


# ECMAScript 2015

장민창 ( [mcjang@hucloud.co.kr](mailto:mcjang@hucloud.co.kr) )

# ECMA Script 2015는 무엇인가?

- ECMA International의 ECMA-262에 근거한 표준 스크립트 언어
- ECMA Script 2015 혹은 ES6 라고 불린다.
- 최초 ECMA Script는 브라우저 언어인 Javascript와 Jscript간 차이를 줄이기 위한 공통 스펙 제안으로 출발 (1992, ECMA-262)
- ECMA? 
  - 1961년 설립된 국제 표준화 기구.
  - European Computer Manufacturers Association.
  - 유럽에서 컴퓨터 시스템을 표준화하기 위해 설립됨.
    - **주요 규격**
    - ECMA-119 – CD-ROM 볼륨 및 파일 구조 표준화
    - **ECMA-262 – ECMAScript 언어 규격 표준화**
    - ECMA-334 – C# 언어 규격 표준화
    - ECMA-335 – CLI(공통 언어 기반) 표준화
    - ECMA-404 – JSON 표준화

# ECMA Script 2015 지원 현황

COMPAT ES

ECMAScript

5

6

2016+

next

intl

non-standard

compatibility table

Flattr

by kangax & webbedspace & zloirock

Fork

549

Sort by Engine types

Show obsolete platforms

Show unstable platforms

V8

SpiderMonkey

JavaScriptCore

Chakra

Carakan

KJS

Other

Minor difference (1 point)

Small feature (2 points)

Medium feature (4 points)

Large feature (8 points)

		97%	Compilers/polyfills					Desktop									
		56%	71%	48%	59%	17%	5%	11%	96%	96%	96%	94%	97%	97%	97%		
Feature name	Current browser	Traceur	Babel + core-js <sup>[2]</sup>	Closure	TypeScript + core-js	es6-shim	Kong 4.14 <sup>[3]</sup>	IE 11	Edge 15	Edge 16	Edge 17 Preview	FF 52 ESR	FF 56	FF 57	FF 58 Beta	FF 59	
Optimisation																	
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	
Syntax																	
default function parameters	7/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	
rest parameters	5/5	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
spread (...) operator	15/15	15/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	
object literal extensions	6/6	6/6	6/6	4/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	
for..of loops	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	9/9	9/9	9/9	7/9	9/9	9/9	9/9	9/9	
octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	
template literals	5/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
RegExp "y" and "u" flags	5/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
destructuring, declarations	22/22	20/22	21/22	20/22	15/22	0/22	0/22	0/22	22/22	22/22	22/22	21/22	22/22	22/22	22/22	22/22	
destructuring, assignment	24/24	23/24	24/24	21/24	19/24	0/24	0/24	0/24	24/24	24/24	24/24	23/24	24/24	24/24	24/24	24/24	
destructuring, parameters	24/24	19/24	21/24	18/24	16/24	0/24	0/24	0/24	23/24	23/24	23/24	21/24	24/24	24/24	24/24	24/24	
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	1/2	2/2	2/2	2/2	2/2	
new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	
Bindings																	
const	16/16	14/16	14/16	14/16	14/16	0/16	2/16	12/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	16/16	
let	12/12	10/12	10/12														
block-level function declaration <sup>[15]</sup>	Yes	Yes	Yes														

https://kangax.github.io/compat-table/es6/

<https://kangax.github.io/compat-table/es6/>

# 목차

- 1. Javascript 기초
- 2. 블록바인딩
- 3. 문자열
- 4. 함수
- 5. 확장된 객체
- 6. 구조 분해
- 7. 심벌(Symbol)
- 8. Set / Map
- 9. 이터레이터 / 제네레이터
- 10. 클래스
- 11. 프로미스와 비동기 프로그래밍
- 12. 모듈로 캡슐화하기

## 1장

# Javascript 기초

ECMAScript 2015를 살펴보기전에 필요한 내용을 학습한다.

# Comment (주석)

- 코드들에 대한 설명
- 복잡한 코드에 대한 풀이를 작성하거나
- 코드에 작성된 참고자료들의 출처를 작성할 때 사용된다.

- Single Line Comment

```
// Double Slash 로 주석을 작성할 수 있다.  
var count = 5; // 코드의 바로 옆에 작성할 수도 있다.
```

- Multi Line Comment

```
/*  
 * 여러 줄의 주석을 작성할 때는 멀티라인 주석을 사용한다.  
 * 주석...  
 */  
var count = 5;
```

# 연산자

- 빈번하게 사용되는 비교 연산자들

## 크다 (Great than)

```
var number1 = 10;  
var number2 = 20;  
var result = number2 > number1;
```

result → true

## 크거나 같다 (Great than or equals)

```
var number1 = 10;  
var number2 = 10;  
var result = number2 >= number1;
```

result → true

## 작다 (Less than)

```
var number1 = 10;  
var number2 = 20;  
var result = number1 < number2;
```

result → true

## 작거나 같다 (Less than or equals)

```
var number1 = 10;  
var number2 = 10;  
var result = number1 <= number2;
```

result → true

## 같다 (Equals)

```
var number1 = 10;  
var number2 = 20;  
var result = number2 == number1;
```

result → false

## 다르다 (Not Equals)

```
var number1 = 10;  
var number2 = 10;  
var result = number2 != number1;
```

result → false

# 연산자

- 여러 비교연산자를 연결하는 논리 연산자

## 그리고 (AND)

```
var number1 = 10;  
var number2 = 20;  
var number3 = 30;  
var result = number1 < number2 && number1 < number3 ;
```

두 개의 조건이 모두 *true* 일 때만 *true*가 된다.

```
var result = 10 < 20 && 10 < 30;
```

```
var result = true && true;
```

result → true

## 또는 (OR)

```
var number1 = 10;  
var number2 = 20;  
var number3 = 5;  
var result = number1 < number2 || number1 < number3 ;
```

두 개 중 하나라도 *true* 일 때 *true*가 된다.

```
var result = 10 < 20 || 10 < 5;
```

```
var result = true || false;
```

result → true



# 반복문 (Loop)

- 동일한 코드의 반복적인 사용이 필요할 때 사용함

```
var number = 1;
```


```
console.log(number + "번 손님 응대 중입니다..");  
number++;
```

```
console.log(number + "번 손님 응대 중입니다..");  
number++;
```

```
console.log(number + "번 손님 응대 중입니다..");  
number++;
```

```
console.log(number + "번 손님 응대 중입니다..");  
number++;
```

같은 코드가 반복됨



# 반복문 (Loop)

- 동일한 코드의 반복적인 사용이 필요할 때 사용함

반복에 필요한 값을 초기화 함.

보통 ++, -- 등을 이용해 반복을 제어함.

```
for ( 반복 값 초기화; 반복 진행 여부 체크; 반복 문장 실행 후 실행될 증감식 ){  
    반복 문장  
}
```

반복 문장이 실행되기 이전에 체크함.

*true* 라면 반복문장을 실행하고, *false* 라면 반복문을 종료함.

```
1  
2  
3  
4  
for ( var i = 0; i < 5; i++ ) {  
    console.log(i + "번째 손님 응대 중입니다.");  
}
```

# 반복문 (Loop)

- 동일한 코드의 반복적인 사용이 필요할 때 사용함

```
for ( var i = 0; i < 5; i++ ) {  
    console.log(i + "번째 손님 응대 중입니다.");  
}
```

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.
1	TRUE	1 번째 손님 응대 중입니다.
2	TRUE	2 번째 손님 응대 중입니다.
3	TRUE	3 번째 손님 응대 중입니다.
4	TRUE	4 번째 손님 응대 중입니다.
5	FALSE	STOP!

# 제어문 (if ~ else if ~ else)

- 코드의 진행 상황을 제어할 때 사용.

```
if ( 수행 여부 체크 ) {  
    수행 코드  
}
```

```
var number1 = 20;  
var number2 = 30;  
if ( number1 이 number2 보다 작다면.. ) {  
    “number1” 이 “number2” 보다 작습니다.를 출력하는 코드  
}
```

조건이 true 일 때 실행한다.

```
var number1 = 20;  
var number2 = 30;  
if ( number1 < number2 ) {  
    console.log(number1 + “이 ” + number2 + “보다 작습니다.”);  
}
```

→ 20이 30보다 작습니다.

# 제어문 (if ~ else if ~ else)

- if ~ else

```
if ( 수행 여부 체크 ) {  
    수행 코드1  
} else {  
    수행 코드2  
}
```

```
var number1 = 40;  
var number2 = 30;  
if ( number1 < number2 보다 작다면.. ) {  
    "number1" 이 "number2" 보다 작습니다.를 출력하는 코드  
} else {  
    "number1" 이 "number2" 보다 큼니다.를 출력하는 코드  
}
```

조건이 *true* 일 때 실행한다.

조건이 *false* 일 때 실행한다.

```
var number1 = 40;  
var number2 = 30;  
if ( number1 < number2 ) {  
    console.log(number1 + "이 " + number2 + "보다 작습니다.");  
} else {  
    console.log(number1 + "이 " + number2 + "보다 큼니다.");  
}
```

→ 40이 30보다 큼니다.

# 제어문 (if ~ else if ~ else)

- if ~ else if ~ else

```
if (수행 여부 체크1) {  
    수행 코드1  
} else if (수행 여부 체크2) {  
    수행 코드2  
} else {  
    수행 코드3  
}
```

```
var number1 = 30;  
var number2 = 20;  
var number3 = 50;  
if (number1 이 number2 보다 작다면..) {  
    "number1" 이 "number2" 보다 작습니다.를 출력하는 코드  
} else if (number1 이 number3 보다 작다면..) {  
    "number1" 이 "number3" 보다 작습니다.를 출력하는 코드  
} else {  
    "number1" 은 "number2", "number3" 보다 큼니다.를 출력하는 코드  
}
```

조건이 *true* 일 때 실행한다.

조건이 *false*이고 조건이 *true* 일 때 실행한다.

모든 조건이 *false* 일 때 실행한다.

# 제어문 (if ~ else if ~ else)

- if 와 논리 연산자

```
var number1 = 10;  
var number2 = 20;  
var number3 = 50;  
  
if ( number1 < number2 && number1 < number3 ) {  
    console.log(number1 + "이 가장 작은 수 입니다.");  
}
```

# 배열

- 관련된 값들의 집합
  - 관련된 값들을 하나로 묶어서 관리한다.
  - 아래와 같이 유사한 성격의 값들을 각각의 변수로 나열하면, 코드에 실수가 생기거나 복잡해지기 쉽다.

```
var korScore = 100;  
var engScore = 90;  
var progScore = 100;
```

- 위 개별 변수를 배열로 만들면?

```
var scores = [100, 90, 100];
```

The diagram illustrates the mapping of individual variables to an array. The variable `engScore` is shown with a downward arrow pointing to the first element (100) of the array `scores`. The variable `korScore` is shown with a curved arrow pointing to the second element (90) of the array. The variable `progScore` is shown with a curved arrow pointing to the third element (100) of the array.

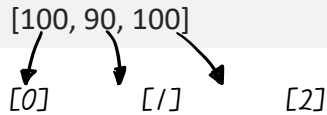
```
console.log( scores );
```

→ [100, 90, 100]



# 배열

- 배열 요소의 집합



- 배열 요소의 개별 참조

```
console.log( scores[0] );
```

→ 100

```
console.log( scores[1] );
```

→ 90

```
console.log( scores[2] );
```

→ 100

존재하지 않는 Index를 사용하면 *undefined*를 출력한다.

```
console.log( scores[3] );
```

→ undefined

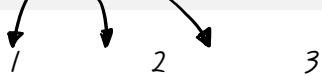
# 배열

- 배열 요소의 개수 확인 하기

```
console.log( scores.length );
```

→ 3

[100, 90, 100]



- 배열 요소 추가하기

```
scores.push(50);
```

↑  
*push* 하게 되면 배열의 마지막 요소로 추가된다.

- 배열 요소 제거하기

```
scores.pop(0);
```

↑  
*pop* 은 요소를 제거한다. 파라미터로 0 또는 -1을 입력할 수 있다.  
0 은 가장 마지막 요소를 제거한다.  
-1은 가장 첫 요소를 제거한다.

# 배열

- 배열과 반복문

```
for ( var i = 0; i < scores.length; i++ ) {  
    console.log(scores[i]);  
}
```

- for-in과 배열

```
for ( var i in scores ) {  
    console.log(scores[i]);  
}
```

- for-in은 배열의 인덱스를 순차 반복한다.

INDEX	0	1	2	3	4	5
VALUE	10	20	30	40	50	60

*scores[0]*   *scores[1]*   *scores[2]*   *scores[3]*   *scores[4]*   *scores[5]*

# 함수

- 일반 함수의 정의와 호출

```
function sayHello() {  
  console.log("안녕하세요?");  
}
```

```
sayHello();
```

→ 안녕하세요?

```
function calcAndPrintNumbers() {  
  var numberOne = 10;  
  var numberTwo = 20;  
  var result = numberOne + numberTwo;  
  console.log( numberOne + " + " + numberTwo + " = " + result );  
}
```

```
calcAndPrintNumbers();
```

→ 10 + 20 = 30

# 함수

- 함수 내에서 다른 함수를 호출할 수도 있다.

```
function calcAndPrintNumbers() {
```

```
  console.log( " 안녕하세요? " );  
  console.log( " 간단한 계산기 입니다. " );  
  console.log( " 이제 숫자를 더해 보겠습니다! " );
```

관련된 작업을 분리시킬 수 있다.  
특히, 반복적으로 사용될 수 있다면 더욱 분리시킬 필요가 있다.

```
  var numberOne = 10;  
  var numberTwo = 20;  
  var result = numberOne + numberTwo;  
  console.log( numberOne + " + " + numberTwo + " = " + result );  
}
```

```
function calcAndPrintNumbers() {
```

```
  sayWelcome();  
  var numberOne = 10;  
  var numberTwo = 20;  
  var result = numberOne + numberTwo;  
  console.log( numberOne + " + " + numberTwo + " = " + result );  
}
```

분리된 함수를 호출한다.

```
function sayWelcome() {
```


```
  console.log( " 안녕하세요? " );  
  console.log( " 간단한 계산기 입니다. " );  
  console.log( " 이제 숫자를 더해 보겠습니다! " );  
}
```

# 함수

- 값을 반환하는 함수

```
function getCalcNumbers() {  
  var numberOne = 10;  
  var numberTwo = 20;  
  var result = numberOne + numberTwo;  
  // console.log( numberOne + " + " + numberTwo + " = " + result );  
  return result;  
}
```

*return* 키워드는 호출자에게 함수의 호출 결과값을 전달할 수 있도록 한다.



```
var calcResult = getCalcNumbers();  
console.log(calcResult);
```

→ 30

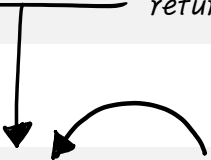
# 함수

- 값을 반환하는 함수

```
function getCalcNumbers() {  
  var numberOne = 10;  
  var numberTwo = 20;  
  var result = numberOne + numberTwo;  
  // console.log( numberOne + " + " + numberTwo + " = " + result );  
  return result;  
}
```

*return 키워드는 호출자에게 함수의 호출 결과값을 전달할 수 있도록 한다.*

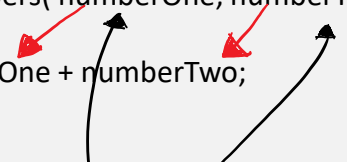
```
var calcResult = getCalcNumbers();  
console.log(calcResult);
```



→ 30

- 함수를 호출할 때 파라미터(매개변수)를 전달할 수도 있다.

```
function getCalcNumbers( numberOne, numberTwo ) {  
  sayWelcome();  
  var result = numberOne + numberTwo;  
  return result;  
}
```



```
var result = getCalcNumbers(10, 60);
```

→ 70

# 함수

- 함수 내의 함수 선언 및 호출

```
function getCalcNumbers ( numberOne, numberTwo ) {  
    var result = calcPlus ( numberOne, numberTwo );  
    function calcPlus ( numberOne, numberTwo ) {  
        return numberOne + numberTwo;  
    }  
    return result;  
}  
  
var result = getCalcNumbers(10, 60);
```

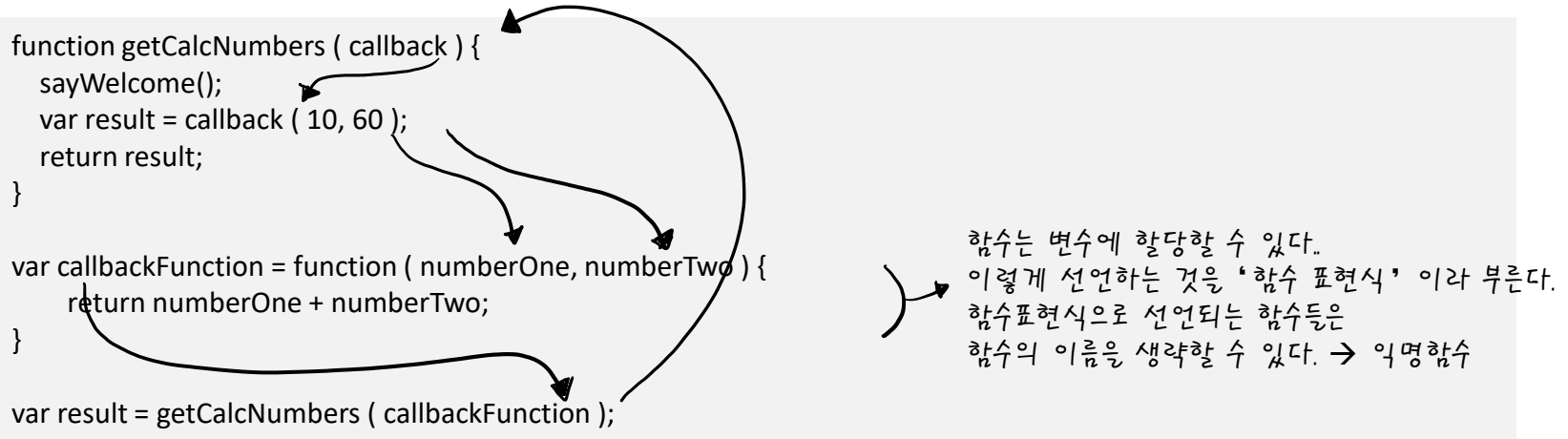
→ 70

- 함수 안에 함수가 다시 포함되어, 감추고 싶은 기능을 구현한다.
- Javascript에는 Access 제한자가 없기 때문에 기능을 감추고 싶다면, 함수 안의 함수로 구현해야 한다.



