

Javascript

Pure Javascript

장민창

mcjang@hucloud.co.kr

Javascript

1. Javascript
2. Javascript 맛보기
3. Javascript IDE 설치
4. Javascript 기본문법

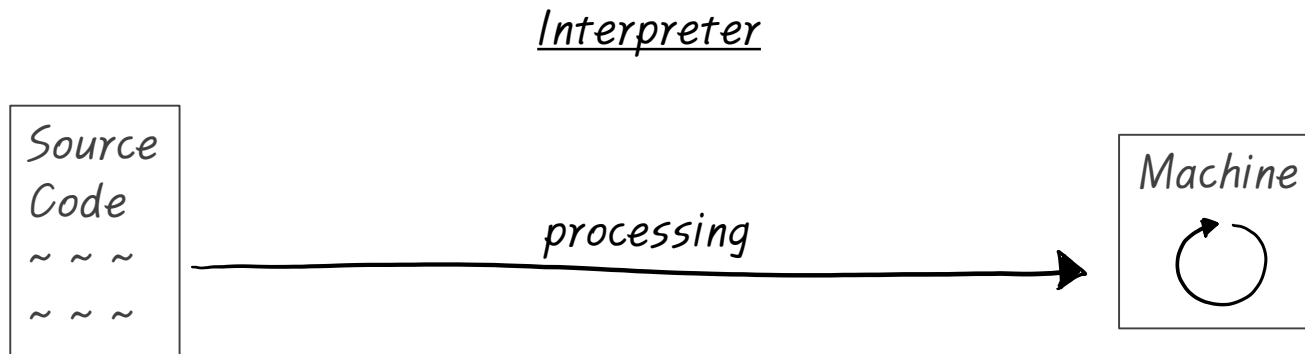
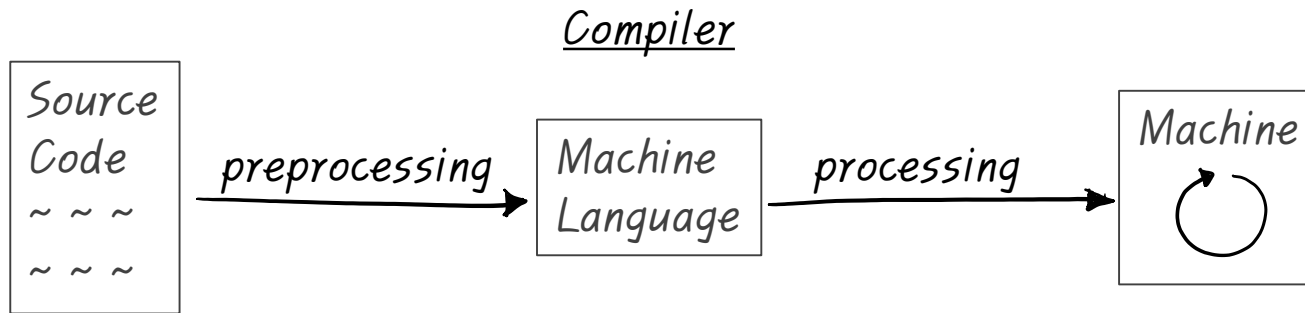
Javascript

JavaScript

- Browser (Client) 에서 동작하는 Client-Side Language
- 이벤트 기반의 동작 언어
 - 예> 사용자가 가입하기 버튼을 클릭할 때 000을 해라.
 - 사용자가 ID를 입력할 때 중복검사 체크를 해라.
- 웹 페이지에서 없어서는 안될 언어
 - Javascript는 동적인 페이지를 만들 수 있도록 도와준다.
- 심지어, 페이지가 하나밖에 존재하지 않는 SPA(Single Page Application)를 만들때에도 Javascript는 필수.
- 2000년대 중반 Web 2.0 열풍은 javascript가 주도함.
 - 기존의 Client – Server 개발 패러다임에 상당히 큰 변화를 일으킴 (Ajax)
- 현재는 Server-Side 에서도 Javascript가 사용된다.
 - MongoDB
 - Node.js

JavaScript

- Javascript는 Compile 언어가 아니다.
- Javascript는 Interpreter 언어.



JavaScript

- ECMA Script 5 가 현재까지 가장 많이 쓰이는 언어 (Client, Server 포함)
 - 함수 지향 언어
 - 다른 언어와는 다른 개념(Scope, Proto, Prototype)으로 혼란이 가중됨.
 - 객체 지향 개념도 지원하지만, 그 형태가 단순하지 않아 사용하기에 어려움
- 최근 ECMA Script 6가 발표되면서 새로운 언어로 재 탄생
 - ECMA Script 5 가 가지던 이미지(Programming Language 가 아니라, Support Tool로써의)를 버림
 - 객체 지향 개념을 지원하기 시작
 - Class 의 개념 도입
 - 다른 언어와 유사한 개념들을 지원함
- ECMA Script 6의 브라우저 지원은 아직 미지수
 - Server-Side 측에서는 적극적으로 지원하는 중
 - 대표적으로 Node.js
- 현재는 ECMA Script 7의 재정 논의되고 있는 중

Javascript 맛보기

JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
 - Web Browser 를 활용하는 방법!



