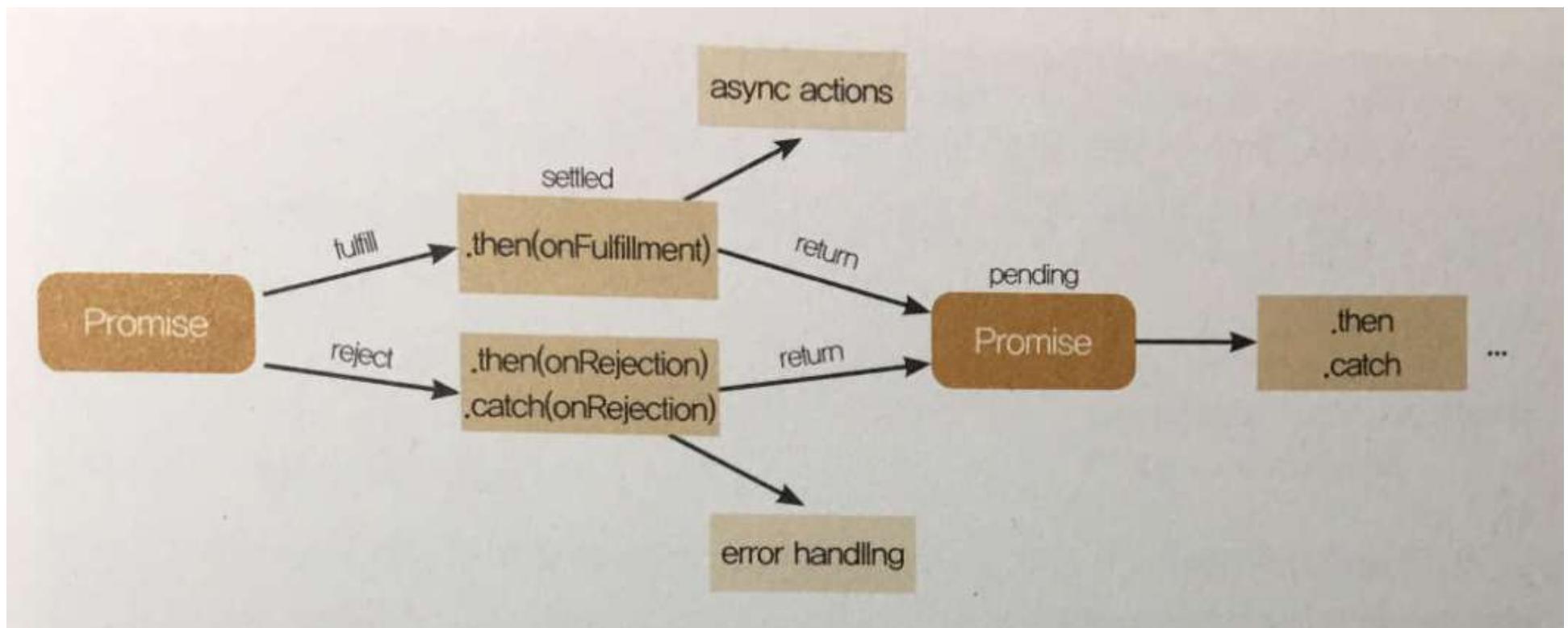


image from : 김규태, <만들면서 이해하는 ECMAScript 6>, 앤써북, 2018, p.95.

Promise (Cont.)

■ Promise 실행 과정

```
<script>
let promise = new Promise(function(resolve, reject){
    setTimeout(function(){
        reject('failed'); // 1초 뒤에 의해 거절인 reject() 호출
    }, 1000);
});

promise.then(function(value){ // 의해 거절되었기 때문에 호출안함.
    console.log(value);
});

promise.catch(function(reason){ // 의해 거절되거나 catch() 인수를 통해 거절 이유 전달
    console.log(reason); // failed
})
</script>
```

Promise (Cont.)

■ Promise methods

- **then (onFulfilled, onRejected)**

- 이행 또는 이행 거절이 되었을 때 인수인 callback 함수 호출을 받는다.
- **onFulfilled**는 이행이 되었을 때 호출 받은 callback 함수
- 인수에 이행 결과를 전달 받는다.
- **onRejected**는 이행 거절이 되었을 때 호출 받는 callback 함수
- 인수에 이행거절 이유를 전달 받는다.

- **catch (onRejected)**

- **onRejected**는 이행 거절이 되었을 때 호출 받는 callback 함수
- 인수에 거절 이유를 전달받는다.

Promise (Cont.)

■ Promise methods

- `Promise.all(iterable)`
 - 한 번에 여러 `Promise` 생성자의 이행 결과를 모아 전달한다.
- `Promise.race(iterable)`
 - 여러 `Promise` 생성자를 경합하여 가장 빠른 `Promise` 생성자의 이행 결과를 전달
 - 인수인 `iterable`은 여러 `Promise` 생성자를 항목으로 지정
 - 그 중 가장 빨리 이행 결정된 결과만 전달.
 - 이 때 하나라도 이행 거부되면 중지하고 이행 거부 이유 전달한다.

Promise (Cont.)

■ Promise methods

```
<script>
    let p1 = new Promise(function(resolve, reject){
        setTimeout(function(){
            resolve('p1 fulfilled');
        }, 2000);
    })

    let p2 = new Promise(function(resolve, reject){
        setTimeout(function(){
            resolve('p2 fulfilled');
        }, 1000);
    })

    let iterable = [p1, p2]; // p1, p2 순으로 열거되는 iterable 생성
    Promise.all(iterable).then(function(value){
        console.log(value); // ['p1 fulfilled', 'p2 fulfilled']
    }, function(reason){
        console.log(reason);
    })
</script>
```

Promise (Cont.)