# 함수

- 함수를 파라미터(매개변수)로 전달



→ 70

- 함수에서 특정 작업이 완료되었을 때, 추가로 실행해야 하는 작업을 기술한 함수.
- 함수가 종료되는 시점이 불분명할 때, 콜백을 사용한다.
  - 예> Ajax와 같은 Network 작업
- 함수가 파라미터로 전달된다.

# 함수

- 함수가 함수를 리턴할 수도 있다.

```
function getCalcNumbers ( numberOne, numberTwo ) {
```

```
  return function () {  
    return numberOne + numberTwo;  
  }  
}
```

함수가 값이 아닌, 함수를 리턴한다.

```
var calc = getCalcNumbers(10, 80);
```

```
var result = calc();
```

리턴된 함수를 실행함.

→ 90

# 객체 리터럴

- Javascript에서 객체를 만드는 가장 일반적인 방법

```
var object = {}; // 객체 생성
```

```
object.name = "Jang Min Chang"; // object 객체에 name field 추가.  
object.job = "Developer"; // object 객체에 job field 추가.
```

```
console.log( object.name );
```

→ Jang Min Chang

```
console.log( object.job );
```

→ Developer

```
console.dir( object );
```

```
▼ Object ⓘ  
  job: "Developer"  
  name: "Jang Min Chang"  
  ▶ __proto__: Object  
> |
```

# 객체 리터럴

- 리터럴 내에는 함수도 포함될 수 있다.

```
var object = {}; // 객체 생성

object.name = "Jang Min Chang"; // object 객체에 name field 추가.
object.job = "Developer"; // object 객체에 job field 추가.
object.hello = function() {
    alert ( "안녕하세요." );
};
```

```
object.hello();
```

이 페이지 내용:

안녕하세요.

☐ 이 페이지가 추가적인 대화를 생성하지 않도록 차단합니다.

확인

## 2장


# 블록 바인딩

ECMAScript 2015에서 새롭게 추가된 Block-Level Scope 에 대해 학습한다.

# 기존 Javascript Variable Scope

- 기존의 범위는 Function base의 범위를 가짐.

```
function getValue(condition) {  
  
  if ( condition ) {  
    var value = "blue";  
  
    // Some code..  
    return value;  
  }  
  else {  
  
    // value는 여기서 undefined로 존재한다.  
    return null;  
  }  
  
  // value는 여기서 undefined로 존재한다.  
  
}
```



```
function getValue(condition) {  
  var value;  
  if ( condition ) {  
    value = "blue";  
  
    // Some code..  
    return value;  
  }  
  else {  
  
    // value는 여기서 undefined로 존재한다.  
    return null;  
  }  
  
  // value는 여기서 undefined로 존재한다.  
  
}
```

- Function 내부에서 선언된 모든 변수들은 Function 선언 아래쪽으로 이동되어 선언된다.
- 기존의 블록개념과 완전히 다른 개념.
- 이 현상을 "호이스팅" 이라 한다.

# undefined?

- Java에서 Null 과 같은 자료형
- 변수가 선언은 되어있지만 데이터가 초기화 되지 않은 상태

```
var variableA = "Value";  
var variableB = null;  
var variableC;
```

```
console.log(variableA);  
console.log(variableB);  
console.log(variableC);
```

Value  
Null  
Undefined

- Null은 개발자가 직접 할당해야 한다.

# ECMAScript Variable Scope

- ECMAScript 2015 에서  
Lexical Scope(Block-Level Scope)로 변경됨.
- Lexical Scope(Block-Level Scope)?
  - 함수 내부
  - 블록 내부( { 와 }를 사용하여 지정 )
- Hoisting(호이스팅)이 더 이상 발생하지 않는다.
- 변수(**let**)와 상수(**const**)로 구분해 지원한다.



# let(변수)

- var와 같은 문법으로 변수를 정의함.
- Hoisting 이 일어나지 않는다.

```
function getValue(condition) {  
  
  if ( condition ) {  
    let value = "blue";  
  
    // Some code..  
    return value;  
  }  
  else {  
  
    // value는 여기에 존재하지 않는다.  
    return null;  
  }  
  
  // value는 여기에 존재하지 않는다.  
  
}
```

- Value는 블록에 따라 제한적으로 존재한다.

# let(변수) – 재정의 금지

- var는 중복 정의시 Error를 발생시키지 않고 값을 덮어 쓴다.

```
var variableA = "Kim Yoona";  
console.log(variableA);
```

Kim Yoona

```
var variableA = "Kim Yuna";  
console.log(variableA);
```

Kim Yuna

- var와 let이 같은 Level에서 정의되면 Error를 발생시킨다.

```
var variableA = "Kim Yoona";  
console.log(variableA);
```

```
let variableA = "Kim Yuna";  
console.log(variableA);
```

SyntaxError: Identifier 'variableA' has already been declared

- 단, 아래처럼 작성하면 Error가 발생하지 않는다.

```
var variableA = "Kim Yoona";  
console.log(variableA);
```

Kim Yoona

```
if ( condition ) {  
  let variableA = "Kim Yuna";  
  console.log(variableA);  
}
```

Kim Yuna

# const(상수)

- 기존 Javascript에서 지원되지 않던 상수를 const 키워드로 지원함.
- 초기화되지 않으면 Error를 발생시킴.

```
// 유효한 상수
const maxItems = 30;

// 문법에러 : 초기화 되지 않음
const name;

console.log(maxItems);
console.log(name);
```

SyntaxError: Missing initializer in const declaration

- let과 같은 Block-Level Scope 를 가진다.

# const(상수)로 객체 선언하기

- const는 상수 뿐만 아니라 객체를 생성/관리하기에 적합하다.
- const는 바인딩(초기화/할당)을 변경하도록 막는 것  
→ 바인딩 된 값의 변경을 막지 않는다.

```
// Const에 객체 리터럴 할당
const person = {
  name : "Seo Tae Ji"
};

// 객체의 값 변경
person.name = " And Boys";

// const에 새로운 객체를 재할당 하려할 경우 에러 발생!
person = {
  name : " And Boys"
};
```

TypeError: Assignment to constant variable.

# Var를 사용할 때의 반복문의 문제점

- var는 전역객체로 활용되기 때문에 특히나 반복문에서 문제점이 많음.

```
var numbers = [];  
  
for ( var i = 0; i < 10; i++ ) {  
  numbers.push(function() {  
    console.log(i)  
  });  
}  
  
numbers.forEach(function(f) {  
  f();  
});
```

10  
10  
...

- 결과의 원인은?

```
numbers.push(function() {  
  console.log(i)  
});
```

- 반복문 내에 존재하는 위 코드의 i 변수는 항상 같은 값을 가지게 됨  
→ 함수내에서 공용으로 사용하는 변수.

# Var를 사용할 때의 반복문의 문제점

- 이전 코드를 제대로 사용하기 위해서는 복잡한 문법의 즉시 실행 함수 표현식을 사용해야 함.

```
var numbers = [];
```

```
for ( var i = 0; i < 10; i++ ) {  
  numbers.push((function(value) {  
    return function() {  
      console.log(value);  
    }  
  })(i));  
}
```

```
numbers.forEach(function(f) {  
  f();  
});
```

0  
1  
...

- 반복문 내에서 공유하고 있는 i 변수를 다른 function의 인자로 전달해 값을 복사해 사용함.

# 반복문 내의 let 사용

- let을 사용할 경우 복잡한 수식 없이 아래처럼 사용 가능하다.

```
var numbers = [];  
  
for ( let i = 0; i < 10; i++ ) {  
  numbers.push(function() {  
    console.log(i);  
  });  
}  
  
numbers.forEach(function(f) {  
  f();  
});
```

0  
1  
...

- let은 값이 참조될 때 마다 복사본을 전달한다.

# 반복문 내의 let 사용

- 반복문에서도 let을 사용할 수 있다.

```
const funcs = [];  
const object = {  
  a: true,  
  b: true,  
  c: true  
};  
  
for ( let key in object ) {  
  funcs.push(function() {  
    console.log(key, object[key]);  
  });  
}  
  
funcs.forEach(function(f) {  
  f();  
});
```

```
a true  
b true  
c true
```



## 3장

# 문자열

ECMAScript2015에 새롭게 추가된 문자열 함수들을 살펴보고 학습한다.

# 부분 문자열 식별하기

- 타 언어에 비해 부족했던 자바스크립트에 몇 가지 유용한 함수가 추가됨.

```
let message = "Hello world!";

console.log(message.startsWith("Hello")); // true
console.log(message.startsWith("hello")); // false
console.log(message.endsWith("!")); // true
console.log(message.includes("o")); // true
console.log(message.includes("O")); // false

console.log(message.startsWith("o", 4)); // true
console.log(message.endsWith("o", 8)); // true
console.log(message.includes("o", 8)); // false
```

- `String.startsWith("")` : 문자열의 시작점에서 주어진 문자를 찾으면 `true`, 그렇지 않으면 `false`를 반환. (시작점을 지정할 수도 있다)
- `String.endsWith("")` : 문자열의 끝에서 주어진 문자를 찾으면 `true`, 그렇지 않으면 `false`를 반환. (찾을 지점을 지정할 수도 있다)
- `String.includes("")` : 문자열의 어느 곳이든 주어진 문자를 찾으면 `true`, 그렇지 않으면 `false`를 반환. (찾을 지점을 지정할 수도 있다)

# 문자 위치 알아내기

- 기타 유용한 문자열 함수

```
let message = "Hello world!";  
  
console.log(message.indexOf("o")); // 4  
console.log(message.lastIndexOf("o")); // 7
```

- `String.indexOf("")` : 문자열에서 주어진 문자의 위치를 반환. 없으면 -1을 반환
- `String.lastIndexOf("")` : 문자열에서 주어진 문자의 가장 마지막 위치를 반환. 없으면 -1을 반환

```
let message = " Hello world! ";  
console.log(message);  
  
message = message.trim();  
console.log(message);
```

- `String.trim()` : 문자열에서 좌우 끝 공백을 모두 제거함.

# 문자 반복 시키기

- 문자열 반복하기

```
console.log("x".repeat(3)); // xxx
console.log("hello".repeat(2)); // hellohello
console.log("abc".repeat(4)) // abcabcabcabc
```

- 원본 문자열을 주어진 횟수만큼 반복해 추가해주는 함수.
- 주로 편의를 위한 함수.
- 특히 텍스트/아이디(DB-PK)를 조작할 때 유용하게 사용될 수 있음.

```
let idx = "10"; // 0000010 으로 변경.
let idxFormat = "0000000";

idx = "0".repeat(idxFormat.length - idx.length) + idx;
// idx = "0".repeat( 7 - 2 ) + id;
console.log(idx);
```

# 템플릿 리터럴 사용

- Javascript 에서 문자열을 이어 붙이는 방법은 복잡하다.

```
const year = 2018;
const month = 1;
const date = 16;

// 오늘은 2018년 1월 16일 입니다.
let message = "오늘은 " + year + "년 " + month + "월 " + date + "일 입니다."
console.log(message);
```

- 또한, 여러줄의 텍스트를 만든다면 더욱 그렇다.

```
const year = 2018;
const month = 1;
const date = 16;

// 오늘은 2018년 1월 16일 입니다.
// 내일은 몇일 인가요?
let message = "오늘은 " + year + "년 " + month + "월 " + date + "일 입니다.\n";
message += "내일은 몇일 인가요?";
console.log(message);
```

- 몇 줄 반복되다보면 매우 어지러워진다.

# 템플릿 리터럴 사용

- 템플릿 리터럴은 간단하게 문자열을 만들어 낼 수 있다.

```
const year = 2018;
const month = 1;
const date = 16;

// 오늘은 2018년 1월 16일 입니다.
let message = `오늘은 ${year}년 ${month}월 ${date}일 입니다.`;
console.log(message);
```

- 템플릿 리터럴은 백틱(`)을 사용해 표현한다.
- 백틱내에 변수를 조합하기 위해서 \${변수명} 을 사용한다.

```
const year = 2018;
const month = 1;
const date = 16;

// 오늘은 2018년 1월 16일 입니다.
let message = `오늘은 ${year}년 ${month_}월 ${date}일 입니다.`;
console.log(message);
```

- 위 처럼 잘못된 변수명 \${month\_} 을 사용하면 Reference Error를 발생시킨다.

# 템플릿 리터럴 사용

- 여러 줄의 텍스트를 만드려면 백틱 내에서 새로운 줄을 만들면 된다.

```
const year = 2018;
const month = 1;
const date = 16;

// 오늘은 2018년 1월 16일 입니다.
// 내일은 몇일 인가요?
let message = `오늘은 ${year}년 ${month}월 ${date}일 입니다.
내일은 몇일 인가요?`;
console.log(message);
```

- `${}` 내에서 계산식도 사용할 수 있다.

```
let count = 10;
let price = 50;

let message = `${count} items cost ${count * price} USD`;
console.log(message);
```

10 items cost 500 USD

## 4장

# 함수

ECMAScript2015에서 변경된 함수들을 살펴보고 학습한다.



# 함수

- ECMAScript의 가장 중요한 객체 중 하나.
- 함수를 파라미터로 혹은 객체로 저장하며, 다양하게 활용할 수 있음.
- ECMAScript 2015 이전 함수의 문제점.
  - 파라미터를 제대로 전달하지 않더라도 정상적으로 실행이 된다!

```
function foo(bar) {  
  console.log(bar);  
}
```

```
foo();  
foo("Bar");  
foo("Bar", "Foo");
```

```
undefined  
Bar  
Bar
```

- ECMAScript는 파라미터를 arguments 라는 객체를 통해 전달한다,.

```
function foo(bar) {  
  console.log(arguments);  
}
```

```
foo();  
foo("Bar");  
foo("Bar", "Foo");
```

```
Arguments(0) []  
Arguments(1) ["Bar"]  
Arguments(2) ["Bar", "Foo"]
```

# 함수

- 이런 특징 때문에 발생했던 비정상적인 코드

```
function foo(bar) {  
  if ( bar == undefined ) {  
    bar = "Init Bar";  
  }  
  console.log(bar);  
}
```

```
foo();  
foo("Bar");  
foo("Bar", "Foo");
```

```
Init Bar  
Bar  
Bar
```

- Undefined 체크 로직 때문에 복잡할 필요가 없는 코드가 복잡해지고 길어짐.
- ECMAScript 2015에서는 이런 불편함을 해소하기 위해 Default Parameter를 제공함.

