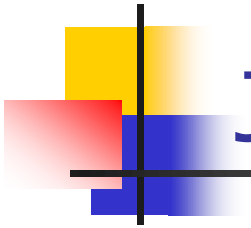
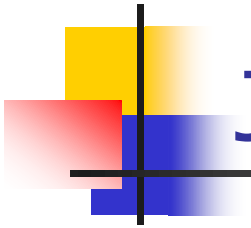
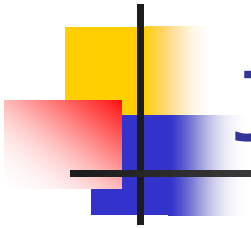


JavaScript 교재



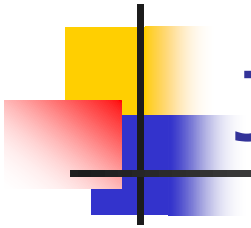
JavaScript

- Overview
  - What is JavaScript
  - Client-side JavaScript
  - Advantages of JavaScript
  - Limitations of JavaScript
  - Where is JavaScript Today?
- Hello World!
  - Internal Scripts
  - External Scripts
- Code structure
  - statements
  - expressions
  - semicolons
  - comments
- Strict mode
  - The “use strict” Directive
  - Declaring Strict Mode
  - Syntax errors



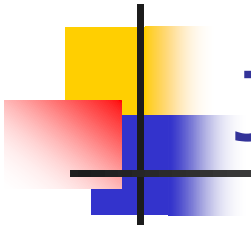
JavaScript

- Data types
  - Definition
  - primitive data types
  - object data type
- Variables
  - Definition
  - Global Variables
  - Local Variables
  - The Scope of a Variable
- Operators
  - Definition
  - Arithmetic Operators
  - Comparison Operators
  - Logical Operators
  - Assignment Operators
  - Bitwise Operators
  - Miscellaneous Operators



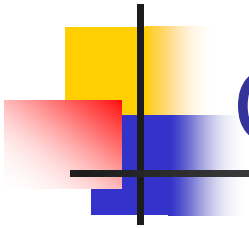
JavaScript

- Control flow and error handling
  - Definition
  - Block statement
  - Conditional statements
  - Loops and iteration
  - Exception handling
- Object
  - Definition
  - Object properties
  - Object methods
  - Object literals
- Functions
  - Definition
  - Function Declaration
  - Function Expression



JavaScript

- Executable Code and Execution Contexts
  - Definition
  - Execution Context
  - Lexical Environment
  - Variable Environment
  - ThisBinding
  - Closures
- Prototype
  - Definition
  - Prototype property and [[prototype]]
  - Prototype chain



Overview

- What is JavaScript?
  - 객체지향 능력을 갖춘 interpreted 프로그래밍 언어
  - 웹 브라우저 내에서 주로 사용하며, 사용자와 상호작용, 동적인 페이지를 생성 할 수 있음
  - 현재 자바스크립트는 브라우저 뿐만 아니라 JavaScript Engine이 탑재된 장치에서 실행 될 수 있음
  - Node.js와 같은 런타임 환경과 같이 server-side 네트워크 프로그래밍에도 사용
  - 초기에 모카(Mocha) > LiveScript, 최종적으로 JavaScript가 되었다.
  - JavaScript는 Java와 직접적인 관련성이 없음
  - JavaScript에 대한 표준은 ECMAScript이다
  - ECMAScript는 언어에 대한 표준 명세

Overview

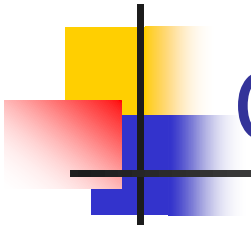
- ECMAScript 역사

판	출판일	이전 판과의 차이점
1	1997 년 6월	초판
2	1998 년 6월	ISO/IEC 16262 국제 표준과 완전히 동일한 규격을 적용하기 위한 변경.
3	1999 년 12 월	강력한 정규 표현식, 향상된 문자열 처리, 새로운 제어문 , try/catch 예외 처리, 엄격한 오류 정의, 수치형 출력의 포매팅 등.
4	버려짐	4번째 판은 언어에 얽힌 정치적 차이로 인해 버려졌다. 이 판을 작업 가운데 일부는 5번째 판을 이루는 기본이 되고 다른 일부는 ECMA스크립트의 기본을 이루고 있다.
5	2009 년 12 월	더 철저한 오류 검사를 제공하고 오류 경향이 있는 구조를 피하는 하부집합인 "strict mode"를 추가한다. 3번째 판의 규격에 있던 수많은 애매한 부분을 명확히 한다.[2]
5.1	2011 년 6월	ECMA스크립트 표준의 제 5.1판은 ISO/IEC 16262:2011 국제 표준 제3판과 함께 한다.
6	2015 년 6월	6판에는 클래스와 모듈 같은 복잡한 응용 프로그램을 작성하기 위한 새로운 문법이 추가되었다. 하지만 이러한 문법의 의미는 5판의 strict mode와 같은 방법으로 정의된다. 이 판은 "ECMAScript Harmony" 혹은 "ES6 Harmony" 등으로 불리기도 한다.
7	2016 년 6월	6판에 이어서 새로운 언어 기능이 추가된 7판을 출판했다.

Overview

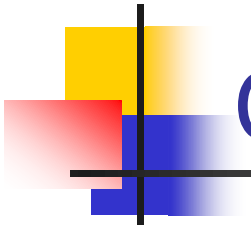
- javascript 역사

버전	출시일	동등한 기술적 내용	넷스케이프 내비게이터	모질라 파이어폭스	인터넷 익스플로러	오페라	사파리	구글 크롬
1.0	1996년 3월		2.0		3.0			
1.1	1996년 8월		3.0					
1.2	1997년 6월		4.0-4.05			3		
1.3	1998년 10월	ECMA-262 1st + 2nd edition	4.06-4.7x		4.0	5[8]		
1.4			넷스케이프 서버			6		
1.5	2000년 11월	ECMA-262 3rd edition	6.0	1.0	5.5 (JScript 5.5), 6 (JScript 5.6), 7 (JScript 5.7), 8 (JScript 5.8)	7.0	3.0-5	1.0-10.0.666
1.6	2005년 11월	1.5 + array extras + array and string generics + E4X		1.5				
1.7	2006년 10월	1.6 + Pythonic generators + iterators + let		2.0				28.0.1500.95
1.8	2008년 6월	1.7 + generator expressions + expression closures		3.0		11.50		
1.8.1		1.8 + native JSON support + minor updates		3.5				
1.8.2	2009년 6월 22일	1.8.1 + minor updates		3.6				
1.8.5	2010년 7월 27일	1.8.2 + new features for ECMA-262 5th edition compliance		4.0				



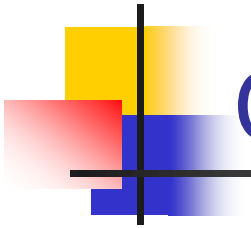
Overview

- Client-side JavaScript
  - 클라이언트 측 자바스크립트는 가장 대표적인 형태
  - 스크립트는 외부 파일로 참조, 또는 HTML 문서 내부에 포함되어질 수도 있음.
  - 사용자가 서버로 요청을 보내기 전 유효성 검사 가능
  - 각종 event 처리 가능



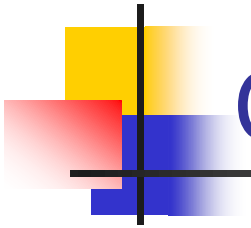
Overview

- Advantages of JavaScript
  - Less server interaction
  - Immediate feedback to the visitors
  - Increased interactivity
  - Richer interfaces



Overview

- Limitations of JavaScript
  - 클라이언트 측 자바스크립트는 파일 읽기/쓰기 불가능, This has been kept for security reason.
  - Cannot used for networking applications because there is no such support available
  - It doesn’t have any multithreading or multiprocessor capabilities
  - 하나의 page에서 또 다른 page를 오픈 할 경우 접근 가능하지만, cross domain 환경에서는 불가능(same origin policy)

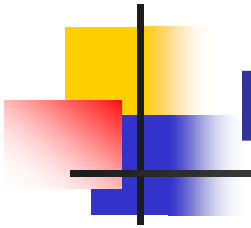


Overview

- Where is JavaScript Today?
  - The ECMAScript 5 standard이 2009년 발표되었다
  - 2016년 6월에 6버전이 발표되었고, 여기는 클래스와 같은 개념들이 포함되었다
  - JavaScript나 Jscript는 ECMAScript를 따르지만 이종 스크립트 간에 비표준화 된 문법들이 존재한다
  - 이종 브라우저 간에 존재하는 비표준 문법들을 잘 파악해야 오류가 없는 크로스 브라우징 가능한 화면을 구현 할 수 있다

Hello World!

- Internal Scripts
  - <script></script> 태그를 사용하여 web page 내부에서 스크립트 사용
  - <script></script> 태그는 문서의 어느 위치에 와도 상관이 없음
  - HTML 문서가 로드 되면서 <script> tag를 만나면 코드를 실행 함



Hello World!

```
<!DOCTYPE HTML>
<html>
<head>
</head>
<body>
        <p>before running script</p>
        <script>
                alert("Hello World!");
        </script>
        <p>after running script</p>
</body>
</html>
```

Hello World!

- External Scripts
  - JavaScript code량이 많다면 js 파일로 분할 할 수 있다
  - Web page외부에 확장자를 js로 갖는 스크립트 파일 생성
  - <script></script> 태그의 src 속성에 해당 js 파일의 경로를 명시
  - <script src="/js/somejsfile.js"></script>
  - src 속성 값이 들어 있는 <script> tag 사이에 JavaScript code가 있어도 이는 무시된다
  - use multiple tags

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
```

Code structure

- statements
  - statements란 브라우저에 의해 실행되게 될 명령들
  - JavaScript statements are composed of: Values, Operators, Expressions, Keywords, and Comments.
  - statements는 semi-colon에 의해서 분리

```
statement1; statement2;
```
bad case

```
statement1;
statement2;
```
good case

Code structure

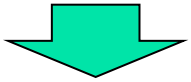
- expressions
  - An expression is any valid set of literals, variables, operators, and expressions that evaluates to a single value.
  - 3가지 종류의 표현식
    - Arithmetic
    - String
    - Logical

Arithmetic	String	Logical
x=7	"my " + "string"	expr1||expr2, expr1&&expr2
3+4	'my' + 'string'	expr1==expr2, expr1>expr2
i++, ++i	mystring += "bet"	

Code structure

- semicolons
  - statement를 분리 하기 위해 사용
  - statement들 간에 line break되어 있으면 semicolon은 생략 될 수 있다

```
sum = sum + 1
sum = sum + 1
```



implicit semicolon or automatic semicolon

```
sum = sum + 1;
sum = sum + 1;
```

  - 모든 case에서 new line이 semicolon을 의미하지 않는다

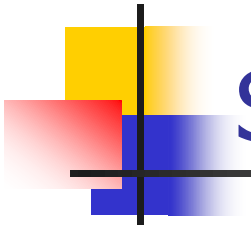
```
alert(3+
1                    결과 -> 6
+2);
```

Code structure

- Comments
  - comments는 스크립트의 어느 위치에도 추가 될 수 있음
  - 자바스크립트 엔진에 의해 실행되지 않는다
  - //, /**/

```
// one-line comments
sum = sum + 1; // 주석
sum = sum + 1;
```

```
/*
multiline comments
*/
sum = sum + 1;
sum = sum + 1;
```



Strict mode

- The “use strict” Directive
  - “use strict” 지시자는 JavaScript1.8.5에 새롭게 등장(ECMAScript version 5)
  - It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.
  - 스크립트 코드가 strict mode에서 실행 되도록 제약

Strict mode

- Declaring Strict Mode
  - 스크립트, 함수의 시작 전에 선언하거나 함수 내부에 선언 될 수 있음

```
"use strict"
x = 3.14;
```

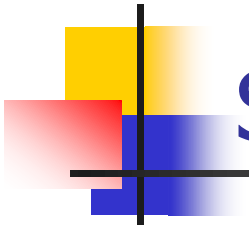
스크립트 시작 시 선언

```
"use strict"
sum(1,2);
function sum(x, y){
  return result = x+y;
}
```

함수 시작 시 선언

