ES6 Fundamentals

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Vue.js

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

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello ES6</title>
</head>
<body>
    <script>

        let subject = 'ES6';
        let str = `오늘의 주제는 ${subject}입니다.`;
        console.log(str);   //오늘의 주제는 ES6입니다.
    </script>
</body>
</html>
```

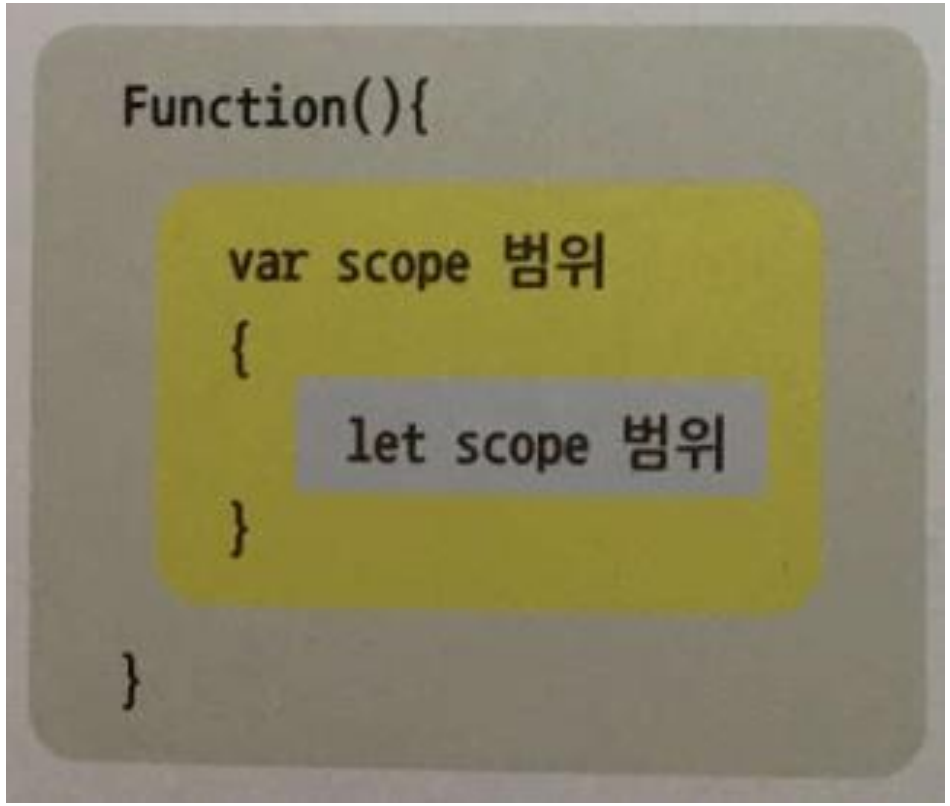
기본 문법

let

- **var** 변수의 문제점
  - 선언문의 생략
  - 중복된 변수명 선언의 가능
  - 함수 Hoisting
  - 개발에 혼란
  - 가독성의 떨어짐

let (Cont.)

- let은 var와 다르게 Block에서 Scope가 설정된다.
  - var는 함수 Block에서 Scope가 설정되지만, 그 외 Block에서는 Scope가 설정되지 않아서 변수가 공유된다.




```
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.)

