ch03 노드의 자바스크립트와 친해지기

자바스크립트 기초

언어로서의 특징

- 스크립트 언어
  - 스크립트 언어
    - 간편한 코딩을 목적으로 만들어진 프로그래밍 언어
  - 단기간에 습득 및 개발 가능
- 인터프리터 언어
  - 인터프리터 언어
    - 프로그램을 컴파일하지 않고 소스코드 레벨에서 한 줄씩 번역하면서 실행
  - 단점
    - 컴파일 언어에 비하여 느림
  - 장점
    - 컴파일과 같은 특별한 단계가 필요없음
    - 코드를 작성하여 바로 실행 가능함

statement 규칙

- 문장의 맨 끝에 세미콜론(;)을 붙인다
  - 생략하는 것도 가능하지만 문장의 단락이 불명확해지므로 생략하지 말것!

```
document.writeln('Hello, World ! ')
```

- 문장의 도중에 공백이나 개행, 탭 포함 가능
  - JavaScript에서는 문장 안의 공백이나 개행, 탭은 무시
  - 하나의 문장이 긴 경우는 적절한 개행이나 인덴트를 가미함으로써 코드를 보기 좋게 할 수 있음

```
document.writeln('Hello, World ! ');
document.
        writeln
                ('Hello, World ! ');
```

statement 규칙

- 대/소문자 구별됨

  document.Writeln('Hello, World ! ');

  - ‘writeln’과 ‘Writeln’은 각각 별개의 명령으로 인식

주석

- comments
  - 스크립트의 동작에는 상관없는 메모와 같은 정보
    - 자기 자신이 기술한 코드
      - 시간이 경과하면 어떠한 목적을 위해 작성했는지 좀처럼 기억해내지 못하는 경우가 자주 있음.
    - 다른 사람이 작성한 코드를 이해
      - 대부분 매우 어려운 일
    - 코드의 요소 요소 주석
      - 그 코드가 무엇을 하고 있으며 무엇을 목적으로 작성되었는지 그 개요를 파악하기 쉬움.
  - 장기적인 유지보수를 전제로 한 애플리케이션에서는 주석의 기술은 필수

기법	개요
// comment	단일 행 주석. '//'에서부터 해당 행의 끝까지를 주석
/* comment */	복수 행 주석. '/*~*/'으로 둘러싸인 블록을 주석

- 한 행으로 끝나는 주석을 기술할 경우: //
- 복수 행에 걸친 주석을 기술할 경우: /* ~ */

03-01 자바스트립트의 객체와 함수 이해하기

자바와 자바스크립트의 변수 타입 비교

- 자바는 자료형(타입)을 명시하는 언어
- 자바스크립트는 자료형을 명시하지 않는 언어
    - 모든 변수는 var 로 선언
- 내부에서는 자료형에 따라 변수 상자의 크기가 달라짐

자바스크립트의 자료형

- **boolean, number, string 이 있으며, 그 외에 undefined, null, Object 자료형이 있음**

자료형	설명
Boolean	[기본 자료형] true와 false의 두 가지 값을 가지는 자료형
Number	[기본 자료형] 64비트 형식의 IEEE 754 값이며 정수나 부동소수 값을 가지는 자료형 몇 가지 상징적인 값을 가질 수 있음: NaN(숫자가 아님), +무한대(Number.MAX_VALUE로 확인), -무한대(Number.MIN_VALUE로 확인)
String	[기본 자료형] 문자열 값을 가지는 자료형
undefined	값을 할당하지 않은 변수의 값
null	존재하지 않는 값을 가리키는 값
Object	객체를 값으로 가지는 자료형 객체는 속성들을 담고 있는 가방(Collection)으로 볼 수 있으며, 대표적인 객체로 Array나 Date를 들 수 있음

식별자의 명명규칙

- 식별자
  - 이름
    - 변수는 물론, 나중에 등장할 함수나 레이블 등이 모두 해당

- 네 가지 명명규칙
  - 문자는 영문자/언더스코프(_)/달러표시($) 중 하나이어야 함
  - 문자 이후에는 첫 번째에서 사용할 수 있는 문자 또는 숫자이어야 함
  - 변수명에 포함된 영문자의 대문자/소문자는 구별됨
  - JavaScript에서 의미를 갖는 예약어가 아니어야 함

식별자의 명명규칙

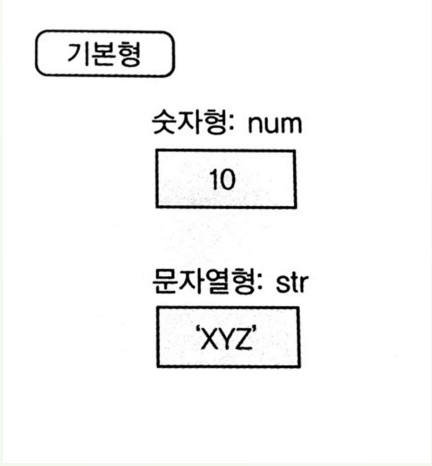
- 기술 방법

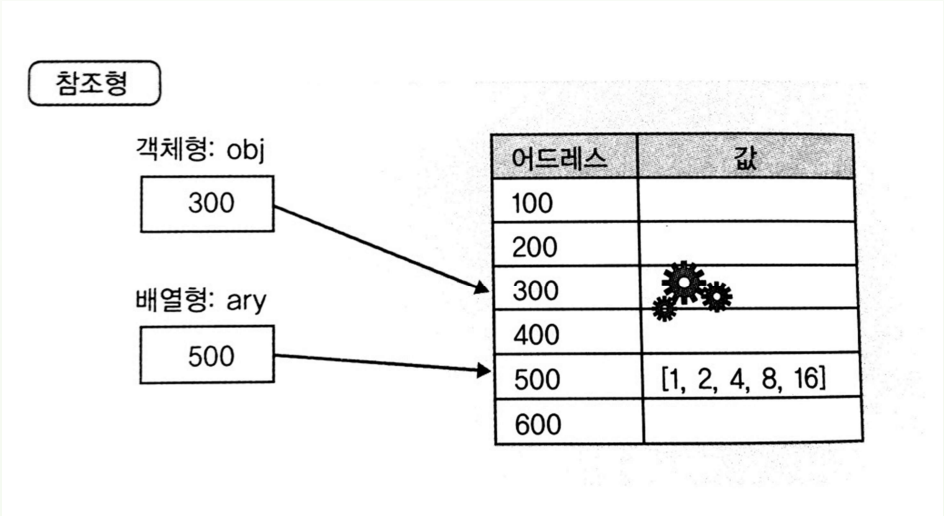
기술 방법	개요	예
camelCase 기법	앞 단어 첫 문자는 소문자, 그 이후의 단어의 첫 문자는 대문자	lastName
Pascal 기법	모든 단어의 첫 문자는 대문자	LastName
언더스코프 기법	모든 단어의 첫 문자는 소문자, 단어 간은 '_'로 연결	last_name

  - 변수명이나 함수명: camelCase 기법
  - 클래스(구조체)명: Pascal 기법

변수: 기본형과 참조형

- 기본형과 참조형
  - 차이점: 값을 변수에 격납하는 방법
- 기본형의 변수
  - 값 그 자체가 직접 격납
- 참조형의 변수
  - 그 참조값(값을 실제로 격납하고 있는 메모리상의 어드레스)을 격납





변수 만들기

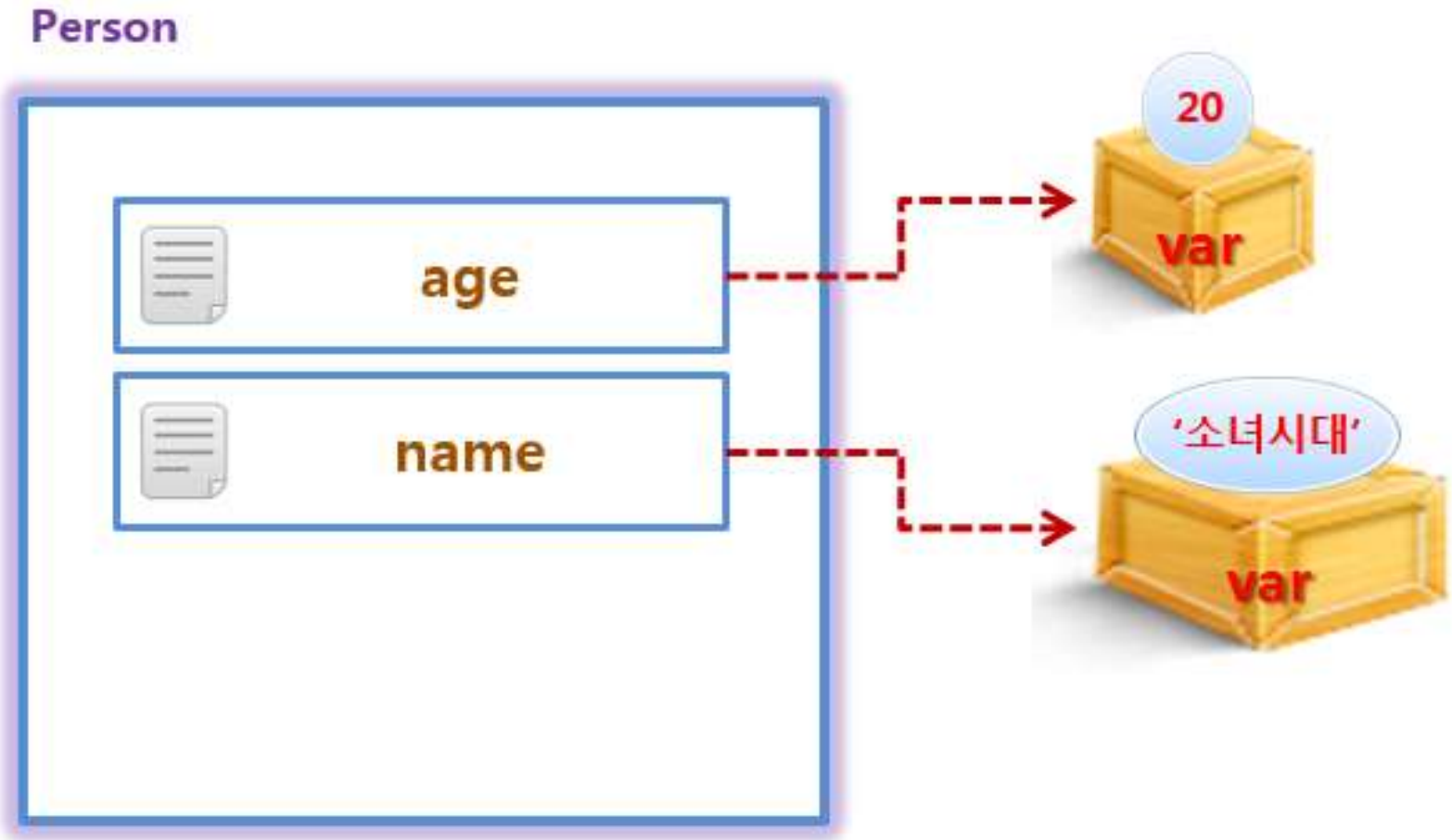
- **변수 앞에는 var 키워드를 붙임**

ch03_test1.js

```
var age = 20;
console.log('나이 : %d', age);
var name = '소녀시대';
console.log('이름 : %s', name);
```