```
x = 3.14;
sum(1,2);
function sum(x, y){
"use strict"
  return result = x+y;
}
```

함수 내부에 선언



Strict mode

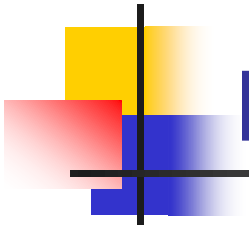
- Syntax errors
  - Octal syntax var n = 023;
  - with statement
  - Using delete on a variable name delete myVariable;
  - Using eval or arguments as variable or function argument name
  - Using one of the newly reserved keywords (in prevision for ECMAScript 2015): implements, interface, let, package, private, protected, public, static, and yield
  - Declaring function in blocks if (a < b) { function f() {} }
  - Obvious errors
    - Declaring twice the same name for a property name in an object literal {a: 1, b: 3, a: 7} This is no longer the case in ECMAScript 2015 (bug 1041128).
    - Declaring two function parameters with the same name function f(a, b, b) {}

Strict mode

```
<!doctype html>
<html>
<head>
<script>
"use strict";
var n = 024;
function foo(){
  var obj = {data:{inner:"inner"}};
  with(obj.data){
    console.log(inner);
  }
}
</script>
</head>
<body>
<input id="name" type=text value="asdf">
</body>
</html>
```

Strict mode

```
"use strict";

//쓸 수 없는 프로퍼티에 할당
var undefined = 5; // TypeError 발생
var Infinity = 5; // TypeError 발생

//쓸 수 없는 프로퍼티에 할당
var obj1 = {};
Object.defineProperty(obj1, "x", { value: 42, writable: false });
obj1.x = 9; // TypeError 발생

//getter-only 프로퍼티에 할당
var obj2 = { get x() { return 17; } };
obj2.x = 5; // TypeError 발생

//확장 불가 객체에 새 프로퍼티 할당
var fixed = {};
Object.preventExtensions(fixed);
fixed.newProp = "ohai"; // TypeError 발생
```



Data types

- Definition
  - 자바스크립트 변수(variable)은 데이터(value)를 담을 수 있다
  - 데이터는 형(data type)이 존재한다
  - 자바스크립트에는 5가지 primitive data type과 object data type이 있다
  - primitive data type과 reference type으로 나누기도 한다
  - 자바스크립트는 느슨한 타입 언어, 혹은 동적 언어이다. 변수의 타입을 미리 선언할 필요가 없음
  - 타입은 프로그램이 처리되는 과정에서 자동으로 파악, 즉 같은 변수에 여러 타입의 값을 넣을 수 있음

Data types

- primitive data types
  - 변경 불가능한 값(immutable value)
  - number, string, boolean, null, undefined

- A number
  - 정수, 부동 소수점(실수) 모두 number type으로 처리된다
  - 정수 만을 표현하기 위한 type은 없음
  - special number value : Infinity, -Infinity, NaN

	의미	
Infinity	+무한대	alert(1/0)
-Infinity	-무한대	alert(-Infinity)
NaN	computational error	alert("abcd"/3);

special number value

Data types

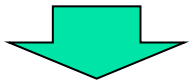
- A string
  - 문자열은 따옴표로 묶어져 있음
  - double quotes, single quotes 동일하게 인식 됨
  - In JavaScript, there is no character type. There's only one type: string. A string may consist of only one character or many of them.
  - 특수 문자를 나타내기 위해 escape character(\, backslash)를 사용함

```
var str = "Hello";
var str2 = 'Single quotes are ok too';

// ES6부터 backstick 기능 제공
var phrase1 = `can embed ${str}`; // embed a variable
alert( `the result is ${1 + 2}` ); // embed an expression
var phrase2 = “can embed ${str}”; // do nothing
```

Data types

- escape character
  - 문자열은 따옴표로 묶여져 있기 때문에 아래의 문장은 잘못 해석 될 수 있다
  - 문자열에 특수 문자를 입력 할 때 escape character를 사용
  - \uFFFF 형식으로 unicode 문자를 모두 표현 가능

```
var y = "We are the so-called "Vikings" from the north.“;
```



backslash(escape character) 사용

```
var y = "We are the so-called \"Vikings\" from the north.";
```

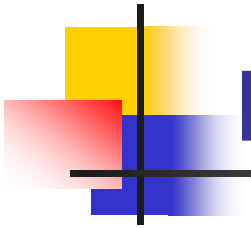
Code	Outputs
\'	single quote
\"	double quote
\\	backslash

\b	Backspace
\r	Carriage Return
\f	Form Feed
\t	Horizontal Tabulator
\v	Vertical Tabulator

Data types

- A boolean
  - 논리 타입
  - true 또는 false

```
(function(){
  var isNameChecked = true;
  var isAgeChecked = false;
  console.log(isNameChecked);
  console.log(isAgeChecked);
  var x=1;
  var y=2;
  console.log(x>y);
  console.log(4>1);
  if ( isNamChecked ){
    console.log("isNamChecked");
  }
})();
```



Data types

- The “null” value
  - 이 특별한 null value는 위에서 언급한 어느 type에도 속하지 않는다
  - 다른 언어들처럼 null pointer
  - nothing, empty, unknown의 의미를 가짐

- The “undefined” value
  - 값이 할당 되지 않았음을 의미함
  - 변수를 선언하고 값을 할당하지 않으면 undefined
  - 함수 실행 시 함수의 return문이 없을 경우 undefined (단, new 생성자 함수() 형태로 호출될 경우에는 제외)

Data types

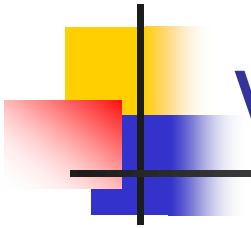
```
(function(){
  var obj = null;
  console.log(obj);
  var age;
  console.log(age);
  var result1 = (function(x, y){
    return x+y;
  })(1,2);
  console.log(result1);
  var result2 = (function(x, y){
    var result = x+y;
  })(1,2);
  console.log(result2);
})();
```

null and undefined 사용 예

Data types

- object data type
  - primitive data type은 하나의 데이터만을 저장
  - object data type은 data collection을 저장 할 수 있음
  - 자바스크립트의 모든 객체는 최상위 Object를 상속함
  - object는 Objects chapter에서 상세하게 다룸

```
(function(){
 // object literal
  var objCase1 = {name:"홍길동", age:15};
 //Object 객체를 통해서 객체 생성
  var objCase2 = new Object();
 // Object.create 함수를 통해서 객체 생성
  var objCase3 = Object.create(Object.prototype);
})();
```



Variables

- Definition
  - variables(변수)는 데이터를 저장 할 수 있는 공간에 이름을 붙인 것과 같다
  - 실제로 메모리는 address로 접근 하는데, 자바스크립트에서 물리적 메모리에 접근 하는 것은 불가능
  - 변수에 데이터를 저장
  - var 키워드를 사용하여 변수 선언
  - 변수를 선언하고 값을 할당하지 않으면(초기화 하지 않으면) undefined 값을 가짐
  - variables are Function-Scoped

Variables

- Global Variables
  - 전역 변수는 스크립트 상의 어디에서나 접근 할 수 있다
- Local Variables
  - 자바스크립트에서 지역 변수는 function 내부에서 선언된 변수
  - 해당 함수 내에서만 접근 가능
  - function에 전달된 parameter도 지역변수로 간주

```
var score = 100; // global
function function check(){
  var score = 200; // local
  console.log(score); // 200; ← shadowing
}
```

Variables

- The scope of a variable
  - 변수의 유효 범위
  - 변수를 접근 할 수 있는 위치
  - 자바스크립트는 변수의 유효 범위는 function-scoped

```
function foo(){
  var x;
}
```

x의 유효 범위는 foo 함수 내부

```
function foo(){
  if ( true ){
    var x=1;
  }
  console.log(x);
}
```

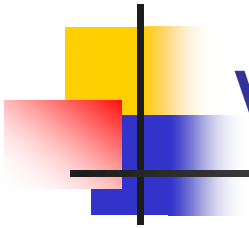
Javascript의 변수 유효범위는 function-scoped 이므로, x의 유효 범위는 foo 함수 내부

Variables

- Lexical scoping
  - 자바스크립트에서 변수는 어휘적으로 유효범위가 결정된다
  - 프로그램의 정적 구조에 의해서 변수의 범위가 결정

```
bar();
function foo(){
  var x = 1;
  console.log(y);
}

function bar(){
  var y = 1;
  foo();
  console.log(x);
}
```

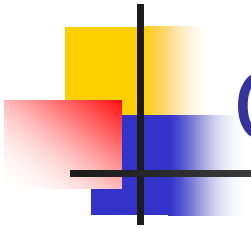
foo 함수와 bar 함수는 구조적으로 중첩되지 않았기 때문에 각각 자신의 로컬 변수만 접근 가능



Variables

- Nested scopes
  - 중첩된 함수 구조
  - 구조적으로 중첩된 함수 구조에서 외부 함수의 변수를 내부함수에서 접근 가능

```
foo(1);
function foo(arg){
  function bar(){
    console.log(arg);
  }
  bar();
}
```



Operators

- Definition
  - 표현식 4+3에서 4,3은 operands(피연산자), '+'는 operator(연산자)라 부름
  - 자바스크립트에서 지원하는 연산자
    - Arithmetic Operators
    - Comparison Operators
    - Logical Operators
    - Assignment Operators
    - Bitwise Operators
    - Miscellaneous Operators
  - 연산자는 우선순위와 결합방향이 존재함

Operators

- Arithmetic Operators
  - 산술 연산자, 산술 연산을 수행
  - Given that **y = 5**, the table below explains the arithmetic operators:

Operator	Description	Example	Result in y	Result in x
+	Addition	x = y + 2	y = 5	x = 7
-	Subtraction	x = y - 2	y = 5	x = 3
*	Multiplication	x = y * 2	y = 5	x = 10
/	Division	x = y / 2	y = 5	x = 2.5
%	Modulus (division remainder)	x = y % 2	y = 5	x = 1
++	Increment	x = ++y	y = 6	x = 6
		x = y++	y = 6	x = 5
--	Decrement	x = --y	y = 4	x = 4
		x = y--	y = 4	x = 5

Arithmetic operators

Operators

- Comparison Operators
  - 비교연산자는 논리식에서 사용 된다
  - 두 값을 비교하여 true/false 값을 반환
  - Given that **x = 5**, the table below explains the comparison operators:

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
===	equal value and equal type	x === "5"	false
		x === 5	true
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== "5"	true
		x !== 5	false
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

Comparison operators

Operators

- Logical Operators
  - 논리 연산자는 주어진 논리식을 판단하여, true/false를 반환
  - Given that **x = 6 and y = 3**, the table below explains the logical operators

Operator	Description	Example
&&	and	(x < 10 && y > 1) is true
||	or	(x === 5 || y === 5) is false
!	not	!(x === y) is true

Logical operators

Operators

- Assignment Operators
  - 할당 연산자는 값(value)을 변수(variable)에 할당하는데 사용
  - Given that **x = 10** and **y = 5**, the table below explains the assignment operators:

Operator	Example	Same As	Result in x
=	x = y	x = y	x = 5
+=	x += y	x = x + y	x = 15
-=	x -= y	x = x - y	x = 5
*=	x *= y	x = x * y	x = 50
/=	x /= y	x = x / y	x = 2
%=	x %= y	x = x % y	x = 0

Assignment operators

Operators

- Bitwise Operators
  - 비트연산 수행
  - 비트연산을 수행하기 전에 피연산자를 32 bit number로 변환되고 결과는 JavaScript number로 다시 변환
  - JavaScript는 32bit signed numbers를 사용하므로 비트연산 시 +,- 부호에 신경 써야 함 (다른 언어도 마찬가지)

Operator	Description	Example	Same as	Result	Decimal
&	AND	x = 5 & 1	0101 & 0001	0001	1
|	OR	x = 5 | 1	0101 | 0001	0101	5
~	NOT	x = ~ 5	~0101	1010	10
^	XOR	x = 5 ^ 1	0101 ^ 0001	0100	4
<<	Left shift	x = 5 << 1	0101 << 1	1010	10
>>	Right shift	x = 5 >> 1	0101 >> 1	0010	2

Bitwise operators

Operators

- Miscellaneous Operators
  - ternary, typeof, delete, in, instanceof, void, comma(,)

- Ternary Operator
  - 3항 연산자, 또는 조건 연산자
  - 조건에 따라서 식을 평가하거나, 값을 반환

Syntax
condition ? expr1 : expr2