# 함수 – Default Parameter

- 함수 선언할 때 파라미터의 기본값을 정의할 수 있다.

```
function foo(bar = "Init Bar") {  
  console.log(bar);  
}
```

```
foo();  
foo("Bar");  
foo("Bar", "Foo");
```

```
Init Bar  
Bar  
Bar
```

- 파라미터가 전달되지 않아 Undefined로 정의된다면, 자동으로 Default Parameter 의 값인 "Init Bar" 로 할당된다.
- 아래 코드도 가능하다.

```
function makeRequest(url, timeout = 3000, callback = function() { return "Basic CallBack"; }) {  
  console.log("Request URL ", url);  
  console.log("Timeout ", timeout);  
  console.log("Callback ", callback());  
  console.log("");  
}
```

```
// URL만 전달  
makeRequest("url");
```

```
// URL, Timeout 전달  
makeRequest("url", 1000);
```

```
// URL, Timeout, Callback 전달  
makeRequest("url", 2000, function() {  
  return "hello";  
});
```

# 함수 – Default Parameter

- Default Parameter의 값으로 함수를 사용할 수도 있다.

```
function basicCallback() {  
  return function() {  
    return "Basic Callback";  
  }  
}  
  
function basicCallback2() {  
  return "Basic Callback2";  
}  
  
function makeRequest(url, timeout = 3000, callback = basicCallback2) {  
  console.log("Request URL ", url);  
  console.log("Timeout ", timeout);  
  console.log("Callback ", callback());  
  console.log("");  
}
```

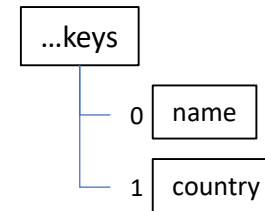
- 필요에 따라 아래처럼 사용할 수도 있다.

```
function makeRequest(url, timeout = 3000, callback = basicCallback()) {  
  console.log("Request URL ", url);  
  console.log("Timeout ", timeout);  
  console.log("Callback ", callback());  
  console.log("");  
}
```

# 함수 – 가변길이 파라미터

- 하나의 파라미터변수에 여러가지 값을 전달해, 배열처럼 사용할 수 있음.
  - arguments 객체를 대체하기 위해 설계됨.

```
function pick(object, ...keys) {  
  const result = {};  
  for ( let i in keys ) {  
    result[keys[i]] = object[keys[i]];  
  }  
  return result;  
}  
  
const object = {  
  name: 'Min Chang',  
  city: 'Seoul',  
  country: 'South Korea'  
};  
const result = pick(object, "name", "country");  
console.log(result);
```



Object {name: "Min Chang", country: "South Korea"}

- 가변길이 파라미터 뒤에는 다른 파라미터를 사용할 수 없다.

```
function pick(object, ...keys, next) {  
  ...  
}
```

- 가변길이 파라미터는 항상 마지막에 위치해야 한다.

# 함수 – Function

- 새로운 함수를 동적으로 생성하게 해준다.

```
const add = new Function("first", "second", "return first + second");  
console.log(add(10, 50));
```

60

- Default Parameter / 가변길이 파라미터를 모두 사용할 수 있다.
  - Default Parameter

```
const add = new Function("first", "second = first", "return first + second");  
console.log(add(10, 50));  
console.log(add(10));
```

60  
20

- 가변길이 파라미터

```
const pickFirst = new Function("...numbers", "return numbers[0]");  
console.log(pickFirst(50, 1, 10));
```

50

- 제약사항
  - 외부의 function을 참조할 수 없다.

```
function add2(first, second) {  
  return first + second;  
}  
const add = new Function("first", "second = first", "return add2(first + second);");
```

ReferenceError: add2 is not defined

# 함수 – 펼침(Spread) 연산자

- 배열을 단일 값으로 풀어해치는 연산자.

```
console.log("배열 출력 : ", [10, 20, 30]);  
console.log("펼침 연산자로 배열 출력 : ", ...[10, 20, 30]);
```

배열 출력 : Array(3) [10, 20, 30]  
펼침 연산자로 배열 출력 : 10 20 30

- 배열에 다른 배열을 추가할 때에도 유용하게 사용할 수 있다.
  - 가변길이 파라미터를 다른 배열에 추가하려 할 때.

```
function push(array, ...args) {  
  array.push(args);  
}
```

```
const array = [10, 20];  
push(array, 30, 50, 60);  
console.log(array);
```

Array(3) [10, 20, Array(3)]

- 가변길이 파라미터를 펼침연산자를 이용해 배열에 추가하려 할 때

```
function push(array, ...args) {  
  array.push(...args);  
}
```

```
const array = [10, 20];  
push(array, 30, 50, 60);  
console.log(array);
```

Array(5) [10, 20, 30, 50, 60]

# 함수 – 화살표(Fat Arrow) 함수

- 함수를 화살표(=>) 로 정의할 수 있는 새로운 방법
- 타 언어의 Lambda와 유사하다.
  - function 키워드를 사용하지 않는다.
  - => 이후 중괄호( { 와 } )의 여부에 따라 Return 유무가 결정된다.
- 일반적인 형태의 익명함수 (함수표현식)

```
const sum = function(first, second) {  
  return first + second;  
}  
console.log(sum(10, 20));
```

- 위 함수는 아래처럼 변경이 가능하다.

```
const sum = (first, second) => first + second;  
console.log(sum(10, 20));
```



# 함수 – 화살표(Fat Arrow) 함수

- 화살표 함수의 다양한 표현법

```
const fn = () => {  
  console.log("반환값이 없는 함수");  
};
```



```
const fn = function() {  
  console.log("반환값이 없는 함수");  
}
```

```
const fn = () => 10;
```



```
const fn = function() {  
  return 10;  
}
```

```
const fn = (value) => value * 2;
```



```
const fn = function(value) {  
  return value * 2;  
}
```

```
const fn = (...value) =>  
  Math.max(...value);
```



```
const fn = function(...value) {  
  return Math.max(...value);  
}
```

```
const fn = (first, second=10) =>  
  first + second;
```



```
const fn = function (first, second=10) {  
  return first + second;  
}
```

# 함수 – 화살표(Fat Arrow) 함수

- 화살표 함수로 즉시실행 함수를 만들수 있다.
  - 일반적인 형태의 즉시 실행 함수

```
const person = (function(name) {  
  return {  
    getName: function() {  
      return name;  
    }  
  };  
})("Seo Tae Ji");  
  
console.log(person.getName());
```

- 화살표 함수 형태의 즉시 실행 함수

```
const person = ((name) => {  
  return {  
    getName: () => name  
  };  
})("Seo Tae Ji");  
  
console.log(person.getName());
```

- 자질구레한 코드들이 사라진다.

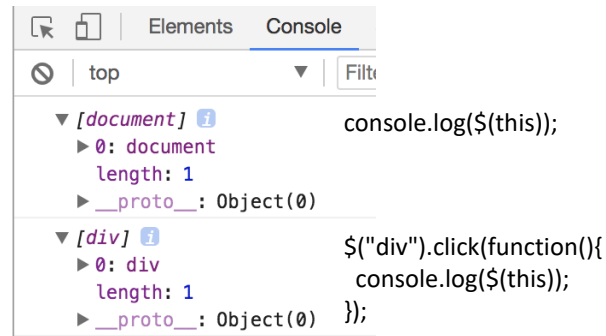
# 함수 – 화살표(Fat Arrow) 함수

- jQuery에서 화살표 함수를 이용할 때 주의할 점.

```
<!DOCTYPE html>
<html lang="ko">
  <head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>
    <script type="text/javascript">
      $.ready(function() {
        console.log($(this));
        $("div").click(function(){
          console.log($(this));
        });
      });
    </script>
  </head>
  <body>
    <div>Div1</div>
    <div>Div2</div>
    <div>Div3</div>
  </body>
</html>
```

- jQuery에서는 이벤트가 일어난 DOM이 this객체로 전달된다.

Div1  
Div2  
Div3



# 함수 – 화살표(Fat Arrow) 함수

- jQuery에서 화살표 함수를 이용할 때 주의할 점.

```
$.ready() => {  
  console.log($(this));  
  $("div").click( () => {  
    console.log($(this));  
  });  
};
```

- 화살표 함수로 바꾸었을 때, \$(this)는 window가 된다.

The screenshot shows a web browser's developer console with the 'Console' tab selected. On the left, there is a list of DOM elements: Div1, Div2, and Div3. The console displays two log messages:

- The first log message is from `console.log($(this));` and shows the `Window` object.
- The second log message is from `console.log($(this));` inside the `click` handler and also shows the `Window` object.

Both log messages show the `Window` object with properties like `frames`, `postMes`, `length: 1`, and `__proto__: Object(0)`. This demonstrates that `this` is not the DOM element as intended.

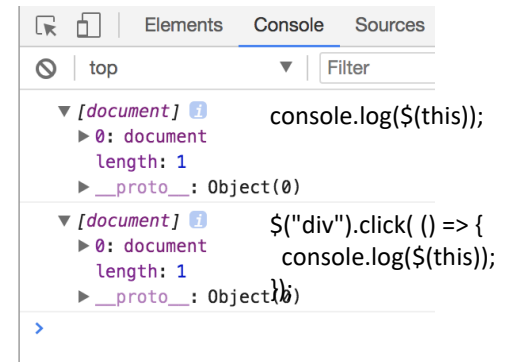
# 함수 – 화살표(Fat Arrow) 함수

- 왜?
  - this는 인접한 function을 기준으로 만들어지기 때문.
- 화살표 함수는 this객체를 가지지 못한다.
  - 인접한 function이 있을 경우, 그 function의 this를 사용하게 됨.
- 따라서 아래와 같은 결과가 나타날 수도 있다.

```
$.ready(function() {  
  console.log($(this));  
  $("div").click( () => {  
    console.log($(this));  
  });  
});
```



Div1  
Div2  
Div3

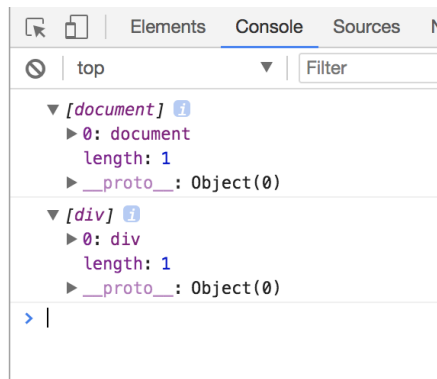


# 함수 – 화살표(Fat Arrow) 함수

- jQuery에서 화살표 함수를 사용하려면...
  - Event 객체의 `currentTarget` 을 가져와야 한다.

```
$.ready(function() {  
  console.log($(this));  
  $("div").on("click", (e) => {  
    const $this = $(e.currentTarget);  
    console.log($this);  
  });  
});
```

Div1  
Div2  
Div3



`console.log($(this));`

```
$("div").on("click", (e) => {  
  const $this = $(e.currentTarget);  
  console.log($this);  
});
```

## 5장

# 확장된 객체

ECMAScript2015에서 확장된 함수, 프로토타입, 객체 리터럴을 살펴보고 학습한다.

# 프로퍼티 생략 가능

- 객체 리터럴을 만들 때 사용되던 일반적인 방법

```
function createPerson(name, age) {  
  return {  
    name: name,  
    age: age  
  };  
}  
  
const person = createPerson("James Dean", 30);
```

- 거의 모든 코드에서 프로퍼티의 이름과 값이 담겨있는 변수의 이름이 동일함.
  - 객체의 프로퍼티에 접근하려면 **객체.프로퍼티명** 혹은 **객체["프로퍼티명"]**
- ECMAScript2015에서 프로퍼티 명과 변수명이 같을 경우 프로퍼티를 생략할 수 있도록 개선됨.

```
function createPerson(name, age) {  
  return {  
    name,  
    age  
  };  
}  
  
const person = createPerson("James Dean", 30);
```



# 간결해진 메소드 정의 방법

- 객체 리터럴내 function을 정의하는 방법도 변경됨.

```
function createPerson(name, age) {  
  return {  
    name,  
    age,  
    sayName: function() {  
      console.log(this.name);  
    }  
  };  
}  
  
const person = createPerson("James Dean", 30);  
person.sayName();
```

- function() 키워드가 생략되면서 간결하게 메소드를 정의할 수 있다.

```
function createPerson(name, age) {  
  return {  
    name,  
    age,  
    sayName() {  
      console.log(this.name);  
    }  
  };  
}  
  
const person = createPerson("James Dean", 30);  
person.sayName();
```

# 객체 복사(Mixin)

- ECMAScript5 이전에 객체를 복사하는 방법

```
const supplier = {  
  name: "Min Chang",  
  city: "Seoul",  
  sayName() {  
    console.log(this.name);  
  }  
};  
  
const receiver = {  
  address: "Guro",  
  city: "Jinju"  
};  
  
function mixin(receiver, supplier) {  
  Object.keys(supplier).forEach(function(key) {  
    receiver[key] = supplier[key];  
  });  
}  
  
mixin(receiver, supplier);  
console.log(supplier);  
console.log(receiver);  
receiver.sayName();
```

- Mixin 패턴을 이용해 얇은 복사(값만 복사) 방법이 많이 사용됨.

# 객체 복사(Object.assign)

- Mixin 패턴이 ECMAScript6에서 지원.
  - 아주 빈번하게 사용되는 패턴 → Script에서 자체 지원되도록 추가됨.

```
const supplier = {  
  name: "Min Chang",  
  city: "Seoul",  
  sayName() {  
    console.log(this.name);  
  }  
};  
  
const receiver = {  
  address: "Guro",  
  city: "Jinju"  
};  
  
Object.assign(receiver, supplier);  
console.log(supplier);  
console.log(receiver);  
receiver.sayName();
```

- Mixin, Object.assign을 사용하면, 중첩되는 Key/Value는 Supplier의 것으로 덮어쓰게 됨.

# 기존의 프로토타입

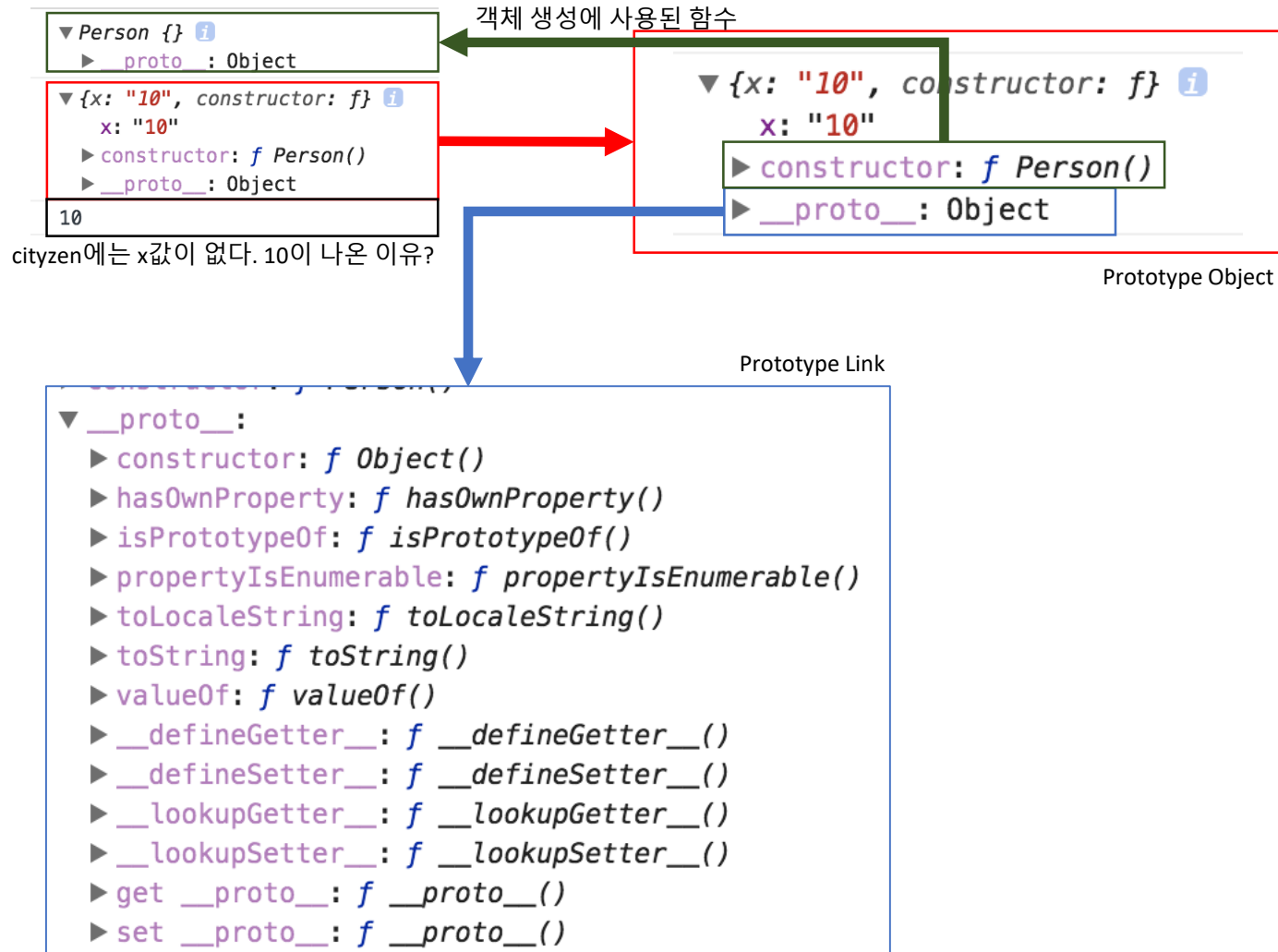
- 프로토타입?
  - 한 객체가 만들어지기 위해 필요한 객체의 모태.
  - 클래스가 존재하지 않는 자바 스크립트에서 객체 지향 프로그래밍을 가능케 해주는 객체의 원형
  - 확장 및 객체의 재사용을 가능하게 해준다.
- Prototype = Prototype Object + Prototype Link
  - Prototype Object : Function의 객체가 가지는 속성
  - Prototype Link : Function 객체를 만들 때 사용된 객체의 원형
    - 일반 Function 객체일 경우 Object 가 Prototype Link가 된다. (\_\_proto\_\_)
- 아래와 같은 코드의 결과로 Prototype 이해하기

```
function Person() {}  
Person.prototype.x = "10";  
  
const citizen = new Person();  
console.log(citizen);  
console.log(Person.prototype);  
console.log(citizen.x);
```