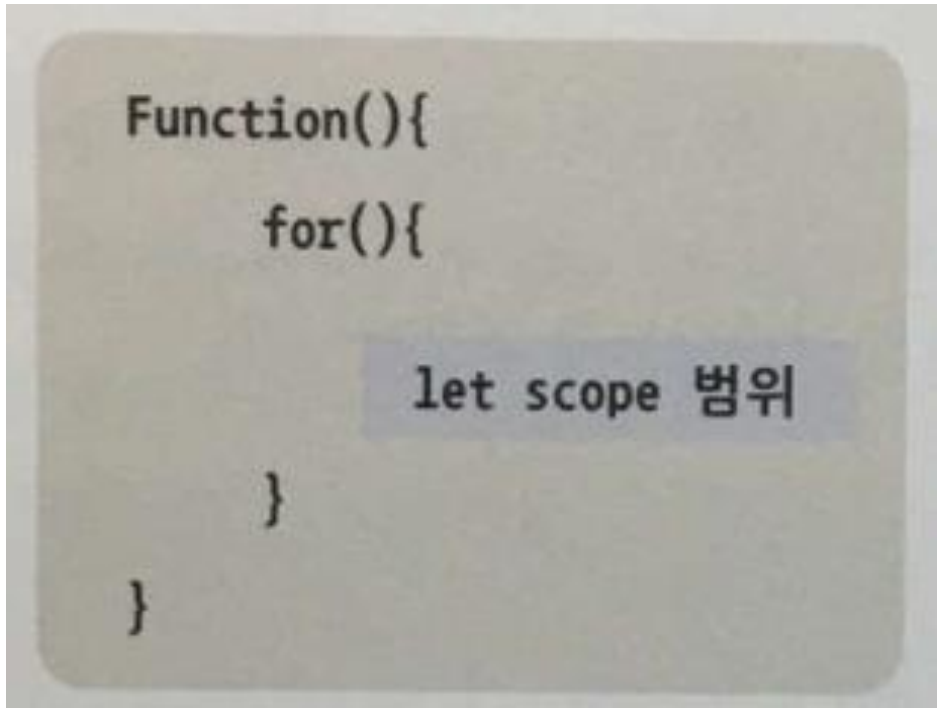
- let은 var와 다르게 Block에서 Scope가 설정된다.
  - var는 함수 Block에서 Scope가 설정되지만, 그 외 Block에서는 Scope가 설정되지 않아서 변수가 공유된다.

```
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.)

- var는 반복문 안에서 변수가 공유되는 문제가 있었는데, let으로 이를 개선했다.
  - 이는 반복문 안에 비동기 함수를 호출할 경우 문제가 발생할 수 있다.




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

let (Cont.)

- var는 반복문 안에서 변수가 공유되는 문제가 있었는데, let으로 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);
}
```

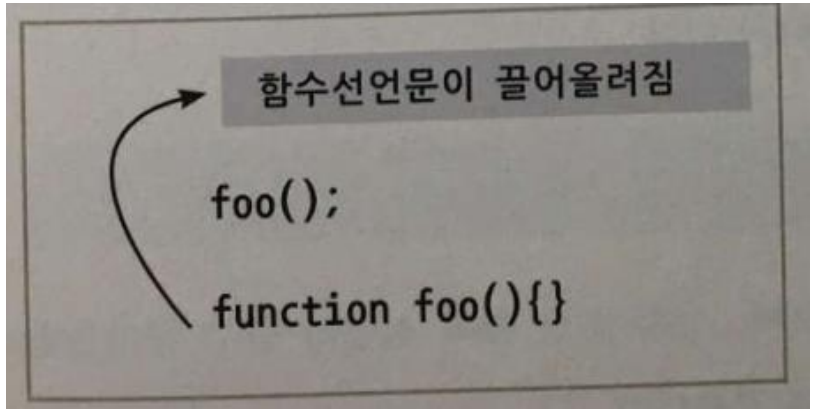
let (Cont.)

- let은 같은 Scope 내에서 변수 중복 선언이 불가능하다.
  - var는 같은 Scope내에서 변수 중복 선언할 때 이전에 선언된 변수가 덮어씌워지지만, let은 이를 허용하지 않는다.
  - 변수 중복 선언시 SyntaxError 발생

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

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 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.)

- let은 함수 끌어올림(Hoisting)이 되지 않는다.
  - var는 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.