```
(function(){
  var age = 17;
  var isAdult = false;
  isAdult = age>=18?true:false;
  console.log(isAdult);
  age>=18?console.log("adult"):console.log("boy");
})();
```

Operators

- typeof Operator
  - typeof 연산자는 피연산자의 타입을 나타내는 문자열을 반환

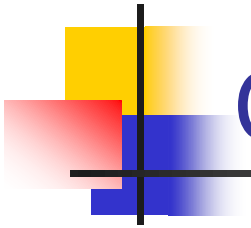
Syntax
typeof 피연산자

  - 반환 가능한 타입 값

타입	결과
undefined	"undefined"
null	"object"
boolean	"boolean"
number	"number"
string	"string"
function object	"function"
any other object	"object"

Operators

```
(function(){
  typeof 37 === 'number';
  typeof 3.14 === 'number';
  typeof Math.LN2 === 'number';
  typeof Infinity === 'number';
  typeof NaN === 'number'; // Despite being "Not-A-Number"

  // Strings
  typeof "" === 'string';
  typeof "bla" === 'string';
  typeof (typeof 1) === 'string'; // typeof always returns a string

  // Booleans
  typeof true === 'boolean';
  typeof false === 'boolean';

  // Undefined
  typeof undefined === 'undefined';
  typeof declaredButUndefinedVariable === 'undefined';
  typeof undeclaredVariable === 'undefined';

  // Objects
  typeof {a:1} === 'object';

  // use Array.isArray or Object.prototype.toString.call
  // to differentiate regular objects from arrays
  typeof [1, 2, 4] === 'object';
  typeof new Date() === 'object';

  // The following is confusing. Don't use!
  typeof new Boolean(true) === 'object';
  typeof new Number(1) === 'object';
  typeof new String("abc") === 'object';

  // Functions
  typeof function(){} === 'function';
});
```



Operators

- instanceof Operator
  - instanceof 연산자는 객체가 자신의 prototype chain에 contructor의 prototype property를 가지는 지 여부를 평가하여 결과 반환
  - instanceof는 prototype과 관련이 있으므로, prototype 관련된 내용을 참고

Syntax
object **instanceof** constructor

Operators

```
(function(){
// defining constructors
function C() {}
function D() {}

var o = new C();

// true, because: Object.getPrototypeOf(o) === C.prototype
console.log(o instanceof C);

// false, because D.prototype is nowhere in o's prototype chain
console.log(o instanceof D);

console.log(o instanceof Object); // true, because:
console.log(C.prototype instanceof Object); // true

C.prototype = {};
var o2 = new C();

console.log(o2 instanceof C); // true

// false, because C.prototype is nowhere in
// o's prototype chain anymore
console.log(o instanceof C);

D.prototype = new C(); // add C to [[Prototype]] linkage of D
var o3 = new D();
console.log(o3 instanceof D); // true
console.log(o3 instanceof C); // true since C.prototype is now in o3's
prototype chain
})();
```

Operators

- delete Operator
  - delete 연산자는 객체로부터 property를 제거 한다
  - 성공하면 ture, non-configurable property일 경우 false
  - delete 연산자는 메모리를 해제 하는 것과 직접적인 관련이 없음
  - delete 연산자는 사용을 권고하지 않는다

Syntax
delete expression

Example
delete object.property **delete object[`property’]**

Operators

- in Operator
  - in 연산자는 specified property 또는 property list가 object에 있으면 true, 그렇지 않으면 false
  - for 문에서 in 연산자를 사용 할 수 있다

Syntax
prop **in** *objectName* (*prop1, prop2, ...*) **in** *objectName*

```
(function(){
  var obj = {name:"홍길동",
             age:500,
             grade:"A"};
  var arr = [1,2,3];
  console.log('name' in obj);
  console.log('age' in obj);
  console.log(0 in arr);
  console.log(1 in arr);
})();
```

Operators

- void Operator
  - void 연산자는 주어진 표현식을 평가하고 undefined 반환
  - 함수의 선언을 표현식으로 취급되도록 할 수 있음, IIFE
  - javascript:로 시작하는 URI를 지원하는 브라우저에서는, URI에 있는 코드의 평가 결과가 undefined가 아니라면 페이지의 콘텐츠를 반환 값으로 대체합니다. void 연산자를 사용하면 undefined를 반환할 수 있습니다.

Syntax
void *expression*

```
(function(){
  void function sayHello(){
    var n = 1;
    console.log(n);
  }();
})();
```

Operators

```
<a href="javascript:void(0);">
  아무일도 일어나지 않음
</a>
<a href="javascript:void(document.body.style.backgroundColor='green');">
  녹색으로 변경되고 페이지 content는 그대로
</a>
```

Operators

- comma Operator
  - comma 연산자는 피연산자를 왼쪽에서 오른쪽으로 평가하고 마지막 피연산자의 값을 반환

Syntax
expr1, expr2, expr3...

```
(function(){
  var x, y, z;
  var a=b=c=3,d=4;
  x=(y=5,z=6);

  console.log(myFunc());
  function myFunc(){
    var x=0;
    return (x+=1, x);
  }
})();
```

Control flow and error handling

- Definition
  - JavaScript 제어문
  - 프로그램의 흐름 제어 및 에러 처리
  - JavaScript에서 흐름 제어를 위해 statements를 제공
    - 조건문
    - 반복문
    - try catch 문
  - 에러가 발생 했을 경우 흐름 제어는 error handling을 통해서 가능

Control flow and error handling

- Block statement
  - 블록 구문은 0개 이상의 statement를 그룹으로 묶는데 사용
  - 블록은 중괄호 쌍으로 구성
  - 제어문과 함께 자주 사용

Syntax
{ *statement_1;* *statement_2;* *...* *statement_N;* *}*

Control flow and error handling

- Conditional statements
  - 조건문
  - 주어진 조건이 참일 경우 or 거짓을 경우의 흐름 제어
  - if else, switch case

- if ... else statement
  - if문은 조건이 true인 경우에 문장을 실행
  - else는 조건이 false일 때

Syntax

if(condition){
  statement_1;
}else{
  statement_2;
}

Control flow and error handling

- else if문를 사용하여 복수 조건을 테스트 할 수 있다

Syntax

```
if(condition_1){
  statement_1;
}else if(condition_2){
  statement_2;
}else if(condition_3){
  statement_3;
}else{
  statement_4;
}
```

Control flow and error handling

- falsy values
  - 조건식에는 꼭 논리식만 올 수 있는 것 아님
  - 조건식에서 false로 평가되는 값들
  - 이외의 값들은 true로 평가

value	description
false	primitive false
undefined	primitive undefined
null	primitive null
0	primitive number
NaN	special number value
the empty string(“”)	공백문자

Control flow and error handling

```
(function(){
  var x=1,y=true,z=false;
  var undefinedVar,nullVar = null;

  if(x){
    console.log('1 => true');
  }
  if(y){
    console.log('true => true');
  }
  if(z){
    console.log('false => true');
  }
  if(undefinedVar){
    console.log('undefinedVar => true');
  }
  if(nullVar){
    console.log('nullVar => true');
  }
})();
```

Control flow and error handling

- switch statement
  - switch문은 표현식의 값과 일치하는 case문을 찾아 구문 실행
  - case문과 일치하는 값이 없으면 default 문 실행
  - default 문은 선택

Syntax

```
switch(expression){
  case value1:
    statements_1;
    [break;]
 case value2:
    statements_2;
    [break;]
 default:
    //statements_default
    [break;]
}
```

Control flow and error handling

- Loops and iteration
  - Loop는 어떤 것을 반복적으로 실행 해야 할 때 빠르고 간편한 방법을 제공
  - 다양한 종류의 반복문 제공
- for statement
  - 조건이 false로 판별될 때 까지 반복 수행

Syntax
for ([초기문]; [조건문]; [증감문]) statement

실행순서
step1. 초기화 구문실행 step2. 조건문 평가, true이면 statement 실행, false이면 for문 종료 step3. 증감문 실행 후 step2에서 3반복

Control flow and error handling

- do ... while statement
  - 표현식이 false로 판별될 때 까지 반복 수행
  - 최소 한번은 무조건 수행

Syntax
do   statement; **while**(*expression*);

- while statement
  - 표현식이 참이면 반복 수행

Syntax
while(*expression*) statement;

Control flow and error handling

```
(function(){
  var i,j;
  for(i=0,j=0; i>5; i++){
    console.log(i);
  }

  var k = 0;
  do{
    k+=1;
    console.log(k);
  }while(k<5);

  var l=0;
  var lSum=0;
  while(l<=100){
    l+=1;
    lSum+=l;
  }
})();
```

Control flow and error handling

- label statement
  - 레이블 구문은 break나 continue 구문과 함께 사용
  - 원하는 식별자를 구문 앞에 입력
  - 자바스크립트에서 사용할 수 있는 식별자 가능

Syntax
label: statement

예제

markLoop:
while(theMark==true){
  doSomething();
}

Control flow and error handling

- break statement
  - break문은 반복문, switch문을 빠져 나올 때 사용
  - label과 함께 사용하여 특정 구문을 빠져 나올 수 있음

Syntax
break; break 레이블;

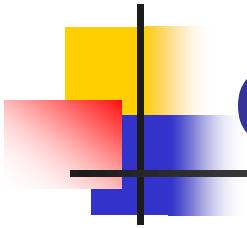
Control flow and error handling

break 단독

```
(function(){
  var i=3;
  var a = [1,2,3];
  for (i = 0; i < a.length; i++) {
    if (a[i] == theValue) {
      break;
    }
  }
})();
```

label과 같이 사용

```
(function(){
var x = 0;
var z = 0
labelCancelLoops: while (true) {
  console.log("Outer loops: " + x);
  x += 1;
  z = 1;
  while (true) {
    console.log("Inner loops: " + z);
    z += 1;
    if (z === 10 && x === 10) {
      break labelCancelLoops;
    } else if (z === 10) {
      break;
    }
  }
}
})();
```



Control flow and error handling

- continue statement
  - while, do-while, for 문과 함께 사용하여 현재 반복구문을 종료하고 반복문의 시작점으로 진입
  - 레이블과 함께 사용하여 특정 구문으로 이동

Syntax
continue; continue 레이블;

Control flow and error handling

continue 단독

```
(function(){
  var i = n = 0;
  while (i < 5) {
    i++;
    if (i == 3) {
      continue;
    }
    n += i;
  }
})();
```

label과 같이 사용

```
(function(){
checkiandj:
  while (i < 4) {
    console.log(i);
    i += 1;
    checkj:
      while (j > 4) {
        console.log(j);
        j -= 1;
        if ((j % 2) == 0) {
          continue checkj;
        }
        console.log(j + " is odd.");
      }
      console.log("i = " + i);
      console.log("j = " + j);
  }
})();
```

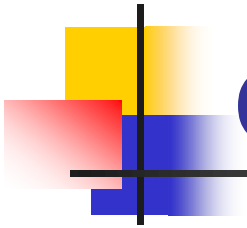
Control flow and error handling

- for ... in statement
  - 객체의 열거 속성에 대해 지정된 변수를 반복

Syntax

for (variable **in** object){
  statements
}

```
(function(){
  var person = {
        firstName:"John",
        lastName:"Doe",
        age:50,
        eyeColor:"blue"
  };
  var eachVar;
  for(eachVar in person){
    console.log(eachVar+" "+person[eachVar]);
  }
})();
```



Control flow and error handling

- Exception handling
  - throw문을 이용하여 던지고(발생시킴), try catch문을 이용하여 예외를 처리 할 수 있음
  - throw, try catch statement

- Exception types
  - 자바스크립트에서 발생하는 예외의 종류
    - ECMAScript Exceptions
    - DOMException and DOMError

Control flow and error handling

- throw statement
  - 예외를 던질 때 사용

Syntax
throw expression;

- try catch statement
  - 예외를 처리 하기 위해 사용

Syntax

try{
  statements**;**
}**catch**(object){
  statements;
}**finally**{
 //자원 해제
};

Control flow and error handling

- 함수 외부에서 예외 처리

```
(function(){
  try{
    throwException();
  }catch(e){
    console.log(e);
  }
  function throwException(){
    throw "Exception";
  }
})();
```

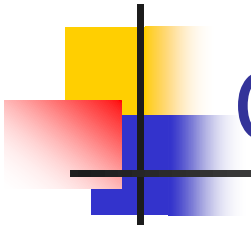
- 함수 내에서 예외 처리