■ Promise methods

```
<script>
  let p1 = new Promise(function(resolve, reject){
    setTimeout(function(){
      resolve('p1 fulfilled');
    }, 2000);
  })

  let p2 = new Promise(function(resolve, reject){
    setTimeout(function(){
      resolve('p2 fulfilled');
    }, 1000);
  })

  let iterable = [p1, p2]; // p1, p2 순으로 열거되는 iterable 생성
  Promise.race(iterable).then(function(value){
    console.log(value); // p2 fulfilled
  }, function(reason){
    console.log(reason);
  })
</script>
```

Promise (Cont.)

비동기 연산 또는 지연 함수 결과
활용 코드 가독성

Promise 사용 시 장점	호출 뒤 얻어지는 결과를 <code>executor()</code> 에서 전달받아 필요한 시점에 사용할 수 있어서 <code>timing</code> 조절이 쉽다.	<code>executor()</code> 에서 전달받은 결과를 이행 결정 또는 이행 거절 함수를 통하여 전달 <code>Chaining</code> 형태로 제공하므로 code 가독성이 이전보다 좋아짐.
-----------------	--	---

Proxy

- 객체에서 일어나는 일 관찰
- 사용자가 객체의 속성을 조회하거나, 할당, 열거, 호출 등을 할 때 관찰 중 이던 **Proxy**가 먼저 이를 알고 객체의 진행 결정한다.
- 메타 프로그래밍 : Runtime에서 동작하기 이전에 동작하도록 하는 것
- 객체에서 일어나는 일의 log를 남기거나 debugging 용도로 유용
- Syntax

```
let proxy = new Proxy(target, handler);
```

- **target** : **Proxy**의 관찰 대상 객체
- **handler** : 객체의 동작 가로챔을 실행하는 함수인 **trap**을 속성으로 하는 객체

연산자



연산자

■ 펼침 연산자(Spread Operator)

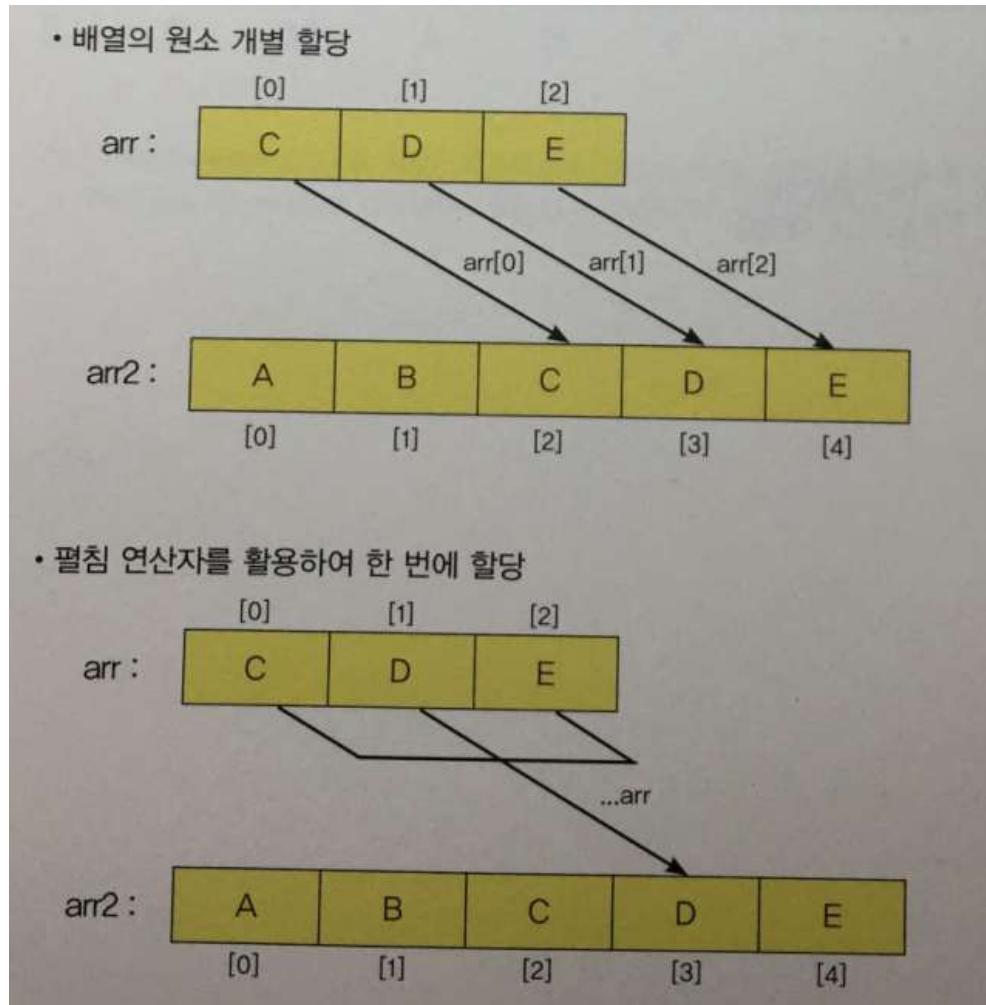
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

■ 비구조화 할당(Destructuring Assignment)

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Spread Syntax

- 펼침 연산자.
- 배열의 원소 또는 객체의 속성 등을 펼쳐서 할당한다.
- 배열 원소 전부를 한 번에 다른 literal 배열 원소에 포함시킬 수 있다.
- 함수 인자나 원소(배열 literal 등)가 여럿 나오는 곳이면 어디든지 사용 가능.
- ...으로 표기



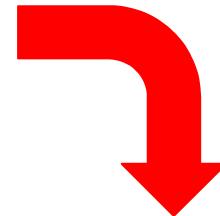
Spread Syntax (Cont.)

- 배열 원소 전부를 한 번에 다른 Literal 배열 원소에 포함시킨다.

```
<script>

let arr = [1, 2, 3];
let arr2 = [0, arr[0], arr[1], arr[2], 4];
console.log(arr2); //0,1,2,3,4

</script>
```



```
<script>

let arr = [1, 2, 3];
let arr2 = [0, ...arr, 4];
console.log(arr2); //0,1,2,3,4

</script>
```

Spread Syntax (Cont.)

- 배열 원소 전부를 한 번에 함수 인수에 전달한다.