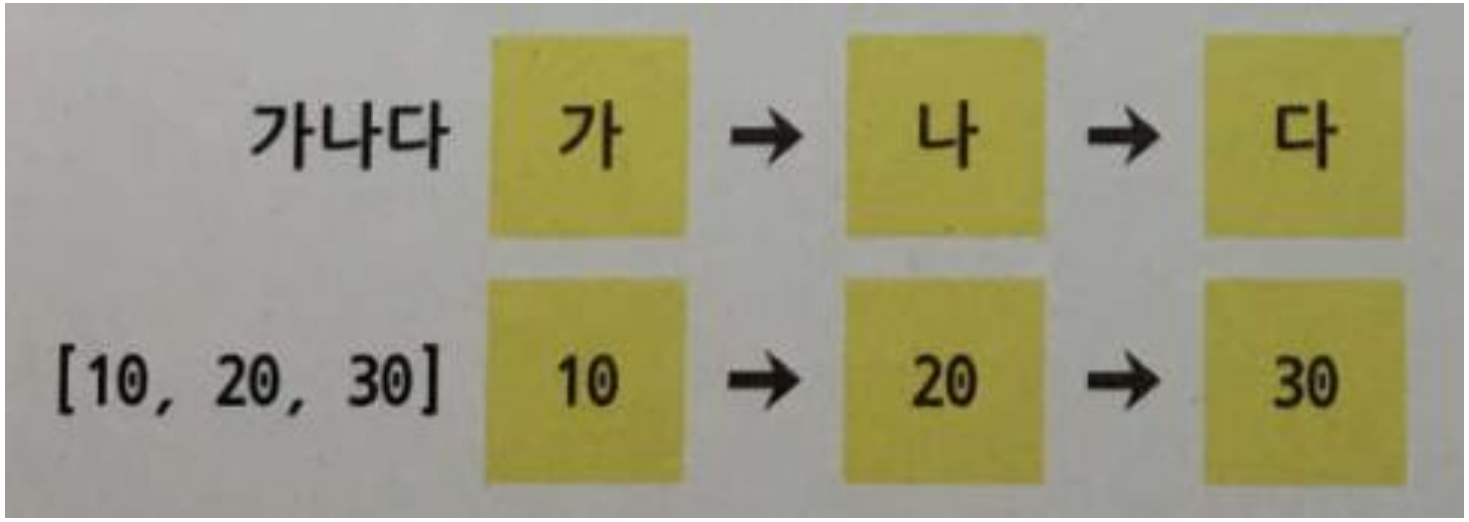
```
const MY_NAME;    //SyntaxError 발생
const MY_NAME = 'Sujan';
MY_NAME = 'Smith';    //TypeError 발생
```

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

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)
  - JavaScript객체 중 **Array**, **String**, **Map**, **Set**, **arguments**
  - **Object** 객체는 제외



iterable Protocol and iterable Object (Cont.)

- iterable Protocol
  - String Iteration

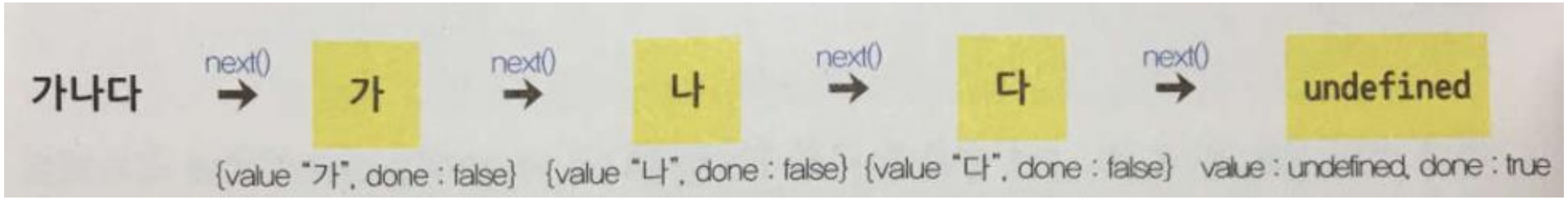
```
let str = '가나다';
for(let value of str){
    console.log(value); //'가', '나', '다'
}
```

  - 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 value와 done을 갖는다.
- value는 실제 값이 할당
- done은 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 호출시 순차적으로 열거되며, 열거된 값이 객체**(value값, done:열거 완료 여부)**이어야 한다.	**iterator** protocol을 따르도록 구현하거나, **iterable** 객체로 부터 **@@iterator** method를 호출하여 참조 가능.

for…of Statement

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

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

for…of Statement (Cont.)