Firefox



chrome



Opera



Safari



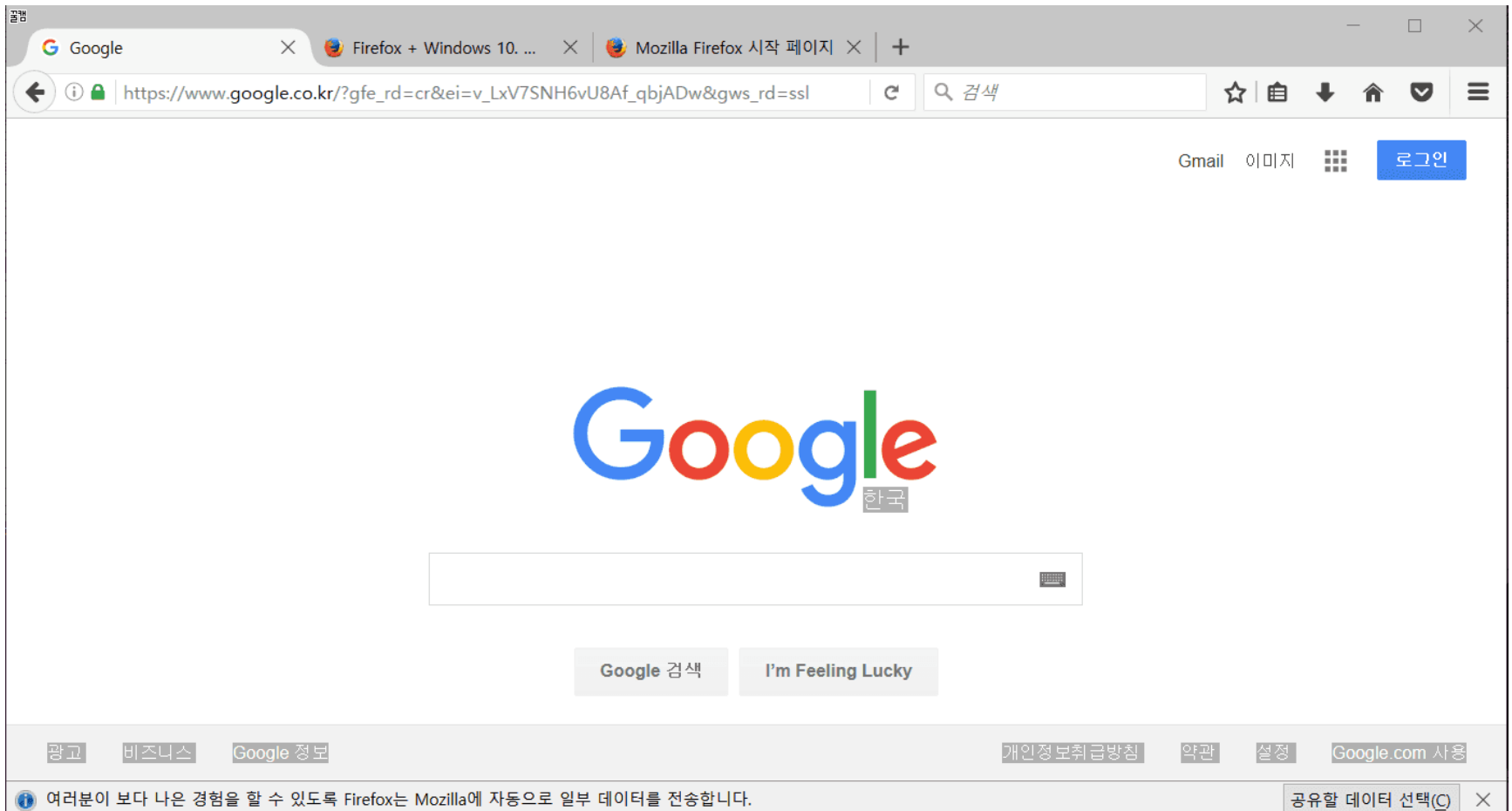
Internet Explorer



MS Edge

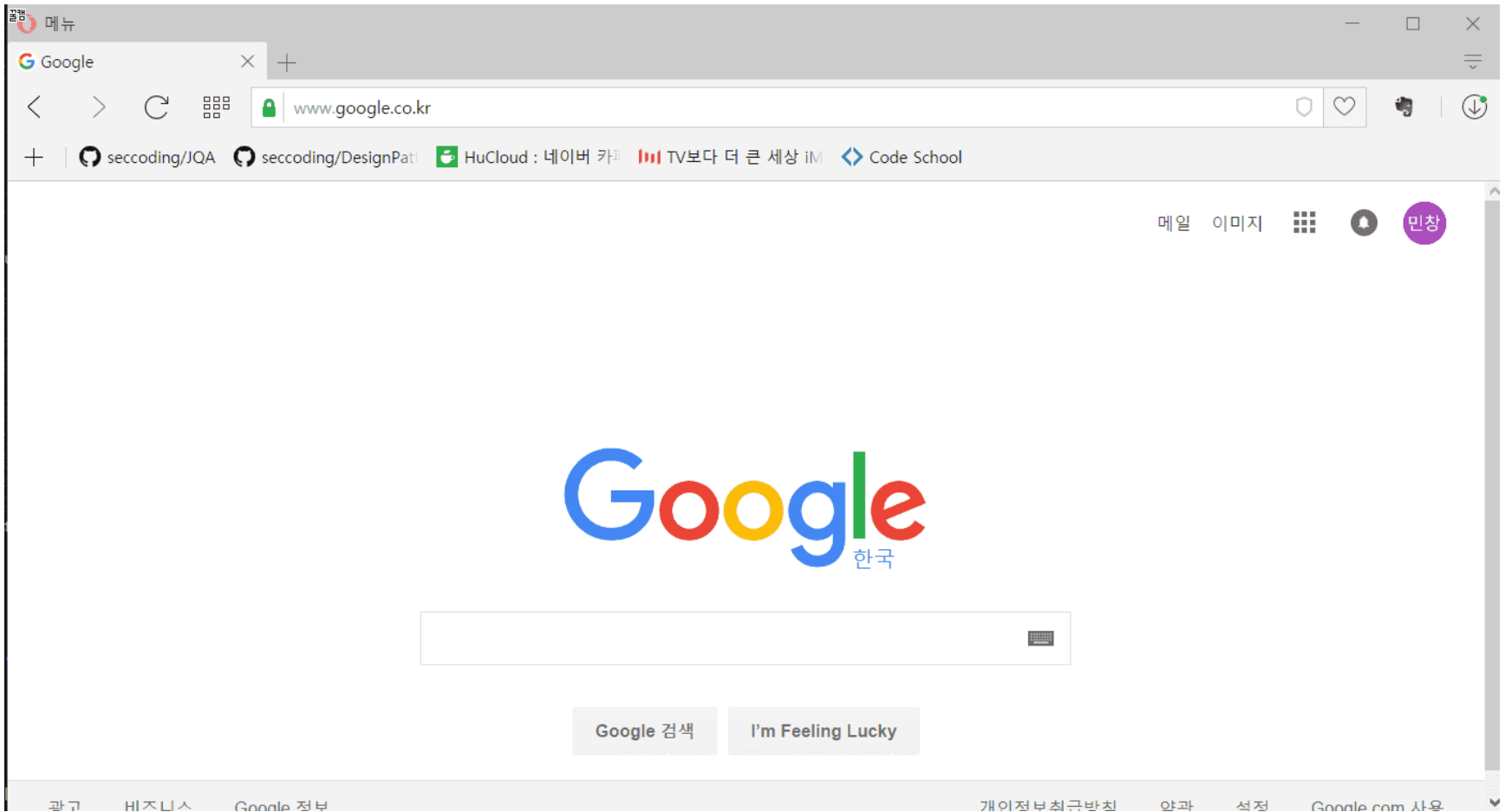
JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
- Firefox Browser



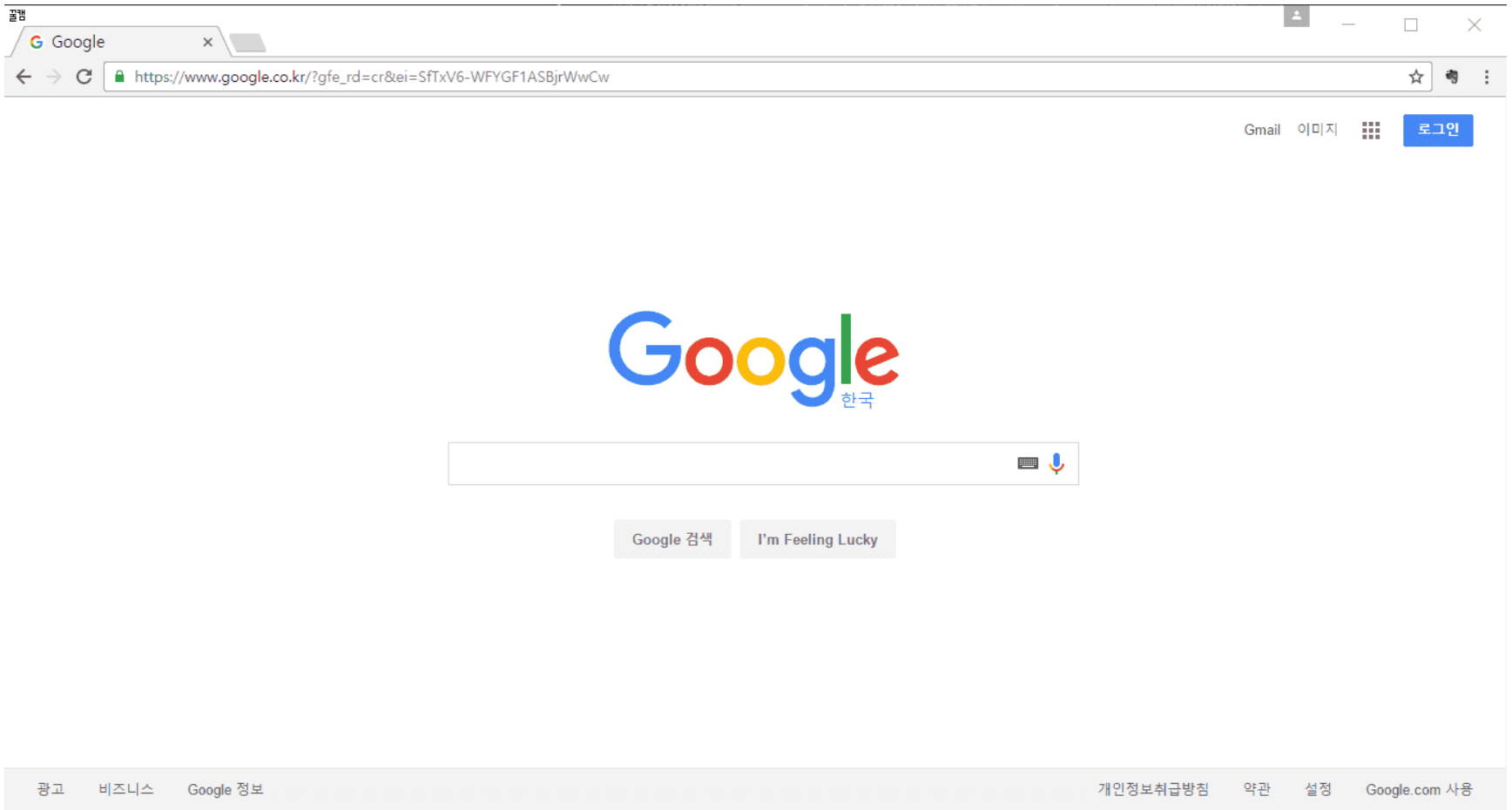
JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
- Opera Browser