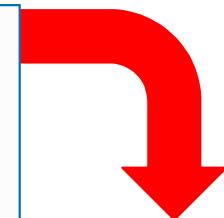
```
<script>

  let arr = [1, 2, 3];

  function foo(p1, p2, p3){
    console.log(p1, p2, p3); //1, 2, 3
  }

  foo(arr[0], arr[1], arr[2]);

</script>
```



```
<script>

  let arr = [1, 2, 3];

  function foo(p1, p2, p3){
    console.log(p1, p2, p3); //1, 2, 3
  }

  foo(...arr);

</script>
```

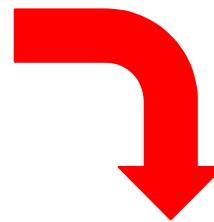
Spread Syntax (Cont.)

- 객체 속성 전부를 한 번에 다른 Literal 객체 속성에 포함시킨다.

```
<script>

let obj = {p1 : 1, p2 : 2};
let obj2 = {p2 : 20, p3 : 30};
obj2.p1 = obj.p1;
obj2.p2 = obj.p2;
console.log(obj2);    // {p2: 2, p3: 30, p1: 1}

</script>
```



```
<script>

let obj = {p1 : 1, p2 : 2};
let obj2 = {p2 : 20, p3 : 30, ...obj};

console.log(obj2);    // {p2: 2, p3: 30, p1: 1}

</script>
```

Spread Syntax (Cont.)

배열의 원소를 **literal** 배열의 원소에 추가하거나 객체의 속성을 **literal** 객체의 속성으로 추가할 때 함수 인수에 배열 원소, 또는 객체 속성값을 전달할 때

기존 방법

배열 원소 또는 객체 속성을 읽어 개별적으로 추가시켜야 되는 번거로움이 있다.

배열 원소 또는 객체 속성을 읽어 개별적으로 인수에 전달해야 되는 번거로움이 있다.

펼침 연산자 사용

배열 원소 또는 객체 속성을 개별이 아닌 한꺼번에 추가한다.

배열 원소 또는 객체 속성을 개별이 아닌 한꺼번에 인수에 전달한다.

Lab. Spread Operator



Destructuring Assignment

- 비구조할당
- 배열 또는 객체에서 변수를 추출해 내는 표현식.
- 배열 원소값 또는 객체 속성값을 배열 Literal이나 객체 Literal 형태의 표현식으로 간편하게 변수를 선언.
- 함수의 전달 인자가 객체 또는 배열일 경우 편리.

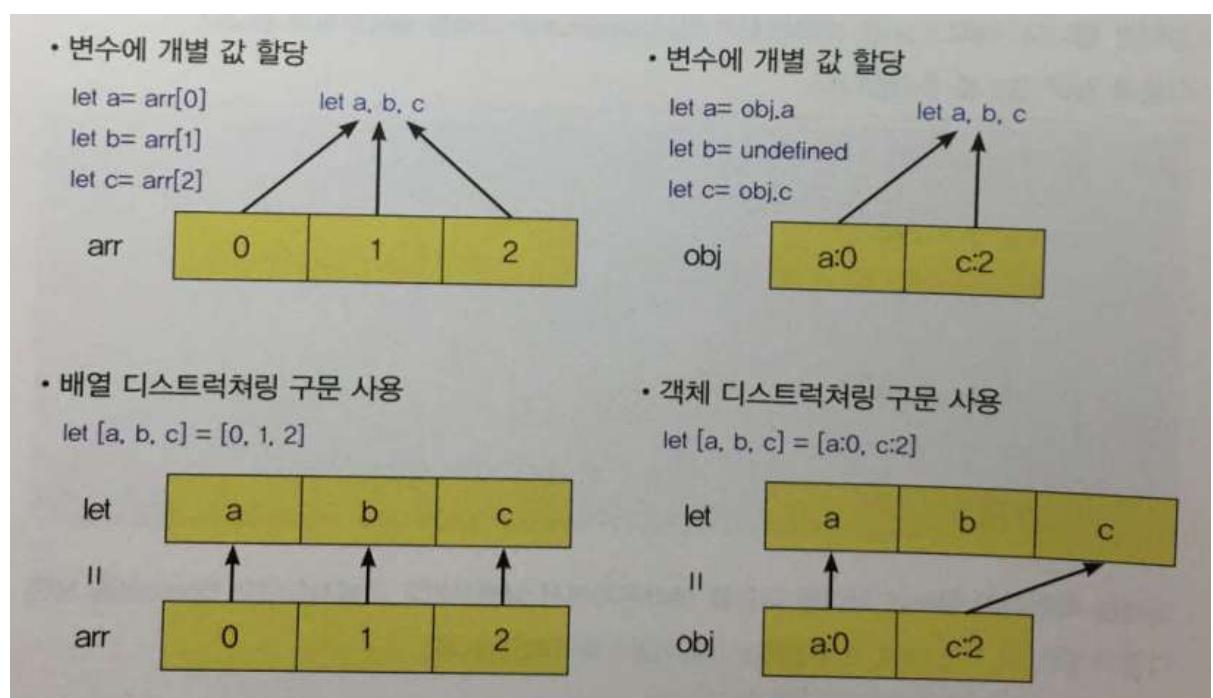


image from : 김규태, <만들면서 이해하는 ECMAScript 6>, 앤써북, 2018, p.120.

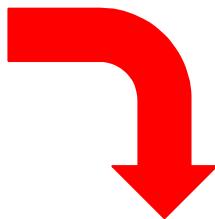
Destructuring Assignment (Cont.)

■ 배열의 Destructuring

```
<script>

let arr = [1, 2, 3];
let a = arr[0];
let b = arr[1];
let c = arr[2];
console.log(a, b, c);    // 1, 2, 3

</script>
```



```
<script>

let [a, b, c] = [1, 2, 3];

console.log(a, b, c);    //1, 2, 3

</script>
```

Destructuring Assignment (Cont.)