자바스크립트의 객체

- 속성들이 이름 - 값 의 형태로 들어가 있음



자바스크립트의 객체

- 객체 생성 방법 3가지
  - ① object literal 방식: {} 를 사용하여 생성
  - ② Object() constructor 사용
  - ③ prototype constructor 함수 사용
    - constructor fucntion의 첫 글자는 대문자로 시작함
    - 예) function Person(){…}

객체 만들기

- 객체 안의 속성 접근
  - ① . 연산자를 이용해 접근
  - ② [ ] 를 붙이고 그 안에 속성 이름을 문자열로 넣어 접근
- 객체 속성 접근 연산 의미
  - 기존의 속성이 있는 경우: 해당 속성에 접근
  - 기존의 속성이 없는 경우: 새로운 속성 추가

ch03_test03.js

```
var Person = {};
Person["age"] = 20;
Person["name"] = '소녀시대';
Person.mobile = '010-1000-1000';
console.log('나이 : %d', Person.age);
console.log('이름 : %s', Person.name);
console.log('전화 : %s', Person["mobile"]);
```

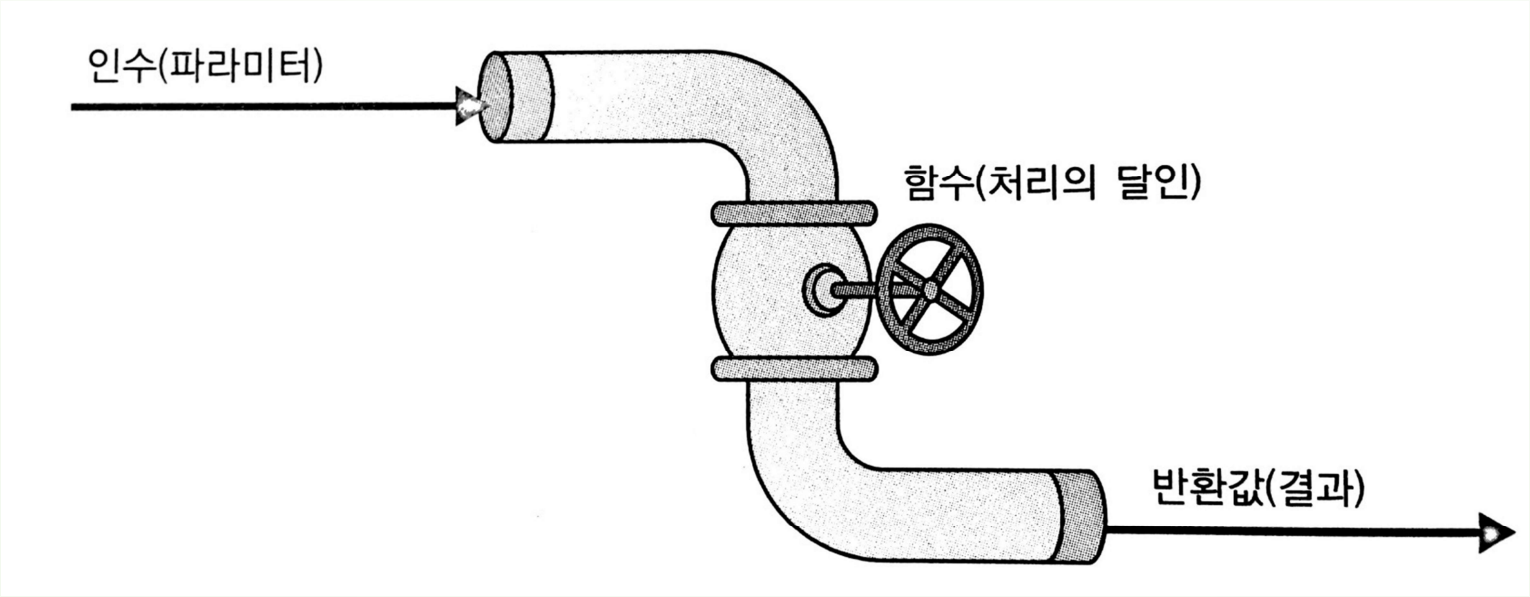
Object 객체

- Object 객체
  - 모든 객체의 공통적인 성질/기능을 제공
  - 모든 객체의 기본 객체다 => 모든 객체의 프로토타입 객체다
- 주요 멤버

멤버	개요
constructor	인스턴스화에서 사용되는 생성자(읽기 전용)
toString()	객체의 문자열 표현을 취득
valueOf()	객체의 기본형 표현(많게는 숫자값)을 취득
hasOwnProperty(prop)	지정한 프로퍼티를 갖는가
propertyIsEnumerable(prop)	for ... in 명령에 의해서 프로퍼티/메소드를 열거할 수 있는가
isPrototypeOf(obj)	호출원의 객체가 지정한 객체의 프로토타입인가

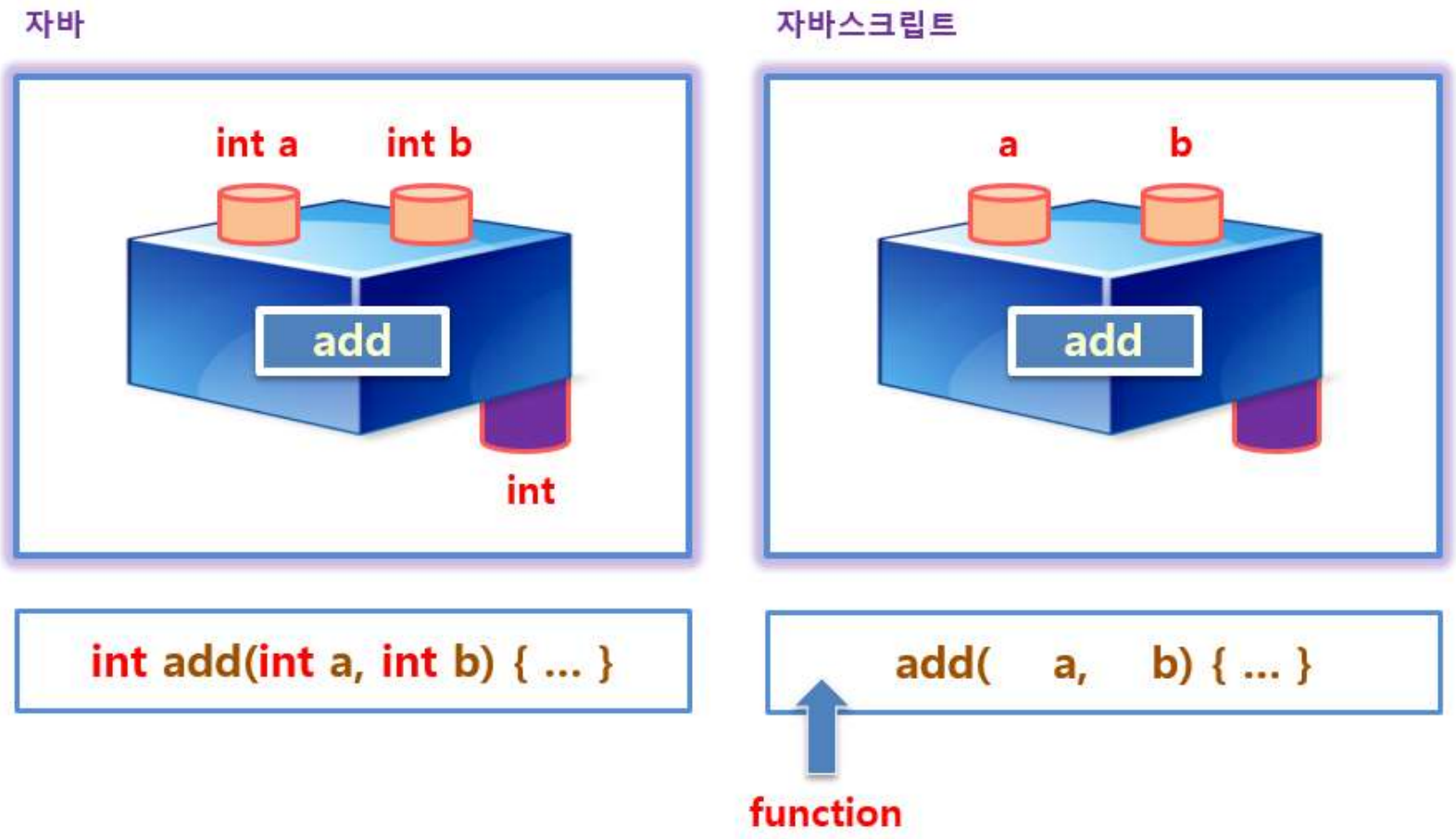
function

- 함수
  - 어떠한 입력값(인수)이 주어짐에 따라 미리 정해진 처리를 행하여 그 결과(반환값)를 반환해주는 구조



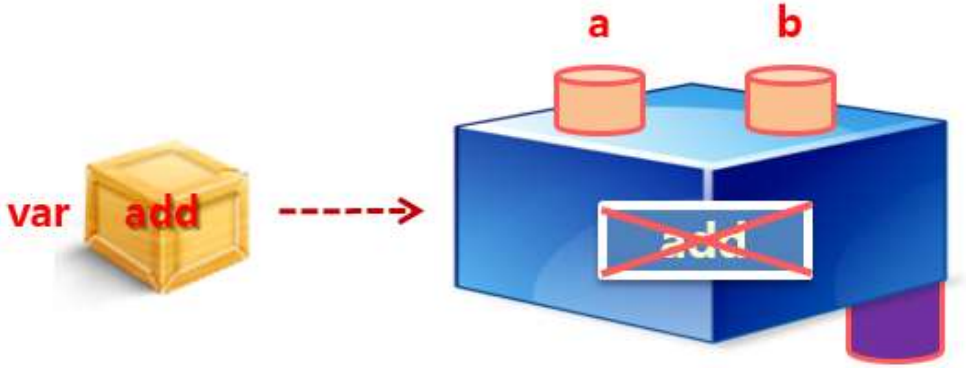
자바와 자바스크립트의 함수 비교

- 파라미터의 타입과 반환되는 값의 타입을 명시하지 않음
- 함수 앞에는 function 키워드를 붙임



함수를 변수에 할당하기

- 자바스크립트에서는 함수를 일급 객체(First class object)로 다룸
  - 함수가 변수에 할당될 수 있음
- 변수에 할당될 경우
  - 두 가지 이름으로 함수를 호출할 수 있음
- 익명함수(Anonymous Function)
  - 함수의 이름이 없는 함수




```
var add = function (    a,    b) { ... };
```