```
(function(){
  throwException();
  function throwException(){
    try{
      throw "Exception";
    }catch(e){
      console.log(e);
    }
  }
})();
```

Control flow and error handling

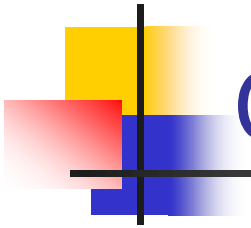
ReferenceError	TypeError	URIError

(function(){
  try{
  method();
  }catch(e){
  console.log(e);
  }
})();
	(function(){
	"use strict"
	  try{
	  var obj=1;
	  obj.name = "홍길동";
	  }catch(e){
	  console.log(e);
	  }
	})();
		(function(){
		  try{
		  decodeURIComponent('/login.do');
		  decodeURIComponent('%');
		  }catch (e){
		  console.log(e);
		  }
		})();



Object

- Definition
  - Object based language, Object Oriented language
  - JavaScript는 객체기반 언어이며 거의 모든 것들이 객체로 되어 있음
  - 객체는 속성과 행위를 갖는다
  - 객체는 attribute들로 구성됨(method, property)
  - 객체의 attribute 중 method는 함수 포인터를 갖는 property이고, 그 외에는 property
  - 모든 객체의 조상 Object



Object

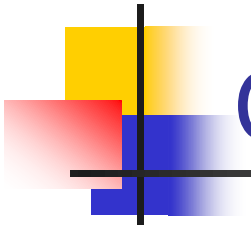
- Object properties
  - 객체의 일부로 이름과 값 사이 연결
  - 객체의 속성을 나타내는 접근 가능한 이름과 활용 가능한 값을 가지는 특별한 형태
  - primitive data type, object type 모두 property의 값으로 허용
  - 새로운 property가 필요 할 경우 동적으로 선언 가능
- Object methods
  - 객체가 가지고 있는 동작
  - 함수 포인터를 참조하는 객체의 attribute(property)
  - 함수는 반복사용 가능한 문장의 독립단위, method는 property의 값인 함수

Object

- Object literals
  - Object를 생성하는 3가지 방법
    - Object의 생성자 함수를 이용한 방법
    - {} 중괄호를 이용한 방법
    - Object의 Create 함수를 이용한 방법 (IE9 이상) – 상속 편에서 설명

literal	syntax
Object 생성자 함수	var obj = **new** Object();
{}	var obj = {};
Object.create 함수	var obj = Object.create(o);

Object literals



Object

```
(function(){
  var carObj = new Object();
  carObj.wheels = 4;
  carObj.name = 'BMW520';
  if ( carObj.wheels === 4 ){
    console.log("4륜 자동차");
  }
})();
```

생성자 함수이용

```
(function(){
  var carObj = {};
  carObj.wheels = 4;
  carObj.name = 'BMW520';
  if ( carObj.wheels === 4 ){
    console.log("4륜 자동차");
  }
})();
```

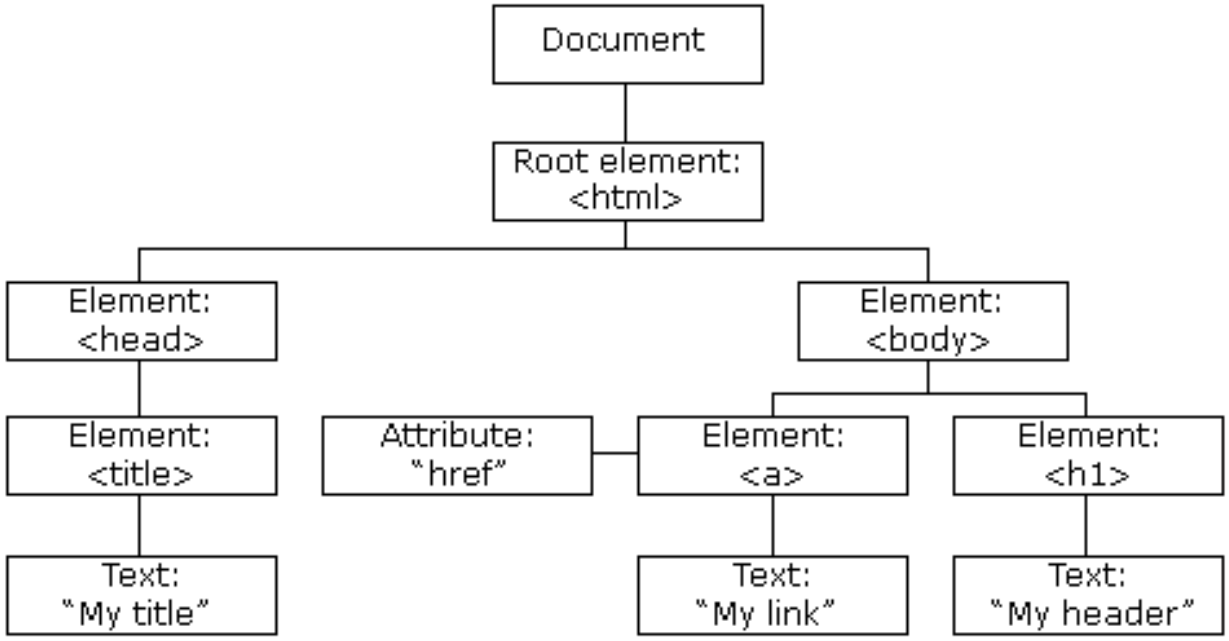
{} 이용

DOM(Document Object Model)

- Definition
  - 문서 객체 모델은 객체지향 모델로써 구조화된 문서를 표현하는 형식
  - 플랫폼 언어 중립적으로 구조화된 문서를 표현하는 W3C의 공식 표준, 프로그래밍 언어가 DOM구조에 접근할 수 있는 방법을 제공하여 문서 구조, 스타일, 내용 등을 변경 가능
  - 브라우저간의 DOM 구현의 차이 때문에 상호 운용성 문제
  - 비표준 확장 기능을 제공하는 IE와 표준을 준수하는 Mozila

DOM(Document Object Model)

- DOM Tree
  - When a web page is loaded, the browser creates a Document Object Model of the page.
  - The **HTML DOM** model is constructed as a tree of **Objects**:



DOM(Document Object Model)

- Nodes
  - DOM 트리에 존재하는 모든 아이템은 Nodes로 정의된다.
  - 가장 많이 사용되는 Node의 종류
    - Element nodes
    - Text nodes
    - Comment nodes

Node type	Value	Example
DOCUMENT_NODE	9	The HTML document itself (the parent of <html>)
ELEMENT_NODE	1	The <body> element
TEXT_NODE	3	Text that is not part of an element
COMMENT_NODE	8	<!- - an HTML comment - ->

DOM(Document Object Model)

- What is the HTML DOM?
  - The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:
    - The HTML elements as objects
    - The **properties** of all HTML elements
    - The **methods** to access all HTML elements
    - The **events** for all HTML elements
  - In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

DOM(Document Object Model)

- How to access Elements in the DOM
  - DOM 트리는 Nodes들로 구성되어 있다.
  - document 객체에 접근하는 것으로부터 모든 nodes에 접근 할 수 있다.
  - DOM의 요소에 접근하려면 CSS selector에 대한 이해가 필요하다
  - document 객체의 DOM 요소 접근 함수

Gets	Selector syntax	Example
ID	#demo	getElementById
Class	.demo	getElementsByClassName
Tag	demo	getElementsByTagName
Selector(single)		querySelector
Selector(all)		querySelectorAll

DOM(Document Object Model)

```
<!DOCTYPE html>
<html lang="ko">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">

  <title>Accessing Elements in the DOM</title>

  <style>
    html { font-family: sans-serif; color: #333; }
    body { max-width: 500px; margin: 0 auto; padding:
0 15px; }
    div, article { padding: 10px; margin: 5px; border:
1px solid #dedede; }
  </style>

</head>
```

```
<body>

  <h1>Accessing Elements in the DOM</h1>

  <h2>ID (#demo)</h2>
  <div id="demo">Access me by ID</div>

  <h2>Class (.demo)</h2>
  <div class="demo">Access me by class (1)</div>
  <div class="demo">Access me by class (2)</div>

  <h2>Tag (article)</h2>
  <article>Access me by tag (1)</article>
  <article>Access me by tag (2)</article>

  <h2>Query Selector</h2>
  <div id="demo-query">Access me by query</div>

  <h2>Query Selector All</h2>
  <div class="demo-query-all">Access me by query all
(1)</div>
  <div class="demo-query-all">Access me by query all
(2)</div>

</body>

</html>
```

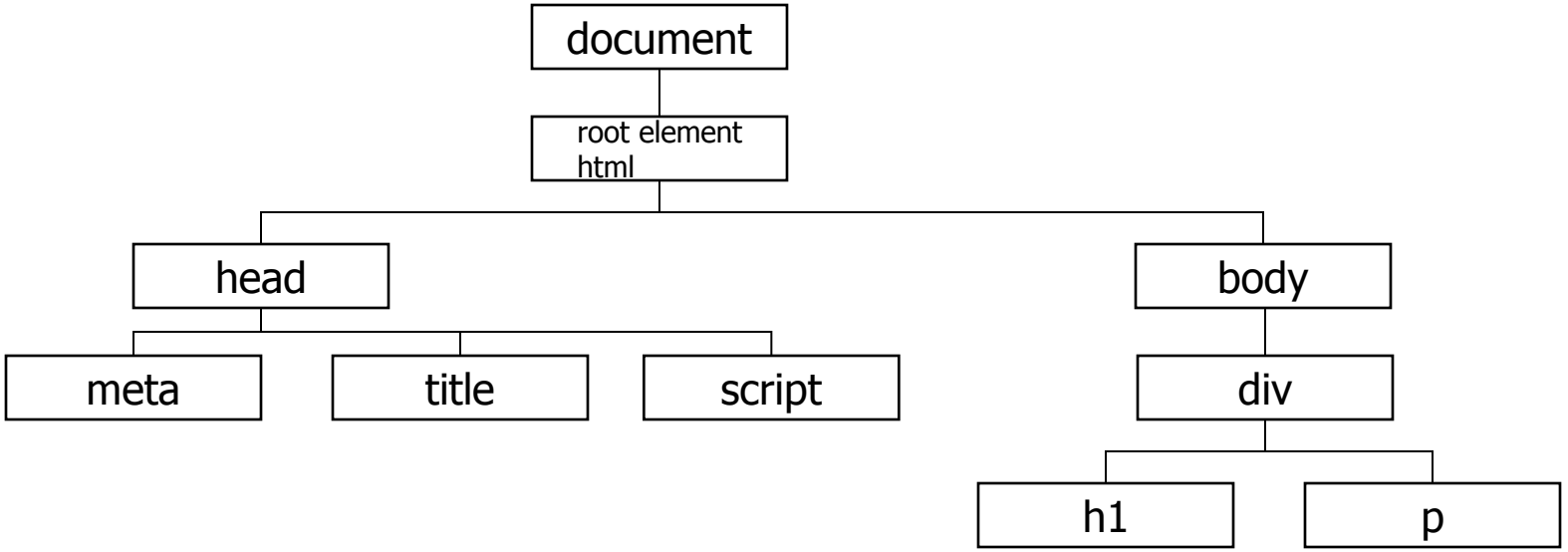
DOM(Document Object Model)

- DOM Navigation(DOM Traversing)

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>DOM Traversing</title>
<script>
window.onload = function(){
  var div = document.getElementById('header');
  for(var i=0; i<div.childNodes.length; i++){
    console.dir(div.childNodes[i]);
  }
}
</script>
</head>
<body>
  <div id='header'>
    <h1>DOM traversing</h1>
    <p>DOM을 탐색한다</p>
  </div>
</body>
</html>
```