- 배열의 Destructuring
 - 일부 원소 생략 가능

```
<script>

  let [a, , c] = [1, 2, 3];

  console.log(a, c); //1, 3

</script>
```

Destructuring Assignment (Cont.)

- 배열의 Destructuring
 - 배열 Destructuring 구문에 기본값 할당

```
<script>

  let [a = 100, b = 200, c = 300] = [undefined, , 1000];

  console.log(a, b, c);    // 100, 200, 1000

</script>
```

Destructuring Assignment (Cont.)

■ 배열의 Destructuring

- 배열 Destructuring 구문에 나머지 매개변수 적용

```
<script>

    let [a, b, ...c] = [1,2,3,4,5,6];

    console.log(a, b, c); //1, 2, [3, 4, 5, 6]

</script>
```

나머지 매개변수 : 함수 전달 인자 중 할당된 원소 외의 나머지 원소를 모두 배열의 형태로 참조하는 매개변수

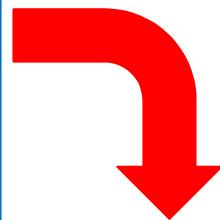
Destructuring Assignment (Cont.)

■ 객체 Destructuring

```
<script>

let obj = {a : 100, b : 200, c : 300};
let a = obj.a;
let b = obj.b;
let c = obj.c;
console.log(a, b, c); //100, 200, 300

</script>
```



```
<script>

let {a, b, c} = {a : 100, b : 200, c : 300};
console.log(a, b, c); //100, 200, 300

</script>
```

Destructuring Assignment (Cont.)

■ 객체 Destructuring

- 변수와 같은 이름의 객체 속성이 없으면 **undefined** 할당된다.

```
<script>

    let {a, b, c} = {a : 100, c : 300};
    console.log(a, b, c); //100, undefined, 300

</script>
```

Destructuring Assignment (Cont.)

■ 객체 Destructuring

- 객체 Destructuring 구문에 기본값 할당이 가능.

```
<script>

  let {a= 1, b = 2, c = 3} = {a : 100, c: undefined};
  console.log(a, b, c); //100, 2, 3

</script>
```

Destructuring Assignment (Cont.)

- 함수 매개변수로 Destructuring 구문을 활용하고 기본값 할당.

```
<script>

    function foo([a, b, c = 300] = [100, 200],
                {d = 400, e} = {d : undefined, e : 500}){
        console.log(a, b, c, d, e); //100, 200, 300, 400, 500
    }

    foo();

</script>
```



Lab. Destructuring Assignment



함수



함수

- 나머지 매개변수(Rest Parameter)와 기본 매개변수(Default Parameter)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters
- 화살표 함수(Arrow Function Expressions)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Rest Parameter

- JavaScript는 함수 인자와 인수의 수가 일치하지 않으면 오류가 발생한다.
- 인자는 선언된 매개변수에 순차적으로 할당되며 나머지는 할당되지 않는다.
- 함수 호출시 함수 내부에는 arguments 객체가 생성되며, 배열과 유사한 형태로 전달 인자를 원소로 저장한다.

```
<script>

    function myFunction(p1){
        if(arguments[1]){ //두번째 인자가 있을 경우
            return p1 + arguments[1];
        }else{
            return p1;
        }
    }

    console.log(myFunction(100, 200)); //300

</script>
```

Rest Parameter (Cont.)

- 함수 호출시 전달 인자가 앞의 매개변수에 순차적으로 전달되고, 나머지 인자가 모두 나머지 매개변수에 할당된다.
- 이때 Type은 배열이 되고, 인자들은 순차적으로 배열의 원소가 된다.

```
<script>

    function myFunction(p1, ...args){
        if(args[0]){ //나머지 매개변수가 있을 경우
            return p1 + args[0];
        }else{
            return p1;
        }
    }

    console.log(myFunction(100, 200)); //300

</script>
```

Rest Parameter (Cont.)

- 나머지 매개변수에 전달 인자가 없을 경우 값은 **undefined**가 아니다.
- 왜냐하면 빈 배열이기 때문이다.

```
<script>

    function myFunction(...args){
        console.log(args.length);      // 0
    }

    myFunction();

</script>
```

Rest Parameter (Cont.)

■ 나머지 매개변수의 제한

- 나머지 매개변수는 반드시 한 개 여야 한다.
- 마지막 위치에만 사용할 수 있다.

```
<script>
    function pick(object, ...keys, last){
        //Uncaught SyntaxError: Rest parameter must be last formal parameter
        let result = Object.create(null);

        for(let i = 0, len = keys.length ; i < len ; i++){
            result[keys[i]] = object[keys[i]];
        }
        return result;
    }
</script>
```

Rest Parameter (Cont.)

- 나머지 매개변수의 제한
 - 객체 literal의 setter에 사용할 수 없다.

```
<script>
  let object = {
    //Uncaught SyntaxError: Setter function argument must not be a rest parameter
    set name(...value){
      //나머지/ 동작
    }
  }
</script>
```

Default Parameter

- JavaScript에서는 매개변수에 기본값을 설정할 수 없기 때문에 전달 인자가 없을 경우에는 **undefined**가 할당된다.

```
<script>

    function myFunction(p1){
        p1 = (typeof p1 != 'undefined') ? p1 : 0;
        console.log(p1);    // 0
    }

    myFunction();

</script>
```

Default Parameter (Cont.)

- 매개변수는 선언시 기본값을 할당이 가능하기 때문에 전달 인자가 없을 경우 기본값으로 설정되고 전달 인자가 있는 경우에는 전달 인자가 할당된다.

```
<script>

    function myFunction(p1 = 100){
        console.log(p1); // 100
    }

    myFunction();

</script>
```

Default Parameter (Cont.)

- 먼저 선언된 매개변수의 값은 나중의 기본 매개변수에 이용 가능하다.

```
<script>

    function myFunction(a, b = 50, c = a + b){
        console.log(c);    // 150 = a(100) + b(50)
    }

    myFunction(100);

</script>
```

Default Parameter (Cont.)