함수를 변수에 할당하기

- 함수를 선언하는 세 가지 방법

① function statement 방식: 함수 이름을 정의

```
function add(a, b){
    return a+b;
}
```

② function expression 방식: 익명 함수로 정의 후 변수에 할당

```
var add = function(a, b){
    return a+b;
};
```

③ Function constructor 사용하여 정의 ➔ 현재 사용하지 않는 방법

```
var add = new Function("a", "b", "return a+b;");
```

- 함수 호출

```
add(10, 10);
```

함수 만들어 실행하기

- 함수를 만들고 실행할 수 있음
- 선언문 (Declaration)

ch03_test03.js

```
function add(a, b) {
    return a + b;
}
var result = add(10, 10);
console.log('더하기 (10, 10) : %d', result);
```

함수 만들어 변수에 할당하기

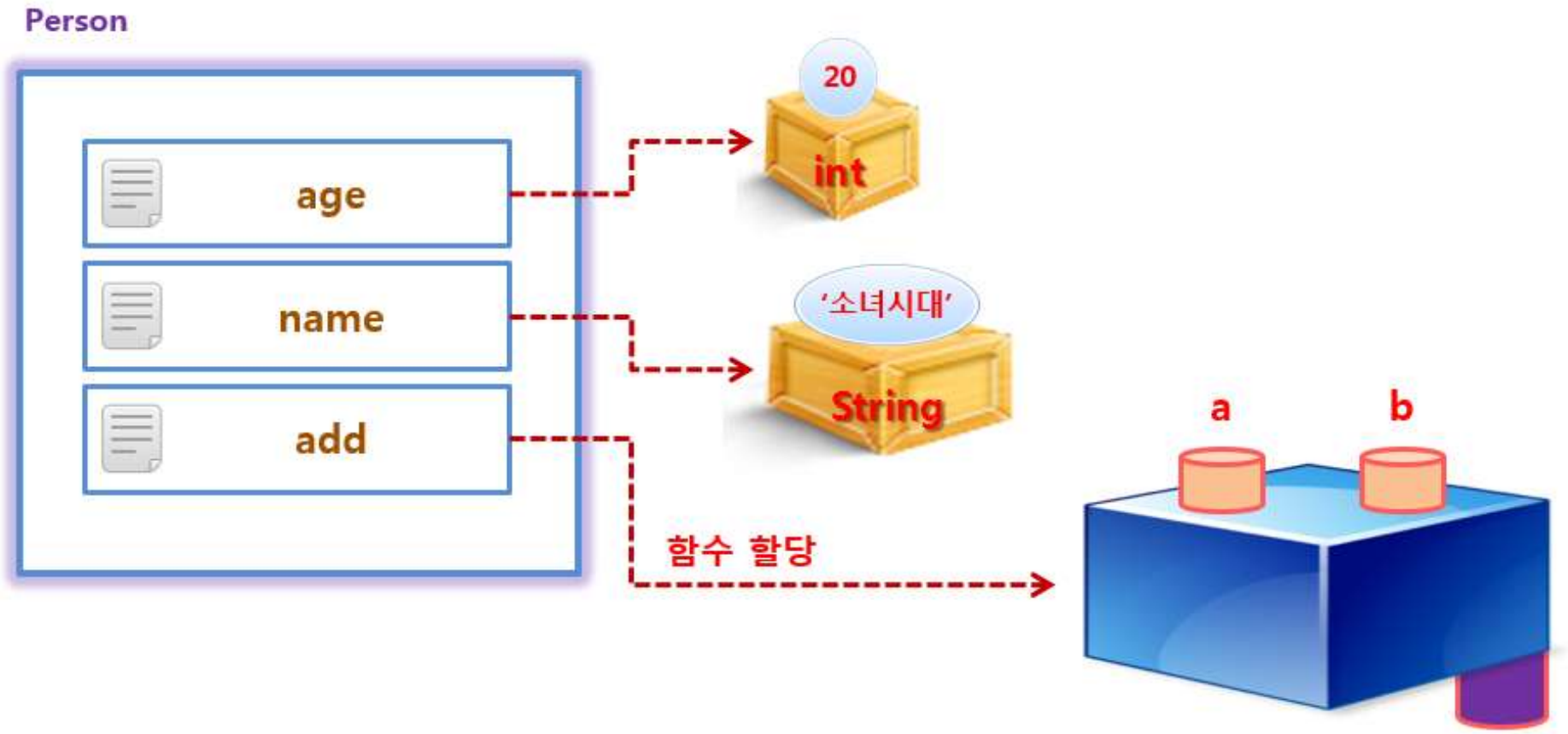
- 변수 이름으로 호출 가능

ch03_test04.js

```
var add = function (a, b) {
    return a + b;
};
var result = add(10, 10);
console.log('더하기 (10, 10) : %d', result);
```

객체의 속성으로 함수 할당하기

- 객체의 속성도 변수처럼 처리되므로 함수 할당 가능



객체의 속성에 함수 할당

ch03_test05.js

```
var Person = {};
Person["age"] = 20;
Person["name"] = '소녀시대';
Person.add = function(a, b) {
    return a + b;
};
console.log('더하기 : %d', Person.add(10, 10));
```

객체를 만들 때 속성 할당하기

ch03_test07.js

```
var Person = {
    age: 20,
    name: '소녀시대',
    add: function(a, b) {
        return a + b;
    }
};
console.log('더하기 : %d', Person.add(10, 10));
```

{}를 이용하여 선언과 동시에
속성 정의 및 초기화하는 경우

함수정의에 있어 네 가지 주의점

return명령의 중간에 개행하지 말것

- JavaScript
  - 세미콜론으로 문장의 끝 인식
  - 세미콜론을 생략했을 경우
    - 적당히 앞뒤의 문맥으로부터 문장의 끝 판단
- return 명령 중간에 라인 바꾸기
  - 불필요한 에러 유발 가능
  - 예)

```
var triangle = function(base, height) {
                return
                base * height / 2;
};
document.writeln('삼각형의 면적 : ' + triangle(5,2));
```

다음과 같이 인식

```
return;
base * height / 2;
```

함수는 데이터형의 일종임

- 함수는 객체임
  - 함수의 변수 대입, 다른 함수의 인수, 반환값으로서의 함수 ⇨ 모두 가능함
- 함수 리터럴로 정의된 변수는 값의 수정이 가능함
- 예)
  - triangle 변수에 함수형의 리터럴 저장
  - 1에서 변수 triangle에 재차 수치형의 값을 저장한다 하더라도 틀린 것이 아님
  - 수치형으로 고쳐 쓸 수 있는 변수를 참조하고 있는 2의 코드도 올바른 것이 됨

```
var triangle = function(base, height) {
                    return base * height / 2;
};


document.writeln(triangle(5,2));  //5
triangle = 0;     // ← ①
document.writeln(triangle);  // 0 ← ②
```

함수는 데이터형의 일종임

```
var triangle = function(base, height) {
        return base * height / 2;
};
document.writeln(triangle);
```


```
var triangle = function(base, height) {
        return base * height / 2;
};
```

- triangle을 변수로서 참조
- triangle에 저장된 함수 정의가 그대로 문자열로 출력
  - 엄밀하게는 Function 객체의 toString 메소드가 호출되어 문자열 표현으로 변환된 것이 출력

function 명령은 정적인 구조 선언

- function 명령
  - 정적인 구조를 선언하기 위한 키워드
  - 코드를 해석/컴파일하는 타이밍에 함수를 등록
  - 실행 시에는 이미 코드 내에 있는 구조의 일부분 ⇨ 함수를 어디에서라도 호출 가능함
  - 주의
    - 함수를 정의한 스크립트 블록 (<script> 태그)는 호출 측의 스크립트 블록보다 앞에 혹은 동일한 스크립트 블록에 기술해야만 함
    - 브라우저는 script 태그 단위로, 순서대로 스크립트를 처리해 나가기 때문임
- 예)

```
document.writeln('삼각형의 면적 : ' + triangle(5,2));
function triangle(base, height) {
        return base * height / 2;
}
```