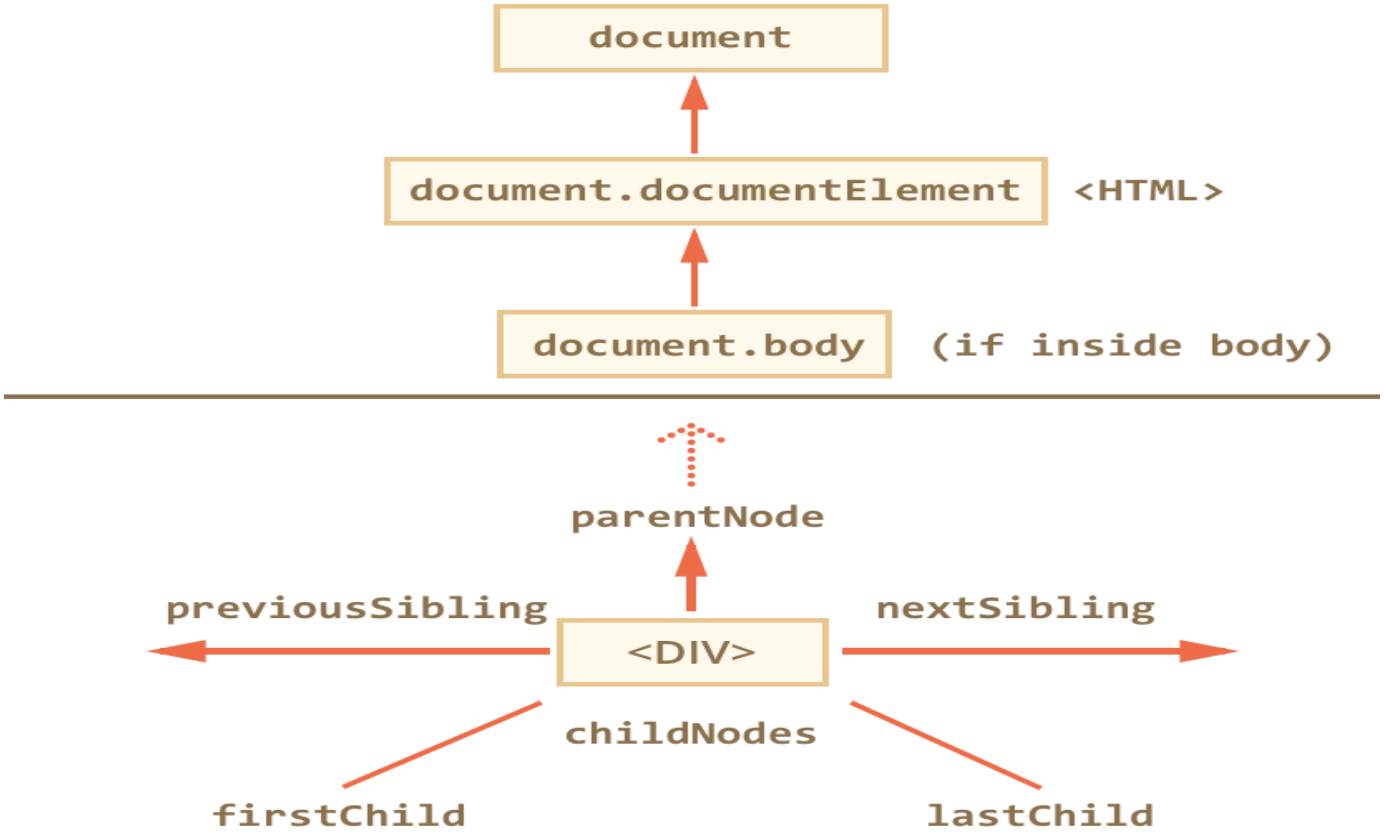
DOM(Document Object Model)

- node tree를 구성
  - html은 head와 body 갖는다
  - head는 meta, title, script를 갖는다
  - body는 div를 갖는다
  - div는 h1, p를 갖는다
  - h1, p는 형제 관계



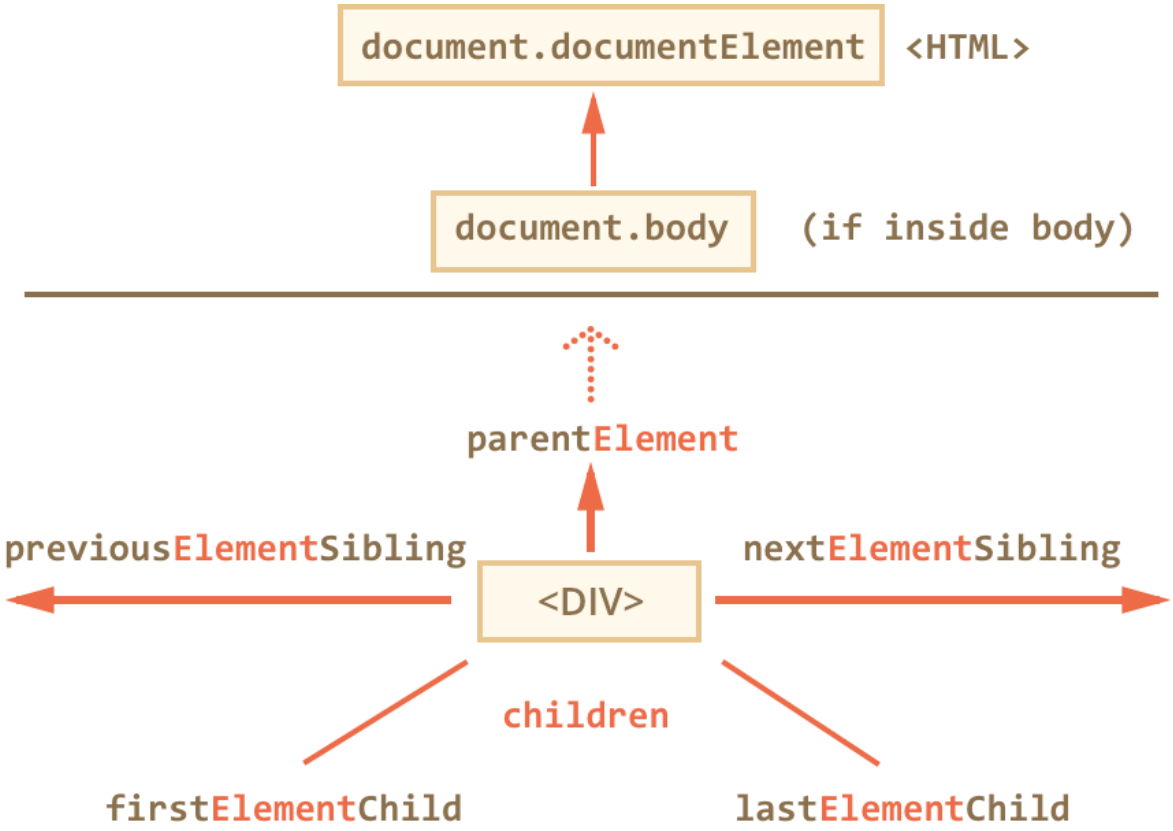
DOM(Document Object Model)

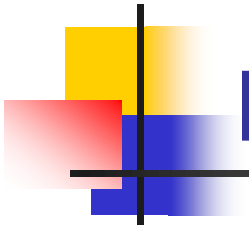
- DOM 탐색



DOM(Document Object Model)

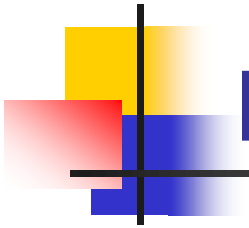
- DOM 탐색 con'd





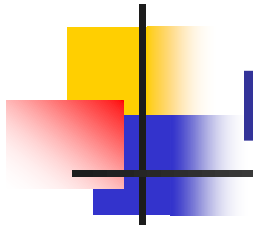
DOM(Document Object Model)

- Events
  - 이벤트는 사용자나 브라우저로부터 발생하는 행위
  - 아래는 웹사이트에서 발생 할 수 있는 이벤트
    - The page finishes loading
    - The user clicks a button
    - The user hovers over a dropdown
    - The user submits a form
    - The user presses a key on their keyboard
  - Event Handlers and Event Listeners
    - Event Handler, Event Listener는 이벤트가 발생 했을 때 실행되는 자바스크립트 함수
    - Inline event handlers, Event handler properties, Event listeners를 이용한 세가지 방식으로 이벤트 헨들러를 지정(등록) 할 수 있음.



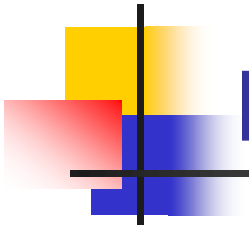
Functions

- Definition
  - 일정한 형식의 명령을 수학 공식처럼 규칙화한 것
  - member of the Object type that is an instance of the standard built-in Function constructor and that may be invoked as a subroutine
  - 함수는 'first-class object'
    - 변수에 담을 수 있다
    - 인자로 전달 할 수 있다
    - 반환 값으로 전달 할 수 있다
  - 함수는 Function 객체의 instance
  - 함수의 정의는 function declaration, function expression
  - 함수 호출 시 return이 없을 경우 undefined 반환
  - 생성자 함수를 이용하여 객체를 생성 할 수 있다
  - call by value, call by reference



Functions

- 함수를 호출하면 stack 자료구조에 execution context가 생성되고 execution context에는 함수 실행에 필요한 여러 정보가 포함 되어 있다
  - LexicalEnvironment
  - VariableEnvironment
  - ThisBinding

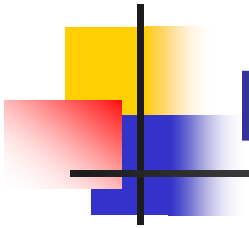


Functions

- Function Declaration
  - 선언식을 사용하여 함수 정의
  - 기명 또는 무기명 함수
  - return 문이 없으면 undefined 반환
  - Hoisting

Syntax

function functionName(parameters){
  function's body;
  return statement; // optional
}



Functions

- Function Expression
  - 표현식을 사용하여 함수 선언
  - 기명 또는 무기명 함수
  - Hoisting 되지 않는다

Syntax

```
var func = function(parameters){
    function's body;
    return statement; // optional
}
```

Executable Code and Execution Context

- 들어가기 전에...
  - 아래 코드의 실행 결과를 예측 할 수 있는가?

```
(function(){
  var ref1 = outer();
  var ref2 = outer();
  ref1(); // ?
  ref1(); // ?
  ref1(); // ?
  ref1(); // ?
  ref2(); // ?
  ref2(); // ?
  ref2(); // ?
  ref2(); // ?
  function outer(){
    var index = 1;
    return function(){
      index++;
      console.log(index);
    };
  }
})();
```

```
(function(){
  var funcArr = [];
  for(var i=1; i<10; i++){
    funcArr[i-1] = function(){
      console.log(i);
    }
  }

  funcArr[0](); // 1?
  funcArr[1](); // 2?
  funcArr[2](); // 3?
})();
```

Executable Code and Execution Context

- Definition
  - ECMAScript에는 실행가능 한 코드가 3가지 있다
    - global code
    - eval code
    - function code
  - 흐름이 실행코드로 이동하면 흐름은 실행코드의 실행과 관련된 execution context로 진입한다
  - execution context는 stack자료 구조에 저장된다
  - exectuion context는 3가지 component를 갖는다
    - LexicalEnvironment
    - VariableEnvironment
    - ThisBinding

Executable Code and Execution Context

- Execution Context
  - When control is transferred to ECMAScript executable code, control is entering an *execution context.*
  - Active execution contexts logically form a stack.
  - The top execution context on this logical stack is the running execution context.
  - A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context.
  - The newly created execution context is pushed onto the stack and becomes the running execution context.

Executable Code and Execution Context

- Lexical Environment
  - A **Lexical Environment** is a specification type used to define the association of **Identifiers** to specific **variables** and **functions** based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an Environment Record and a possibly null reference to an outer Lexical Environment.
  - Lexical Environment는 일반적으로 Function Declaration, With Statement, Catch절과 같은 syntactic 구조와 관련이 있고, 저러한 코드가 평가 될 때 새로운 Lexical Environment가 생성된다

Executable Code and Execution Context

- Environment Records
  - declarative environment record
    - 구문요소인 함수 선언, 변수 선언, catch절 등을 정의
    - 함수 scope 내에 정의된 식별자(identifier)를 bind
  - object environment record
    - binding object라 불리는 객체와 관련이 있다
    - binding object객체의 property를 bind
    - with절에 object를 기술하면 with절 내부에서는 object의 property를 식별자처럼 사용 가능한 것

- outer environment reference
  - The outer environment reference is used to model the logical nesting of Lexical Environment values.
  - 함수를 둘러싼 외부 함수의 Lexical Environment를 참조
  - 함수 내부에 2개의 함수가 있다면 두 함수의 outer는 동일한 외부 함수의 Lexical Environment를 참조
  - 함수 객체는 [[scope]]라 불리는 internal property를 가지고 있고, 함수가 평가될 때 active execution context의 LexicalEnvironment를 기록 함

Executable Code and Execution Context

- Variable Environment
  - Identifies the Lexical Environment whose environment record holds bindings created by VariableStatements and FunctionDeclarations within this execution context.
  - 특정 Execution context의 LexicalEnvironment와 VariableEnvironment는 항상 Lexical Environments
  - execution context가 생성 될 때 둘의 값은 동일하다
  - LexicalEnvironment는 코드의 실행 동안 변경되지만 VariableEnvironment는 절대 변경되지 않는다