- 전달 인자에 **undefined** 할당시 기본 매개변수는 초기값이 된다.

```
<script>

    function myFunction(p1 = 100, p2){
        console.log(p1, p2);    // 100, 200
    }

    myFunction(undefined, 200);

</script>
```

Default Parameter (Cont.)

	나머지 매개변수	기본 매개변수
장점	<ul style="list-style-type: none">전달 인자의 수가 일정하지 않을 경우 <code>arguments</code> 객체를 사용 없이 가능배열 인수에 <code>index 0</code>부터 차례로 전달받는다.	<ul style="list-style-type: none">전달 인자를 설정하지 않을 경우 초기값이 필요하면 별도로 예외처리를 해주어야 하는 번거로움이 있었다.기본 매개 변수를 사용하면 초기값 설정이 가능인수 선언 시 먼저 선언된 인수의 값을 활용 가능.



Lab. Rest & Default Parameter



Arrow Function Expressions

- 함수표기를 화살표(=>)로 하여 구문을 짧게 줄여 준다.
- 코드 작성량을 줄여주어 작성 시간 단축에 도움.
- 일반 함수와 다르게 함수 block안에서 **this, arguments, super, new, target** 등의 key 값을 생성하지 않는다.

```
function foo() {  
    ...  
}
```



```
foo() => { ... }
```

Arrow Function Expressions (Cont.)

```
<script>

  var add = function(a, b){
    return a + b;
  }

  console.log(add(5, 6)); //11

</script>
```



```
<script>

  let add = (a, b) => {
    return a + b;
  }

  console.log(add(5, 6)); //11

</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function은 block 구문을 생략하고 표현식을 사용할 수 있다.

```
<script>
  let add = (a, b) => {
    console.log(a + b);    // 11
  }
  add(5, 6);
</script>
```



```
<script>
  let add = (a, b) => console.log(a + b);
  add(5, 6);
</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function은 block 구문을 생략하고 표현식을 사용할 수 있다.
- 하지만, block 구문 생략시 **return**은 사용할 수 없고, **SyntaxError**가 발생 한다.

```
<script>

  let add = (a, b) => return a + b;

  add(5, 6);

</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function은 단일 인자만 넘겨받는 경우 괄호 생략 가능.

```
<script>

  let printText = (message) => document.write(message);

  printText('Hello, World');

</script>
```



```
<script>

  let printText = message => document.write(message);

  printText('Hello, World');

</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function도 매개변수에 기본값과 Destructuring 구문 사용 가능.

```
<script>

  let add = ({a = 100, b = 200}) => {
    console.log(a, b);    //200, 200
    return a + b;
  }

  console.log(add({a : 200}));  // 400

</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function도 일반 함수처럼 method로 사용가능.

```
<script>

  const calculation = {
    add : (a, b) => {
      return a + b;
    }
  }

  let sum = calculation.add(100, 200);
  console.log(sum);

</script>
```

Arrow Function Expressions (Cont.)

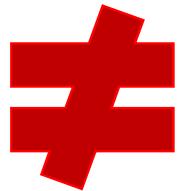
- Arrow Function은 일반 함수와 다르게 **this**를 생성하지 않는다.

```
<script>

var obj = {
    foo:function(){
        console.log(this);
    }
}

obj.foo();

</script>
```



```
<script>

var obj = {
    foo:() => {
        console.log(this); //window 객체 참조
    }
}

obj.foo();

</script>
```

Arrow Function Expressions (Cont.)

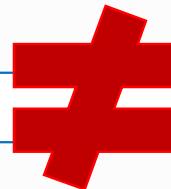
- Arrow Function은 일반 함수와 다르게 **arguments**를 생성하지 않는다.

```
<script>

  var foo = function(a, b){
    console.log(arguments);
  }

  foo(100, 200); //Arguments(2) [100, 200, callee: f, Symbol(Symbol.iterator): f]

</script>
```



```
<script>

  var foo = (a, b) => {
    console.log(arguments); //ReferenceError : arguments is not defined
  }

  foo(100, 200);

</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function은 **new** 연산자 호출이 불가능하다.

```
<script>

  let foo = () => {};
  let f = new foo(); //TypeError : foo is not a constructor

</script>
```

Arrow Function Expressions (Cont.)

- Arrow Function은 **prototype** 속성이 존재하지 않는다.

```
<script>

  let foo = () => {};
  let p = foo.prototype;
  console.log(p);    //undefined

</script>
```

Arrow Function Expressions (Cont.)

	일반 함수	화살표 함수
표기	<code>function(){} function</code> 키워드와 괄호, 블록 구문을 생략 불가능	<code>() => {} () => {}</code> 인수가 하나인 경우 괄호를 생략할 수 있고, 구문이 한 줄로 끝날 경우 블록 구문 생략 가능
this 생성	함수가 객체의 속성에 참조되었거나, <code>new</code> 연산자 호출시 <code>this</code> 의 참조값을 객체가 된다.	<code>this</code> 를 생성하지 않는다.
arguments 생성	전달 인자를 리스트로 하는 <code>arguments</code> 를 생성	<code>arguments</code> 를 생성하지 않는다.
<code>new</code> 연산자 호출	<code>new</code> 연산자 호출시 인스턴스를 생성	<code>new</code> 연산자 호출 불가능
prototype 속성	prototype 속성이 존재한다.	prototype 속성이 존재하지 않는다.

Lab. Arrow Function



클래스와 모듈



Class & Module

■ Class

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

■ Module

Class

■ Class 선언

- Class 선언문으로 선언

```
class Name {}
```

```
class Name extends Super_name {}
```

```
<script>

    class Member {
        getName() {
            return "이름";
        }
    };

    let obj = new Member();
    console.log(obj.getName());

</script>
```

Class (Cont.)

■ 생성자 함수

- Class의 instance 생성 시에 한번 호출되는 함수
- 주로 초기 설정 목적으로 사용
- Instance 호출 시 전달한 인자값이 생성자 함수 매개변수로 전달된다.
- Class block 안에 선언하며, **constructor** 키워드를 사용하여 선언
- **function**을 붙이지 않는다.

```
<script>

  class Display {
    constructor(x, y){
      this.x = x;
      this.y = y;
      console.log(this.x, this.y);
    }
  }

  const display = new Display(100, 200);
</script>
```

Class (Cont.)