▼ Person {} ⓘ	console.log(citizen);
▶ __proto__: Object	
▼ {x: "10", constructor: f} ⓘ	console.log(Person.prototype);
▶ x: "10"	
▶ constructor: f Person()	
▶ __proto__: Object	
10	console.log(citizen.x);

# 기존의 프로토타입

- Prototype Object / Prototype Link



# 기존의 프로토타입

- 조금 더 상세하게 사용해보기

```
function Person(firstName) {  
  this.firstName = firstName;  
}  
  
Person.prototype.walk = function(){  
  console.log("I am walking!");  
};  
Person.prototype.sayHello = function(){  
  console.log("Hello, I'm " + this.firstName);  
};  
  
const citizen = new Person("Min Chang Jang");  
console.log(citizen);  
console.log(Person.prototype);  
citizen.walk();  
citizen.sayHello();
```

```
▼ Person {firstName: "Min Chang Jang"} ⓘ  
  firstName: "Min Chang Jang"  
  ► __proto__: Object  
  
▼ {walk: f, sayHello: f, constructor: f} ⓘ  
  ► sayHello: f ()  
  ► walk: f ()  
  ► constructor: f Person(firstName)  
  ► __proto__: Object  
  
I am walking!  
Hello, I'm Min Chang Jang
```

- citizen에는 walk(), sayHello() 메소드가 없어도 실행이 되는 이유는?
  - Property Chaining 때문.

# 기존의 프로토타입

- Property Chaining

```
function Person(firstName) {  
  this.firstName = firstName;  
}  
  
Person.prototype.walk = function(){  
  console.log("I am walking!");  
};  
Person.prototype.sayHello = function(){  
  console.log("Hello, I'm " + this.firstName);  
};  
  
const citizen = new Person("Min Chang Jang");  
citizen.walk();  
citizen.sayHello();
```

- Function 또는 Property 를 먼저 객체(Function)에서 찾고, 없으면 Prototype Object에서 찾고, 없다면 Prototype Link에서 찾는다.



# 기존의 프로토타입

- 프로토타입을 이용한 객체 상속( Object.create(Super Prototype) )

```
function Person(firstName) {
  this.firstName = firstName;
}
Person.prototype.walk = function(){
  console.log("I am walking!");
};
Person.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName);
};

const citizen = new Person("Min Chang Jang");
console.log(citizen);
console.log(Person.prototype);
citizen.walk();
citizen.sayHello();

function Student(firstName, subject) {
  Person.call(this, firstName);
  this.subject = subject
}

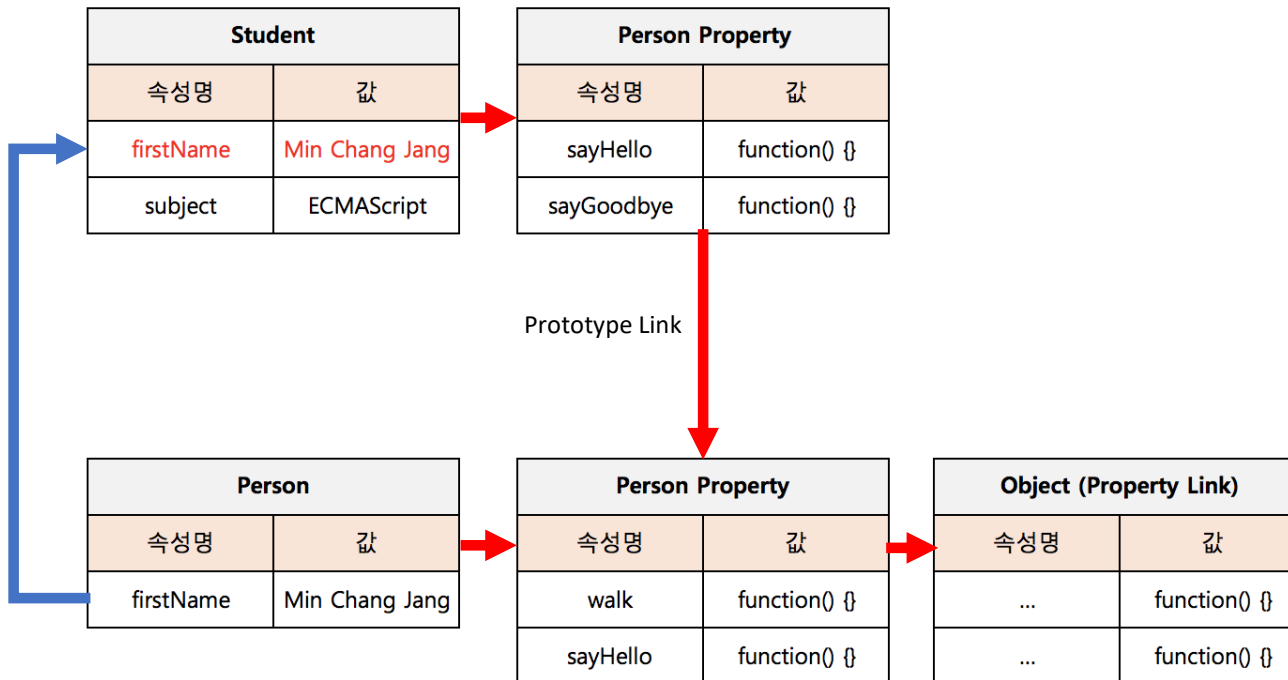
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;
Student.prototype.sayHello = function(){
  console.log("Hello, I'm " + this.firstName + ". I'm studying " + this.subject + ".");
};
Student.prototype.sayGoodBye = function(){
  console.log("Goodbye!");
};

const student = new Student("Min Chang Jang", "ECMAScript");
console.log(student);
console.log(Student.prototype);
student.walk();
student.sayHello();
student.sayGoodBye();
```



# 기존의 프로토타입

- 프로토타입을 이용한 객체 상속( Object.create(Super Prototype) )



# ECMAScript 2015 프로토타입

- Prototype의 한계점.
  - Prototype은 function 객체에만 존재함.
  - 따라서 Non-Function 객체에서는 Prototype을 활용한 상속/확장이 불가능함.
- ECMAScript 2015 이후부터 객체 리터럴에도 프로토타입을 변경할 수 있도록 지원함.
  - 단, 상속의 개념이 아닌 단순 변경.
- Object.setPrototypeOf();
  - 객체 리터럴의 프로토타입을 변경할 수 있도록 하는 명령.

# ECMAScript 2015 프로토타입

- 프로토타입 변경 예제

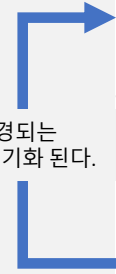
```
const person = {
  firstName: "Min Chang Jang",
  work() {
    console.log("I'm working");
  },
  sayHello() {
    console.log("Hello, I'm " + this.firstName);
  }
}

const student = {
  subject: "ECMAScript",
  sayHello() {
    console.log("Hello, I'm " + this.firstName + ". I'm studying " + this.subject);
  },
  sayGoodBye() {
    console.log("Goodbye.");
  }
}
```

```
const friend = Object.create(person);
console.log(friend);
friend.work();
friend.sayHello();

Object.setPrototypeOf(friend, student);
console.log(friend);
friend.sayHello();
friend.sayGoodBye();
```

마지막으로 변경되는  
프로토타입으로 동기화 된다.



```
▼ {} ⓘ
  ▼ __proto__:
    ▶ sayGoodBye: f sayGoodBye()
    ▶ sayHello: f sayHello()
    subject: "ECMAScript"
    ▶ __proto__: Object
I'm working
Hello, I'm Min Chang Jang

▼ {} ⓘ
  ▼ __proto__:
    ▶ sayGoodBye: f sayGoodBye()
    ▶ sayHello: f sayHello()
    subject: "ECMAScript"
    ▶ __proto__: Object
Hello, I'm undefined. I'm studying ECMAScript
Goodbye.
```

# ECMAScript 2015 프로토타입

- Super Prototype 참조를 통한 쉬운 Prototype 접근
  - Object.create() 와 달리, 객체를 상속하여 사용이 가능하다.

```
const person = {
  firstName: "Min Chang Jang",
  work() {
    console.log("I'm working");
  },
  sayHello() {
    console.log("Hello, I'm " + this.firstName);
  }
}

const student = {
  subject: "ECMAScript",
  sayHello() {
    super.sayHello();
    console.log("Hello, I'm " + this.firstName + ". I'm studying " + this.subject);
  },
  sayGoodBye() {
    console.log("Goodbye.");
  }
}
```

**Object.setPrototypeOf(student, person);**

```
console.log(person);
person.work();
person.sayHello();
```

```
console.log(student);
student.work();
student.sayHello();
student.sayGoodBye();
```

```
const friend = Object.create(student);
console.log(friend);
friend.work();
friend.sayHello();
friend.sayGoodBye();
```

# ECMAScript 2015 프로토타입

- Super Prototype 참조를 통한 쉬운 Prototype 접근
  - Object.create() 와 달리, 객체를 상속하여 사용이 가능하다.

```
▼ {firstName: "Min Chang Jang", work: f, sayHello: f} ⓘ  
  firstName: "Min Chang Jang"  
  ▶ sayHello: f sayHello()  
  ▶ work: f work()  
  ▶ __proto__: Object
```

I'm working

Hello, I'm Min Chang Jang

```
▼ {subject: "ECMAScript", sayHello: f, sayGoodBye: f} ⓘ  
  ▶ sayGoodBye: f sayGoodBye()  
  ▶ sayHello: f sayHello()  
  subject: "ECMAScript"
```

```
▼ __proto__:  
  firstName: "Min Chang Jang"  
  ▶ sayHello: f sayHello()  
  ▶ work: f work()  
  ▶ __proto__: Object
```

I'm working

Hello, I'm Min Chang Jang

Hello, I'm Min Chang Jang. I'm studying ECMAScript

Goodbye.

```
▼ {} ⓘ  
  ▼ __proto__:  
    ▶ sayGoodBye: f sayGoodBye()  
    ▶ sayHello: f sayHello()  
    subject: "ECMAScript"
```

```
▼ __proto__:  
  firstName: "Min Chang Jang"  
  ▶ sayHello: f sayHello()  
  ▶ work: f work()  
  ▶ __proto__: Object
```

I'm working

Hello, I'm Min Chang Jang

Hello, I'm Min Chang Jang. I'm studying ECMAScript

Goodbye.

## 6장

# 구조 분해(해체)

ECMAScript2015에서 추가된 구조 분해(해체) 방법을 살펴보고 학습한다.

# 구조 분해(해체)

- 객체 리터럴이나 배열을 해체해 최소단위의 변수로 저장하는 방법
- 기존에 객체리터럴이나 배열에서 데이터를 가져오는 방법

객체 리터럴에서 기존의 방법으로 추출

```
let options = {  
  repeat: true,  
  save: false  
};  
  
// 추출  
let repeat = options.repeat;  
let save = options.save;
```

배열에서 기존의 방법으로 추출

```
let colors = ["red", "green", "blue"];  
  
let red = colors[0];  
let green = colors[1];  
let blue = colors[2];  
  
console.log(red, green, blue);
```

객체 리터럴에서 구조 분해해 추출

```
let options = {  
  repeat: true,  
  save: false  
};  
  
// 추출  
let {repeat, save} = options;  
  
console.log(repeat);  
console.log(save);
```

배열에서 구조 분해해 추출

```
let colors = ["red", "green", "blue"];  
let [red, green, blue] = colors;  
  
console.log(red, green, blue);
```

# 구조 분해(해체)

- 선언된 변수에 구조 분해 후 할당

```
let node = {  
  type: "Identifier",  
  name: "foo"  
};
```

```
let type = "Literal"  
let name = 5;
```

```
console.log(type, name);
```

Literal 5

```
{type, name} = node;
```

```
console.log(type, name);
```

Identifier foo

- 미리 선언된 변수에 구조 분해 후 재 할당문에는 반드시 괄호가 필요하다.



# 구조 분해(해체)

- 기본 값 할당

```
let node = {  
  type: "Identifier",  
  name: "foo"  
};  
  
let {type, name, value} = node;  
  
console.log(type, name, value);
```

Identifier foo undefined

- 구조 분해시, 지정된 값이 없을 경우 undefined 로 지정됨.
- 이를 막기 위해 기본값을 할당할 수 있다.

```
let node = {  
  type: "Identifier",  
  name: "foo"  
};  
  
let {type, name, value = true} = node;  
  
console.log(type, name, value);
```

Identifier foo true

# 구조 분해(해체)

- 이름이 다른 변수에 할당하기

```
let node = {  
  type: "Identifier",  
  name: "foo"  
};  
  
let {type: localType, name: localName} = node;  
  
console.log(localType, localName);
```

Identifier foo

- 구조 분해는 객체 리터럴 내의 이름이 같은 변수에 할당된다.
- 다른 이름의 변수에 할당하고자 한다면, 아래와 같은 패턴으로 작성한다.

```
let {프로퍼티 명: 변수명, ...} = 객체 리터럴;
```

# 구조 분해(해체)

- 중첩 구조의 객체 분해

```
let node = {  
  type: "Identifier",  
  name: "foo",  
  loc: {  
    start: {  
      line: 1,  
      column: 1  
    },  
    end: {  
      line: 1,  
      column: 4  
    }  
  }  
};
```

```
let { loc: { start } } = node;
```

```
console.log(start.line, start.column);
```

1 1

- 복잡한 중첩 구조의 객체를 분해할 때는 리터럴 문법을 사용할 수 있다.

```
let {프로퍼티 명: {프로퍼티 명}, ...} = 객체 리터럴;
```

- 최종 중괄호의 프로퍼티명이 지역변수의 이름이 된다.

# 구조 분해(해체)

- 배열의 구조 분해 할당

```
let colors = ["red", "green", "blue"];

let firstColor = "black";
let secondColor = "purple";

console.log(firstColor, secondColor);

[firstColor, secondColor] = colors;

console.log(firstColor, secondColor);
```

black purple

red green

- 배열의 구조 분해시 객체의 분해처럼 괄호를 필요로하지 않는다.
- 구조 분해시 모든 배열의 개수를 맞출 필요가 없다. (단, 순서는 맞추어야 한다)

- 중첩된 배열을 분해할 때 대괄호를 한번 더 사용한다.

```
let colors = ["red", ["green", "lightGreen"], "blue"];

let firstColor = "black";
let secondColor = "purple";

console.log(firstColor, secondColor);

[firstColor, [secondColor1, secondColor2], blue] = colors;

console.log(firstColor, secondColor1, secondColor2, blue);
```

# 구조 분해(해체)

- 구조분해와 나머지 연산자를 이용해 배열 복사하기

```
let colors = ["red", "green", "blue"];
```

```
let [...colonedColors] = colors;  
colonedColors.push("cyan");
```

```
console.log(colors);  
console.log(colonedColors);
```

```
red green blue  
red green blue cyan
```

# 혼합된 구조 분해(해체)

- 객체 구조 분해와 배열 구조 분해를 함께 사용해 복잡한 표현식 만들기

```
let node = {  
  type: "Identifier",  
  name: "foo",  
  loc: {  
    start: {  
      line: 1,  
      column: 1  
    },  
    end: {  
      line: 1,  
      column: 4  
    }  
  },  
  range: [0, 3]  
};  
  
let {  
  loc: {start},  
  range: [ startIndex, endIndex ]  
} = node;  
  
console.log(start.line, start.column);  
console.log(startIndex, endIndex);
```

# 파라미터 구조 분해(해체)

- 객체를 파라미터로 보낼 때 function의 파라미터로 구조 분해 할 수 있다.

```
function setAttribute(name, {url, method}) {  
  console.log("name", name);  
  console.log("url", url);  
  console.log("method", method);  
}  
  
setAttribute("searchForm", {  
  url: "http://localhost",  
  method: "post"  
});
```

- 혹은 기본값을 이용해 아래처럼 사용할 수도 있다.

```
function setAttribute(name, {  
  url = "http://localhost",  
  method = "post"  
} = {}) {  
  console.log("name", name);  
  console.log("url", url);  
  console.log("method", method);  
}  
  
setAttribute("searchForm");
```

## 7장

# 심벌(Symbol)

ECMAScript2015에서 추가된 심벌을 살펴보고 학습한다.