- 문자열이 **@@iterable** 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…of문은 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
}
```

Template Literal

- 문자열 안에 표현식을 포함시킬 수 있고, 여러 줄 작성을 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 문자열을 출력할 수 있다.→ Tagged template literal

Template Literal (Cont.)

- 여러 줄 문자열
  - 문자열을 여러 줄로 작성하려면 \n을 입력해야 했다.
  - 또는 + 연산자 사용
  - Template Literal은 +연산자 없이 여러 줄 작성이 가능
  - 줄 바꿈시 자동으로 \n문자가 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 끝맺음 하고 +연산자로 표현식을 연결하여 작성해야 했다.
  - Template Literal은 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);
```

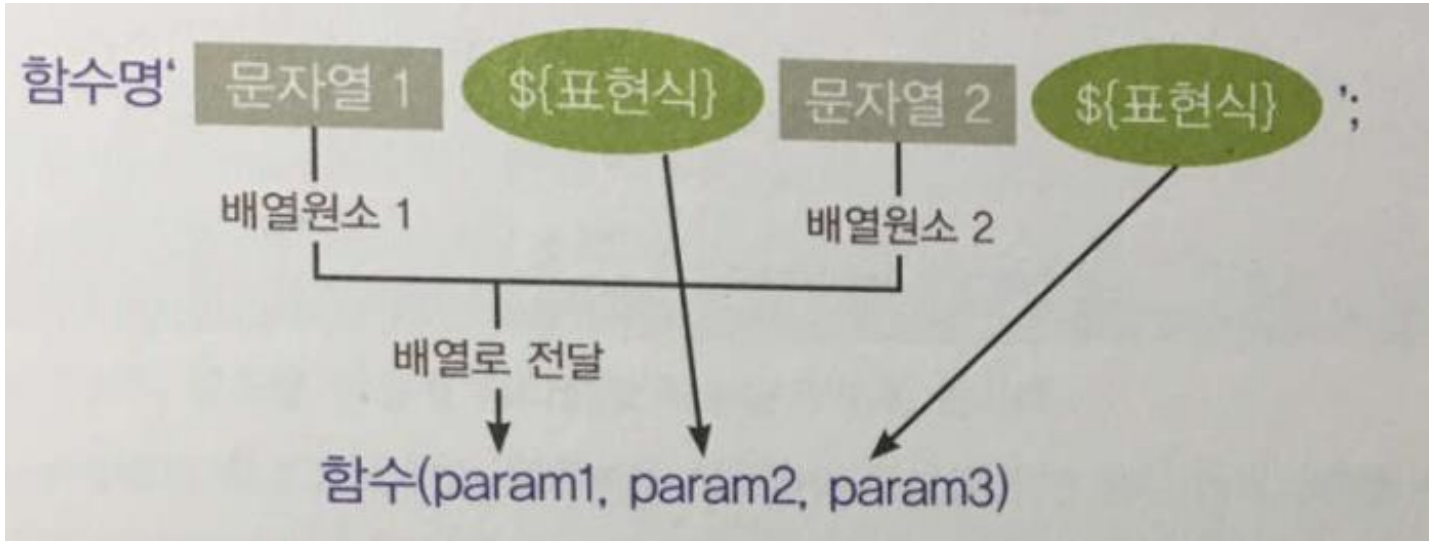
Template Literal (Cont.)