■ Prototype 함수

- Instance를 통해서 호출 가능한 함수
- Class 선언문 block 내부에 함수를 선언
- **function** 은 불이지 않는다.

```
<script>  
  var Display = function(){}
  
  Display.prototype.foo = function(){}
</script>
```



```
<script>
  class Display{
    foo(){}
  }
</script>
```

Class (Cont.)

■ Prototype 함수

```
<script>

    class Display{
        foo(){
            console.log(this);
        }
    }

    const display = new Display();
    display.foo();

</script>
```

Class (Cont.)

■ 정적 메소드(Static Method)

- Prototype method와는 다르게 instance를 통하지 않고 class 이름 뒤에 바로 method 호출
- 주로 Utility 함수를 정의할 때 사용
- Method 이름 앞에 **static** keyword를 붙여 정의

```
<script>

    class Display{
        static foo(){
            console.log(this);
        }
    }

    Display.foo();

</script>
```

Class (Cont.)

■ 상속

- 이전에는 prototype chain mechanism을 활용해야.
- ES6에서는 **extends** keyword를 사용하여 보다 쉽게 상속 구현

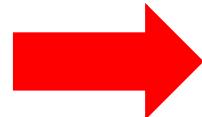
```
<script>

  var Parent = function(){}
  Parent.prototype.foo = function(){
    console.log('foo');
  }

  var Child = function(){}
  var parent = new Parent();
  Child.prototype = parent;

  var child = new Child();
  child.foo();

</script>
```



```
<script>

  class Display{
    constructor(){}
  }

  class Rect extends Display{
    constructor(){
      super();
    }
  }

  const rect = new Rect();

</script>
```

Class (Cont.)

■ 상속

```
class Display{
    constructor(x, y){
        this.x = x;
        this.y = y;
    }
    LogPosition(){
        console.log(this.x, this.y);
    }
}

class Rect extends Display{
    constructor(x, y, width, height){
        super(x, y);
        this.width = width;
        this.height = height;
    }
    LogScale(){
        console.log(this.width, this.height);
        super.LogPosition();
    }
}
```

Class (Cont.)

■ Prototype Method Overriding

- 부모의 method를 재정의하기
- 상속받은 class의 method 호출시
자식 class의 method 목록을 찾아
호출하며, 없을 경우에는 부모 class
를 조회하여 호출

```
class Display{  
    constructor(x, y){  
        this.x = x;  
        this.y = y;  
    }  
    LogPosition(){  
        console.log(this.x, this.y);  
    }  
}  
class Rect extends Display{  
    constructor(x, y, width, height){  
        super(x, y);  
        this.width = width;  
        this.height = height;  
    }  
    LogPosition(){ //Method Overriding  
        console.log('Method Overriding');  
        super.LogPosition();  
    }  
}  
const rect = new Rect(10, 20, 100, 200);  
rect.LogPosition();
```

Class (Cont.)

기존 방식	Class
Prototype method	<ul style="list-style-type: none">함수 <code>prototype</code> 속성에 추가하여 사용매번 <code>prototype keyword</code>를 작성해야 하는 번거로움어느 위치에서도 <code>method</code> 추가 <code>code</code>를 작성 가능그룹화가 되어 있지 않아서 가독성이 떨어짐 <ul style="list-style-type: none">Class block 안에 <code>prototype</code> 생략<code>Method</code>가 class block 안에서 작성그룹화가독성이 좋다.
상속	<ul style="list-style-type: none">자식 <code>class</code>의 <code>prototype</code> 속성에 부모 <code>class</code>의 <code>instance</code>를 추가하여 구현.부모 <code>class</code>의 초기화를 추가로 구현해야정해진 방식이 없으므로 구현방식이 모두 다름<code>Code</code>의 가독성이 매우 떨어짐. <ul style="list-style-type: none">Class 선언 뒤에 <code>extends keyword</code>를 붙여서 상속<code>super</code> 함수를 호출하여 부모 <code>class</code>를 간단하게 초기화 가능선언방식이 정해져 있어서 <code>code</code>의 가독성이 좋다.



Lab. Class



Module

- JavaScript code의 재사용성을 높인다.
- Program과 Library를 Module 단위로 나눔 가능
- 주요 기능
 - 자신만의 독립적인 실행영역(scope)을 갖기
 - 전역변수와 지역변수를 나누어 선언
 - 비동기로 module을 load하여 사용
- **<script type="module">**로 선언
- **import**와 **export** keyword 사용
- 변수와 함수 등 비공개 API / 공개 API로 선언가능

Module (Cont.)

■ Module化 필요성

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Document</title>
7 </head>
8 <body>
9     <script src="js/a.js"></script>
10    <script src="js/b.js"></script>
11    <script>
12        |   getTotal(); // 200
13    </script>
14 </body>
15 </html>
```

```
1 //js/a.js
2
3 var total = 100;
4 function getTotal() {
5     |   return total;
6 }
```

```
1 //js/b.js
2 |
3 var total = 200;
```

Module (Cont.)

■ Export

- Module마다 개별.js 파일에 JavaScript code로 구현하며 원하는 개수만큼 변수를 export 가능.
- 변수, 함수, class, 기타 entity를 export 가능.
- **export** 변수, 함수

```
export { variableName };  
export { variableName1, variableName2, variableName3 };  
export { variableName as myvariableName };  
export { variableName as default }  
export { variableName as default, variableName1 as myvariableName1,  
        variableName2 };  
export default function(){};  
export { variableName1, variableName2 } from "myAnotherModule";  
export * from "myAnotherModule";
```

Module (Cont.)

■ Export : [js/example.js]

```
//data export
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

//함수 export
export function sum(num1, num2){
    return num1 + num2;
}

//class export
export class Rectangle {
    constructor(length, width){
        this.length = length;
        this.width = width;
    }
}
```