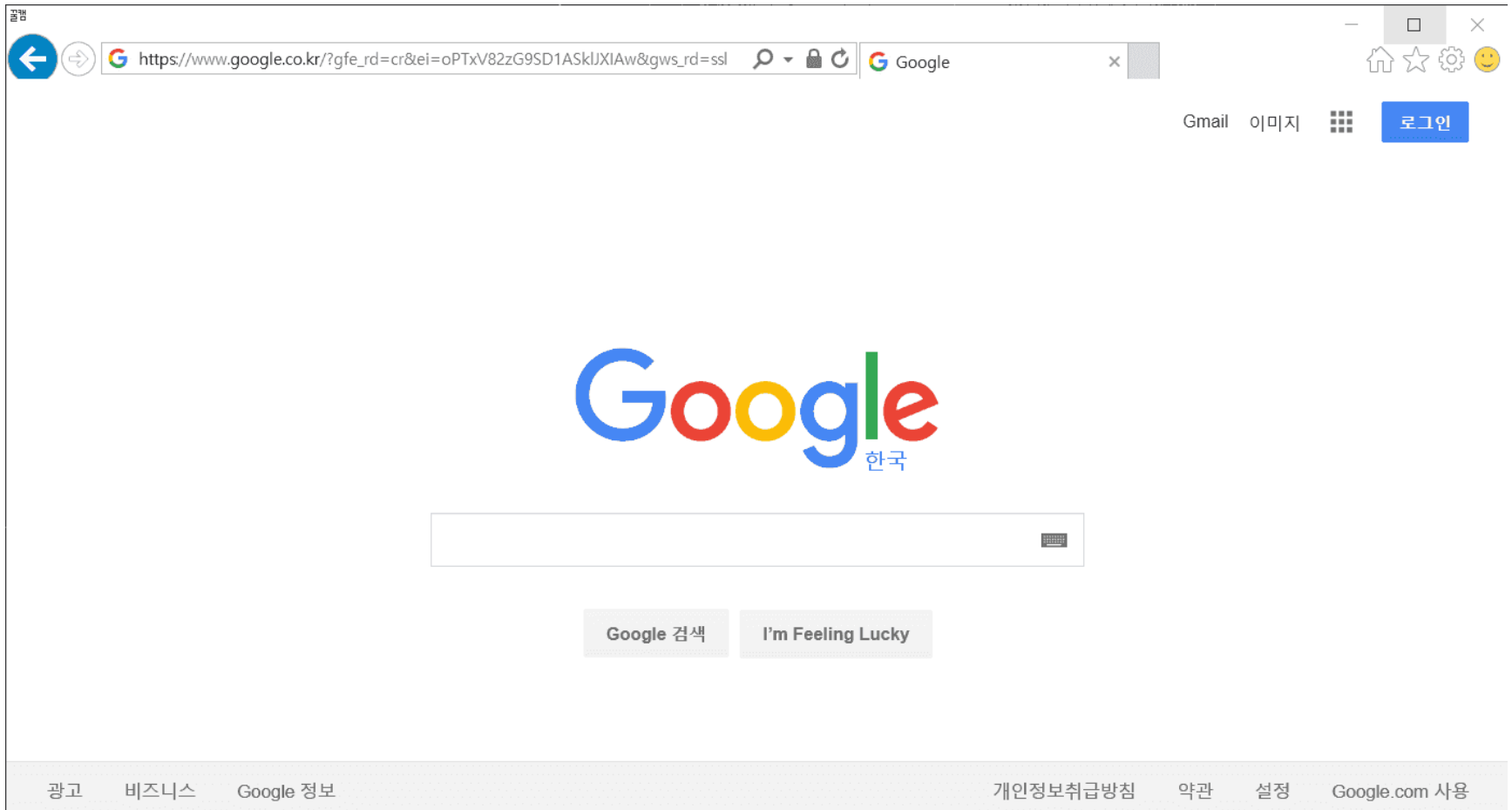
JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
- Chrome Browser



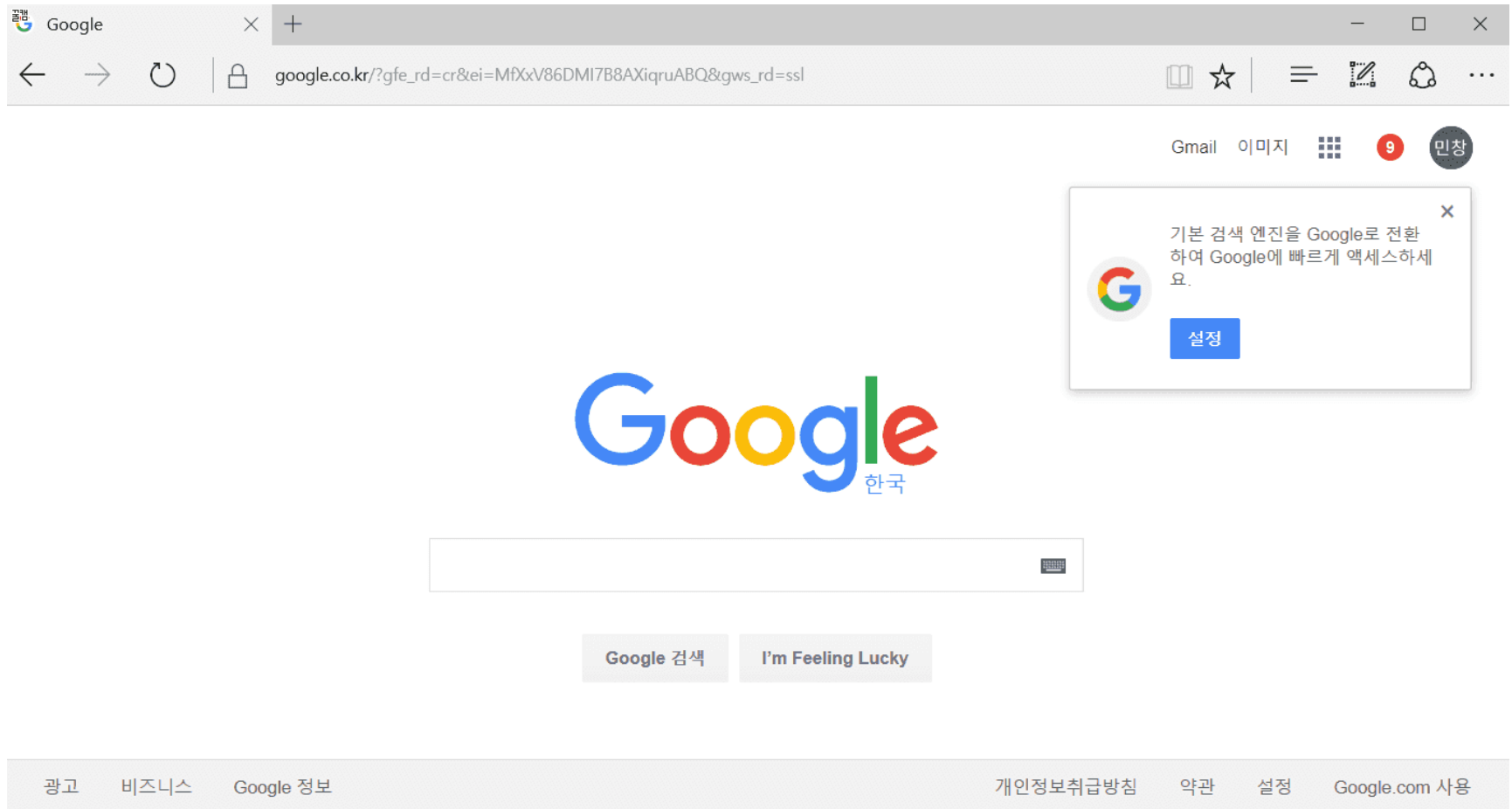
JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
- Internet Explorer



JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
- MS Edge



JavaScript

- Javascript 를 사용할 수 있는 가장 간단한 방법
- 경고창 띄어보기 `alert("안녕하세요");`



JavaScript

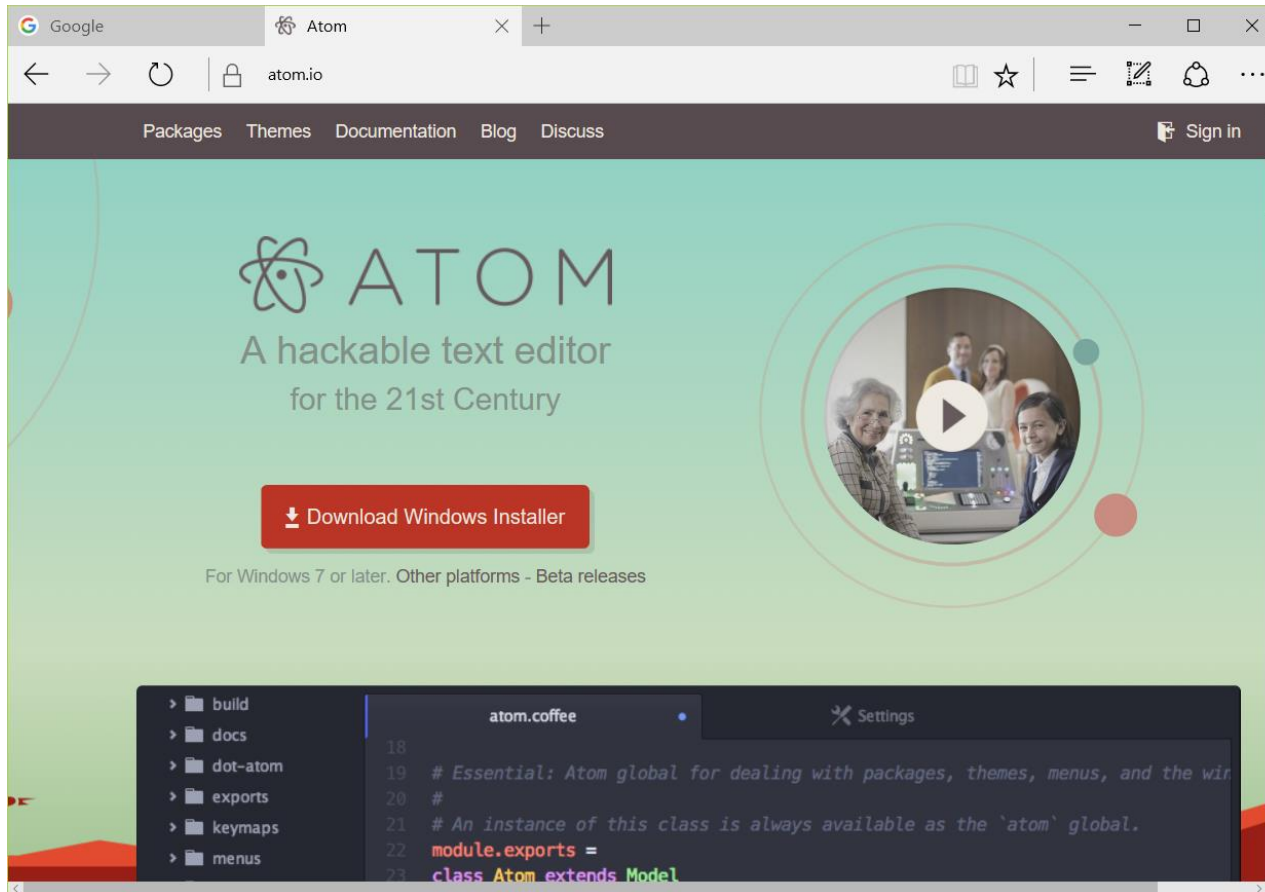
- Javascript 를 사용할 수 있는 가장 간단한 방법
- 확인 창 띄어보기 `confirm("계속 하시겠습니까?");`



Javascript IDE 소개 및 설치

Javascript IDE 소개 및 설치

- <https://atom.io>
 - Download Windows Installer 클릭 해 파일 다운로드 후 설치 진행



Javascript IDE 소개 및 설치

- Atom 설치 후 실행



Javascript 기본 문법 살펴보기

변수

변수

- 데이터를 계산할 목적으로 잠시 보관할 저장소
- 예> 계산기

$$\boxed{10} + \boxed{50} = \boxed{60}$$

$$\boxed{\text{numberA}} + \boxed{\text{numberB}} = \boxed{\text{result}}$$

- 변수의 선언 방법
 - **var** 키워드를 사용한다.
`var 변수명;`
 - 명령의 끝은 반드시 **;(세미콜론)** 으로 끝낸다.

- 변수의 할당
 - **=** 키워드를 사용한다.

`var 변수명 = 값;`

데이터의 표현

데이터의 표현

- Javascript는 다른 컴파일 언어(C, Java, C++ 등)와 다르게 데이터 타입이 없다.
- Interpreter 언어의 특성상, 변수 속의 데이터를 특정할 수 없기 때문.
- 단, 정수, 실수, 문자열, 불린 등의 값 표현은 가능함.

데이터의 표현

- 정수의 표현

```
var count = 10;  
var age = 40;  
var number = 660;
```

- 실수의 표현

```
var pi = 3.14;  
var floatingNumber = 50.13;
```

- 불린의 표현

```
var isTrue = true;  
var isNotTrue = false;
```

- 문자의 표현

```
var str1 = "문자열 표현입니다.";   
var str2 = '문자열 표현입니다.';
```


데이터의 표현

- 그 외 특수한 표현들
 - 정의는 되었지만, 값이 할당되지 않았을 때 사용될 표현

```
undefined
```

- 값이 비어 있을 경우 사용할 수 있는 표현
- undefined와 다르게 직접 할당해야 한다.

```
null
```

```
var name = null;
```

변수 값 변경

변수 값 변경

- 변수의 값은 언제든지 변경될 수 있다.

```
var count = 10;  
count = 15;  
  
var number = 660;  
number = 177;
```

- var** 키워드는 변수의 정의를 담당한다.
- var** 키워드를 삭제한 후 값을 할당하면, 변수 값의 변경을 의미한다.
- var** 키워드를 삭제하지 않고 변수를 할당하더라도 값의 재할당이 이루어진다.

```
var count = 10;  
var count = 15;  
  
var number = 660;  
var number = 177;
```

변수 값 변경

- 연산에 의한 변경
- 변수의 값을 변경하는 경우는 보통 연산에 의한 변경이 대부분.

```
var count = 10;  
count = 40 + 60;
```

변수의 사용 - 출력

변수의 사용 - 출력

- 변수를 출력.

```
alert("안녕하세요.");
```

- **alert(...)** 은 화면에 경고창을 띄어주는 역할을 함
- **alert(...)** 에 변수를 사용하면 변수의 내용이 경고창으로 나타난

- ```
var helloMessage = "안녕하세요.";
alert(helloMessage);
```

- **console.log(...)** 는 개발자 도구에 변수의 값을 출력하는 역할을 함.
- 실제 개발 환경에서 Debugging 을 할 때, 주로 사용됨.

```
var helloMessage = "안녕하세요.";
console.log(helloMessage);
```

이름, 회사명을 저장할 수 있는 변수를 만들고  
alert(...); 과 console.log(...); 를 이용해 출력해보세요.