- 보간 표현법
  - 일반 문자열에 표현식을 삽입하려면 문자열을 끝맺음 하고 +연산자로 표현식을 연결하여 작성해야 했다.
  - Template Literal은 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 Literal이 올 경우 함수를 호출한다.
  - 함수의 인수로 Template Literal이 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 Literal이 올 경우 함수를 호출한다.
  - 함수의 인수로 Template Literal이 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);
```

내장 객체

Generator

- **generator** 함수로부터 반환된 값.
- **iterable** protocol과 **iterator** protocol을 따른다.
- 앞 Slide처럼 위의 2 protocol의 규약을 준수하는 객체는 **@@iterator()**와 **next()**를 구현해야 하는데, 이를 작성하기가 쉽지 않다.
- 이를 좀 더 쉽게 구현하도록 하는 것이 **generator** 함수이다.

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

Generator (Cont.)

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

Generator (Cont.)

- **iterable** protocol / **iterator** protocol을 준수하는 **iterable** 객체를 작성한 Code와 **generator** 함수를 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>
```

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 읽어 온다.

```
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` : `Map`에 추가된 항목 수

Map (Cont.)

- `Map` Methods
  - `set(key, value)` : `Map`에 새로운 항목 추가하고 Instance 반환
  - `get(key)` : key를 갖는 항목의 value 값 반환
  - `clear()` : `Map`의 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)` : `Map`에 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.

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

```
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.)

- **entries()**
  - **Map**의 항목을 열거할 수 있는 **iterator** 객체 반환
  - **iterator** 객체에 **next()** 호출 시 반환되는 객체의 value 속성값은 **Map**의 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
  - **유의할 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 알려 줌.

```
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}
```

Set

- Map과 다르게 value들로 구성된 Collection이다.
- 물론, Array도 value들로만 구성된 Collection 이지만, 차이가 있다.
- Set은 Array처럼 index로 값을 읽을 수 없고, 열거를 통해서만 값을 얻을 수 있다.
- Set은 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) : Set**에 새로운 항목을 추가하고 **Set** instance를 반환
  - **clear()** : **Set**의 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)** : **Set**에 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()**
  - **Set**의 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>
```

Symbol

- 새로 추가된 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.
- Syntax

```
let symbol = Symbol(description);
```

- **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>
```

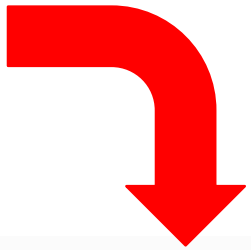
Symbol (Cont.)

