

# 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

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

# 기본 문법



# let

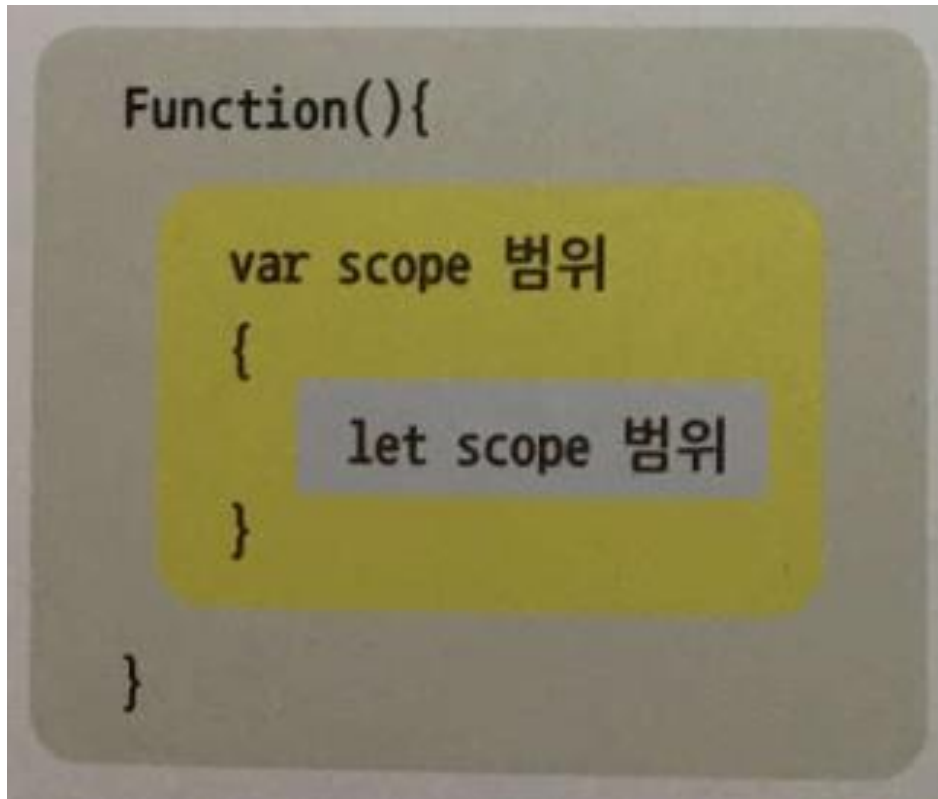
## ■ var 변수의 문제점

- 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.



```
var a = 100; // 변수 a 선언
```

```
function f(){  
    var a = 200; // 함수 Block 안에서 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);  
}
```

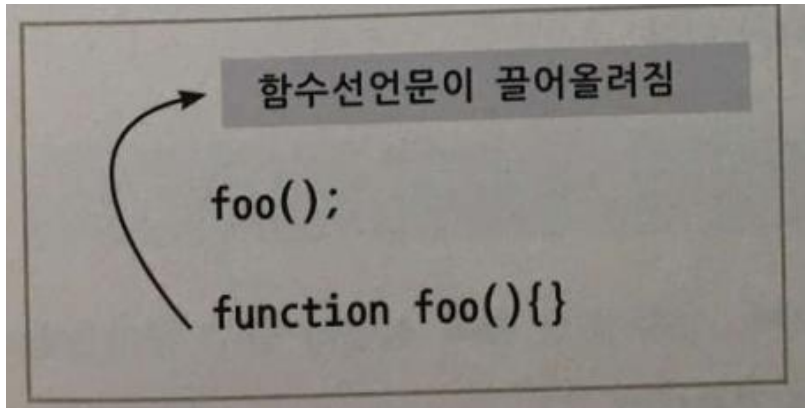
## let (Cont.)

- **let**은 같은 Scope 내에서 변수 중복 선언이 불가능하다.
  - **var**는 같은 Scope내에서 변수 중복 선언할 때 이전에 선언된 변수가 덮어쓰워지지만, **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.