# 변수의 사용 - 출력

- 화면에 출력하기
- HTML 문서내에 Javascript 변수의 내용을 출력하기
- HTML 문서내 script 태그를 만들어 아래와 같이 코딩함

```
<html>
 <head>
 <script type="text/javascript">
 var helloMessage = "안녕하세요.";
 document.write(helloMessage);
 </script>
 </head>
 <body>
 </body>
</html>
```

이름, 회사명을 저장할 수 있는 변수를 만들고  
document.write(...); 를 이용해 출력해보세요.

# 변수의 사용 - 연산



# 변수의 사용 - 연산

- Javascript는 변수의 연산을 위해 기본적으로 사칙 연산(+, -, \*, /)을 지원함

- 더하기

숫자 + 숫자

```
var add = 9 + 7;
```

→ 16

- 빼기

숫자 - 숫자

```
var sub = 60 - 15;
```

→ 45

- 곱하기

숫자 \* 숫자

```
var mul = 9 * 9;
```

→ 81

- 나누기의 몫

숫자 / 숫자

```
var div = 10 / 5;
```

→ 2

# 변수의 사용 - 연산

- Javascript는 변수의 연산을 위해 기본적으로 사칙 연산(+, -, \*, /)을 지원함
  - 나누기의 나머지

숫자 % 숫자

```
var rem = 10 % 3;
```

→ 1

- 연산의 우선순위 ( PEMDAS : 괄호 - 제곱 - 곱셈 - 나눗셈 - 덧셈 - 뺄셈 )

$2 \times 2 + 2 = ?$

→ 6

$2 \times (2 + 2) = ?$

$2 \times 4 = ?$

→ 8

$2 \times (2 \times (7 - 2)) = ?$

$2 \times (2 \times 5) = ?$

$2 \times 10 = ?$

→ 20

# 변수의 사용 - 연산

- 단항 연산자

```
var number = 10;
number = number + 1;
```

→ 11

```
var number = 10;
number += 1;
```

→ 11

```
var number = 10;
number++;
```

→ 11

```
var number = 10;
number = number - 1;
```

→ 9

```
var number = 10;
number -= 1;
```

→ 9

```
var number = 10;
number--;
```

→ 9

```
var number = 10;
number = number * 10;
```

→ 100

```
var number = 10;
number *= 10;
```

→ 100

```
var number = 10;
number = number / 2;
```

→ 5

```
var number = 10;
number /= 2;
```

→ 5

# 변수의 사용 - 연산

- 문자열 연산

문자열 + 모든 자료형

- 문자열 + 모든 자료형

- 문자열과 더해진 모든 것들은 문자열로써 더해진다.

문자열 + 숫자

```
var stringAndNumber = "반갑습니다." + 10;
```

문자열 + 불린

```
var stringAndBoolean = "이 값은 " + true + " 입니다. " ;
```

문자열 + 문자열

```
var stringAndString = "반갑습니다, " + " JavaScript 세계에 오신 것을 환영합니다! " ;
```

# 변수의 사용 - 참조

# 변수의 사용 - 참조

- 변수는 다른 변수의 값으로 참조될 수 있다.

```
var count = 5;
var myNumber = count;
console.log(myNumber);
```

→ 5

- 연산자의 항으로도 사용될 수 있다.

```
var count = 5;
var myNumber = count + 5;
console.log(myNumber);
```

→ 10

- 문자열 연산에도 사용될 수 있다.

```
var count = 5;
var myNumber = count + "당신은 " + count + " 번째 입니다.";
console.log(myNumber);
```

→ 당신은 5 번째 입니다.

# 주석

# 주석

---

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

- Single line Command

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

- Multi line Command

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



**반복문 - for**

# 반복문 - for

- 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 - for

- 비교 연산자

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

# 반복문 - for

- 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.

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

# 반복문 - for

- 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
---	---------	----

# 반복문 - for

- 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.

# 반복문 - for

- 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.

# 반복문 - for

- 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.



# 반복문 - for

- 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.

# 반복문 - for

- 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.

# 반복문 - for

- 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!

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.




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

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.



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

i	i > 0 ?	출력
---	---------	----

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.

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

i	i > 0 ?	출력
5	TRUE	5 번째 손님 응대 중입니다.

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.


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

i	i > 0 ?	출력
5	TRUE	5 번째 손님 응대 중입니다.
4	TRUE	4 번째 손님 응대 중입니다.

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.



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

i	i > 0 ?	출력
5	TRUE	5 번째 손님 응대 중입니다.
4	TRUE	4 번째 손님 응대 중입니다.
3	TRUE	3 번째 손님 응대 중입니다.



# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.

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

i	i > 0 ?	출력
5	TRUE	5 번째 손님 응대 중입니다.
4	TRUE	4 번째 손님 응대 중입니다.
3	TRUE	3 번째 손님 응대 중입니다.
2	TRUE	2 번째 손님 응대 중입니다.

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.

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

i	i > 0 ?	출력
5	TRUE	5 번째 손님 응대 중입니다.
4	TRUE	4 번째 손님 응대 중입니다.
3	TRUE	3 번째 손님 응대 중입니다.
2	TRUE	2 번째 손님 응대 중입니다.
1	TRUE	1 번째 손님 응대 중입니다.

# 반복문 - for

- 동일한 코드의 반복적인 사용이 필요할 때 사용함 (역순 진행)

초기값과 조건식, 증감식이 변화함.

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

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

# 반복문 - for

- for 안에 for 사용하기

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

# 반복문 - for

- for 안에 for 사용하기

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

i	j	i < 2 ?	j < 2 ?	출력
---	---	---------	---------	----

# 반복문 - for

- for 안에 for 사용하기

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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.

# 반복문 - for

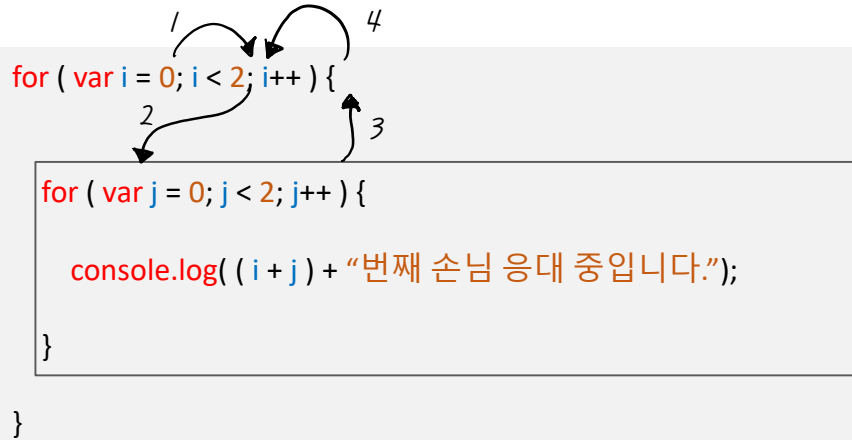
- for 안에 for 사용하기

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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.
0	1	TRUE	TRUE	1 번째 손님 응대 중입니다.