함수 리터럴/Function 생성자는 실행시에 판단됨

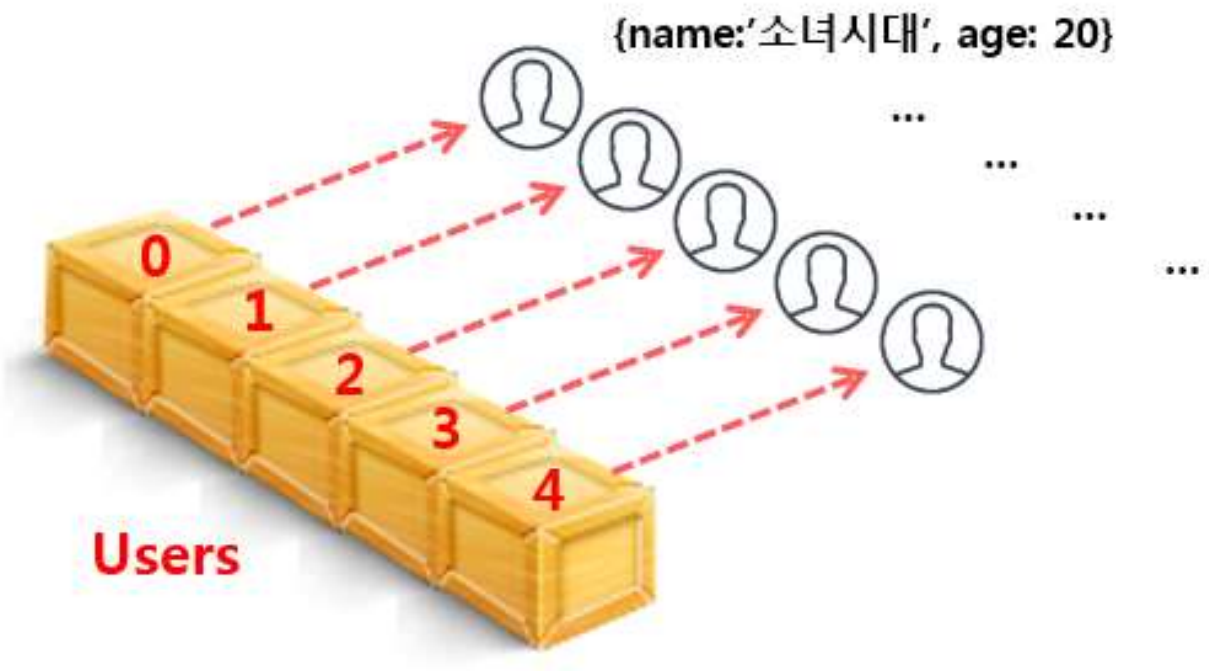
- 함수 리터럴/Function 생성자
  - 실행 시(대입 시)에 판단됨
  - 함수 리터럴/Function 생성자 함수 정의문 ⇨ 호출원의 코드보다 먼저 기술해야 함

```
document.writeln('삼각형의 면적 : ' + triangle(5,2)); //ERROR


var triangle = function(base, height) {
                return base * height / 2;
};
```

03-02 배열 이해하기

배열이 만들어지는 모양

- 배열
  - 여러 개의 데이터를 하나의 변수에 담아둘 수 있는 방법
- 배열의 요소는 대괄호를 이용해 접근할 수 있음
  - 0부터 시작하는 index 사용

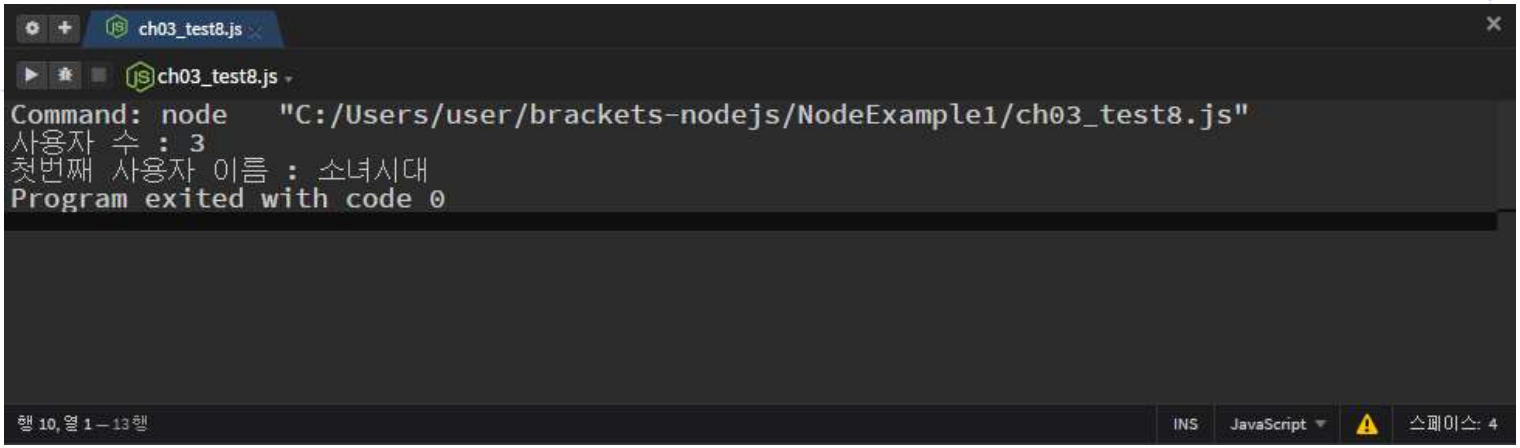


배열에 원소 추가

- push 함수를 호출하여 배열의 마지막에 원소를 추가할 수 있음

ch03_test08.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}];
Users.push({name:'티아라', age:23});
console.log('사용자 수 : %d', Users.length);
console.log('첫 번째 사용자 이름 : %s', Users[0].name);
```

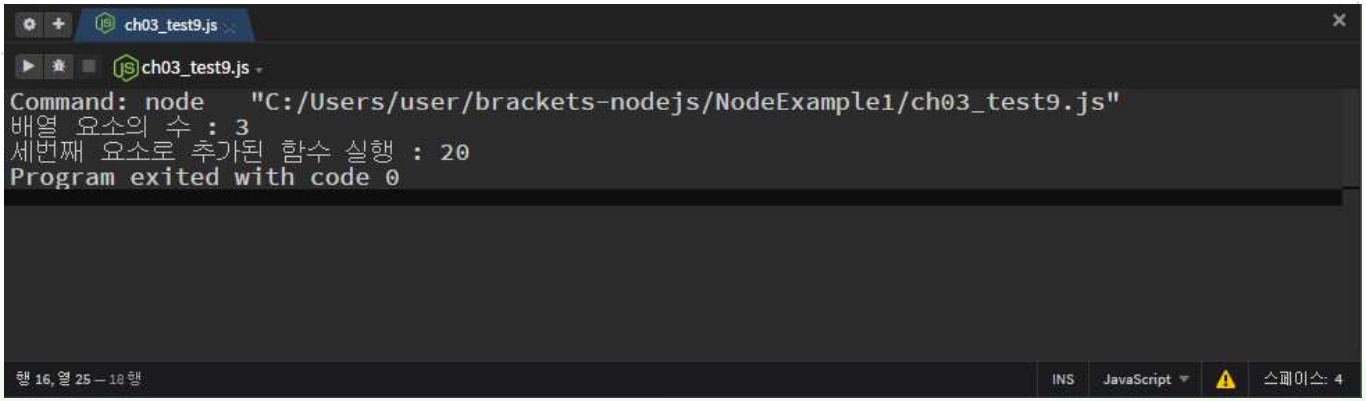


배열 원소로 함수 추가

- **변수의 자료형과 상관없이 배열에 추가 가능**

ch03_test09.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}];
var add = function(a, b) {
    return a + b;
};
Users.push(add)
console.log('배열 요소의 수 : %d', Users.length);
console.log('세 번째 요소로 추가된 함수 실행 : %d', Users[2](10, 10));
```

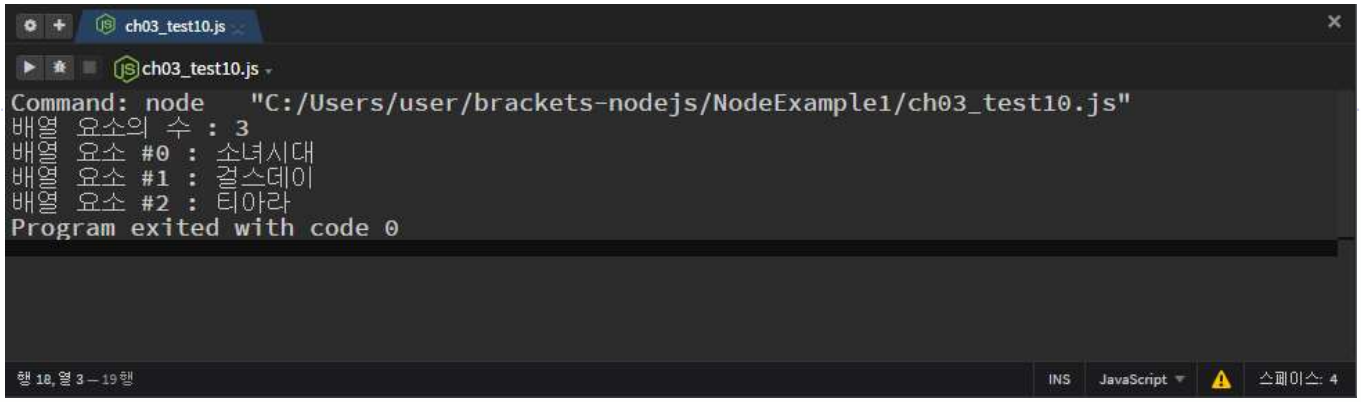


배열의 원소를 하나씩 확인하기

- **for 문에서 index를 사용하는 방법**

ch03_test10.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22},
{name:'티아라', age:23}];
console.log('배열 요소의 수 : %d', Users.length);