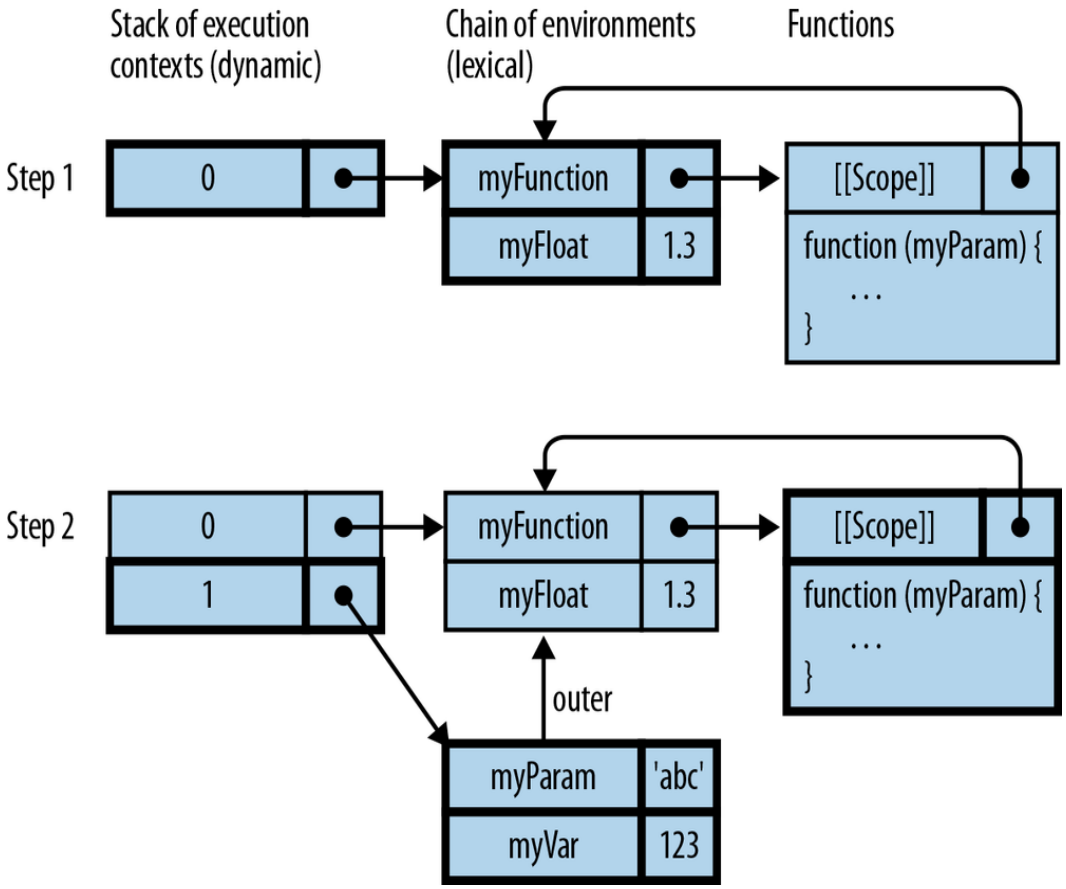
Executable Code and Execution Context

- ThisBinding
  - The value associated with the this keyword within ECMAScript code associated with this execution context.
  - 실행가능한 코드가 실행 될 때 execution context이 생성되고 ThisBinding 객체가 결정된다

실행가능 코드	**case**	**ThisBinding object**
global code	always	global object
eval code	- no calling context - not being evaluated by a direct call to eval	global object
	else	calling execution context
function code	- caller provided thisArg	thisArg
	- thisArg is null or undefined	global object
	- thisArg is not Object	ToObject(thisArg)

Executable Code and Execution Context


```
function myFunction(myParam){
  var myVar = 123;
  return myFloat;
}
var myFloat = 1.3;
// Step 1
myFunction('abc'); // Step2
```



Executable Code and Execution Context

- Closures
  - Functions stay connected to their birth scope.
  - 함수 객체가 생성 될 때 internal property인 [[scope]] 속성에 현재 실행 중인 execution context의 LexicalEnvironment가 기록된다
  - 함수가 중첩구조로 되어 있을 때 외부 함수가 종료 되더라도 외부 함수의 변수(LexicalEnvironment)를 접근하는 내부 함수의 reference가 살아있으면 외부함수의 LexicalEnvironment가 유지되는 현상

Executable Code and Execution Context


```
(function(){
  var x;
  var y = increment(1);
  x=y();
  x=y();
  function increment(base){
    base = base||0;
    return function(){
      return base++;
    }
  }
})();
```

step1

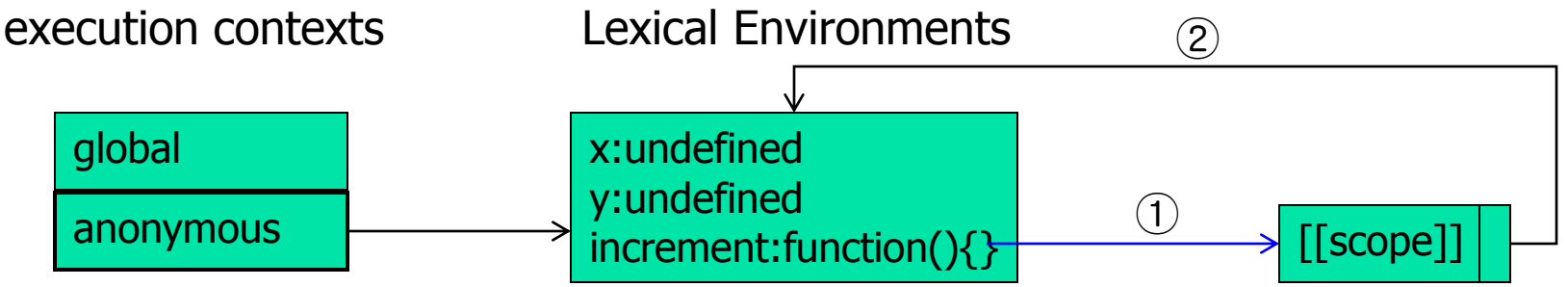
step2

step3

step4

Executable Code and Execution Context

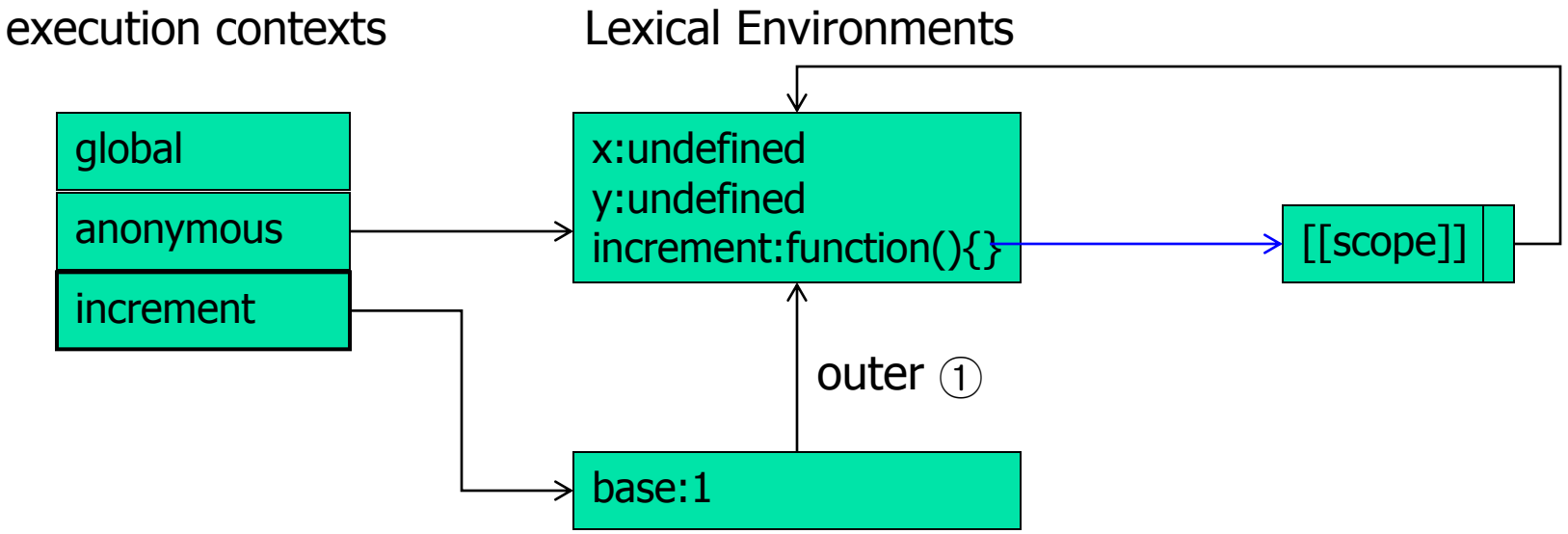
❖ step1. Anonymous(IIFE)함수가 실행되기 전, increment 함수의 평가가 이루어진 후



① 함수 객체는 internal property인 [[scope]]가 존재한다
② 함수 객체는 함수가 평가 될 때 internal property [[scope]]에 자신을 에워싸고 있는(running execution context) execution context의 LexicalEnvironment를 기록한다