# 반복문 - for

- for 안에 for 사용하기



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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.
0	1	TRUE	TRUE	1 번째 손님 응대 중입니다.
0	2	TRUE	FALSE	j for STOP!



# 반복문 - for

- for 안에 for 사용하기

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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.
0	1	TRUE	TRUE	1 번째 손님 응대 중입니다.
0	2	TRUE	FALSE	j for STOP!
1	0	TRUE	TRUE	1 번째 손님 응대 중입니다.

# 반복문 - for

- for 안에 for 사용하기

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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.
0	1	TRUE	TRUE	1 번째 손님 응대 중입니다.
0	2	TRUE	FALSE	j for STOP!
1	0	TRUE	TRUE	1 번째 손님 응대 중입니다.
1	1	TRUE	TRUE	2 번째 손님 응대 중입니다.

# 반복문 - for

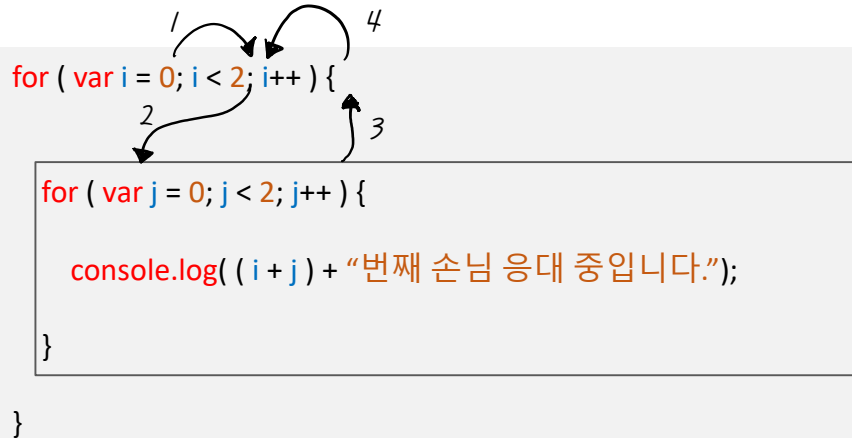
- for 안에 for 사용하기

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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.
0	1	TRUE	TRUE	1 번째 손님 응대 중입니다.
0	2	TRUE	FALSE	j for STOP!
1	0	TRUE	TRUE	1 번째 손님 응대 중입니다.
1	1	TRUE	TRUE	2 번째 손님 응대 중입니다.
1	2	TRUE	FALSE	j for STOP!

# 반복문 - for

- for 안에 for 사용하기



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

i	j	i < 2 ?	j < 2 ?	출력
0	0	TRUE	TRUE	0 번째 손님 응대 중입니다.
0	1	TRUE	TRUE	1 번째 손님 응대 중입니다.
0	2	TRUE	FALSE	j for STOP!
1	0	TRUE	TRUE	1 번째 손님 응대 중입니다.
1	1	TRUE	TRUE	2 번째 손님 응대 중입니다.
1	2	TRUE	FALSE	j for STOP!
2	0	FALSE	STOP!	i for STOP!

# 반복문 - for

---

for를 이용해 1 부터 100 사이 중 3의 배수만 출력해보세요.

for와 증감 연산자를 이용해 1 부터 100 사이 중 짝수만 출력해보세요.

중첩 for를 이용해 구구단을 출력해 보세요.

# 반복문 - while

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.

// 반복을 제어할 변수

var i = 0;

while (반복 진행 여부 체크) {

반복 문장

}

이 조건문이 *true* 일 때만 반복을 수행한다.  
만약 *false* 라면 절대 반복을 수행하지 않는다.

// 반복을 제어할 변수

var i = 0;

while (i > 5) {

console.log(i + "번째 손님 응대 중 입니다..");

}

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
var i = 0;
while (i < 5) {

 console.log(i + "번째 손님 응대 중 입니다..");

}
```

i	i < 5 ?	출력
---	---------	----



# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
var i = 0;
while (i < 5) {

 console.log(i + "번째 손님 응대 중 입니다..");

}
```

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
var i = 0;
while (i < 5) {

 console.log(i + "번째 손님 응대 중 입니다..");

}
```

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
var i = 0;
while (i < 5) {

 console.log(i + "번째 손님 응대 중 입니다..");

}
```

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
var i = 0;
while (i < 5) {

 console.log(i + "번째 손님 응대 중 입니다..");

}
```

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
var i = 0;
while (i < 5) {

 console.log(i + "번째 손님 응대 중 입니다..");

}
```

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.
0	TRUE	0 번째 손님 응대 중입니다.
...	TRUE	...

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
```

```
var i = 0;
```

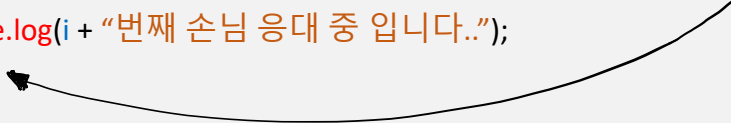
```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

*while* 반복문은 언제나 증감식이 존재해야 한다.



# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
```

```
var i = 0;
```

```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

*while* 반복문은 언제나 증감식이 존재해야 한다.

i	i < 5 ?	출력
---	---------	----

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
```

```
var i = 0;
```

```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

*while* 반복문은 언제나 증감식이 존재해야 한다.

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.



# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

```
// 반복을 제어할 변수
```

```
var i = 0;
```

```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

*while* 반복문은 언제나 증감식이 존재해야 한다.

i	i < 5 ?	출력
0	TRUE	0 번째 손님 응대 중입니다.
1	TRUE	1 번째 손님 응대 중입니다.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
- While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

// 반복을 제어할 변수

```
var i = 0;
```

```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

while 반복문은 언제나 증감식이 존재해야 한다.

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.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

// 반복을 제어할 변수

```
var i = 0;
```

```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

while 반복문은 언제나 증감식이 존재해야 한다.

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.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
- While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

// 반복을 제어할 변수

```
var i = 0;
```

```
while (i < 5) {
```

```
 console.log(i + "번째 손님 응대 중 입니다..");
```

```
 i++;
```

```
}
```

while 반복문은 언제나 증감식이 존재해야 한다.

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.

# 반복문 - while

- for 반복문을 while 반복문으로 표현이 가능함.
  - While 반복문은 자칫 잘못하면 무한반복으로 이어질 수 있다.

// 반복을 제어할 변수

var i = 0;

while ( i < 5 ) {

console.log(i + "번째 손님 응대 중 입니다..");

i++;

}

while 반복문은 언제나 증감식이 존재해야 한다.

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!

# 반복문 - while

---

while를 이용해 1 부터 100 사이 중 3의 배수만 출력해보세요.

while을 이용해 구구단 3단을 출력해 보세요.

# 조건문 - if

# 조건문 - if

- 코드의 진행 상황을 제어할 때 사용함.

```
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.



# 조건문 - if

- 코드의 진행 상황을 제어할 때 사용함.
- 이것 아니면 저것.

```
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.

# 조건문 - if

- 코드의 진행 상황을 제어할 때 사용함.
- 이것 아니면 저것, 그것도 아니라면!

```
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 false이고 조건이 true 일 때 실행한다.

모든 조건이 false 일 때 실행한다.

# 조건문 - if

- 코드의 진행 상황을 제어할 때 사용함.
- 이것 아니면 저것, 그것도 아니라면!

```
var number1 = 30;
var number2 = 20;
var number3 = 50;
if (number1 < number2) {
 console.log(number1 + "이 " + number2 + "보다 작습니다.");
} else if (number1 < number3) {
 console.log(number1 + "이 " + number3 + "보다 작습니다.");
} else {
 console.log(number1 + "이 " + number2 + ", " + number2 + "보다 큼니다.");
}
```

→ 30이 50보다 작습니다.