for (var i = 0; i < Users.length; i++) {
    console.log('배열 요소 #' + i + ' : %s', Users[i].name);
}
```



forEach 구문 사용하기

- **forEach 구문을 이용해 배열 원소를 하나씩 확인할 수 있음**

ch03_test10.js - 추가코드

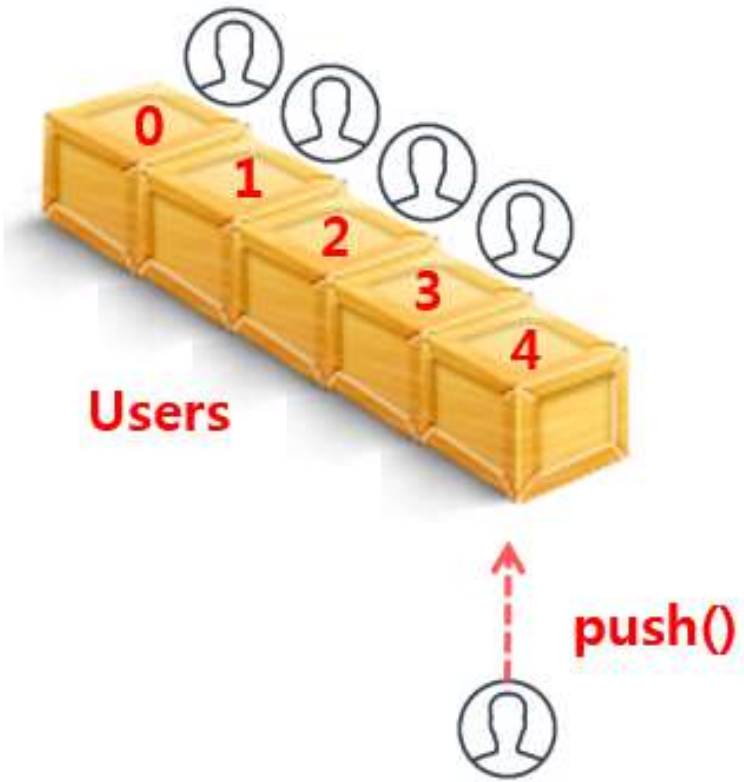
```
console.log('₩nforEach 구문 사용하기');
Users.forEach(function(item, index) {
    console.log('배열 요소 #' + index + ' : %s', item.name);
});
```

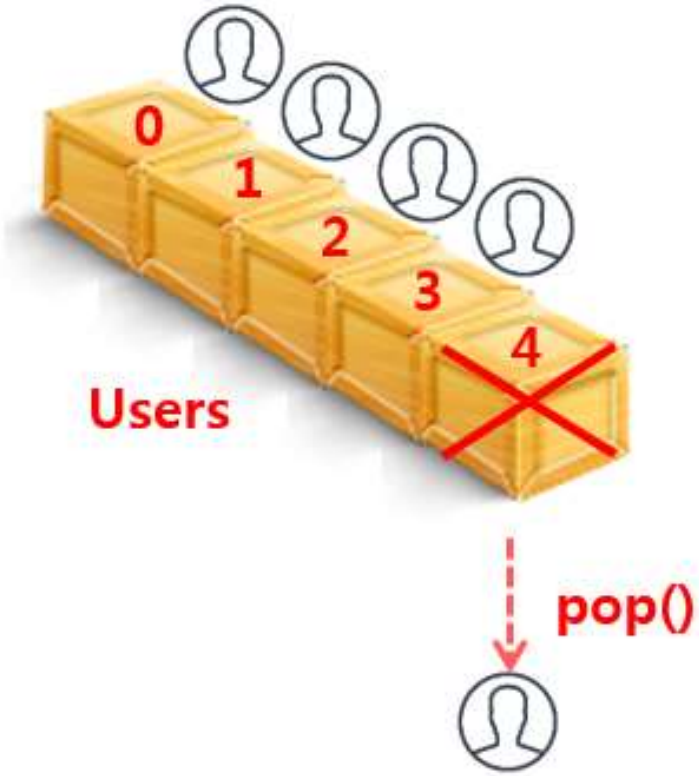
배열에 값 추가 및 삭제하기

- 배열의 끝에 원소를 추가하거나 삭제할 때 : push, pop
- 배열의 앞에 원소를 추가하거나 삭제할 때 : unshift, shift
- 여러 개의 원소를 한꺼번에 추가하거나 삭제할 때 : splice

속성 / 메소드 이름	설명
push(object)	배열의 끝에 요소를 추가합니다.
pop( )	배열의 끝에 있는 요소를 삭제합니다.
unshift( )	배열의 앞에 요소를 추가합니다.
shift( )	배열의 앞에 있는 요소를 삭제합니다.
splice(index, removeCount [,object])	여러 개의 객체를 요소로 추가하거나 삭제합니다.
slice(index, copyCount)	여러 개의 요소를 잘라내어 새로운 배열 객체로 만듭니다.

push와 pop





unshift와 shift

delete

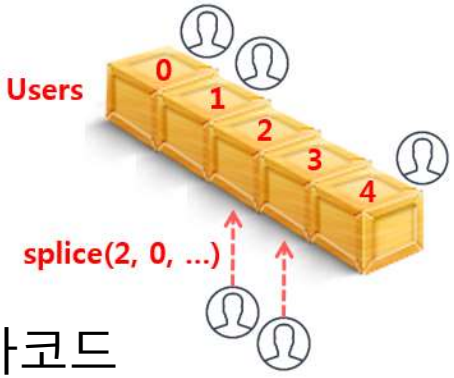
- **delete**
  - **배열에서 지정 요소를 삭제함**
  - **단, 저장 공간은 그대로 남아있음**

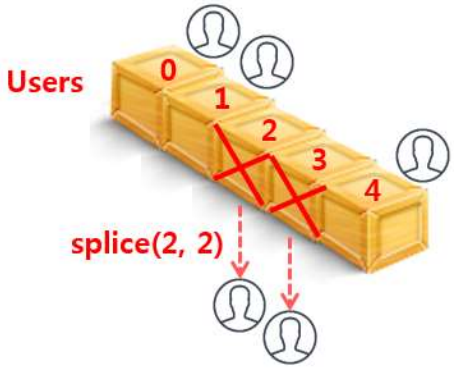
ch03_test13.js

```
var Users = [{name:'소녀시대',
age:20},{name:'걸스데이',age:22},{name:'티아라',age:23}];
console.log('delete 삭제 전 배열 요소 수: %d', Users.length);

delete Users[1];
console.log('delete 삭제 후');
console.dir(Users);
```

splice

- splice
  - 1st 파라미터: 인덱스 값으로 배열의 몇 번째 원소부터 처리할 것인지 지정
  - 2nd 파라미터: 삭제할 요소의 개수
    - 요소를 추가할 경우에는 0 으로 지정

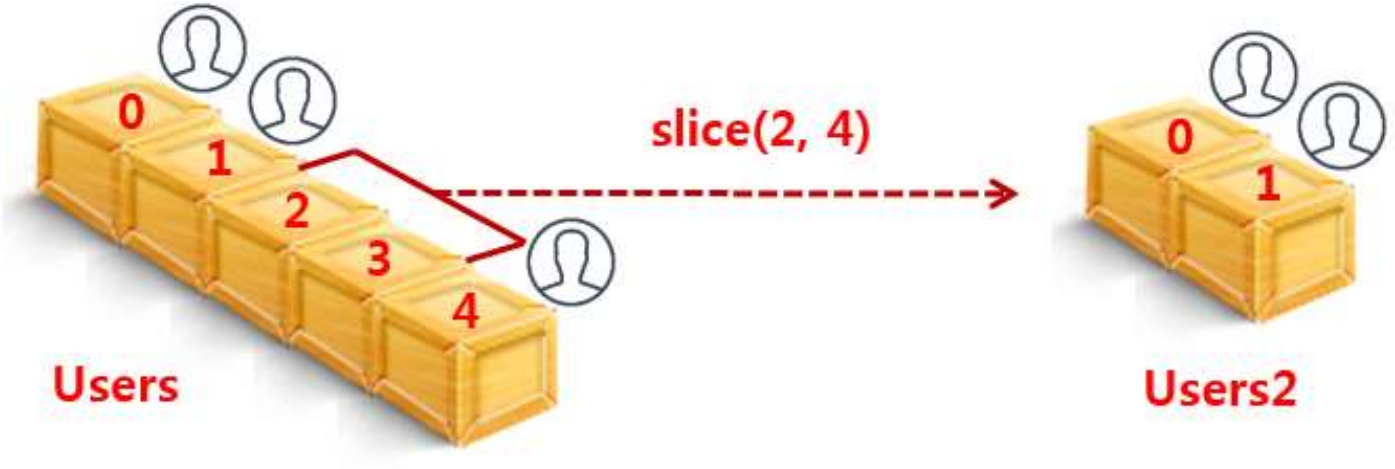




ch03_test13.js - 추가코드

```
Users.splice(1, 0, {name:'애프터스쿨', age:25});
console.log('splice()로 요소를 인덱스 1에 추가한 후');
console.dir(Users);
Users.splice(2, 1);
console.log('splice()로 인덱스 1의 요소를 1개 삭제한 후');
console.dir(Users);
```

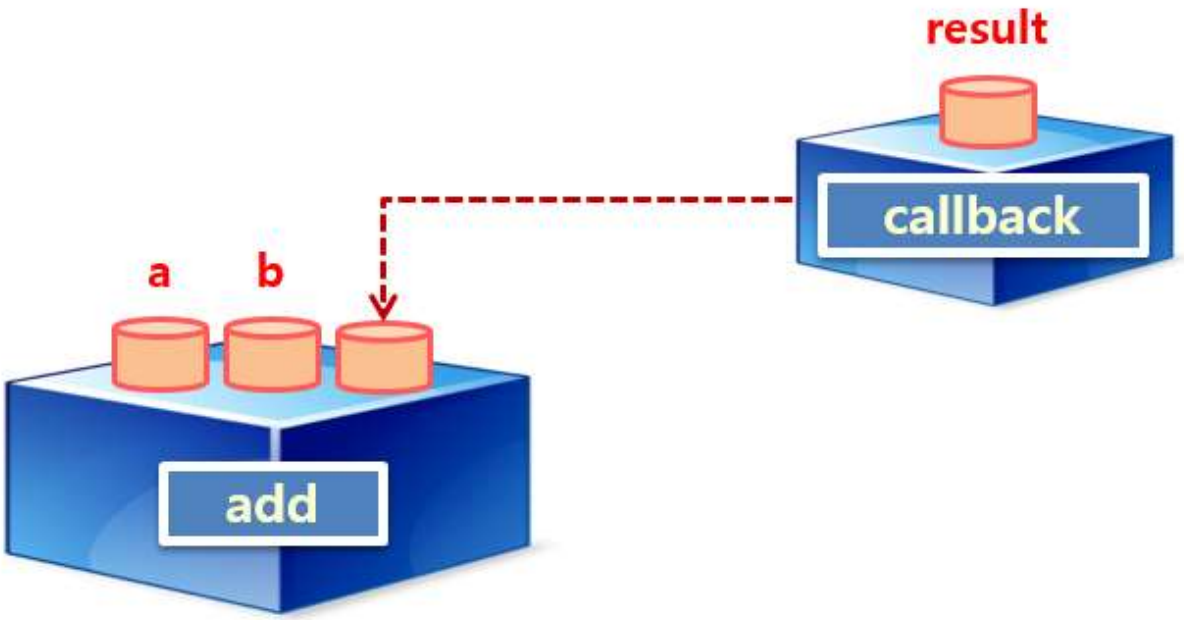
- slice
  - 배열의 일부 요소를 복사하여 새로운 배열을 만듬
  - 1st 파라미터: 복사할 요소의 시작 위치
  - 2nd 파라미터: 복사할 마지막 요소의 다음 위치



03-03 콜백함수 이해하기

함수를 파라미터로 전달하기

- 변수에 함수를 할당할 수 있으므로 함수를 호출할 때 파라미터로 다른 함수 전달 가능



콜백함수를 파라미터로 전달

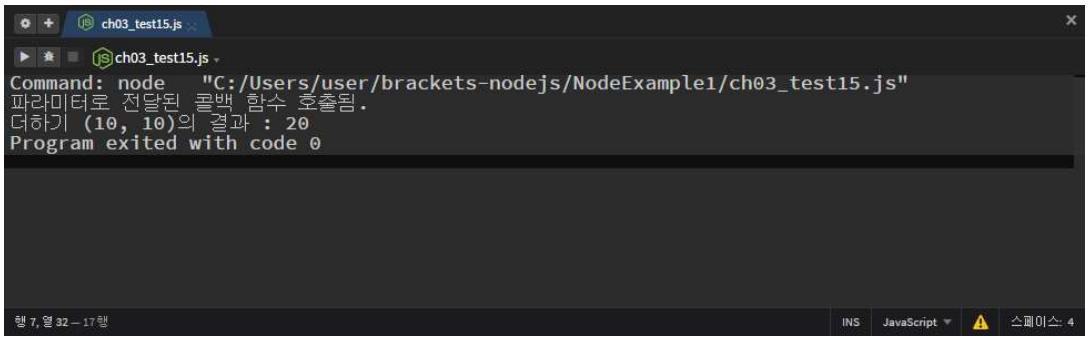
```
function add( a, b, callback) {
    var result = a + b;
    callback(result);
}
```

콜백 함수를 파라미터로 전달하기

- 콜백함수(callback function)
  - 함수를 파라미터로 전달했을 때 특정 시점에 그 함수를 실행시켜 주는 경우
  - 비동기 프로그래밍에서 주로 사용
  - 함수가 실행되고 있는 중간에 호출되어 상태 정보를 전달하거나 결과값을 처리하는데 사용

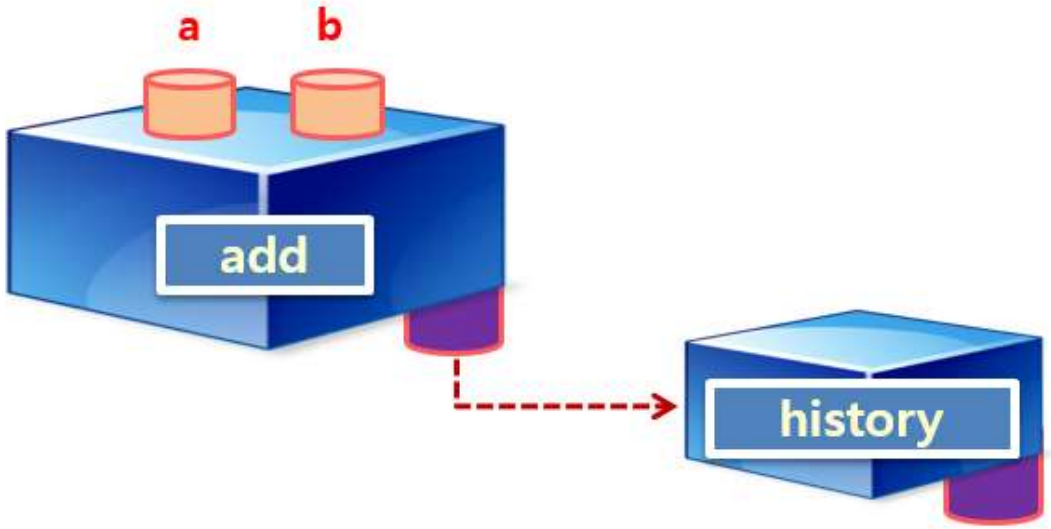
ch03_test15.js