Executable Code and Execution Context

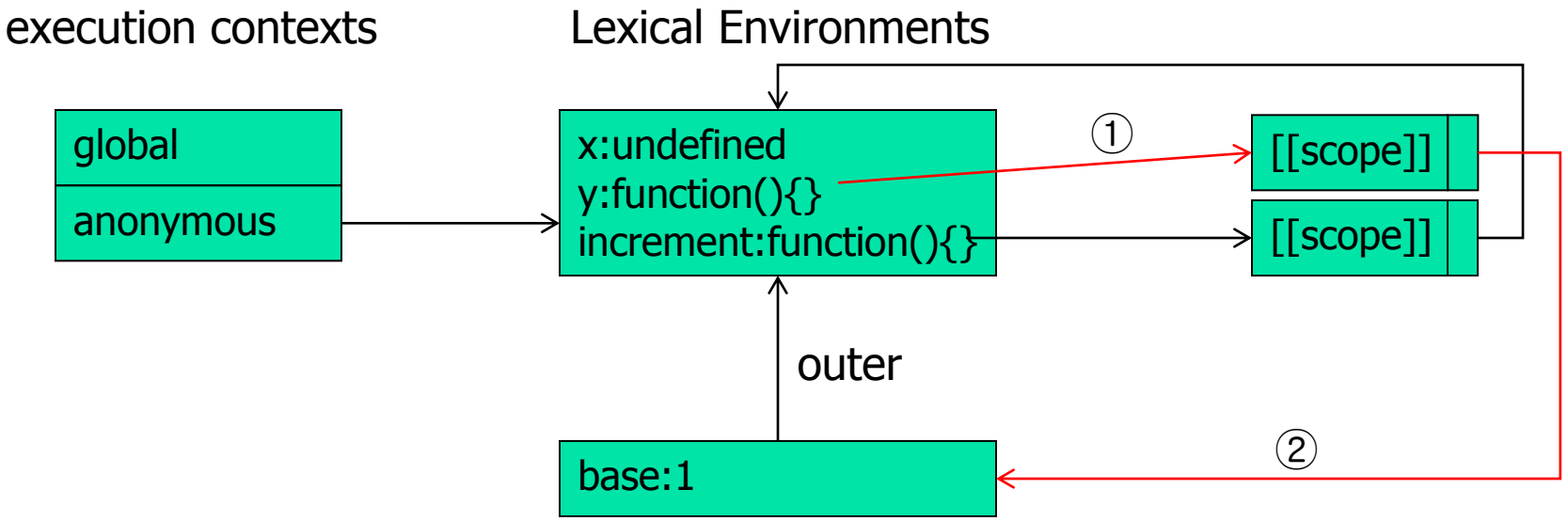
❖ step2. increment 함수가 실행 되는 중



① LexicalEnvironment의 구성요소인 outer는 함수 객체의 내부 속성인 [[scope]]를 통해서 값이 설정

Executable Code and Execution Context

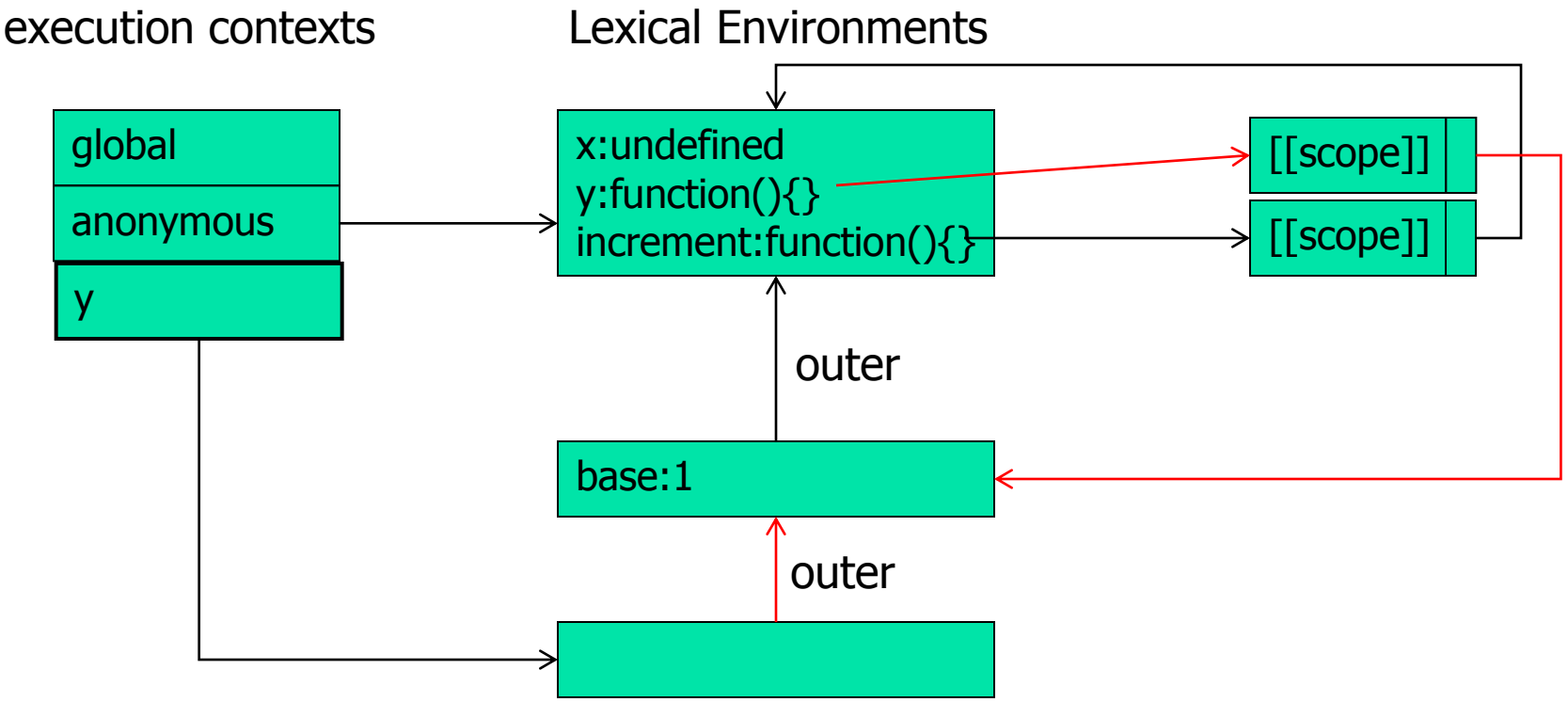
❖ step2. increment 함수가 실행 완료되고, 할당 연산이 완료 되었을 때



① 변수 y는 함수 객체를 할당 받음
② 내부 함수가 외부 함수의 식별자를 참조하고, 내부 함수의 reference가 살아 있을 경우, 외부 함수의 LexicalEnvironment가 heap에 존재한다

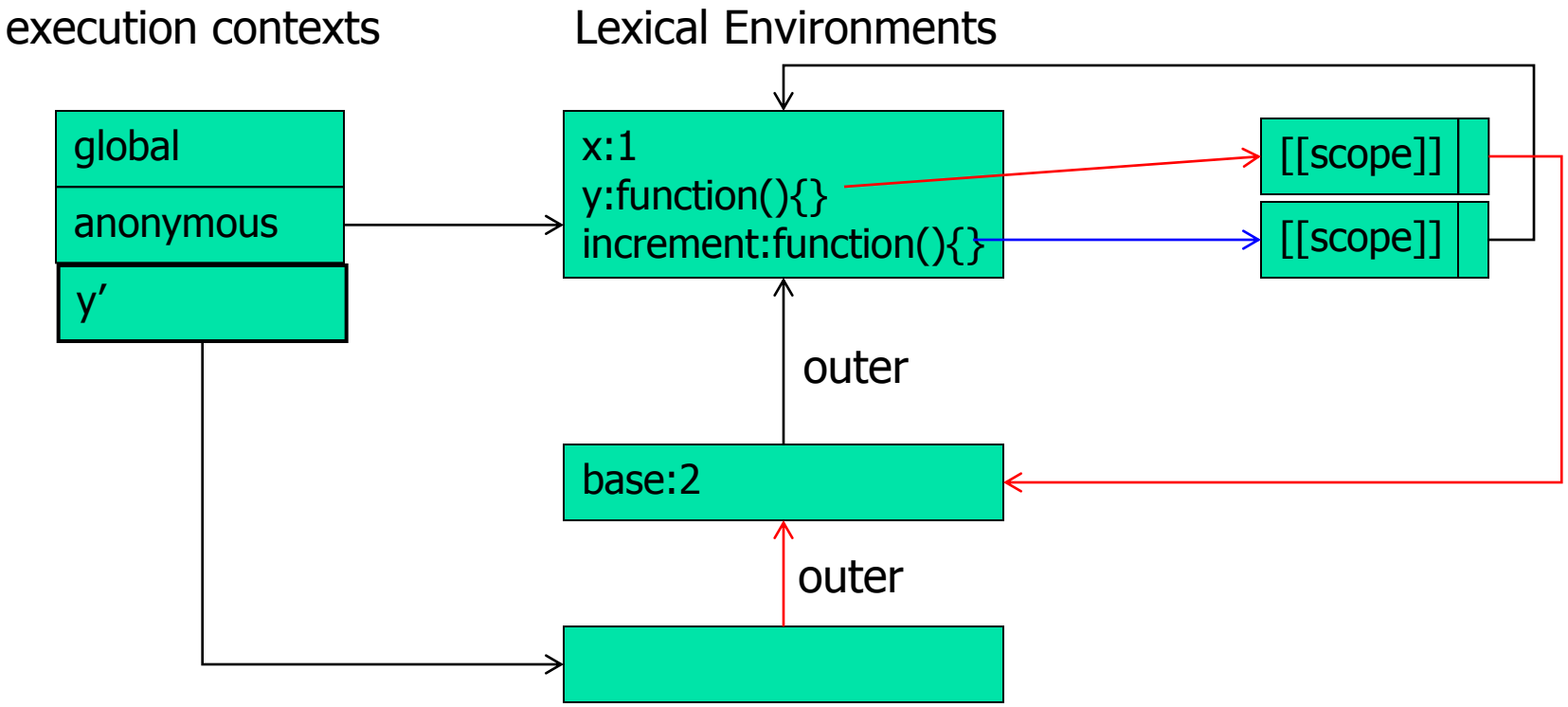
Executable Code and Execution Context

❖ step3. 함수 포인터 y를 실행 중 상황



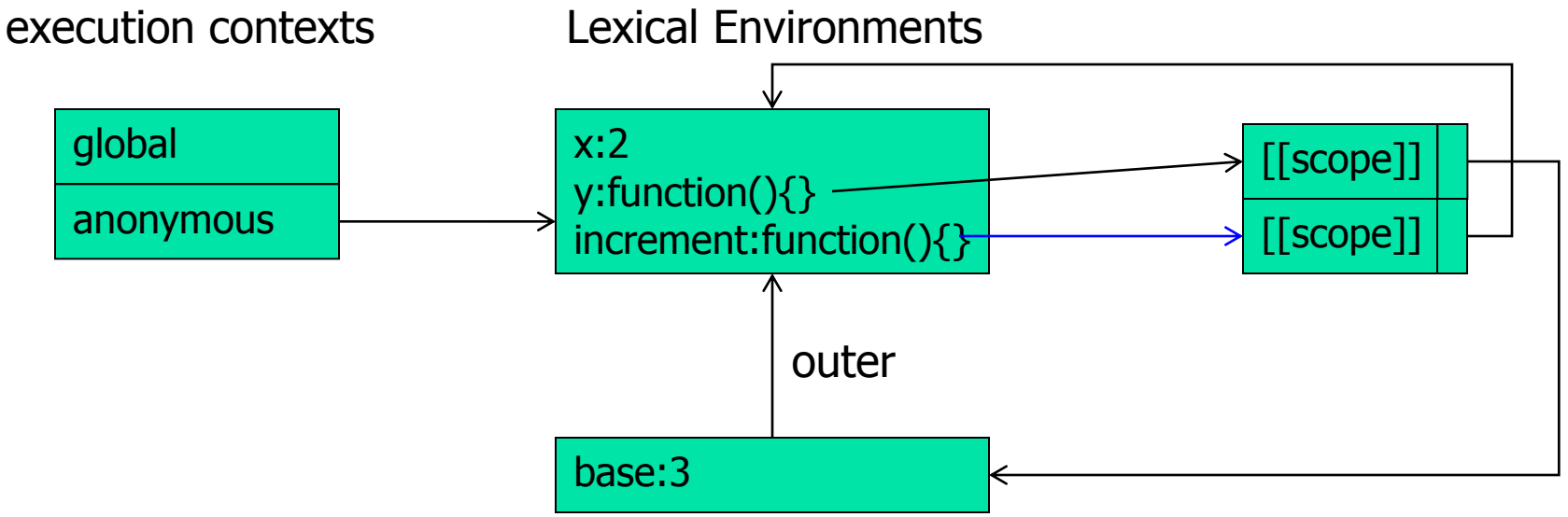
Executable Code and Execution Context

❖ step4. 함수 포인터 y를 실행 중 상황



Executable Code and Execution Context

❖ step4. y 실행을 완료하고 할당연산이 종료 된 후



Executable Code and Execution Context

- closure를 잘못 사용 하는 예
  - 현재 scope의 변수에 영향을 받는 복수의 내부 함수를 선언 하려고 할 때

```
(function(){
  function foo(){
    var resultArr = [];
    for(var i=0; i<10; i++){
      var f = function(){
        return i;
      }
      resultArr.push(f);
    }
    return resultArr;
  }
  console.log(foo()[4]());
  console.log(foo()[5]());
})();
```

변수 i는 foo 함수의 지역변수

f에 함수 객체를 할당, 함수 객체는 foo 함수의 i를 참조

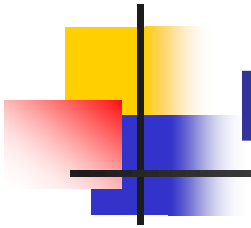
?

figure 1. closure의 잘못된 사용

Executable Code and Execution Context

- *figure 1* 의 문제점을 IIFE를 사용하여 해결

```
(function(){
  function foo(){
    var resultArr = [];
    for(var i=0; i<10; i++){
      var f = (function(param){
        return function(){
          return param;
        }
      })(i);
      resultArr.push(f);
    }
    return resultArr;
  }
  console.log(foo()[4]());
  console.log(foo()[5]());
})();
```



Prototype

- Definition
  - JavaScript 모든 객체는 prototype property와 internal property인 [[prototype]]를 갖는다
  - prototype property는 new 생성자() 실행 시, 새롭게 생성된 객체의 [[prototype]]의 값으로 설정된다
  - 객체 내부의 속성은 prototype chain을 따라 식별자를 찾을 때까지 prototype chain을 탐색한다

Prototype

- Prototype property and [[prototype]]
  - 모든 객체는 prototype property를 갖는다.
  - 해당 함수 객체로 새로운 객체를 생성 할 때 공통으로 사용되어야 하는 속성들을 정의
  - new연산자와 생성자 함수로 prototype property를 참조하는 [[prototype]]속성을 갖는 새로운 객체가 생성
  - [[prototype]]은 ECMAScript 명세에 있는 속성임

Prototype

```
(function(){
  var person = {describe:function(){
    console.log(this.name+" "+this.age);
  }};
  var person1 = Object.create(person,{name:{writable:true,value:"철수"},age:{writable:true,value:25}});
  var person2 = Object.create(person,{name:{writable:true,value:"영희"},age:{writable:true,value:31}});
  person1.describe();
  person2.describe();
})();
```