```
//이 함수는 Module에 비공개
function subtract(num1, num2){
    return num1 - num2;
}

//함수 정의
function multiply(num1, num2){
    return num1 * num2;
}

//위에서 정의한 함수를 export
export { multiply };
```

Module (Cont.)

■ Import

- `import { 불러올 변수 또는 함수 이름 } from '파일 경로';`
`import x from "module-relative-path";`
`import { x } from "module-relative-path";`
`import { x1 as x2 } from "module-relative-path";`
`import { x1, x2 } from "module-relative-path";`
`import { x1, x2 as x3 } from "module-relative-path";`
`import x, { x1, x2 } from "module-relative-path";`
`import "module-relative-path";`
`import * as x from "module-relative-path";`
`import x1, * as x2 from "module-relative-path";`

Module (Cont.)

■ 한 개의 binding만 import 하기

- Browser와 Node.js간의 호환을 위해서, import 파일 문자열의 시작에 `/`나 `./`, `../`를 포함해야 한다.

```
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Hello, Vue.js World</title>
7   <script type="module">
8     // 단 하나의 식별자 import
9     import { sum } from "./js/example.js";
10
11    console.log(sum(1, 2));
12
13    sum = 1;    //Uncaught TypeError : Assignment to constant variable.
14    //import 된 binding에는 재할당이 불가능하기 때문
15  </script>
16 </head>
17 <body>
```

Module (Cont.)

- 여러 개의 binding import 하기
 - 여러 개의 binding을 import 하려면, 명시적으로 나열

```
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Hello, Vue.js World</title>
7      <script type="module">
8          import { sum, multiply, magicNumber } from './js/example.js';
9
10         console.log(sum(1, magicNumber));    //8
11         console.log(multiply(1,2));        //2
12     </script>
13 </head>
14 <body>
```

Module (Cont.)

■ Module 전체 import 하기

```
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Hello, Vue.js World</title>
7     <script type="module">
8         import * as example from './js/example.js';
9
10        console.log(example.sum(1, example.magicNumber));    //8
11        console.log(example.multiply(4, 8));      //32
12    </script>
13 </head>
14 <body>
```

Module (Cont.)

■ 주의 사항

- import 문에서 module을 사용한 횟수와 상관없이 한 번만 실행

```
import { sum } from './js/example.js';
import { multiply } from './js/example.js';
import { magicNumber } from './js/example.js';
```

- 여기서는 example.js는 한 번만 실행됨.

Module (Cont.)

■ Module의 기본값

- module의 기본값은 **default** 키워드로 지정된 변수나 함수, 클래스이다.
- module마다 하나의 **export** 기본값을 설정할 수 있다.
- 단 한 개만 export 해야 한다. 여러 개 사용하면 에러 발생
- **Names Module과 달리 {}(중괄호)를 사용하지 않는다.**

```
<script type="module">
    import sum from './js/example.js';

    console.log(sum(5, 9));
</script>
```

```
1 //first way
2 export default function(num1, num2){
3     return num1 + num2;
4 }
5
6 //second way
7 function sum(num1, num2){
8     return num1 + num2;
9 }
10
11 export default sum;
12
13 //third way
14 function sum(num1, num2){
15     return num1 + num2;
16 }
17
18 export { sum as default};
19
```

Module (Cont.)

Module 사용 시 장점

독립적인 실행영역
(scope) 생성

- Module을 사용하면 자동으로 독립적인 scope 생성
- 작업 시간 단축
- 외부 library나 불필요한 code 작성 불필요

비동기 load

- Module은 재사용이 쉽도록 미리 작성 가능
- 필요할 때 load하여 사용하여 재생산성 높여 준다.



Lab. Module

Appendix



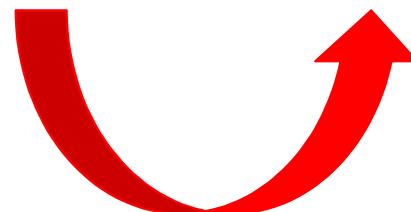
Computed Property Name

■ ES5

```
<script>
  var person = {
    "first name" : "Nicholas"
  }
  console.log(person["first name"]); //Nicholas
  /////////////////////////////////
  var lastName = "last name";
  var member = {
    LastName : "Zakas"
  }
  console.log(member[lastName]); //undefined
</script>
```

■ ES6

```
<script>
  let lastName = "last name";
  let person = {
    "first name" : "Michael",
    [lastName] : "Jackson"
  };
  console.log(person["first name"]); //Michael
  console.log(person[lastName]); //Jackson
</script>
```



Computed Property Name (Cont.)

- 객체 literal안에 대괄호는 프로퍼티 이름을 계산한다는 의미
- 대괄호(【】)의 내용은 문자열로 평가됨.
- 문자열과 변수를 조합하여 Object의 property 이름으로 사용가능
- 조합하려는 이름을 대괄호(【】)안에 문자열로 작성한 형태
 - 문자열 조합

```
<script>

    let item = {
        ["one" + "two"] : 12
    };

    console.log(item.onetwo); //12
</script>
```

Computed Property name (Cont.)

- 객체 literal안에 대괄호는 프로퍼티 이름을 계산한다는 의미
- 대괄호([])의 내용은 문자열로 평가됨.
- 문자열과 변수를 조합하여 Object의 property 이름으로 사용가능
- 조합하려는 이름을 대괄호([])안에 문자열로 작성한 형태
 - 변수 값과 문자열 조합

```
<script>
    let item = "tennis";
    let sports = {
        [item] : 1,
        [item + "Game"] : "Wimbledon",
        [item + "Method"](){
            return this[item];
        }
    }

    console.log(sports.tennis);          //1
    console.log(sports.tennisGame);      //Wimbledon
    console.log(sports.tennisMethod());   //1
</script>
```

Exponentiation Operator

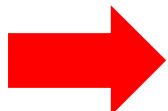
- 거듭 제곱 연산자
- ******
- ES7에 추가

```
<script>
    console.log(3 ** 2);          //9
    console.log(3 ** 3);          //27
    console.log(Math.pow(3, 3));   //27
</script>
```

Property Initializer Shorthand

■ 프로퍼티 초기화 단축

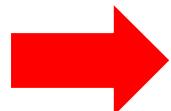
```
<script>
  function createPerson(name, age) {
    return {
      name: name,
      age: age
    };
  }
</script>
```



```
<script>
  function createPerson(name, age) {
    return {
      name,
      age
    };
  }
</script>
```

Concise Method

```
<script>
  var person = {
    name: "Nicholas",
    sayName: function() {
      console.log(this.name);
    }
  };
</script>
```



```
<script>
  var person = {
    name: "Nicholas",
    sayName() {
      console.log(this.name);
    }
  };
</script>
```

Descriptor

- 속성 이름과 속성 값으로 구성.