```
function add(a, b, callback) {
    var result = a + b;
    callback(result);
}

add(10, 10, function(result) {
    console.log('파라미터로 전달된 콜백 함수 호출됨.');
    console.log('더하기 (10, 10)의 결과 : %d', result);
});
```



함수에서 반환하는 값이 함수인 경우

- 함수의 결과를 반환할 때 함수를 반환할 수 있음



함수를 리턴

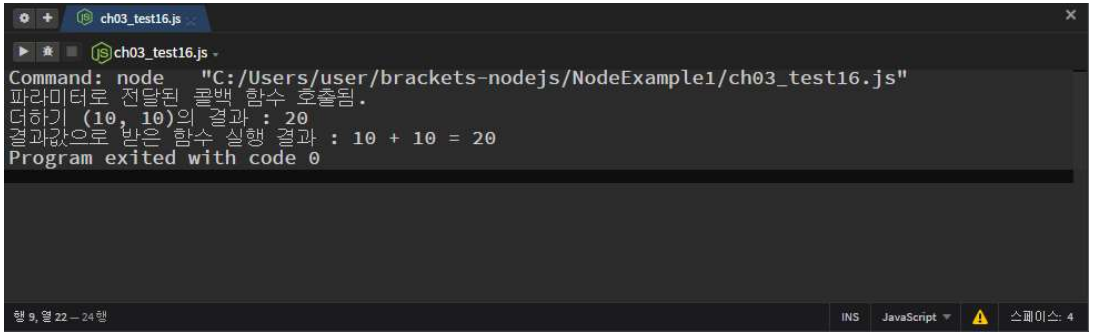
```
function add( a, b, callback) {
    var result = a + b;
    callback(result);

    var history = function() {
        return a + ' + ' + b + ' = ' + result;
    };
    return history;
}
```

더하기 함수를 실행했을 때 기록을 남겨두었다가 출력하기

- 함수를 실행했을 때 함수를 반환 받고 반환된 함수를 실행하여 결과를 확인하는 방법

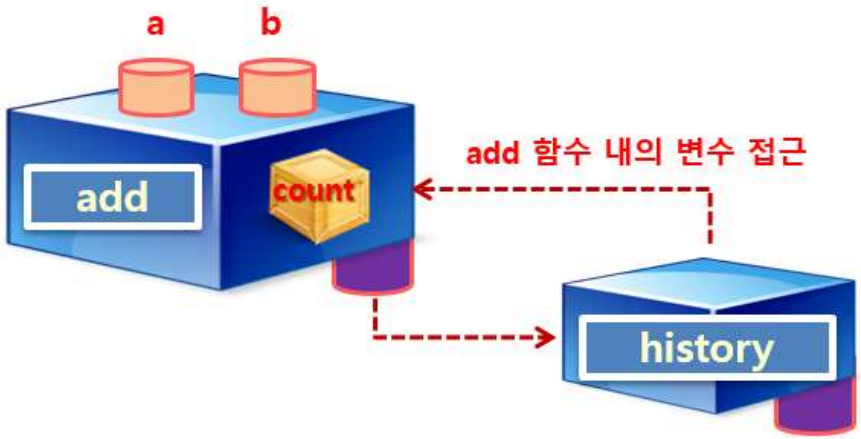
ch03_test14.js

```
function add(a, b, callback) {
    var result = a + b;
    callback(result);
    var history = function() {
        return a + ' + ' + b + ' = ' + result;
    };
    return history;
}
var add_history = add(10, 10, function(result) {
    console.log('파라미터로 전달된 콜백 함수 호출됨.');
    console.log('더하기 (10, 10)의 결과 : %d', result);
});
console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
```



함수에서 반환된 함수 안에서 변수에 접근하는 경우

- **closure**
  - **처음 실행한 함수 안에서 접근하던 변수는 반환된 함수에서 계속 접근 가능**
  - **이미 생명 주기가 끝난 외부 함수의 변수를 참조하는 함수**




```
function add( a, b, callback) {
  var count = 0;

    var history = function() {
       count++;
       return count + ' : ' + a + ' + ' + b + ' = ' + result;
    };
    return history;
}
```

closure에서 사용할 경우에 free variable이 됨

inner function

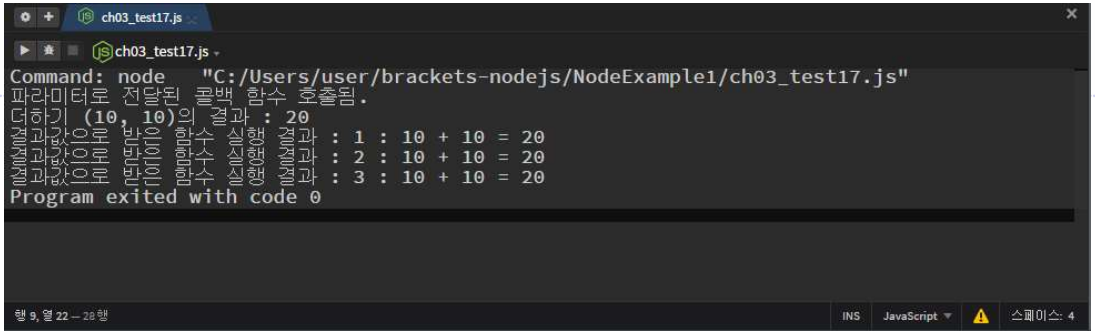
반환된 함수에서 함수 내부의 변수 접근

- **반환된 함수에서 이 함수를 반환했던 함수 내부의 변수를 접근하는 방법**

ch03_test17.js