- 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>
```

연산자

Spread Syntax

- 펼침 연산자.
- 배열의 원소 또는 객체의 속성 등을 펼쳐서 할당한다.
- 배열 원소 전부를 한 번에 다른 literal 배열 원소에 포함시킬 수 있다.
- ... 으로 표기

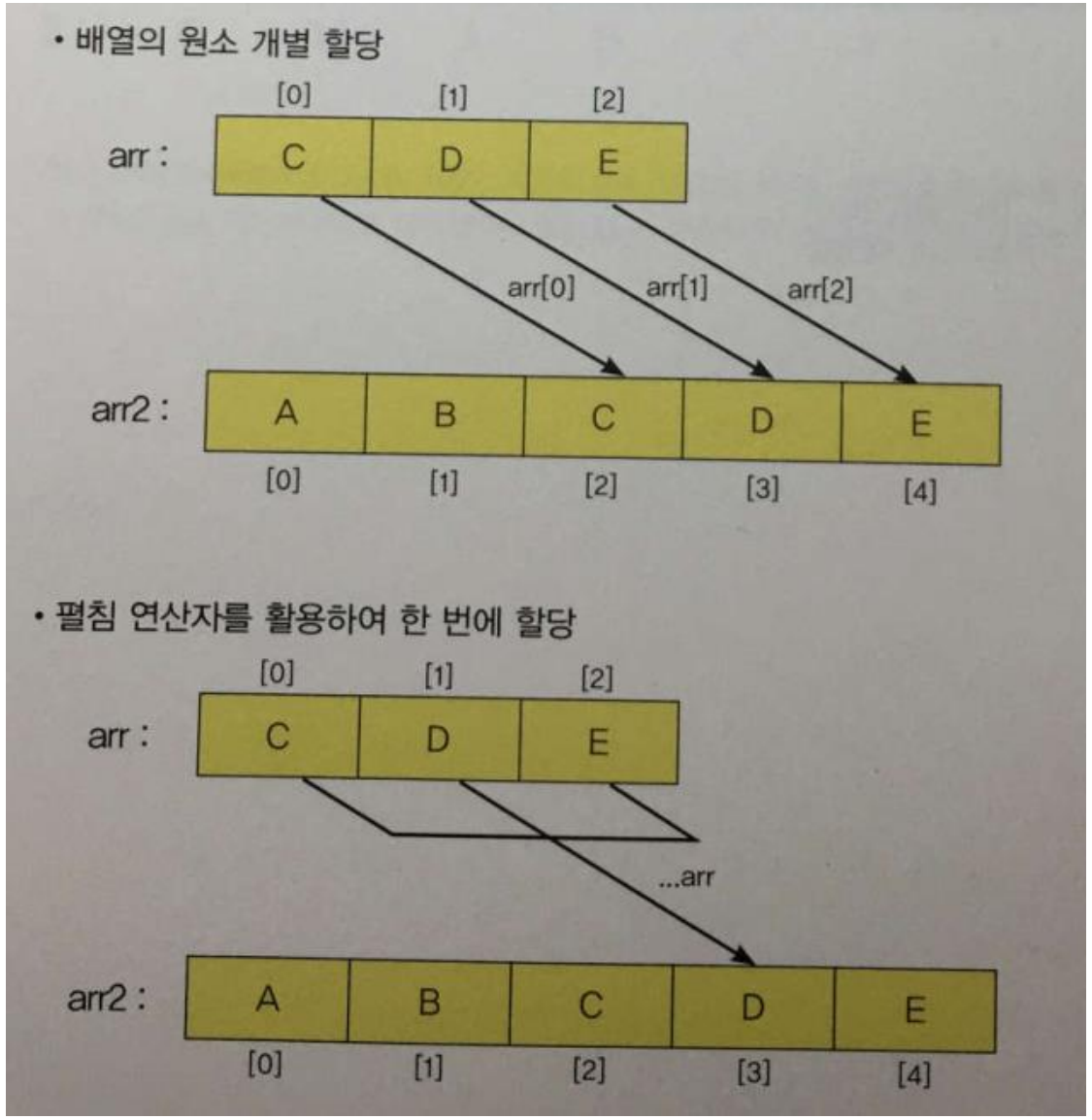


image from : 김규태, <만들면서 이해하는 ECMAScript 6>, 앤써북, 2018, p.114.

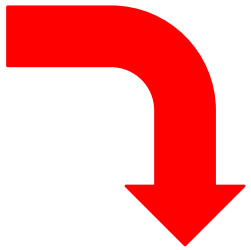
Spread Syntax (Cont.)

- 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.

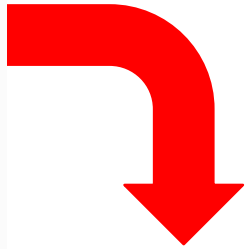
```
<script>

    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>
```

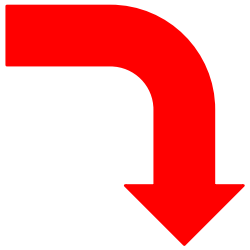
Spread Syntax (Cont.)

- 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>
```

Destructuring

- 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.