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

## 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
```

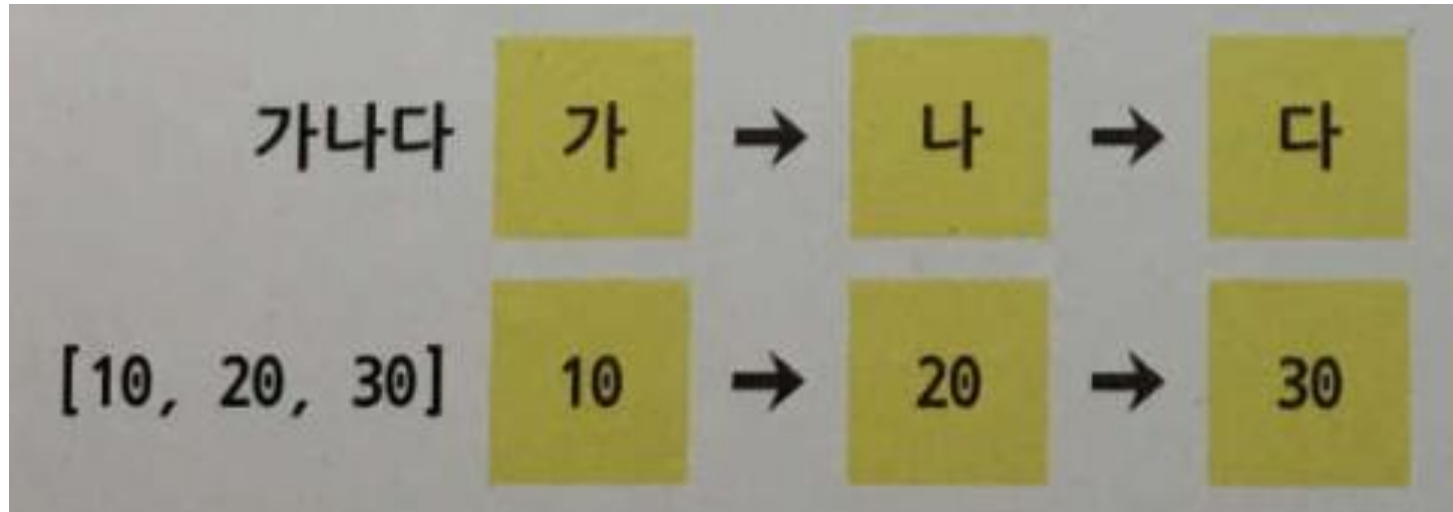
# let 과 const 정리

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

# 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'  
}
```

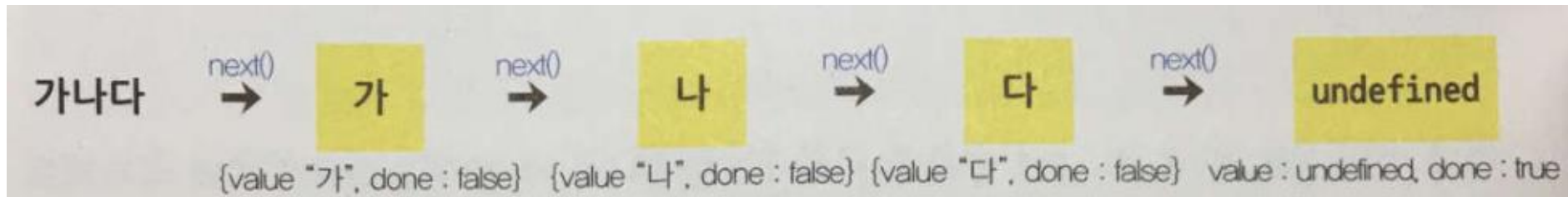
### ● Array Iteration

```
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 **value**와 **done**을 갖는다.
- **value**는 실제 값이 할당
- **done**은 열거의 끝임을 알려준다.
  - 열거가 끝인 경우 **true**
  - 그렇지 않을 경우 **false**
- Iterator 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 Iterator 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 Iterator 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}
```

## 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 Iterator 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  
}
```



# Template Literal

- 문자열 안에 표현식을 포함시킬 수 있고, 여러 줄 작성을 허용하여 간편하게 문자열을 만들 수 있도록 해준다.
- 문자열과 다르게 따옴표 대신 역따옴표(`) 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 Literal의 값이 함수에 전달되며, 함수에서 값을 조작하여 Template 문자열을 출력할 수 있다. → Tagged template literal

# Template Literal (Cont.)

## ■ 여러 줄 문자열

- 문자열을 여러 줄로 작성하려면 `\n`을 입력해야 했다.
- 또는 `+` 연산자 사용
- Template Literal은 `+`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 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 끝맺음하고 **+**연산자로 표현식을 연결하여 작성해야 했다.
- Template Literal은 문자열 끝맺음없이 보간 표현법을 이용하여 보다 쉽게 작성이 가능하다.

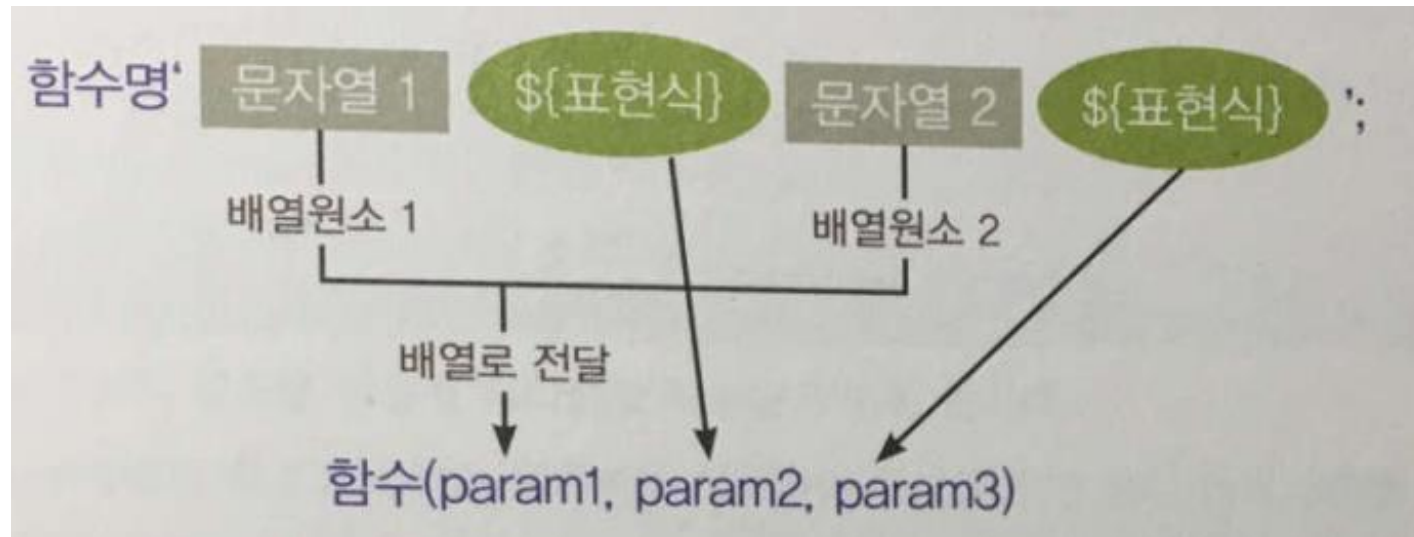
## ■ Template Literal에 표현식 포함