```
function add(a, b, callback) {
    var result = a + b;
    callback(result);
    var count = 0;
    var history = function() {
        count++;
        return count + ' : ' + a + ' + ' + b + ' = ' + result;
    };
    return history;
}
var add_history = add(10, 10, function(result) {
    console.log('파라미터로 전달된 콜백 함수 호출됨.');
    console.log('더하기 (10, 10)의 결과 : %d', result);
});
console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
```

계속 참조함



03-04 프로토타입 객체 만들기

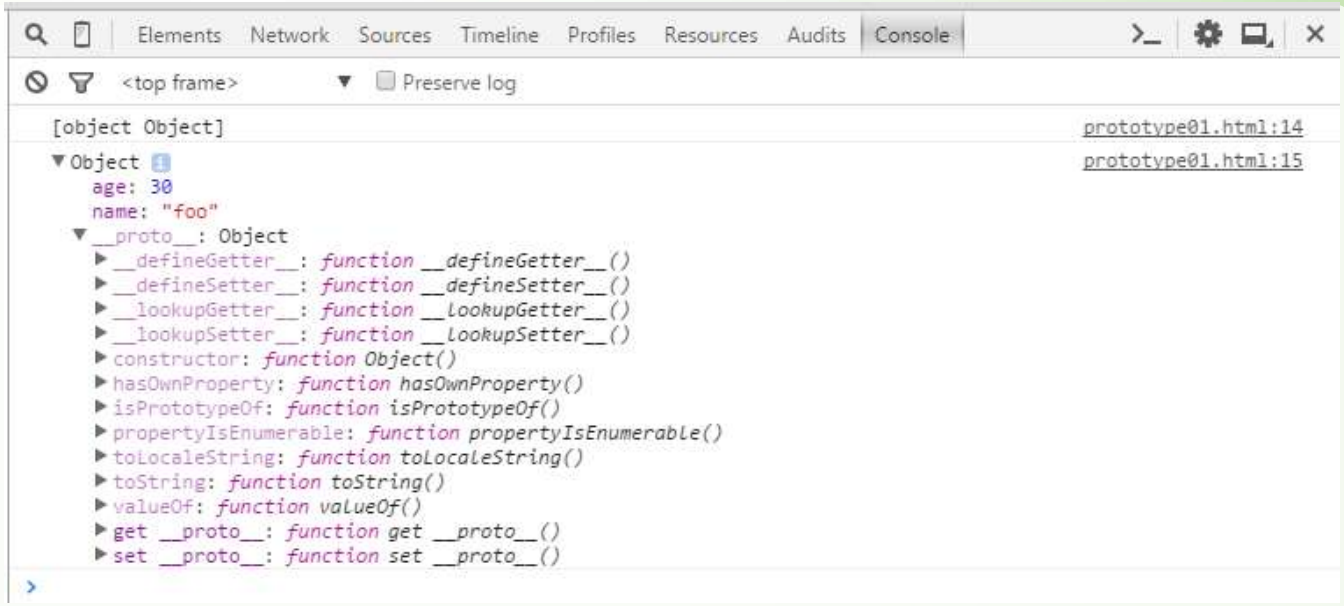
자바스크립트에서 객체지향 방식으로 만들기

- 프로토타입(prototype) 객체
  - 객체의 원형을 가리킴
  - 프로토타입이 클래스(Class)의 역할을 함
- 객체의 속성
  - _proto_ 속성
    - 현 객체의 prototype 객체를 가리킴
    - [[Prototype]] 히든 속성
  - prototype 속성
    - 생성자 함수 객체에 정의되어 있고 함수의 prototype 객체를 가리킴
- 객체 생성 방식에 따른 prototype 객체
  - object literal 방식을 생성된 객체
    - _proto_ 속성이 Object(Object생성자 함수의 prototype 객체)으로 됨
  - 생성자 함수를 이용하여 생성된 객체
    - _proto_ 속성이 생성자 함수의 prototype 객체로 되어 있음

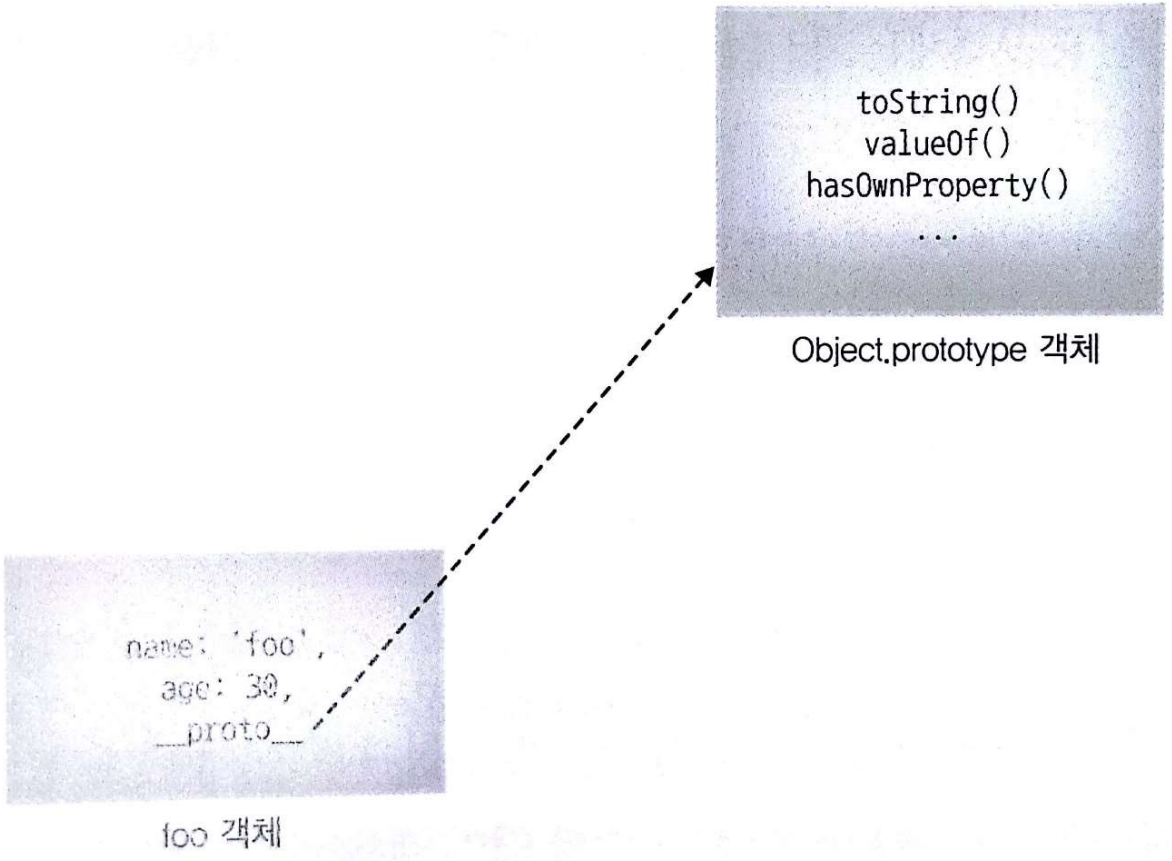
프로토타입 객체(prototype object)

- 프로토타입
  - 자바스크립트의 모든 객체는 자신의 부모 역할을 하는 객체와 연결되어 있음.
  - 프로토타입 객체 or 프로토타입

```
var foo = {
                name: 'foo';
                age: 30;
};

console.log(foo.toString());
console.dir(foo);
```



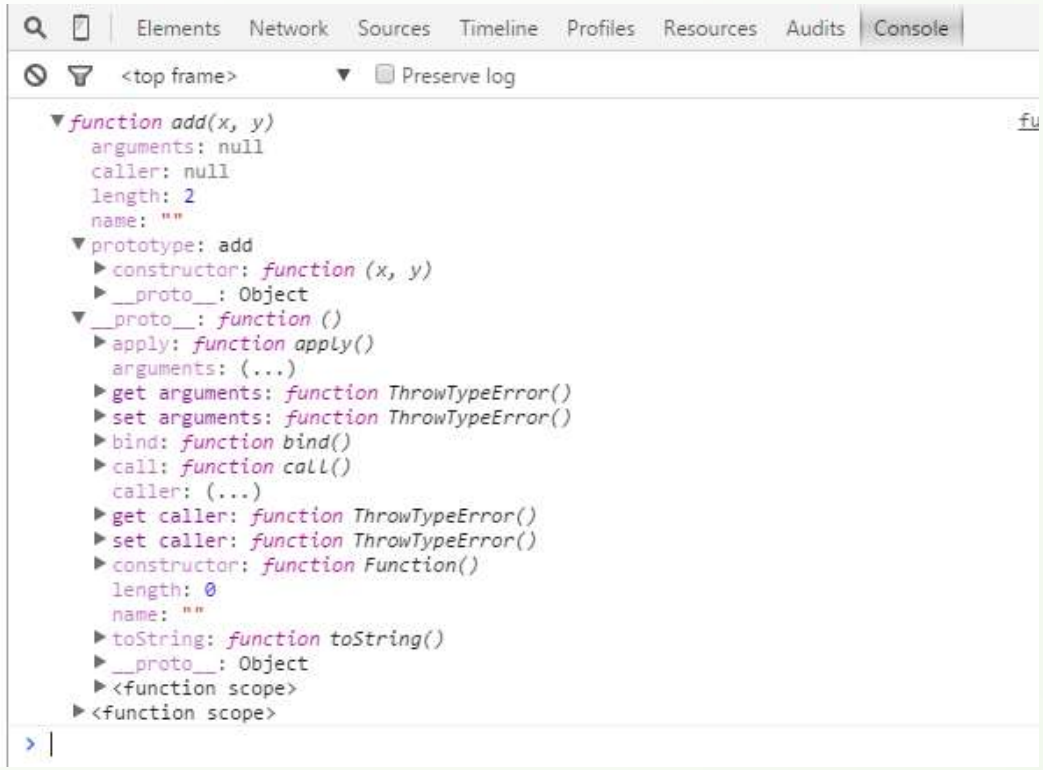
- _proto_ 프로퍼티
  - ECMAScript 명세
    - 모든 객체는 자신의 프로토타입을 가리키는 [[Prototype]]라는 숨겨진 프로퍼티를 가진다
  - 크롬
    - _proto_
    - 부모 객체를 가리키는 프로퍼티
- 프로토타입 체이닝을 위한 프로퍼티


toString()
valueOf()
hasOwnProperty()
...
Object.prototype 객체
name: 'foo',
age: 30,
__proto__
foo 객체

function object

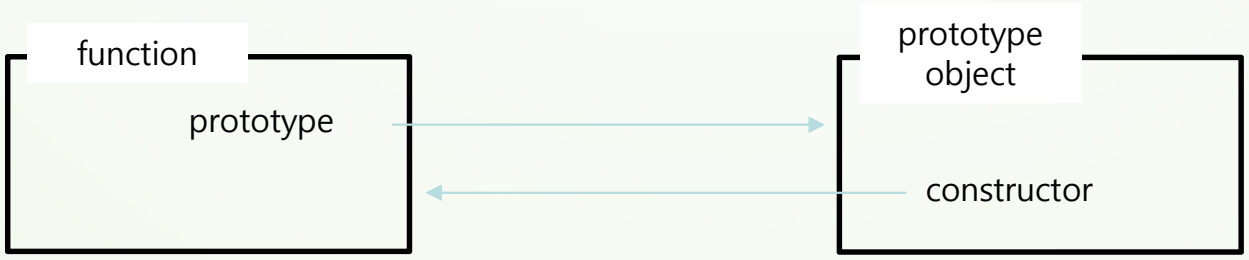
- 함수도 객체임
- 함수 객체만의 고유 프로퍼티가 있음
  - name: 함수의 이름
  - caller: 자신을 호출한 함수
  - _proto_: 자신의 프로토타입 객체를 나타냄
  - arguments
  - length
  - caller
  - prototype