# 심벌(Symbol)

- 문자열, 숫자, 불(boolean), null, undefined와 더불어 ECMAScript2015에서 추가된 원시타입(Primitive Type)
- 타입 자체로서 의미가 부여됨. 이런 측면에서 Java의 enum과 유사함.
- 객체 리터럴에서 "비공개 프로퍼티"를 만드려 할 때 주로 사용된다.
- 필요에 따라 비 공유 / 공유 심볼을 만들 수 있다.
- Symbol 생성 방법

```
const firstName = Symbol();  
console.log(firstName);
```

# 심벌(Symbol)

- Symbol을 이용해 객체리터럴을 생성할 수 있다.

```
const firstName = Symbol();
```

```
const person = {};  
person[firstName] = "Min Chang Jang";  
console.log(person);
```

Object {Symbol(): "Min Chang Jang"}

- Symbol로 정의된 프로퍼티/값은 반복문에도 출력되지 않는다.

```
const firstName = Symbol();
```

```
const person = {};  
person[firstName] = "Min Chang Jang";  
person["secondName"] = "Seo Tae Ji";  
console.log(person);  
console.log(person[firstName]);
```

```
for ( let prop in person ) {  
  console.log(person[prop]);  
}
```

Object {secondName: "Seo Tae Ji", Symbol(): "Min Chang Jang"}  
Min Chang Jang

Seo Tae Ji

# 심벌(Symbol)

- Symbol을 객체 리터럴에 넣으려면, 대괄호로 감싸야 한다.

```
const firstName = Symbol();
```

```
const person = {  
  [firstName]: "Min Chang Jang",  
  secondName: "Seo Tae Ji"  
};  
console.log(person);  
console.log(person[firstName]);  
console.log(person["secondName"]);  
console.log(person.secondName);
```

```
Object {secondName: "Seo Tae Ji", Symbol(): "Min Chang Jang"}  
Min Chang Jang  
Seo Tae Ji  
Seo Tae Ji
```

- Symbol을 이용하면, 객체안의 객체도 숨길 수 있다.

```
const name = Symbol();
```

```
const person = {  
  [name]: {  
    firstName: "Min Chang",  
    lastName: "Jang"  
  },  
  dept: "Research And Develop"  
};
```

```
console.log(person);  
console.log(person[name].firstName);  
console.log(person[name].lastName);
```

```
Object {dept: "Research And Develop", Symbol(): Object}  
Min Chang  
Jang
```

# 심벌(Symbol)

- 객체리터럴의 프로퍼티값을 읽기 전용으로 만드려면 `Object.defineProperty(Object, Property, Accessible)`를 사용한다.

```
const name = Symbol();

const person = {
  [name]: {
    firstName: "Min Chang",
    lastName: "Jang"
  },
  dept: "Research And Develop"
};

Object.defineProperty(person, name, {writable: false});

person[name] = {
  firstName: "Tae Ji",
  lastName: "Seo"
}

console.log(person[name].firstName);
console.log(person[name].lastName);
```

Min Chang  
Jang

# 심벌(Symbol)

- Object.defineProperty(Object, Property)를 사용할 수도 있다.

```
const name = Symbol();

const person = {
  [name]: {
    firstName: "Min Chang",
    lastName: "Jang"
  }
};

Object.defineProperty(person, name, {writable: false});
Object.defineProperties(person, {
  lastName: {
    value: "Research And Develop",
    writable: false
  }
});

person[name] = {
  firstName: "Tae Ji",
  lastName: "Seo"
}

person.lastName = "Sales";

console.log(person[name].firstName);
console.log(person[name].lastName);
console.log(person.lastName);
```

Min Chang  
Jang  
Research And Develop

# 심벌(Symbol)

- Symbol을 사용할 때, 각 심벌을 읽기 쉽게 하기 위해선, 서술 문자형을 사용해 주는 것이 좋다.

```
const firstName = Symbol("first name");
const person = {};

person[firstName] = "Min Chang";

console.log("first name" in person);
console.log(person[firstName]);
console.log(firstName);
```

- 공유되는 심벌 만들기

```
const uid = Symbol.for("uid");
const object = {};

object[uid] = "12345";

console.log(object[uid]);
console.log(uid);

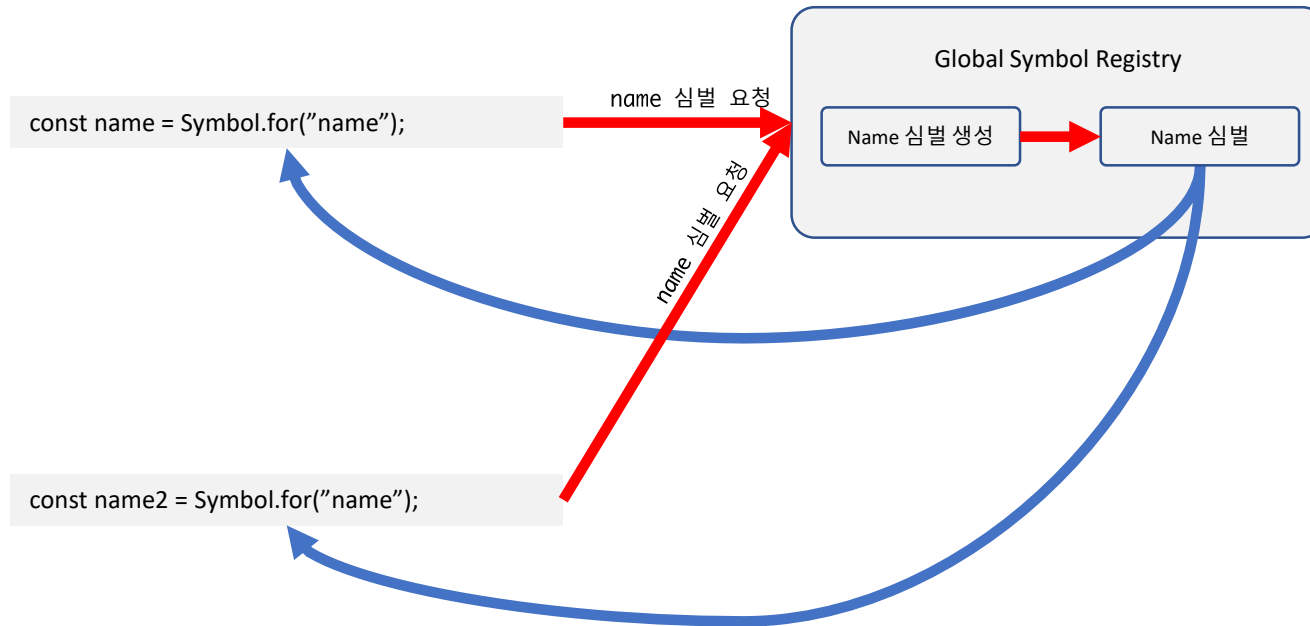
const uid2 = Symbol.for("uid");
console.log(uid === uid2);
console.log(object[uid2]);
console.log(uid2);
```

12345  
Symbol(uid)

true  
12345  
Symbol(uid)

# 심벌(Symbol)

- Symbol.for("name") 은 "전역 심벌 레지스트리(Global Symbol Registry)" 에 "name"을 키로하는 심벌이 존재 하는지 확인 후 없다면, 생성, 있다면 그 심벌을 반환한다.



# 심벌(Symbol)

- 지역 심벌(Local Symbol)은 서로 다른 객체를 반환한다.

```
const uid = Symbol("uid");  
const uid2 = Symbol("uid");
```

```
console.log(uid == uid2);  
console.log(uid === uid2);
```

```
false  
false
```

- 따라서, Symbol을 여러번 사용할 경우 공유 심벌을 사용하는 것이 좋다.



## 8장

# Set / Map

ECMAScript2015에서 추가된 Set과 Map을 살펴보고 학습한다.

# Set / Map

- 기존의 Javascript는 타 언어의 Set / List / Map을 지원하지 않아, Array나 객체를 이용해 완전하지 않은 방법으로 흉내만 내는 수준

```
function set() {  
  
  var items = [];  
  
  return {  
    add: function(item) {  
      if ( !this.has(item) ) {  
        items.push(item);  
      }  
    },  
    has: function(item) {  
      for(var i in items) {  
        if ( items[i] === item ) {  
          return true;  
        }  
      }  
      return false;  
    },  
    get: function() { return items; },  
    clear: function() { items = []; },  
    remove: function(item) {  
      for(var i in items) {  
        if ( items[i] == item ) {  
          items.splice(i, 1);  
          break;  
        }  
      }  
    },  
    size: function() { return items.length; }  
  };  
}
```

```
function map() {  
  
  var items = {};  
  
  return {  
    add: function(key, value) {  
      items[key] = value;  
    },  
    get: function(key) {  
      return items[key];  
    },  
    clear: function() {  
      items = {}  
    },  
    remove: function(key) {  
      delete items[key];  
    },  
    size: function() {  
      var size = 0;  
      for (var key in items) {  
        if (items.hasOwnProperty(key)) size++;  
      }  
      return size;  
    }  
  };  
}
```

# Set / Map

- Array는 숫자인덱스만 지원 / 객체 리터럴은 문자열 프로퍼티만 지원해 비슷하게 흉내를 내지만, 유연하지 않은 객체가 됨.

```
var map = map();  
map.add("Alphabets", "ABCDEF...");  
map.add(5, "number five");  
map.add("5", "String number five");
```

```
console.log(map.get(5));  
console.log(map.get("5"));
```

```
String number five  
String number five
```

- ECMAScript 2015에서는 Set(Weak Set)과 Map(Weak Map)을 지원해 기존의 한계점을 극복하게 해준다.

# Set

- 중복은 없고 순서는 있는 값의 리스트

```
const set = new Set();

set.add(5);
set.add("5");

console.log(set.size);

set.forEach(function(value, key, ownerSet) {
  console.log(value, key, ownerSet);
});
```

- `new Set();` 명령으로 Set을 생성함.

```
const set = new Set();
```

- `add(값)` 명령을 통해 데이터를 삽입할 수 있다.

```
set.add(5);
set.add("5");
```

- 이 때 중복된 값은 제거된다.
- 위 예제에서 5와 "5"는 다른 값으로 처리되어 저장된다.

```
set.add({name: "Min Chang"});
set.add({name: "Tae Ji"});
```

- 객체를 저장 할 수도 있다.

# Set

- 중복은 없고 순서는 있는 값의 리스트

- size 프로퍼티를 이용해 set 내에 저장된 개수를 확인할 수 있다.

```
console.log(set.size);
```

- forEach 메소드를 이용해, set에 저장된 값을 불러 확인할 수 있다.

```
set.forEach(function(value, key, ownerSet) {  
  console.log(value, key, ownerSet);  
});
```

- value : Set에서 다음 위치의 값 (개별 요소의 값)
- key : 첫 번째 인자와 같은 값
- ownerSet : 값을 읽어들이는 원본 Set ( set 객체와 ownerSet은 완전히 같은 객체)

- delete(값) 메소드를 이용해 set에 저장된 값을 삭제할 수 있다.

```
const set = new Set();  
  
const minchang = {name: "Min Chang"};  
  
set.add(minchang);  
set.add({name: "Tae Ji"});  
console.log(set.size);  
  
set.delete(minchang);  
console.log(set.size);
```

리터럴은 만들때마다 새로운 객체가 만들어지기 때문에, 객체를 담고 있는 변수만 해당된다.

원시값은 값 자체로도 삭제가 가능하다.

2

1

# Set

- 중복은 없고 순서는 있는 값의 리스트
  - `has()` 메소드를 이용해 값이 이미 존재하는지 확인할 수 있다.

```
const set = new Set();

set.add("5");
set.add(6);

console.log(set.has(5));      false
console.log(set.has("5"));    true
```

- Set은 배열값으로 초기화 시킬 수 있다.

```
const set = new Set([1, 2, 3, 4, 5, 5, 5, 5]); // 8개
console.log(set.size);
console.log(set.has(5));

set.forEach(function(value) {
  console.log(value);
});
```

5  
True  
  
1  
2  
3  
4  
5

- 중복된 값이 있는 배열을 Set에 초기화 할 경우, 중복된 값은 모두 제거된다.

# WeakSet

- Set = 객체를 참조하는 방식 = Strong Set 이라고 부른다.
- Set에 저장된 객체의 원본이 제거되더라도 Set에는 객체에 대한 참조가 남아 있음. ➔ 가비지컬렉션이 될 수 없다.



- Object(0x1456), Object(0x1457) 이 제거되더라도 Set의 참조는 사라지지 않기 때문에, 메모리 누수가 발생한다.
- WeakSet은 참조객체가 사라질 경우 WeakSet 내의 참조도 삭제시킨다.

# WeakSet

- WeakSet은 객체만 저장할 수 있다.  
→ 원시타입(Primitive Type)은 저장할 수 없다.

```
const set = new WeakSet([1, 2, 3, 4, 5, 5, 5, 5]);
```

TypeError: Invalid value used in weak set

- Set과 WeakSet의 차이점
  - 1. 객체가 아닌 값이 add()에 전달되면 에러를 발생시킨다.
  - 2. for-of를 사용할 수 없다.
  - 3. WeakSet의 내용을 프로그램적으로 확인할 방법이 없다.
  - 4. forEach()메소드가 없다.
  - 5. size 프로퍼티가 없다.



# Map

- Key와 Value를 쌍으로 만들어 데이터를 저장.
  - Set과 유사한 형태로 사용할 수 있다.

```
const map = new Map();  
map.set("name", "Min Chang")  
map.set("age", 25);
```

```
console.log(map.size);  
console.log(map.has("name"));  
console.log(map.get("name"));
```

```
2  
true  
Min Chang
```

```
console.log(map.has("age"));  
console.log(map.get("age"));
```

```
true  
25
```

```
map.delete("name");  
console.log(map.has("name"));  
console.log(map.get("name"));  
console.log(map.size);
```

```
false  
undefined  
1
```

```
map.clear();  
console.log(map.has("name"));  
console.log(map.get("name"));
```

```
false  
undefined
```

```
console.log(map.has("age"));  
console.log(map.get("age"));  
console.log(map.size);
```

```
false  
undefined  
0
```

# Map

- 생성자를 통한 데이터 초기화는 대괄호([ 와 ])를 사용한다.

```
const map = new Map([["name", "Min Chang"], ["age", 25]]);
```

```
console.log(map.has("name"));  
console.log(map.get("name"));
```

true  
Min Chang

```
console.log(map.has("age"));  
console.log(map.get("age"));
```

true  
25

```
console.log(map.size);
```

2

- Map의 forEach(); → Set 과 형태가 동일하다.

```
const map = new Map([["name", "Min Chang"], ["age", 25]]);
```

```
map.forEach(function(value, key, ownerMap) {  
  console.log(value, key, ownerMap == map);  
});
```

- value : Map의 다음 위치의 value
- key : 값에 대한 키
- map : 값을 읽어들이고 있는 Map

# WeakMap

- Set과 WeakSet의 관계처럼 Map과 WeakMap도 동일하게 객체를 관리함.
- Map에 저장된 객체의 원본이 제거되더라도 Map에는 객체에 대한 참조가 남아 있음. ➔ 가비지컬렉션이 될 수 없다.
- WeakSet처럼 WeakMap에 객체가 아닌 값으로 키를 사용할 수 없다.
  - 1. 객체가 아닌 값이 set()에 전달되면 에러를 발생시킨다.
  - 2. for-of를 사용할 수 없다.
  - 3. WeakMap의 내용을 프로그램적으로 확인할 방법이 없다.
  - 4. forEach()메소드가 없다.
  - 5. size 프로퍼티가 없다.

## 9장

# 이터레이터 / 제네레이터

반복문을 대체할 이터레이터와 제네레이터를 학습한다.