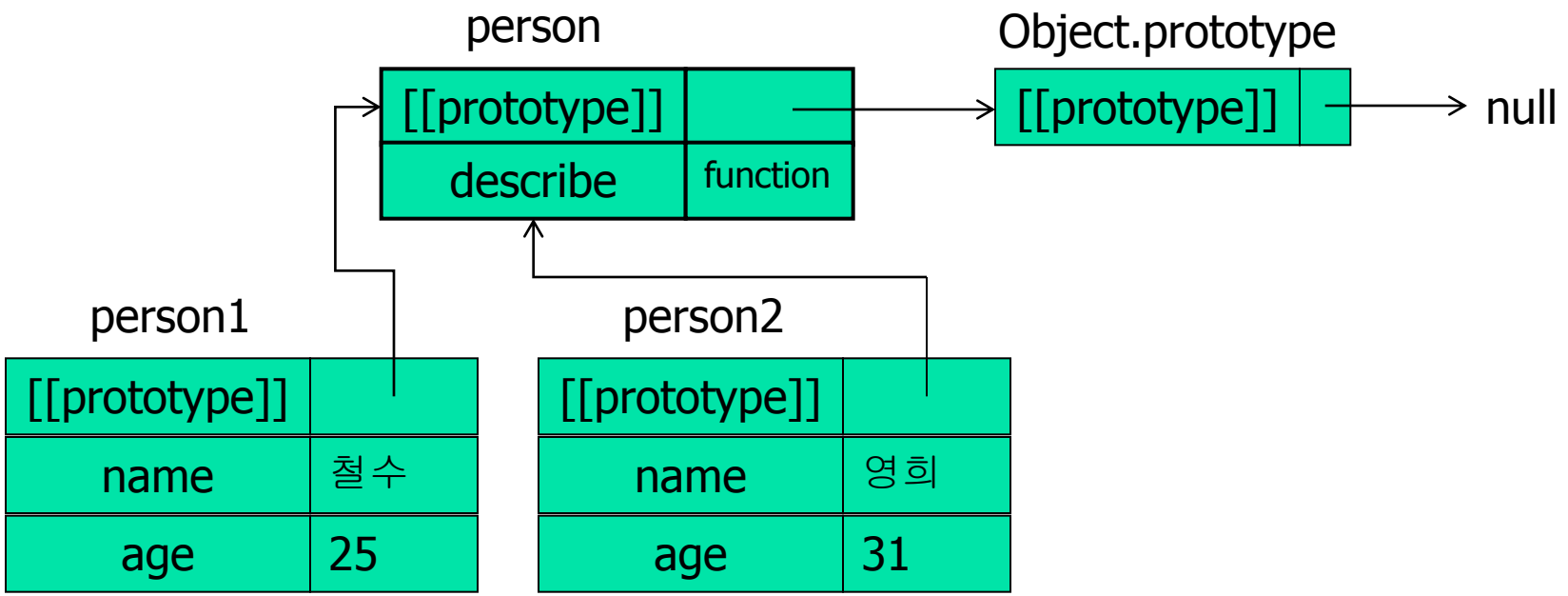


figure 1. Object를 이용한 상속

Prototype

```
(function(){
  function Animal(species, livingPlace){
    this.typeOfSpecies = species;
    this.whereLives = livingPlace;
    this.move = function(){
      console.log("animal moves");
    }
    this.describe = function(){
      console.log("species "+this.typeOfSpecies);
      console.log("whereLives "+this.whereLives);
    };
  }
  var dog = new Animal("개과", "육지");
  var cat = new Animal("고양이과", "육지");
  dog.describe();
  cat.describe();
})();
```

figure 2-1. function객체를 이용한 객체 생성

Prototype

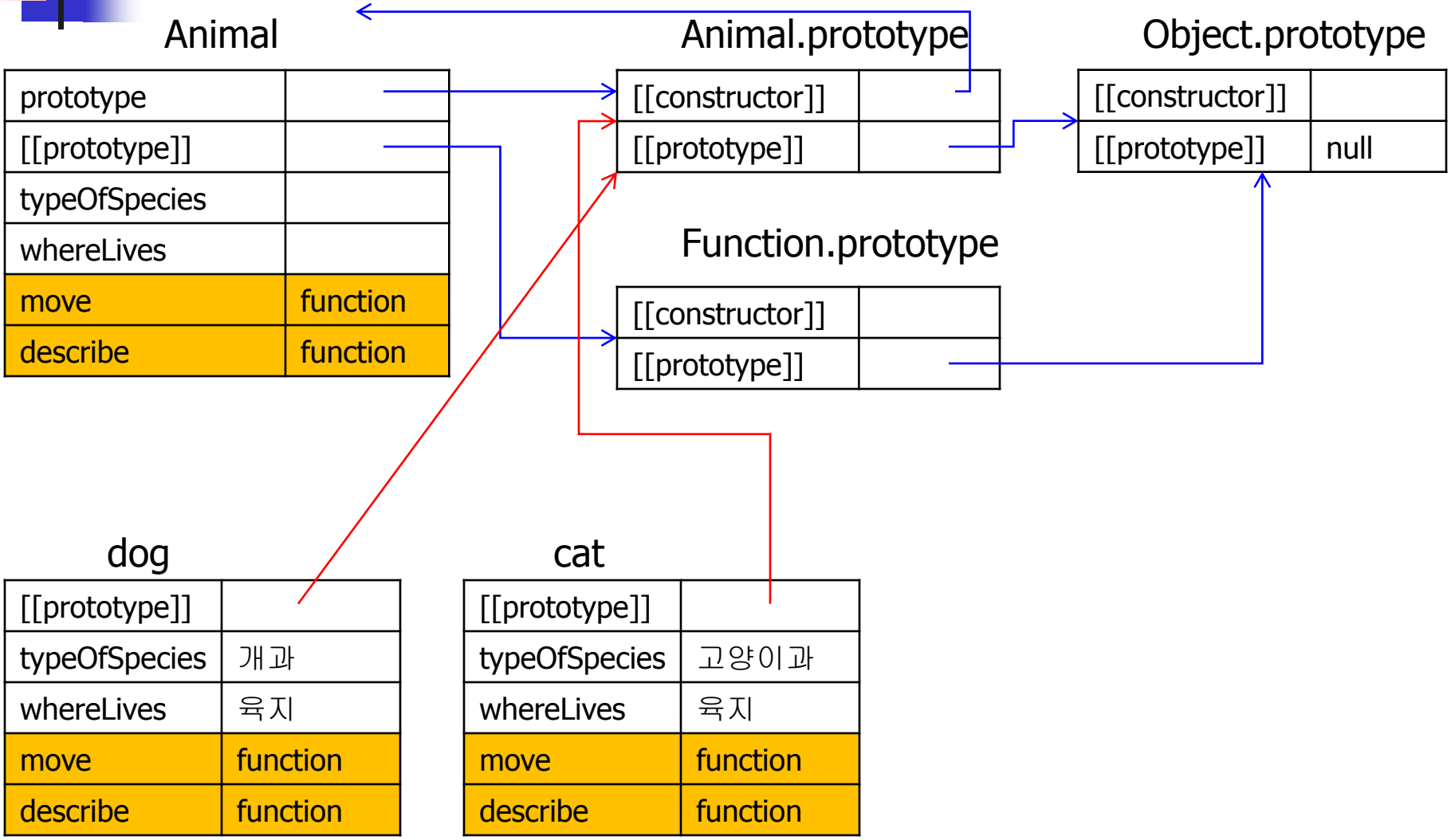


figure 2-2. figure 2-1의 prototype chain

Prototype

```
(function(){
  function Animal(species, livingPlace){
    this.typeOfSpecies = species;
    this.whereLives = livingPlace;
  }
  Animal.prototype.describe = function(){
    console.log("species "+this.typeOfSpecies);
    console.log("whereLives "+this.whereLives);
  }
  Animal.prototype.move = function(){
    console.log("animal moves");
  }
  var dog = new Animal("개과", "육지");
  var cat = new Animal("고양이과", "육지");
  dog.describe();
  cat.describe();
})();
```

figure 3-1. function객체의 prototype 객체에 속성 추가

Prototype

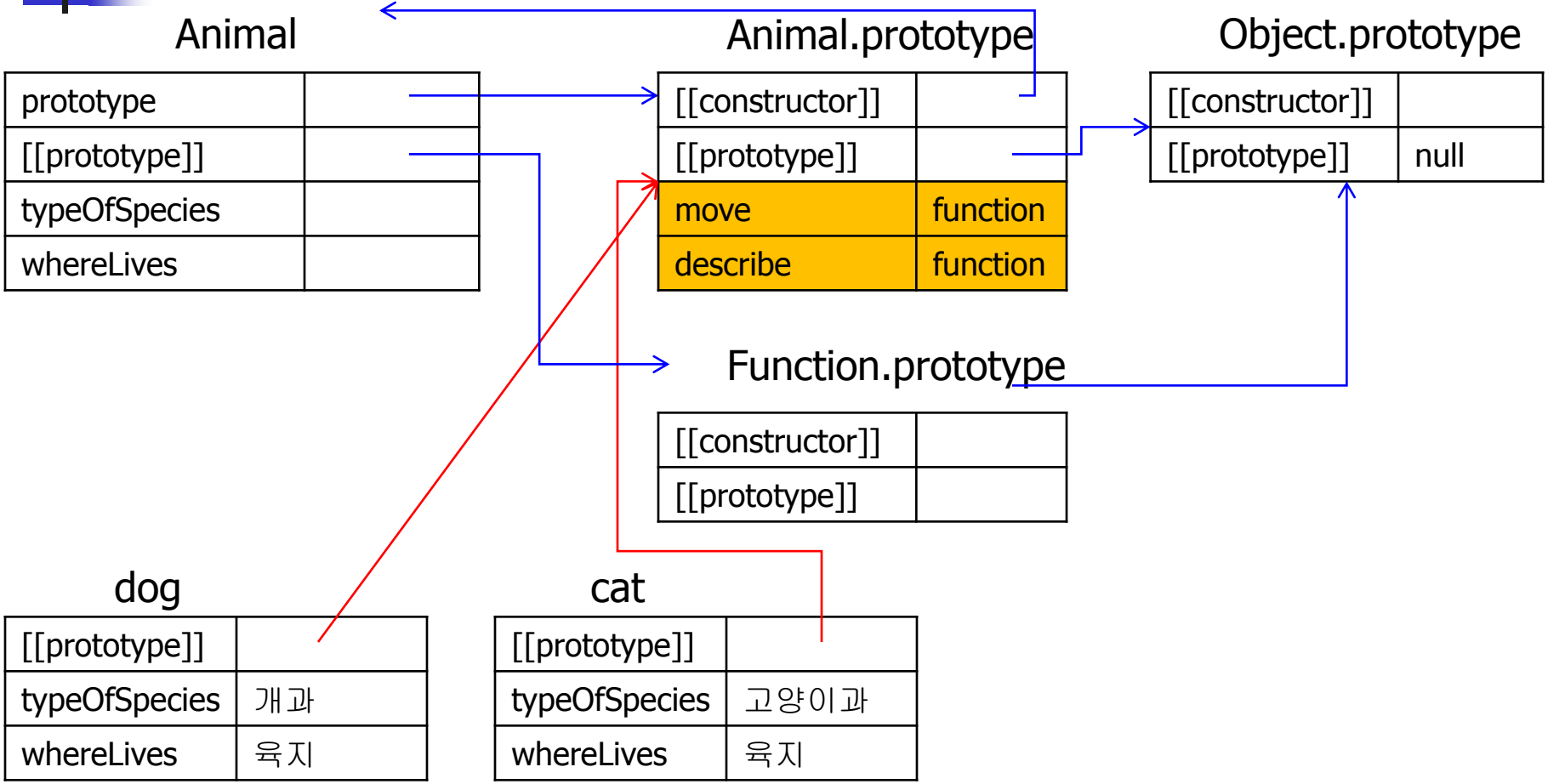
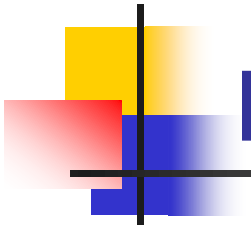


figure 3-2. figure 3-1의 prototype chain



Prototype

- Prototype chain
  - 모든 객체는 internal property인 [[prototype]]을 갖고 있다.
  - [[prototype]] 속성에 의해 연결된 객체 chain을 prototype chain이라 지칭함
  - figure 1에서 person1, person2의 [[prototype]]은 person 객체를 참조하고, person 객체의 [[prototype]]은 Object.prototype 객체를 참조, 이렇듯 [[prototype]]들로 연결된 상태