# 조건문 - if

- 논리 연산자
  - 둘 이상의 조건을 명시할 때 사용한다.

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

# 조건문 - if

---

- 둘 이상의 조건

```
var number1 = 10;
var number2 = 20;
var number3 = 50;

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

# 조건문 - if

- 반복문과 함께 사용하는 if

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 ...

```
if (조건문){
```

수행코드

```
}
```

```
}
```

반복 문장이 실행되기 이전에 체크함.

true 라면 반복문장을 실행하고, false 라면 반복문을 종료함.

보통 반복문의 반복 값을 기준으로 조건문을 작성한다.

# 조건문 - if

---

- 반복문과 함께 사용하는 if

```
for (var i = 0; i < 5; i++) {

 console.log(i + "번째 손님 응대 중입니다.");

 if (i == 3) {
 console.log(i + "번째 손님! 안계신가요?");
 }

}
```

# 조건문 - if

---

```
var highQualityWool = 1; // 고급 양모
var lowQualityWool = 2; // 저급 양모
```

```
var selectedQuality = highQualityWool; // 현재 선택한 양모
var quantity = 30; // 선택한 양모의 개수
```

고급 양모의 가격은 개당 50,000원, 저급 양모의 가격은 10,000원 입니다.  
price 라는 변수를 만들어 가격을 계산해 저장한 후, 아래 포맷에 맞추어 출력해 보세요.

“고급 양모를 30개 선택하셨습니다. 가격은 1500000원 입니다.”



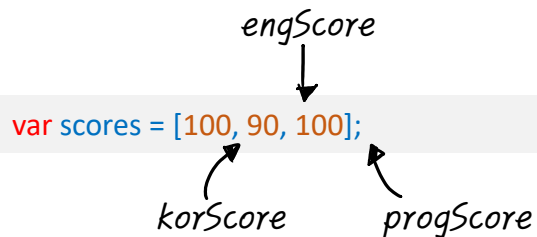
**배열**

# 배열

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

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

- 위 개별 변수를 배열로 만들면?



The diagram illustrates the mapping of individual variables to elements in an array. It shows the code `var scores = [100, 90, 100];` with three arrows pointing from the values 100, 90, and 100 to the labels `korScore`, `engScore`, and `progScore` respectively. The `engScore` label is positioned above the array, while `korScore` and `progScore` are positioned below it.

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

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

→ [100, 90, 100]

# 배열

- 배열 요소의 집합

[100, 90, 100]  
↓   ↓   ↓  
[0] [1] [2]

- 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]);
```

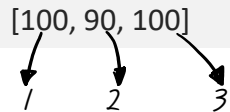
→ undefined

# 배열

- 배열 요소의 개수 확인하기

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

→ 3



- 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

---

- 배열 요소 반복문으로 출력하기

```
for (var i = 0; i < scores.length; i++) {

 console.log(scores[i]);

}
```

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

- for-in 으로 배열 요소 출력하기

```
for (var i in scores) {

 console.log(scores[i]);

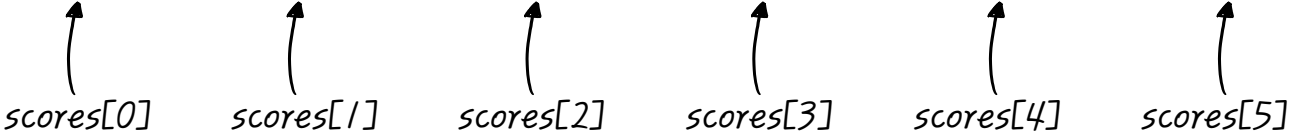
}
```

# 배열

- 배열 요소 반복문으로 출력하기

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

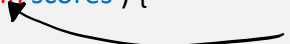
*scores[0]*   *scores[1]*   *scores[2]*   *scores[3]*   *scores[4]*   *scores[5]*

A diagram showing six arrows pointing upwards from the text *scores[0]* through *scores[5]* to the corresponding 'VALUE' cells in the table above. Each arrow starts from the text and points to the center of the value cell.

- for-in
  - 배열의 index를 순차대로 가져옴

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

0, 1, 2, 3, 4, 5 가 순서대로 할당 됨

A diagram showing an arrow pointing from the variable *i* in the `for` loop to the text "0, 1, 2, 3, 4, 5 가 순서대로 할당 됨".

# 배열

---

scores 배열 변수를 만들어, 6개의 숫자를 할당하세요.

totalScore 변수를 만들고 배열에 담긴 6개의 숫자를 모두 더해 할당하세요.

averageScore 변수를 만들고, totalScore와 scores.length 를 이용해  
평균을 구해 출력해 보세요..

# 함수 - 일반 함수



# 함수 - 일반 함수

- 작업 단위를 정하고, 일부를 분리시켜 관리함.

라면 끓이기

1. 물을 넣은 냄비를 준비한다.
2. 가스버너에 냄비를 올리고 불을 켜다.
3. 물이 끓으면, 라면 스프와 면을 넣는다.
4. 4분 더 끓이고, 불을 끈다.
5. 맛있게 먹는다.

```
function function_name() {
```

... 작업 코드 ...

```
}
```

```
function 라면_끓이기() {
 냄비에 물을 넣는다.
 가스버너에 냄비를 올린다.
 가스버너의 불을 켜다.
 if (물이 끓는다.) {
 냄비에 라면스프를 넣는다.
 냄비에 면을 넣는다.
 }
 if (물이 끓은지 4분이 지났다) {
 불을 끈다.
 맛있게 먹는다.
 }
}
```

```
라면_끓이기();
라면_끓이기();
라면_끓이기();
라면_끓이기();
라면_끓이기();
```

# 함수 - 일반 함수

- 함수의 사용 예

```
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 - 일반 함수

- 작업 단위를 정하고, 일부를 분리시켜 관리함.

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

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

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

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

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

분리된 함수를 호출한다.

**함수 – 리턴 함수**

# 함수 - 리턴 함수

- 일반 함수가 결과 값을 반환함.

```
function getCalcNumbers() {
 sayWelcome();
 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 – 매개변수가 있는 함수**

# 함수 – 매개변수가 있는 함수

- 함수에 필요한 데이터를 전달함

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

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



The diagram illustrates the flow of data from the function call to the function definition. Two black arrows originate from the arguments '10' and '60' in the function call 'getCalcNumbers(10, 60)'. One arrow points to the parameter 'numberOne' in the function definition 'function getCalcNumbers( numberOne, numberTwo ) {', and the other points to the parameter 'numberTwo'. A red arrow points from the parameter 'numberOne' to the variable 'result' in the assignment 'var result = numberOne + numberTwo;'.

→ 70

# 함수 – 매개변수가 있는 함수

- 함수에 필요한 데이터를 전달함

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

```
var result = getCalcNumbers(10, 60, 70);
```

파라미터의 개수가 맞지 않더라도,  
함수는 정상적으로 실행된다.

→ 70



# 함수 – 매개변수가 있는 함수

- 함수에 필요한 데이터를 전달함

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

파라미터는 받으나, 전달되지 않았을 경우, *undefined*으로 처리된다.

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

파라미터의 개수가 맞지 않더라도,  
함수는 정상적으로 실행된다.

→ NaN

```
function getCalcNumbers(numberOne, numberTwo) {
 sayWelcome();
 if (numberTwo == undefined) {
 numberTwo = 0;
 }
 var result = numberOne + numberTwo;
 return result;
}
```

*undefined* 로 정의되었는지 확인하고, 그럴 경우 0으로 초기화 시킨다.