# 이터레이터 / 제네레이터

- 컬렉션의 위치변수 추적을 위해 초기화가 필수인 for문 방식에서 컬렉션의 다음 요소를 반환하는 이터레이터로 옮겨가는 추세.
- 컬렉션의 데이터를 쉽게 사용할 수 있도록 지원한다.
- ECMAScript 2015에 추가된 for-of문도 이터레이터와 함께 동작한다.
- 전개(펼침)연산자(...)도 이터레이터를 사용한다.
- 비동기 프로그래밍을 할 때에도 이터레이터를 사용할 수 있다.
- 반복문의 문제점?
  - 코드가 직관적이지만, 1. 중첩되거나 2. 반복문내의 객체를 유지시키려 할때 복잡도가 증가한다. → 예러가 발생하기 쉽다. & 많은 양의 코드를 작성해야 한다.

```
var colors = ["red", "green", "blue"];

for ( var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
// 혹은
for ( var i in colors ) {
  console.log(colors[i]);
}
```

# 이터레이터?

- 반복을 위해 설계된 인터페이스.
- 이터레이터 객체는 next() 메서드와 value, done 프로퍼티를 가지고 있다.
  - next() → 컬렉션의 객체 하나를 반환.
  - value → 다음 값 (혹은 객체)
  - done → 반복의 종료 여부 (true / false)
- ECMAScript 2015 이전에 이터레이터를 만드는 방법

```
function createIterator(items) {  
  var i = 0;  
  
  return {  
    next: function() {  
      var done = ( i >= items.length );  
      var value = !done ? items[i++] : undefined;  
  
      return {  
        done : done,  
        value: value  
      };  
    }  
  };  
}
```

```
var iterator = createIterator([1, 2, 3]);  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());
```

```
Object {done: false, value: 1}  
Object {done: false, value: 2}  
Object {done: false, value: 3}  
Object {done: true, value: undefined}
```

# 제네레이터?

- 이터레이터를 반환하는 함수
- function 뒤에 별표(\*) 로 표현
- yield 키워드 사용

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

```
var iterator = createIterator();  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 2, done: false}  
Object {value: 3, done: false}  
Object {value: undefined, done: true}
```

- 중요 특징
  - 제네레이터 함수 내에서 yield가 실행되면, 그 즉시 값을 반환하고 함수의 진행이 일시 중지된다.
  - 다음 next()의 호출이 있을 때, 중지한 지점에서 다시 함수가 실행된다.
  - 함수표현식을 지원한다.

# 함수표현식으로 만드는 제네레이터

- 함수표현식으로 제네레이터를 만들어 간편하게 전달할 수도 있다.

```
const createIterator = function *(items) {  
  for (let i in items) {  
    yield items[i];  
  }  
}
```

```
var iterator = createIterator([1,2,3]);  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 2, done: false}  
Object {value: 3, done: false}  
Object {value: undefined, done: true}
```

- 단, 화살표 함수로 제네레이터를 만들 수 없다.
  - 별표(\*)를 지원하지 않음.



# 객체 안에 만드는 제네레이터

- 제네레이터는 객체 안에 만들수도 있다.

```
const object = {  
  createIterator: function *(items) {  
    for (let i in items) {  
      yield items[i];  
    }  
  },  
  
  *anotherIterator(items) {  
    for (let i in items) {  
      yield items[i];  
    }  
  }  
}  
  
var iterator = object.createIterator([1,2,3]);  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 2, done: false}  
Object {value: 3, done: false}  
Object {value: undefined, done: true}
```

# for-of와 이터러블(Iterable)

- 이터러블

- Symbol.iterable 프로퍼티를 가지고 있는 객체.
- 이터레이터와 밀접한 관련이 있음.
- Array, Set, Map = 이터러블 = 이터레이터에서 사용 가능 = for-of 로 사용.

```
// 배열 상수 선언  
const values = [1, 2, 3];
```

```
// for-of로 반복하며 출력  
for (const value of values) {  
  console.log(value);  
}
```

1  
2  
3

```
// 배열 상수를 이터레이터로 변환  
const iteratorValues = values[Symbol.iterator]();
```

```
// iterator.next(); 로 출력  
console.log(iteratorValues.next());  
console.log(iteratorValues.next());  
console.log(iteratorValues.next());  
console.log(iteratorValues.next());
```

Object {value: 1, done: false}  
Object {value: 2, done: false}  
Object {value: 3, done: false}  
Object {value: undefined, done: true}

- for-of를 이용해 컬렉션을 순회할 때, 내부적으로 이터레이터가 수행된다.
  - 내부 이터레이터가 next()를 호출하며 컬렉션의 값을 value 변수에 할당해 반복 시켜준다.

# for-of와 이터러블(Iterable)

- for-of로 이터레이터 반복하기

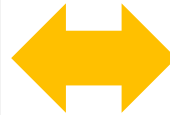
```
const values = [1, 2, 3];  
const iteratorValues = values[Symbol.iterator]();  
  
for (const value of iteratorValues) {  
  console.log(value);  
}
```

1  
2  
3

# 이터러블 만들기

- Array, Set, Map을 제외한 개발자가 직접 만든 객체를 이터러블로 변환
  - Symbol.iterator 메소드를 이용한다.

```
let collection = {  
  items: [],  
  *[Symbol.iterator]() {  
    for (let item of this.items) {  
      yield item;  
    }  
  }  
};  
  
collection.items.push(1);  
collection.items.push(2);  
collection.items.push(3);  
  
const iter = collection[Symbol.iterator]();  
console.log(iter.next());  
console.log(iter.next());  
console.log(iter.next());  
console.log(iter.next());
```



```
let collection = {  
  items: [],  
  *[Symbol.iterator]() {  
    for (let item of this.items) {  
      yield item;  
    }  
  }  
};  
  
collection.items.push(1);  
collection.items.push(2);  
collection.items.push(3);  
  
for ( let x of collection ) {  
  console.log(x);  
}
```

# 내장 이터레이터(컬렉션 이터레이터)

- 컬렉션 내부에 어떤 키/값이 있는지 확인하고자 할 때, 유용하게 사용할 수 있는 내장 이터레이터
  - `entries()`
    - 키와 값을 쌍으로 갖는 이터레이터 반환
  - `values()`
    - 컬렉션의 값을 갖는 이터레이터 반환
  - `keys()`
    - 컬렉션의 키를 갖는 이터레이터 반환
- `entries()`, `values()`, `keys()`는 `next()`가 호출될 때마다 실행된다.

# 내장 이터레이터(컬렉션 이터레이터)

- entries()
  - 키와 값을 쌍으로 갖는 이터레이터 반환

```
let colors = ["red", "green", "blue"];
let tracking = new Set([1234,5678,9012]);
let data = new Map();
data.set("title", "Javascript ECMA Script 2015");
data.set("format", "Power Point");
```

```
for ( let entry of colors.entries() ) {
  console.log(entry);
}
```

```
for ( let entry of tracking.entries() ) {
  console.log(entry);
}
```

```
for ( let entry of data.entries() ) {
  console.log(entry);
}
```

```
Array(2) [0, "red"]
Array(2) [1, "green"]
Array(2) [2, "blue"]
```

```
Array(2) [1234, 1234]
Array(2) [5678, 5678]
Array(2) [9012, 9012]
```

```
Array(2) ["title", "Javascript ECMA Script 2015"]
Array(2) ["format", "Power Point"]
```

# 내장 이터레이터(컬렉션 이터레이터)

- values()
  - 컬렉션의 값을 갖는 이터레이터 반환

```
let colors = ["red", "green", "blue"];
let tracking = new Set([1234,5678,9012]);
let data = new Map();
data.set("title", "Javascript ECMA Script 2015");
data.set("format", "Power Point");
```

```
for ( let value of colors ) {
  console.log(value);
}
```

```
for ( let value of tracking.values() ) {
  console.log(value);
}
```

```
for ( let value of data.values() ) {
  console.log(value);
}
```

red  
green  
blue

1234  
5678  
9012

Javascript ECMA Script 2015  
Power Point

# 내장 이터레이터(컬렉션 이터레이터)

- keys()
  - 컬렉션의 키를 갖는 이터레이터 반환

```
let colors = ["red", "green", "blue"];  
let tracking = new Set([1234,5678,9012]);  
let data = new Map();  
data.set("title", "Javascript ECMA Script 2015");  
data.set("format", "Power Point");
```

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

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

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

0

1

2

1234

5678

9012

title

format



# 내장 이터레이터(컬렉션 이터레이터)

- 컬렉션 별로 기본 이터레이터가 다르다.
  - Array & Set = values(), Map = entries()

```
let colors = ["red", "green", "blue"];
let tracking = new Set([1234,5678,9012]);
let data = new Map();
data.set("title", "Javascript ECMA Script 2015");
data.set("format", "Power Point");
```

```
for ( let value of colors ) {
  console.log(value);
}
```

```
for ( let num of tracking ) {
  console.log(num);
}
```

```
for ( let entry of data ) {
  console.log(entry);
}
```

```
red
green
blue
```

```
1234
5678
9012
```

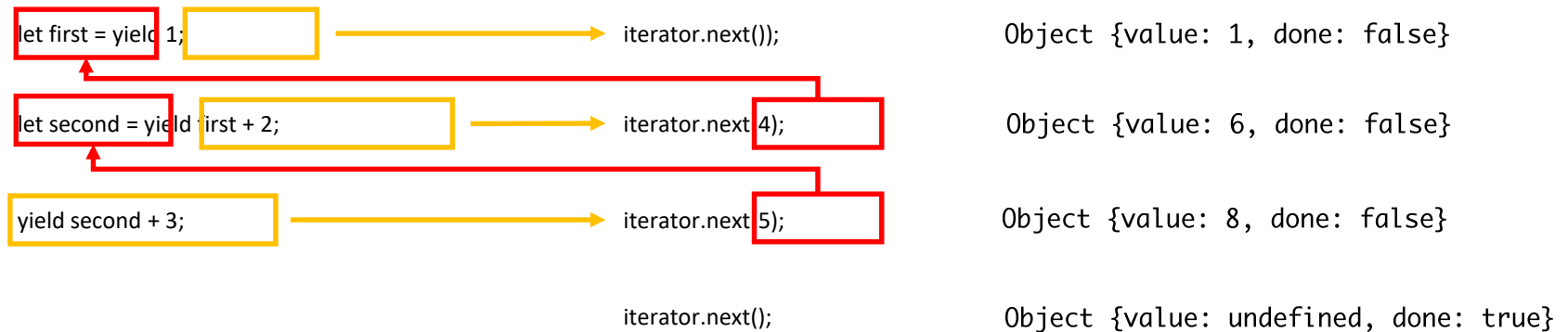
```
Array(2) ["title", "Javascript ECMA Script 2015"]
Array(2) ["format", "Power Point"]
```

# 이터레이터에 인자 전달

- 이터레이터에 있는 next(); 메소드에 인자를 전달할 수 있다.
  - 전달된 인자 값은 yield의 결과로 변수에 할당된다.

```
function *createIterator() {  
  let first = yield 1;  
  let second = yield first + 2;  
  yield second + 3;  
}  
  
let iterator = createIterator();  
  
console.log(iterator.next());  
console.log(iterator.next(4));  
console.log(iterator.next(5));  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 6, done: false}  
Object {value: 8, done: false}  
Object {value: undefined, done: true}
```



# 제네레이터 종료하기

- 제네레이터 == 함수
  - 함수 실행을 종료하기 위해 return 사용가능하다.
  - 또한, return과 함께, next()의 결과를 명시할 수도 있다.

```
function *createIterator() {  
  let first = yield 1;  
  let second = yield first + 2;  
  if ( second == 4 ) {  
    return 10;  
  }  
  yield second + 5;  
}  
  
let iterator = createIterator();  
console.log(iterator.next());  
console.log(iterator.next(2)); // first에 할당  
console.log(iterator.next(3)); // second에 할당  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 4, done: false}  
Object {value: 8, done: false}  
Object {value: undefined, done: true}
```

- 지정 값을 할당했을 때, return이 실행되고 10을 반환 후 함수 종료

```
let iterator = createIterator();  
console.log(iterator.next());  
console.log(iterator.next(2)); // first에 할당  
console.log(iterator.next(4)); // second에 할당  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 4, done: false}  
Object {value: 10, done: false}  
Object {value: undefined, done: true}
```

# 제네레이터 위임하기

- 여러개의 제네레이터를 하나의 제네레이터로 묶어 처리할 수 있다.

```
function *createNumberIterator() {  
  yield 1;  
  yield 2;  
}  
  
function *createColorIterator() {  
  yield "red";  
  yield "green";  
}  
  
function *createCombinedIterator() {  
  yield *createNumberIterator();  
  yield *createColorIterator();  
  yield true;  
}  
  
let iterator = createCombinedIterator();  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());
```

```
Object {value: 1, done: false}  
Object {value: 2, done: false}  
Object {value: "red", done: false}  
Object {value: "green", done: false}  
Object {value: true, done: false}  
Object {value: undefined, done: true}
```

- 제네레이터 안에서 다른 제네레이터를 호출하려면 yield와 별표(\*)를 반드시 사용해야 한다.

# 제네레이터 위임하기

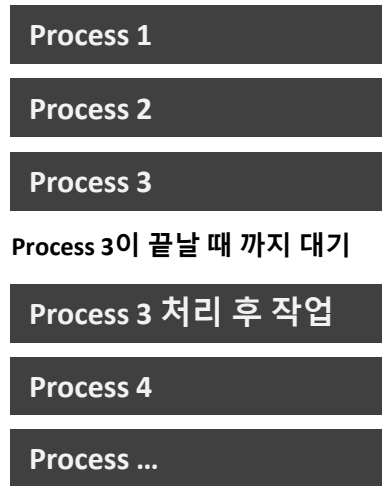
- 여러개의 제네레이터를 하나의 제네레이터로 묶어 처리할 수 있다.
  - 제네레이터가 반환하는 값을 다른 제네레이터에 전달 할수도 있다.

```
function *createNumberIterator() {  
  yield 1;  
  yield 2;  
  return 3;  
}  
  
function *createRepeatingIterator(count) {  
  for ( let i = 0; i < count; i++ ) {  
    yield "repeat" + i;  
  }  
}  
  
function *createCombinedIterator() {  
  let result = yield *createNumberIterator();  
  yield *createRepeatingIterator(result);  
}  
  
var iterator = createCombinedIterator();  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());
```

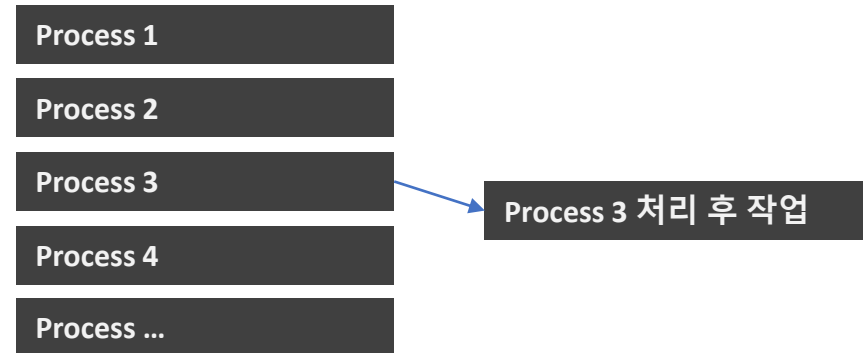
```
Object {value: 1, done: false}  
Object {value: 2, done: false}  
Object {value: "repeat0", done: false}  
Object {value: "repeat1", done: false}  
Object {value: "repeat2", done: false}  
Object {value: undefined, done: true}  
Object {value: undefined, done: true}
```

# 여기서 잠깐!

- 동기와 비동기



동기



비동기

# 제네레이터는 동기? 비동기?

- 일반적인 Javascript 코드(특히 function)들은 동기로 동작한다.

```
function foo() {  
  for ( let i = 0; i < 100; i++ ) {  
    console.log("이 작업이 끝날 때까지 다른 작업은 할 수 없습니다!");  
  }  
}  
  
foo();  
console.log("다른 작업!");
```

이 작업이 끝날 때까지 다른 작업은 할 수 없습니다!  
이 작업이 끝날 때까지 다른 작업은 할 수 없습니다!  
이 작업이 끝날 때까지 다른 작업은 할 수 없습니다!  
...  
다른 작업!