```
<script>
    Object.defineProperty({}, "book", {
        value : 123,
        enumerable : true
    });
</script>
```

- Property name : book
- Property description : value, enumerable

Descriptor (Cont.)

Type	Property Name	Property Value Type	Default Value	Description
Data	value	JavaScript Data Type	undefined	Property 값으로 사용
	writable	true, false	false	false: 속성 값 변경 불가
Access	get	function, undefined	undefined	Property get 함수
	set	function, undefined	undefined	Property set 함수
공용	enumerable	true, false	false	false: for~in으로 열거 불가
	configurable	true, false	false	false : property 삭제 불가

Descriptor (Cont.)

■ get, set 속성

```
<script>
let obj = {};
Object.defineProperty(obj, "book", {
    get:function(){
        return "JavaScript ES6";
    }
});
console.log(obj.book); //JavaScript ES6
</script>
```

```
<script>
let obj = {};
Object.defineProperty(obj, "item", {
    set:function(param){
        this.sports = param;
    }
});
obj.item = "Baseball"
console.log(obj.sports); //Baseball
</script>
```

Getter / Setter

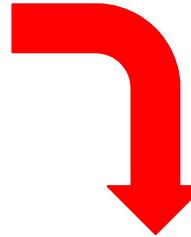
```
<script>
  let obj = {
    value : 32000,
    get getValue(){
      return this.value;
    }
  };
  console.log(obj.getValue); //32000
</script>
```

```
<script>
  let obj = {
    set setValue(value){
      this.value = value;
    }
  };
  obj.setValue = 32000;
  console.log(obj.value); //32000
</script>
```

__proto__ Property

```
<script>
  var x = { name : 'Sally' };
  var y = Object.create(x, { age : {value : 24}});

  console.log(y.name); //Sally
  console.log(y.age); //24
</script>
```



```
<script>
  let y = { name : 'Sally', __proto__ : { age : 24}};
  console.log(y.name);
  console.log(y.age);
</script>
```

Object.is() method

- 두 값의 동등 여부 판단

- Cf)

- `====` : 값과 타입 모두 비교
- `==` : 값 타입은 비교하지 않고 값만 비교
- `Object.is()` : 값과 값 타입 모두 비교

```
<script>
    console.log(Object.is(0, -0));          //false
    console.log(0 === -0);                  //true
    console.log(Object.is(NaN, 0/0));       //true
    console.log(NaN === 0/0);               //false
    console.log(Object.is(NaN, NaN));      //true
    console.log(NaN === NaN);              //false
</script>
```

Object.setPrototypeOf (object, prototype) method

- 객체 [[prototype]] property 값을 할당하는 method

```
<script>
  let x = { name : 'Sally' }
  let y = { age : 24 }
```

```
Object.setPrototypeOf(y, x);
```

```
console.log(y.name); //Sally
console.log(y.age); //24
</script>
```

```
<script>
  let Sports = function(){}
  Sports.prototype.getCount = function(){
    return 123;
  };
  let protoObj = Object.setPrototypeOf({}, Sports.prototype);
  console.log(protoObj.getCount()); //123
</script>
```

Object.assign(targetObj, sourceObj) method

- 하나, 또는 그 이상의 sourceObj에서 모든 열거 가능한 자기 property들을 targetObj로 복사하고 이 targetObj를 반환하는 method

```
<script>
  let x = { name : 'Sally' };
  let y = { age : 24, __proto__ : x };
  let z = {
    gender : '남성',
    get getValue() { return 10; },
    city : {}
  };
  Object.defineProperty(z, 'tel', {enumerable : false});

  let m = {};

  Object.assign(m, y, z);

  console.log(m.age); //24
  console.log(m.gender); //남성
  console.log(m.getValue); //10
  console.log(m.name); //undefined
  console.log(m.city == z.city); //true
</script>
```

Object.assign(targetObj, sourceObj) method (Cont.)

```
<script>
  let sports = {
    event : '축구',
    player : 11
  }
  let dup = sports;           // 주소복사

  sports.player = 55;
  console.log(dup.player);   //55

  dup.event = '농구';
  console.log(sports.event);
</script>
```

```
<script>
  let sports = {
    event : '축구',
    player : 11
  }
  let dup = {};
  for(var key in sports)
    dup[key] = sports[key];
  //값복사 즉 연동하지 않으려면 일일히 복사해야 함.

  sports.player = 33;
  console.log(dup.player);   //11

  dup.event = '농구';
  console.log(sports.event); //축구
</script>
```

```
<script>
  let sports = {
    event : '축구',
    player : 11
  }
  let dup = Object.assign({}, sports); // 값복사
  console.log(dup.player);          //11

  sports.player = 33;
  console.log(sports.player);       //33

  sports.event = '수영';
  console.log(dup.event);          //축구
</script>
```



Number Literal

- ES6에서 2진수 표기법추가
 - 첫 번째 숫자 0을 작성하고 두 번째에 소문자(또는 대문자) b 작성
- ES6에서 8진수 표기법 재정의
 - 첫 번째 숫자 0을 작성하고 두 번째에 소문자(또는 대문자) o 작성

```
<script>
    let binary = 0b0101;
    let octal = 0o0101;

    console.log("이진수 : " + binary); //5
    console.log("8진수 : " + octal); //65
</script>
```