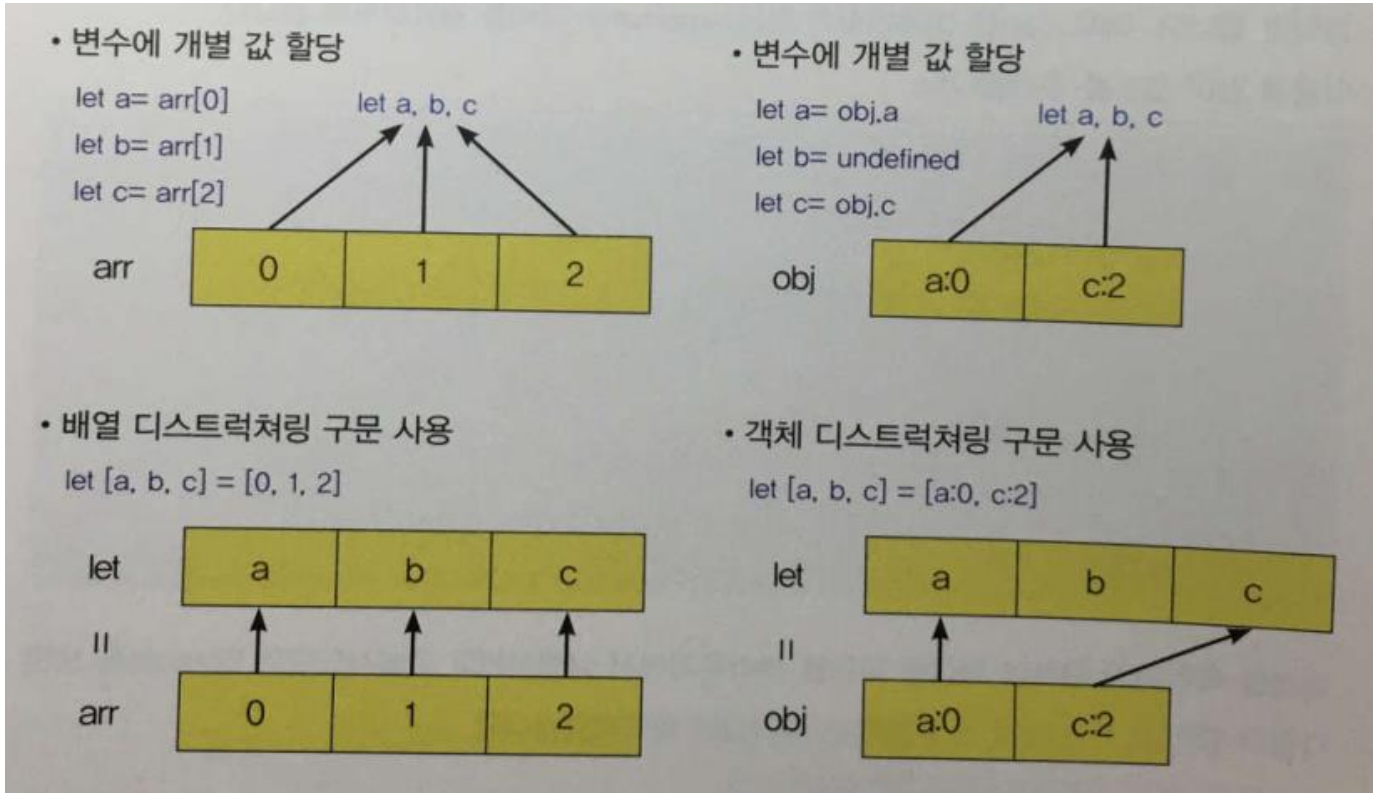


image from : 김규태, <만들면서 이해하는 ECMAScript 6>, 앤써북, 2018, p.120.

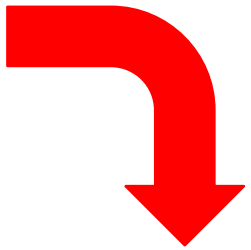
Destructuring (Cont.)

- 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 (Cont.)

- 배열의 Destructuring
  - 일부 원소 생략 가능

```
<script>

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

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

</script>
```

Destructuring (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 (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

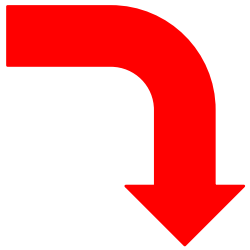
Destructuring (Cont.)

- 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 (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 (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 (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>
```

함수

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>
```

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>
```

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 발생한다.

```
<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.)

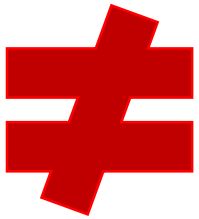
- Arrow Function은 일반 함수와 다르게 **this**를 생성하지 않는다.

```
<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: ƒ, Symbol(Symbol.iterator): ƒ]

</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>
```