- 이 코드를 비동기로 동작시키려면?

```
function foo(anotherFoo) {  
  for ( let i = 0; i < 100; i++ ) {  
    anotherFoo();  
    console.log("콜백으로 다른 함수와 함께 동작합니다!");  
    console.log("하지만 지정된 콜백만 함께 동작합니다!");  
  }  
}  
  
foo(function() {  
  console.log("다른 작업!");  
});
```

# 제네레이터는 동기? 비동기?

- 제네레이터가 나오기 전까지는 비동기작업을 콜백으로 처리했음  
→ 동시에 처리할 비동기 작업이 많아질 수록 이른바 "콜백지옥"을 만나게 된다.

```
function foo(anotherFoo1, anotherFoo2, anotherFoo3) {  
  for ( let i = 0; i < 100; i++ ) {  
    anotherFoo1();  
    console.log("콜백으로 다른 함수와 함께 동작합니다!");  
    anotherFoo2();  
  }  
  anotherFoo3();  
}  
  
foo(function() {  
  console.log("다른 작업1");  
}, function() {  
  console.log("다른 작업2");  
}, function() {  
  console.log("다른 작업3");  
});
```

- 제네레이터는 콜백을 사용한 비동기처리를 단순화 시켜준다.



# 제네레이터를 이용한 비동기 처리

- 제네레이터를 이용해 한 function이 수행되는 동안 다른 function을 처리할 수 있다.

```
function *createIterator() {  
  for ( let i = 0; i < 100; i++ ) {  
    yield i;  
  }  
}  
  
function foo() {  
  let iterator = createIterator();  
  for ( let value of iterator ) {  
    console.log("다른 작업1");  
    console.log(value);  
    console.log("다른 작업2");  
  }  
}  
  
foo();
```

# 제네레이터를 이용한 비동기 처리

- 일련의 비동기 작업을 순서대로 수행할 때, 아래처럼 사용할 수 있다.

```
function *orderCoffee(phoneNumber) {  
  const id = getId(phoneNumber);  
  yield id;  
  const name = getName(id);  
  yield name;  
  const email = getEmail(name);  
  yield email;  
  return order(name, "coffee");  
}  
  
function getId(phoneNumber) { return "mcjang"; }  
  
function getName(id) { return "Min Chang"; }  
  
function getEmail(name) { return mc.jang@hucloud.co.kr }  
  
function order(name, drink) { return `${name}님, ${drink} 나왔습니다.` }  
  
function run(foo) {  
  const iterator = foo("010-1234-5678");  
  const id = iterator.next();  
  console.log(`ID : ${id.value}`);  
  const name = iterator.next();  
  console.log(`Name : ${name.value}`);  
  const email = iterator.next();  
  console.log(`Email : ${email.value}`);  
  const order = iterator.next();  
  console.log(`Order : ${order.value}`);  
}  
  
run(orderCoffee);
```

# 제네레이터를 이용한 비동기 처리

- 작업량이 많은 경우, `.next()`를 호출하는 패턴도 복잡해지기 때문에 실행 과정을 단순화 시킬 필요가 있다.

```
function run(foo, phoneNumber) {  
  
  const task = foo(phoneNumber);  
  
  let result = task.next();  
  console.log(result.value);  
  
  function step() {  
    if ( !result.done ) {  
      result = task.next();  
      console.log(result.value);  
      step();  
    }  
  }  
  
  step();  
}  
  
run(orderCoffee, "010-1234-5678");
```

```
function run(foo, phoneNumber) {  
  
  const task = foo(phoneNumber);  
  
  let result = task.next();  
  console.log(result.value);  
  
  function step() {  
    if ( !result.done ) {  
      result = task.next(result.value);  
      console.log(result.value);  
      step();  
    }  
  }  
  
  step();  
}  
  
run(orderCoffee, "010-1234-5678");
```

`Next()`에 인자를 전달할 필요가 있을 경우 위같이 사용할 수 있다.

- 제네레이터에 더 많은 작업들이 많다고 하더라도 복잡하지 않게 호출할 수 있다.

## 10장

# 클래스

ECMAScript 2015에 새롭게 등장한 클래스를 학습한다.

# 유사 클래스와 클래스

- ECMAScript 2015 이전에는 Class를 지원하지 않음.  
→ 개발자들이 유사 클래스 형태로 만들어 사용함.
- 다른 Javascript 라이브러리들이 유사클래스 형태로 사용,  
→ ECMAScript 2015에서 정식으로 클래스를 지원.

# ECMAScript 2015이전의 유사 클래스

- Function을 이용해 유사 클래스를 생성.
  - new 키워드를 사용해 객체를 생성함.
  - prototype을 이용해 동적으로 메소드를 할당함.

```
function PersonType(name) {  
  this.name = name;  
}  
  
PersonType.prototype.sayName = function() {  
  console.log("My name is " + this.name);  
}  
  
const person = new PersonType("Min Chang");  
console.log(person.name);  
person.sayName();
```

Min Chang  
My name is Min Chang

- 클래스를 흉내내는 많은 라이브러리들이 이 패턴을 바탕으로 제작함.

# ECMAScript 2015의 클래스

- Class 키워드를 이용해 클래스를 선언함.
  - 클래스를 만드는 기본 바탕은 앞선 function.
  - 생성자, 메소드를 모두 지원한다.

```
class PersonClass {  
  constructor(name) {  
    this.name = name;  
  }  
  
  sayName() {  
    console.log("My name is " + this.name );  
  }  
}  
  
const person = new PersonClass("Min Chang");  
console.log(person.name);  
person.sayName();
```

Min Chang  
My name is Min Chang

- 객체 리터럴과 문법이 유사함.
  - 클래스의 요소들 사이에 콤마(,)가 필요 없음

# 클래스의 특징

- ECMAScript 2015 클래스의 특징
  - 1. 함수 선언과 달리 클래스 선언은 호이스팅 되지 않는다.
  - 2. 클래스 선언 내의 모든 코드는 엄격 모드인 strict 모드에서 실행된다.
  - 3. 모든 메소드는 외부에서 열거(출력)할 수 없다.
  - 4. new 없이 클래스 생성자를 호출할 수 없다.
    - new 없이 객체를 생성할 수 없다.



# 정적 멤버(Static Member) 생성하기

- ECMAScript 2015 이전의 유사 클래스에서 정적 멤버를 추가하기

```
function PersonType(name) {  
  this.name = name;  
}  
  
PersonType.create = function(name) {  
  return new PersonType(name);  
};  
  
PersonType.prototype.sayName = function() {  
  console.log("My name is " + this.name);  
};  
  
var person = new PersonType.create("Min Chang");  
console.log(person.name);  
person.sayName();
```

Min Chang  
My name is Min Chang

- Function 을 이용한 유사 클래스에서 정적 멤버를 추가하기 위해서는 PersonType에 멤버를 추가함으로써 정의할 수 있다.
- 반면, Class를 이용한 클래스에서 정적 멤버를 추가하기 위해서는 static 키워드만 사용하면 된다.

# 정적 멤버(Static Member) 생성하기

- ECMAScript 2015의 클래스에서 정적 멤버를 추가하기

```
class PersonClass {  
  constructor(name) {  
    this.name = name;  
  }  
  
  sayName() {  
    console.log("My name is " + this.name);  
  }  
  
  static create(name) {  
    return new PersonClass(name);  
  }  
}  
  
const person = PersonClass.create("Min Chang");  
console.log(person.name);  
person.sayName();
```

Min Chang  
My name is Min Chang

- 생성자(constructor)에는 static을 정의할 수 없다.
- 정적 멤버는 반드시 클래스에서만 접근할 수 있다.
  - 객체에서 정적 멤버에 접근하면 에러가 발생한다.

# Get / Set

- Class의 멤버에 접근할 수 있도록 get/set 키워드를 제공한다.
  - 멤버에 직접 접근하는 방법보다 get/set을 통해 접근하는 방법이 안정적.
  - **멤버의 이름과 get/set의 이름이 같을 경우 “무한반복”이 발생할 수 있다.**

```
class PersonClass {  
  constructor(name) {  
    this._name = name;  
  }  
  
  get name() {  
    console.log("Getter!");  
    return this._name;  
  }  
  set name(name) {  
    console.log("Setter!");  
    this._name = name;  
  }  
  
  sayName() {  
    console.log("My name is " + this._name);  
  }  
  static create(name) {  
    return new PersonClass(name);  
  }  
}  
  
const person = PersonClass.create("Min Chang");  
person.name = "Tae Ji";  
console.log(person.name);  
person.sayName();
```

Tae Ji  
My name is Tae Ji

# 파생 클래스와 상속

- Extends 키워드를 이용해 클래스 상속이 가능하다.

```
class PersonType {
  constructor(name) {
    this.name = name;
  }

  sayName() {
    console.log("My name is " + this.name);
  }
}

class Student extends PersonType {
  constructor(name, schoolName) {
    super(name);
    this.schoolName = schoolName;
  }

  saySchoolName() {
    console.log("I'm going to " + this.schoolName + " school");
  }
}

const student = new Student("Min Chang", "None");
console.log(student.name, student.schoolName);
student.sayName();
student.saySchoolName();
```

- 상속을 받은 클래스에서 상속한 클래스의 멤버를 사용할 수 있다.

# 파생 클래스와 상속

- Super 키워드를 통해 상속한 클래스의 멤버에 접근할 수 있다.

```
class Student extends PersonType {  
  constructor(name, schoolName) {  
    super(name);  
    this.schoolName = schoolName;  
  }  
  
  saySchoolName() {  
    console.log("I'm going to " + this.schoolName + " school");  
  }  
  
  sayName() {  
    super.sayName();  
    this.saySchoolName();  
  }  
}  
  
const student = new Student("Min Chang", "None");  
console.log(student.name, student.schoolName);  
student.saySchoolName();
```

- 파생 클래스에서 상속한 클래스의 멤버를 재 정의할 경우, 파생 클래스의 멤버를 사용한다. (오버라이딩)

## 11장

# 프로미스와 비동기 프로그래밍

비동기 프로그래밍을 위한 프로미스를 학습한다.

# Javascript의 비동기 프로그래밍

- Javascript의 개발 목적
  - 사용자 웹 페이지에서 사용될 목적
  - 주된 기능은 사용자의 반응하기.
    - 마우스 이벤트(클릭, 이동 등)
    - 키보드 입력(keyup, keydown)
- Javascript는 애초부터 비동기에 반응하기 위한 목적으로 개발됨.

```
<!DOCTYPE html>
<html lang="ko">
  <head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript">
      window.onload = function() {
        var btn = document.querySelector("#btn");
        btn.onclick = function(event) {
          alert("클릭!");
        }
      }
    </script>
  </head>
  <body>
    <input type="button" id="btn" value="클릭" />
  </body>
</html>
```

주로 함수표현식이나 콜백으로 비동기를 처리한다.

또한, 처리 결과에 따라 다른 방법들을 제공하게 됨에 따라 코드가 복잡해지기 시작한다.

# Javascript의 비동기 프로그래밍

- 자바스크립트에서의 확정되지 않은 상태에 대한 처리 방법

```
var btn = document.querySelector("#btn");
btn.onclick = function(event) {
  validateName(function() {
    alert("이름을 입력하세요!");
  }, function () {
    alert("성공!");
  });
}

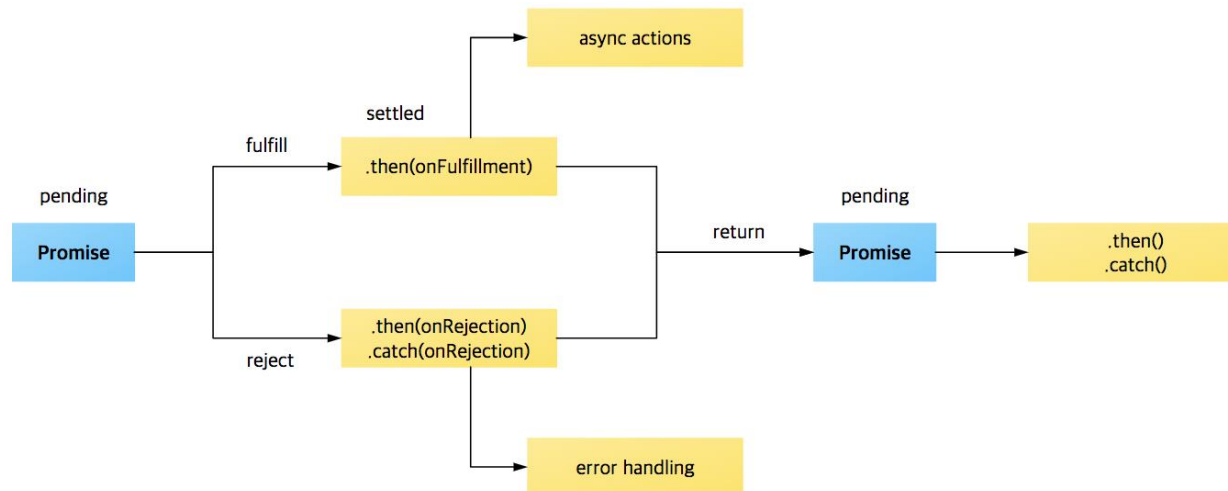
function validateName(error, success) {
  var name = document.querySelector("#name");
  if ( name.value == "" ) {
    error();
  }
  else {
    success();
  }
}
```

- 콜백이 늘어나 자칫 콜백지옥이 빠지기 쉽다.
- 프로미스는 이런 처리에 대한 쉬운 해결책을 제공한다.



# 프로미스

- 프로미스는 비동기 처리 결과가 확정되지 않은 이벤트에 대한 처리 방법을 제공한다.
- 이것을 위해 프로미스는 "보류(pending)", "성공(fulfilled)", "실패(rejected)" 상태를 제공한다.
- Promise의 상태 변화



# 프로미스

- 프로미스가 제공하는 상태에 따라 개발자는 .then() 혹은 .catch() 메소드를 제공한다.

```
<script type="text/javascript">
window.onload = function() {
  let btn = document.querySelector("#btn");
  btn.onclick = function(event) {
    let promise = validate("#name");

    promise.then(function(element) { // 성공
      alert(element.id + " 입력되었습니다.");
    }).catch(function(value) { // 실패
      alert(value);
    });
  };
}
```

```
function validate(selector) {
  return new Promise(function(resolve, reject) {
    var element = document.querySelector(selector);
    if ( element.value == "" ) {
      reject(element.dataset.error); // 실패
    } else {
      resolve(element); // 성공
    }
  });
}
</script>
```

...

```
<input type="text" id="name" data-error="이름을 입력하세요." placeholder="이름을 입력하세요." />
<input type="button" id="btn" value="클릭" />
```

```
btn.onclick = function(event) {
  let promise = validate("#name");

  promise.then(function(element) { // 성공
    alert(element.id + " 입력되었습니다.");
  }, function(value) { // 실패
    alert(value);
  });
}
```

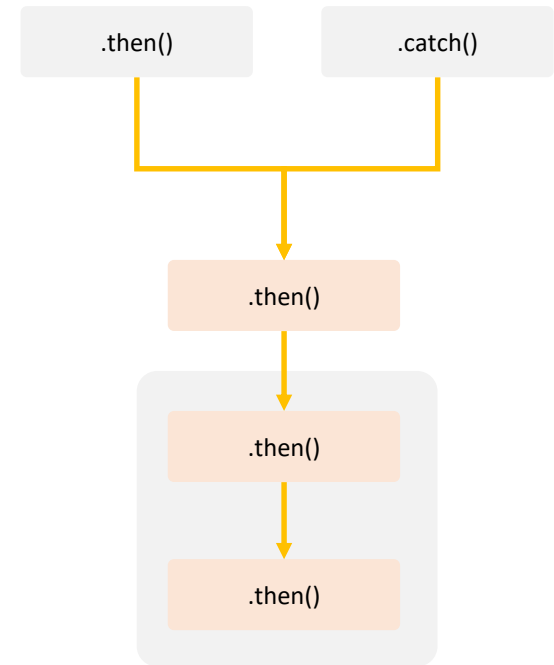
.then() 은 두개의 파라미터를 가진다.  
.then(성공, 실패);

# 프로미스 연결하기

- 프로미스는 상태에 따른 처리가 주된 목적이지만, 프로미스 내에서 다른 프로미스를 만들어 비동기 처리를 연속으로 처리할 수 있도록 한다.
- `.then()`, `.catch()`는 수행될 때마다 다른 프로미스를 만들어 반환한다.

```
let btn = document.querySelector("#btn");
btn.onclick = function(event) {
  let promise = validate("#name");
  promise.then(function(element) {
    alert(element.id + " 입력되었습니다.");
  }).catch(function(value) {
    alert(value);
  }).then(function() {
    alert("다음 프로미스!");
  });
}

function validate(selector) {
  return new Promise(function(resolve, reject) {
    var element = document.querySelector(selector);
    if ( element.value == "" ) {
      reject(element.dataset.error);
    } else {
      resolve(element);
    }
  });
}
```

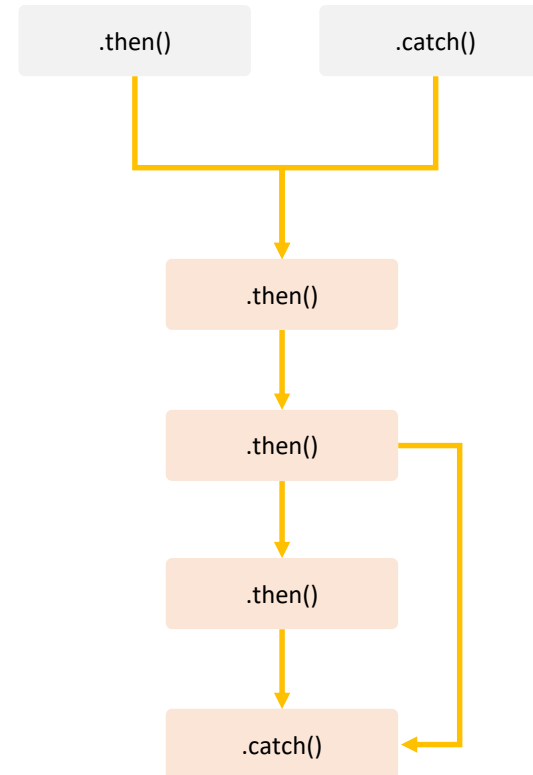


Then이 계속 이어지게할 수도 있다.