```
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 큼니다.';  
}
```

```
let a = 100;  
let b = 200;  
let str = tagged`A와 B 둘 중 ${a}, ${b}`;  
console.log(str);
```

# Type 배열

- 배열과 매우 유사한 객체
- **Binary Data**를 보다 빨리 접근하고 조작하도록 하기 위해 추가되었다.
- JavaScript에서 File을 불러서 처리하거나, Audio나 Video 처리를 위한 Binary Data의 접근이 최근 많아졌다.
- 배열이 원소의 개수가 가변적이고 모든 값을 허용한 반면, Type 배열은 Binary Data만 허용하고 원하는 Bit를 선택할 수 있다.
- 대부분의 배열 API를 동일하게 제공하지만, **push**와 **pop**는 제외다.
- **Buffer**와 **View**로 나뉜다.
- **Buffer(ArrayBuffer)**는 단순히 Data Chunk를 나타내는 객체
- 스스로 읽고 쓸 수 없고, **View**를 통해서 저장된 Data의 조작이 가능



# Type 배열 (Cont.)

## ■ Buffer

- **ArrayBuffer** Class의 생성자 객체
- Class 호출시 지정한 Byte 크기의 Buffer 생성
- 직접 Data의 조작이 불가능
- 특정 Type의 View 생성자 객체를 통해 Read, Write 가능

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

# Type 배열 (Cont.)

## ■ View

- Typed Array Views
- 유형별로 여러 Class 제공
- **UintClamedArray**는 별도로 0 ~ 255 사이의 숫자를 허용하는 Type
- Class 호출 시 지정한 Buffer의 Byte만큼 담을 수 있는 배열형태의 생성자를 생성하고, Class 이름의 Bit에 따라 원소의 수가 결정되고 초기값으로 0을 지정한다.

# Type 배열 (Cont.)

## ■ View

	Byte	값의 범위	설명
Int8Array	1	-128 ~ 127	8비트 정수형
Uint8Array	1	0 ~ 255	8비트 양의 정수형
Uint8ClampedArray	1	0 ~ 255	8비트 양의 정수형(0 ~ 255만 허용)
Int16Array	2	-32768 ~ 32767	16비트 정수형
Uint16Array	2	0 ~ 65535	16비트 양의 정수형
Int32Array	4	-2147483648 ~ 2147483647	32비트 정수형
Uint32Array	4	0 ~ 4294967295	32비트 양의 정수형
Float32Array	4	$1.2 \times 10^{-38} \sim 3.4 \times 10^{38}$	32비트 부동소수점형
Float64Array	8	$5.0 \times 10^{-324} \sim 1.8 \times 10^{308}$	64비트 부동소수점형

## Type 배열 (Cont.)

- 다음 Code를 보자.
- 16 Byte Buffer를 생성하여 32 Bit 정수형 View를 선언하고 Buffer를 지정한다.
- 16 Byte를 Bit로 바꾸면 128 Bit이므로, 32 Bit View에서는 4개로 나뉘어 원소 4개의 초기값이 0인 32 Bit 양의 정수형 배열이 된다.

```
const buffer = new ArrayBuffer(16);  
const view = new Uint32Array(buffer);  
console.log(view); //Uint32Array(4) [0, 0, 0, 0]
```