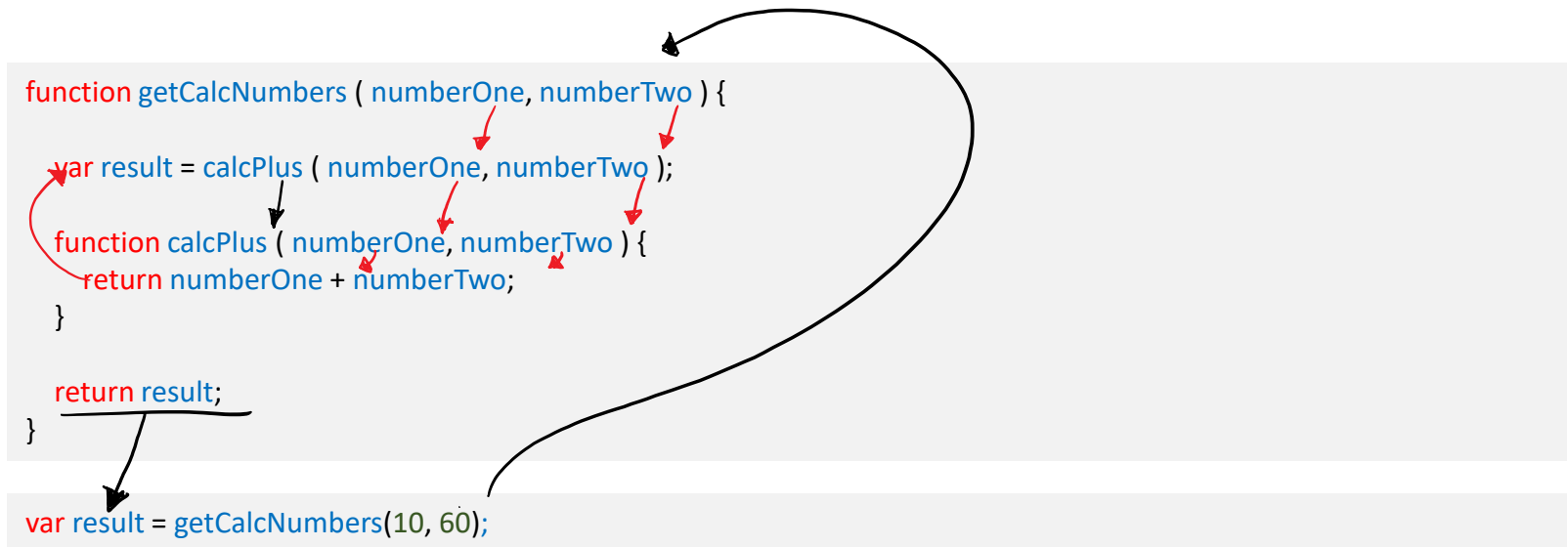
```
var result = getCalcNumbers(10);
```

→ 10

# 함수 - 중첩 함수

# 함수 - 중첩 함수

- 함수안에 함수가 포함 되어있는 형태



→ 70

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

# 함수 - 콜백

# 함수 - 콜백

- 처리 후 작업
  - 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.

```
function getCalcNumbers (callback) {
 sayWelcome();
 var result = callback (10, 60);
 return result;
}

var callbackFunction = function (numberOne, numberTwo) {
 return numberOne + numberTwo;
}

var result = getCalcNumbers (callbackFunction);
```

함수는 변수에 할당할 수 있다.  
이렇게 선언하는 것을 '함수 표현식' 이라 부른다.  
함수표현식으로 선언되는 함수들은  
함수의 이름을 생략할 수 있다. → 익명함수

→ 70

**함수 - 함수를 리턴하는 함수**

# 함수 - 함수를 리턴하는 함수

- 중첩 함수와 유사하지만, 함수를 리턴하는 점이 다름.
  - Private 한 변수나 함수를 가릴 때 사용함.

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

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

함수가 값이 아닌, 함수를 리턴한다.

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

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

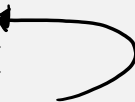
리턴된 함수를 실행함.

→ 90

# 함수 - 함수를 리턴하는 함수

- 중첩 함수와 유사하지만, 함수를 리턴하는 점이 다름.
  - Private 한 변수나 함수를 가릴 때 사용함.

```
function counter () {
 var count = 0;
 return function () {
 return ++count;
 };
}
```



*count 변수에 접근해 변수를 더함.*

```
var count = counter();
var result = count();
```

→ 1

```
result = count();
```

→ 2

```
result = count();
```

→ 3



# 변수의 영역

# 변수의 영역

- 다른 언어에서 변수의 영역

```
function foo () {
 var count = 0;
 if (true) {
 var bar = 10;
 }
}
```

변수 *count*의 영역

변수 *bar*의 영역

- Javascript 에서 변수의 영역
  - Javascript는 function기반의 영역을 사용함.
  - function 내부에서 선언된 변수는 function의 모든 영역에서 사용가능하다.
    - 호이스팅

```
function foo () {
 var count = 0;
 if (true) {
 var bar = 10;
 }
}
```

변수 *count*, *bar*의 영역

# 내장함수 – alert()

# 내장함수 – alert()

---

- 사용자에게 경고창 보여주기

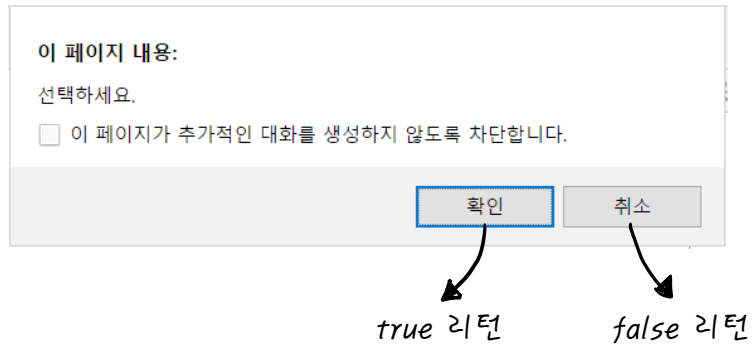
```
alert("알림 메시지");
alert(10);
alert("알림 메시지" + 60);
alert(70 + 60);
alert(foo());
```

**내장함수 – confirm()**

# 내장함수 – confirm()

- 사용자에게 예 / 아니오 선택창 보여주기

```
confirm ("선택하세요.");
```



```
var result = confirm ("선택하세요.");
if (result) {
 // 확인을 클릭했을 때
} else {
 // 취소를 클릭했을 때
}
```

# 객체 - 리터럴

# 객체 - 리터럴

- Javascript에서 가장 기본적인 객체 생성 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

- Javascript에서 가장 기본적인 객체 생성 방법
  - 멤버변수 / 함수를 선언할 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

# 객체 - 클래스

# 객체 - 클래스

- New 키워드를 이용해 객체를 만들어낼 수 있다.

```
function Car () {
 this.carBrand = "Hyundai";
 this.carName = "i40";
}
```

```
var car = new Car();
console.log(car.carBrand);
```

→ Hyundai

```
console.log(car.carName);
```

→ i40

```
console.dir(car);
```

```
▼ Car ⓘ
 carBrand: "Hyundai"
 carName: "i40"
 ▶ __proto__: Object
```

# 객체 - 클래스

- New 키워드를 이용해 객체를 만들어낼 수 있다.

```
function Car (carBrand, carName) {
 this.carBrand = carBrand;
 this.carName = carName;
}
```

```
var car = new Car ("KIA", "K7");
console.log(car.carBrand);
```

→ KIA

```
console.log(car.carName);
```

→ K7

```
console.dir(car);
```

```
⇒ ▼ Car ⓘ
 carBrand: "KIA"
 carName: "K7"
 ▶ __proto__: Object
```

# 객체 - 클래스

- New 키워드를 이용해 객체를 만들어낼 수 있다.

```
function Car (carBrand, carName) {
 this.carBrand = carBrand;
 this.carName = carName;
 this.showMyCar = function () {
 console.log(this.carBrand);
 console.log(this.carName);
 };
}
```

```
var car = new Car ("KIA", "K7");
car.showMyCar();
```

→ KIA

→ K7

# 감사합니다.

---

Javascript

장민창

mcjang@hucloud.co.kr