# 연결된 프로미스에 에러 처리하기

- 프로미스 중간에 Error가 발생하면, catch()로 처리할 수 있다.

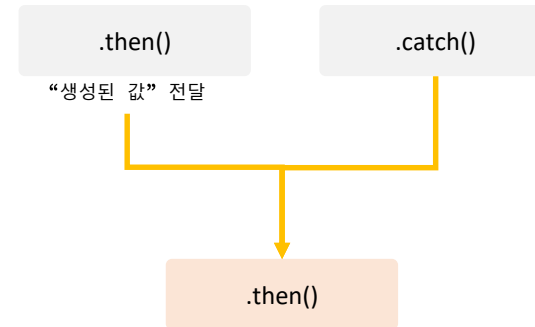
```
promise.then(function(element) {  
  alert(element.id + " 입력되었습니다.");  
}).catch(function(value) {  
  alert(value);  
}).then(function() {  
  alert("다음 프로미스!1");  
}).then(function() {  
  alert("다음 프로미스!2");  
  throw new Error("에러 발생!");  
}).then(function() {  
  alert("다음 프로미스!3");  
}).catch(function(error) {  
  alert(error);  
});
```



# 프로미스 연결에서 값 반환하기

- .catch()나 .then() 에서 다음 .then()으로 넘어갈 때, 값을 전달할 수 있다.

```
promise.then(function(element) {  
  alert(element.id + " 입력되었습니다.");  
  return "생성된 값";  
}).catch(function(value) {  
  alert(value);  
}).then(function(value) {  
  if ( value ) {  
    alert(value);  
  }  
});
```



# 여러개의 프로미스에 응답하기

- 일반적인 프로미스는 한번에 하나의 프로미스만 응답한다.
- ECMAScript 2015 에서는 여러개의 프로미스를 동시에 처리할 수 있도록 지원한다.
  - `Promise.all()`
    - 여러개의 프로미스가 모두 성공했을 때 값을 반환한다.
    - ShortCut 평가 수행.
  - `Promise.race()`
    - 여러개의 프로미스 중 먼저 끝난 값을 반환한다. 나머지는 무시된다.

# Promise.all()

- 여러개로 연결된 프로미스들이 모두 완료되었을 때 최종 결과값을 반환.
- 여러개의 비동기 작업이 모두 완료된 이후 작업을 수행할 때 유용하게 사용.

```
let btn = document.querySelector("#btn");
btn.onclick = function(event) {
  let namePromise = validate("#name");
  let agePromise = validate("#age");
  let deptPromise = validate("#dept");

  let validateAll = Promise.all([namePromise, agePromise, deptPromise]);

  validateAll.then(function(value) {
    console.log(value[0].id);
    console.log(value[1].id);
    console.log(value[2].id);
  });
}

function validate(selector) {
  return new Promise(function(resolve, reject) {
    let element = document.querySelector(selector);
    if ( element.value == "" ) {
      reject(element.dataset.error);
    } else {
      resolve(element);
    }
  });
}
```

# Promise.all()

- 여러개의 프로미스를 Promise.all(); 내의 파라미터에 배열로 전달한다.

```
let validateAll = Promise.all([namePromise, agePromise, deptPromise]);
```

- 개별 프로미스가 처리된 순서대로 하나의 프로미스에 저장된다.

```
validateAll.then(function(value) {  
  console.log(value[0].id);  
  console.log(value[1].id);  
  console.log(value[2].id);  
});
```

- 만약, Promise.all()에 지정된 프로미스 중 하나라도 실패하면 Promise.all()은 중지된다. (ShortCut)

```
validateAll.then(function(value) {  
  console.log(value[0].id);  
  console.log(value[1].id);  
  console.log(value[2].id);  
}).catch(function(value) {  
  console.log(value);  
});
```



# Promise.race()

- 여러개의 프로미스가 동시에 실행된 후 먼저 값을 반환한 프로미스만 결과로 전달한다.

```
let btn = document.querySelector("#btn");
btn.onclick = function(event) {
  let namePromise = validate("#name");
  let agePromise = validate("#age");
  let deptPromise = validate("#dept");

  let validateRace = Promise.race([namePromise, agePromise, deptPromise]);

  validateRace.then(function(value) {
    console.log(value.id);
  }).catch(function(value) {
    console.log(value);
  });
}

function validate(selector) {
  return new Promise(function(resolve, reject) {
    var element = document.querySelector(selector);
    if ( element.value == "" ) {
      reject(element.dataset.error);
    } else {
      resolve(element);
    }
  });
}
```

# Promise.race()

- 여러개의 프로미스를 Promise.race(); 내의 파라미터에 배열로 전달한다.

```
let validateRace = Promise.race([namePromise, agePromise, deptPromise]);
```

- 등록된 프로미스 중 먼저 처리된 프로미스만 반환된다.

```
validateRace.then(function(value) {  
  console.log(value.id);  
}).catch(function(value) {  
  console.log(value);  
});
```

# 프로미스와 제네레이터

- 제네레이터와 프로미스를 이용하면, 비동기 코드를 아주 유연하게 처리할 수 있다.

```
let btn = document.querySelector("#btn");
btn.onclick = function(event) {
  run(function *() {
    try {
      let namePromise = yield validate("#name");
      console.log(namePromise.id + " 입력됨");
      let age = yield foo();
      console.log(age + " 입력됨");
      let deptPromise = yield validate("#dept");
      console.log(deptPromise.id + " 입력됨");
    } catch(e) {
      console.log(e);
    }
  });
}

function foo() {
  return 50;
}

function validate(selector) {
  return new Promise(function(resolve, reject) {
    var element = document.querySelector(selector);
    if ( element.value == "" ) {
      reject(element.dataset.error);
    }
    else {
      resolve(element);
    }
  });
}
```

Promise.resolve()에 프로미스가 전달되면 그대로 통과시킨다.  
단, 일반함수/값일 경우 프로미스로 감싸서 반환한다.

```
function run(taskFunc) {
  let task = taskFunc();
  let result = task.next();

  (function step() {
    if(!result.done) {
      let promise = Promise.resolve(result.value);
      promise.then(function(value) {
        result = task.next(value);
        step();
      }).catch(function(error) {
        result = task.throw(error);
        step();
      });
    }
  })();
}
```

## 12장

# 모듈로 캡슐화하기

Javascript에는 존재하지 않던 모듈을 학습하고, 사용방법을 실습한다.

# 모든 것을 공유하는 Javascript

- 기존의 Javascript는 패키지 등 코드를 분할 할 수 있는 방법이 없음.
- 따라서 아래와 같은 코드가 있을 경우, 함수나 변수의 이름이 충돌할 가능성이 높았고, 이는 곧 에러를 발생시키게 됨.

```
<script type="text/javascript" src="common.js"></script>  
<script type="text/javascript" src="ui.js"></script>  
<script type="text/javascript" src="wyswyg.js"></script>
```

- 자바스크립트에서 최상위에 선언된 함수나 변수는 Global Scope에 등록된다.
- 다른 스크립트를 로드하더라도 Global Scope에 등록된다.
- ECMAScript 2015에서는 이런 문제를 해결하기 위해 “모듈” 시스템을 도입.
  - 함수나 변수의 이름을 충돌시킬 수 있는 가능성을 낮춰줌.
- 모듈은 스크립트 파일 내에서 원하는 함수나 변수만 골라 사용할 수 있다.

# 모듈이란

- 모든 것을 공유하는 구조와 다르게, 모듈의 최상위 수준에서 만들어진 변수는 Global Scope에 등록되지 않는다.
- 모듈에 등록된 변수나 함수는 모듈내의 Global Scope에 등록된다.
- 모듈에 등록된 변수나 함수는 외부에서 자유롭게 사용할 수 없다.
  - 외부에서 사용하기 위해서는 반드시 export 키워드를 사용해 주어야 한다.
  - export된 함수나 변수를 외부에서 사용하려면 import 키워드를 사용해야 한다.

# Export 기본

- 다른 모듈에 코드 일부를 노출시키기 위해 `export` 키워드 사용.

```
// 데이터 익스포트
export var color = "red";
export let name = "Min Chang";
export const magicNumber = 7;

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

// 비공개 함수
function subtract(num1, num2) {
  return num1 - num2;
}

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

// 위에서 정의한 함수 익스포트
export { multiply }
```

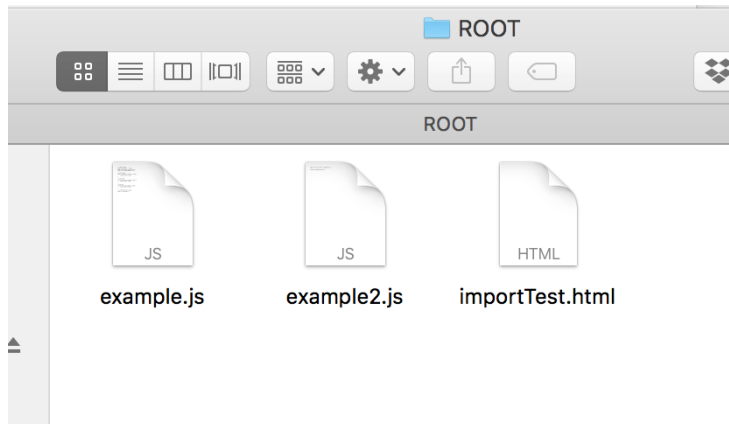
Export 키워드로 선언되지 않은 함수나 변수는 외부에서 사용할 수 없다.

# Import 기본

- Export 한 모듈이 있을 때, import 키워드를 이용해 접근할 수 있다.

```
// example2.js  
  
import { sum } from "./example.js";  
  
console.log(sum(10, 20));
```

- CommonsJS를 사용하는 Node.js에서는 import를 사용할 수 없다.
  - require(); 사용
- 이 코드는 Local에서 실행될 수 없다. 반드시 서버(tomcat 등)에 등록해 테스트 해야 한다.



```
// importTest.html  
  
<!DOCTYPE html>  
<html lang="ko">  
  <head>  
    <meta charset="UTF-8">  
    <title>Document</title>  
    <script type="module">  
      import { sum } from "./example.js";  
      console.log(sum(10, 20));  
    </script>  
  </head>  
  <body>  
  </body>  
</html>
```



# Import 기본

- Export된 변수나 함수는 필요에 따라 하나 또는 여러개 또는 모두 import할 수 있다.

```
// 하나 임포트하기
```

```
import { sum } from "./example.js";
```

```
console.log(sum(10, 20));
```

```
// 여러개 임포트하기
```

```
import { sum, multiply } from "./example.js";
```

```
console.log(sum(10, 20));
```

```
console.log(multiply(10, 20));
```

```
// 모두 임포트하기
```

```
import * as example from "./example.js";
```

```
console.log(example.sum(10, 20));
```

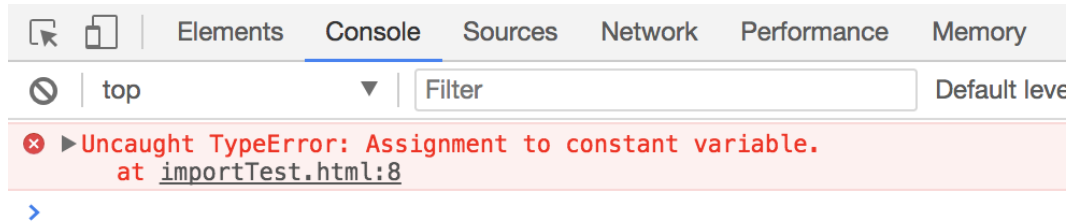
```
console.log(example.multiply(10, 20));
```

```
console.log(example.magicNumber);
```

# Import 의 특징

- Export된 변수나 함수를 Import 했을 경우, 함수 및 변수 등은 모두 읽기 전용으로 바인딩 된다.

```
import { sum } from "./example.js";
sum = function() {
  return "Hello";
}
console.log(sum());
```

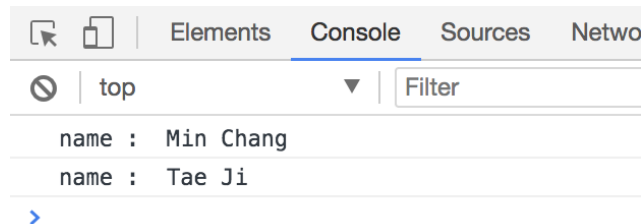


- 단, 함수를 통한 모듈내의 값은 변경이 가능하다.

```
// 데이터 익스포트
export let name = "Min Chang";

export function setName(newName) {
  name = newName;
}
```

```
import { name, setName } from "./example.js";
console.log("name : ", name);
setName("Tae Ji");
console.log("name : ", name);
```



# 별칭으로 Import 하기

- 일반적으로 Export한 이름 그대로 Import 됨.
- 때때로, 중복된 이름 혹은 너무 긴 이름의 변수, 함수, 클래스가 있을 경우 새로운 이름으로 Import해 올수 있다.

```
import { sum as add } from "./example.js";  
// Uncaught ReferenceError: sum is not defined  
// console.log(sum(10, 20));  
console.log(add(10, 20));
```

# 모듈의 대표(변수/함수/클래스) 만들기

- 모듈에서 여러개의 변수/함수/클래스를 export할 수도 있지만 대표 변수/함수/클래스 하나만 export할 수도 있다.
- default 키워드를 사용해 대표 변수/함수/클래스를 만들수 있다.
- 하나의 모듈에는 하나의 default만 사용할 수 있다.

```
// Default 함수 만들기
export default function(num1, num2) {
  return num1 + num2;
}
```

모듈에 Default가 등록되어 있으면, 그 자체가 모듈이 되므로 이름을 작성하지 않아도 된다.

```
import sum from "./example.js";
console.log(sum(10, 20));
```

Default 모듈을 임포트 하려면, 임의의 이름을 부여한다.

```
function sum(num1, num2) {
  return num1 + num2;
}

export default sum;
```

함수를 정의한 후 export default 를 사용해도 된다.

```
function sum(num1, num2) {
  return num1 + num2;
}

export { sum as default };
```

혹은 as default 식별자를 사용할 수도 있다.

# Default와 Export의 동시 Import

- Default 로 정의된 변수/함수/클래스와 동시에 일반적인 export를 함께 Import 할 수도 있다.
- 아래와 같이 정의된 모듈이 있을 때

```
export let color = "red";  
  
export default function (num1, num2) {  
  return num1 + num2;  
}
```

- 아래와 같은 방법들로 Import할 수 있다.

```
import sum, {color} from "./example.js";  
console.log(sum(10, 20));  
console.log(color);
```

```
import {default as sum, color} from "./example.js";  
console.log(sum(10, 20));  
console.log(color);
```

# Import를 다시 Export하기

- 이미 정의된 모듈을 이용해 더 확장된 모듈을 만들고 싶을 때, Import 후 Export할 수 있다.

```
// example.js  
  
export let color = "red";  
  
export default function (num1, num2) {  
  return num1 + num2;  
}
```



```
import sum from "./example.js";  
export { sum };
```



```
import { sum } from "./example2.js";  
console.log(sum(10, 20));
```

```
// example.js  
  
export let color = "red";  
  
export function sum (num1, num2) {  
  return num1 + num2;  
}
```



```
export {sum} from "./example.js";
```



```
import { sum } from "./example2.js";  
console.log(sum(10, 20));
```

수고하셨습니다.