```
var add = function (x,
y) {
    return x*y;
};

console.dir(add);
```

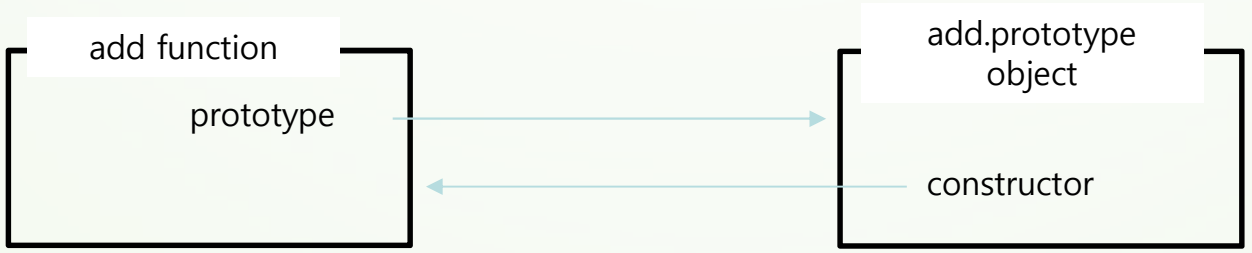


prototype 프로퍼티

- prototype 프로퍼티
  - [[Prototype]](_proto_) 과는 다른 프로퍼티임
  - 차이점
    - _proto_
      - 객체 입장에서 자신의 부모 역할을 하는 프로토타입 객체를 가리킴
    - prototype
      - 이 함수가 생성자로 사용될 때 이 함수를 통해 생성된 객체의 부모 역할을 하는 프로토타입 객체를 가리킴

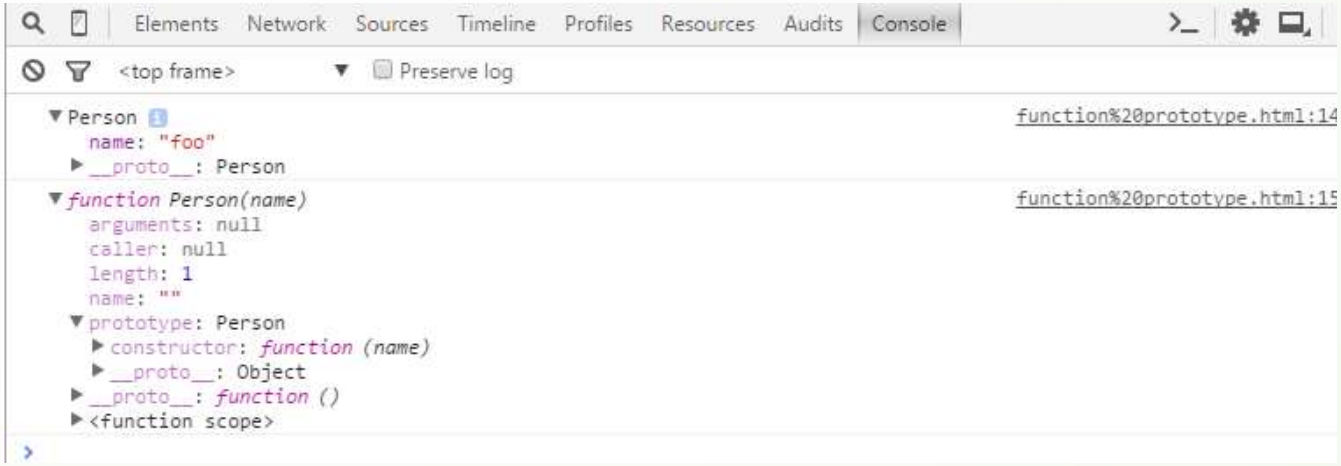


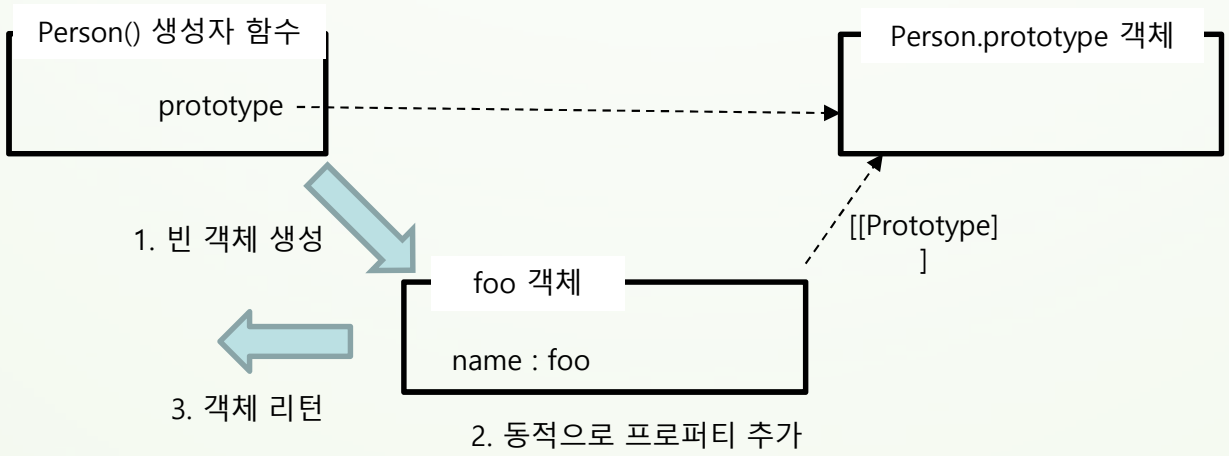
```
function add(x, y) {
    return x*y;
}

console.dir(add.prototype);
console.dir(add.prototype.constructor);
```



constructor 함수가 호출될 때

```
var Person = function (name) {
    this.name = name;
}

var foo = new Person('foo');
console.dir(foo);
console.dir(Person);
```

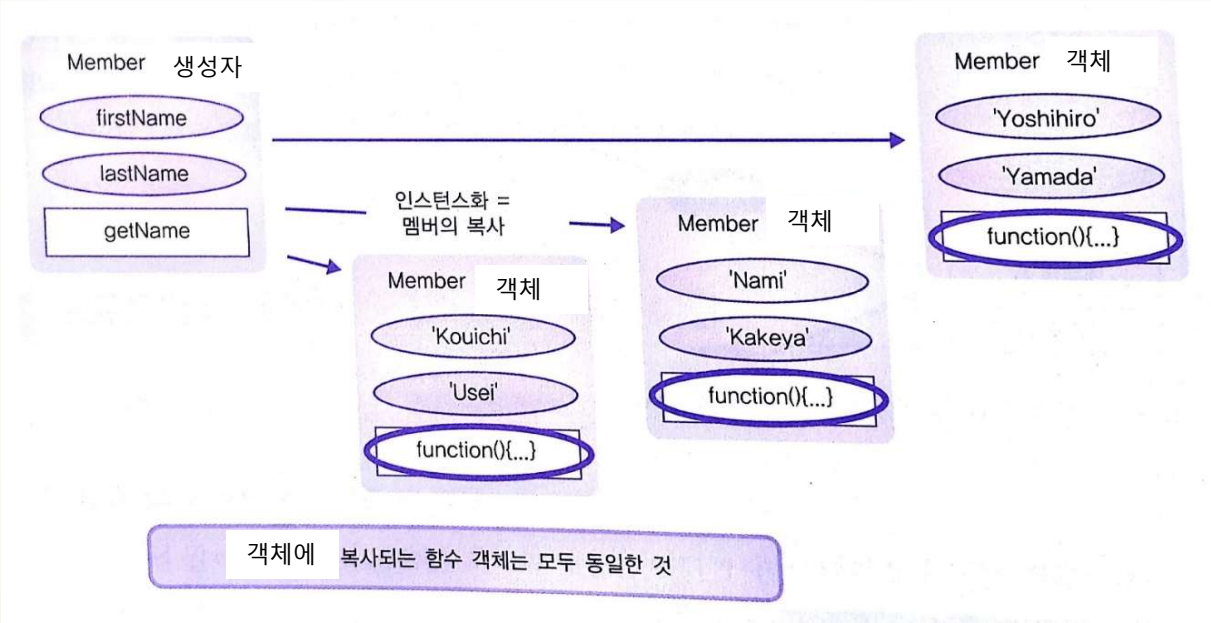




생성자의 문제점과 프로토타입

메소드는 프로토타입 객체에 추가한다

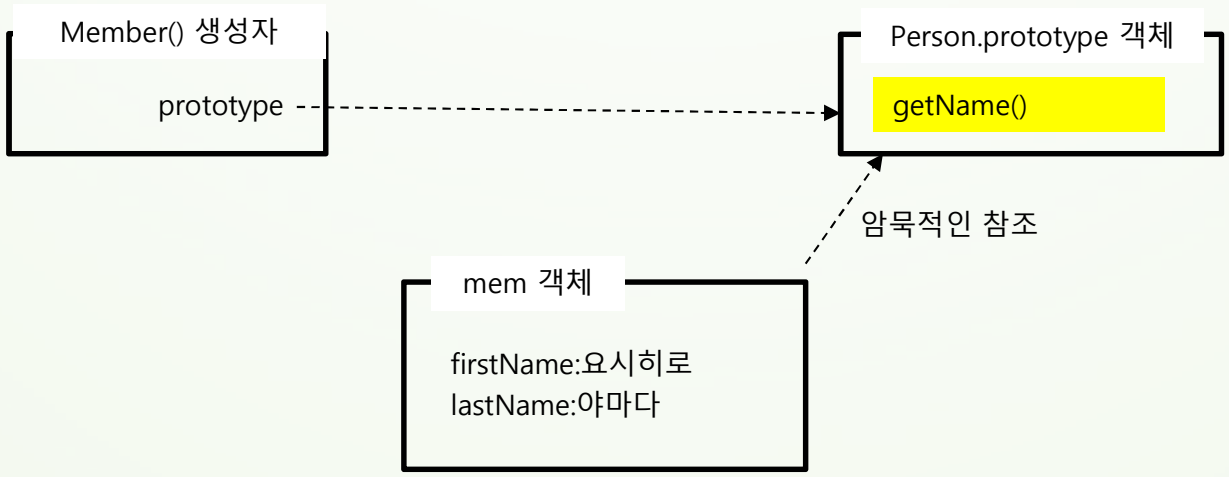
- 생성자 함수에 메소드 정의
  - 메소드의 수에 비례하여 메모리를 소비함
  - 생성자는 객체를 생성할 때 마다 각 메소드를 위한 메모리를 확보함



```
var Member = function(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
};

Member.prototype.getName = function(){
        return this.lastName + ' ' + this.firstName;
};

var mem = new Member('요시히로', '야마다');
document.writeln(mem.getName());    // 야마다 요시히로
```

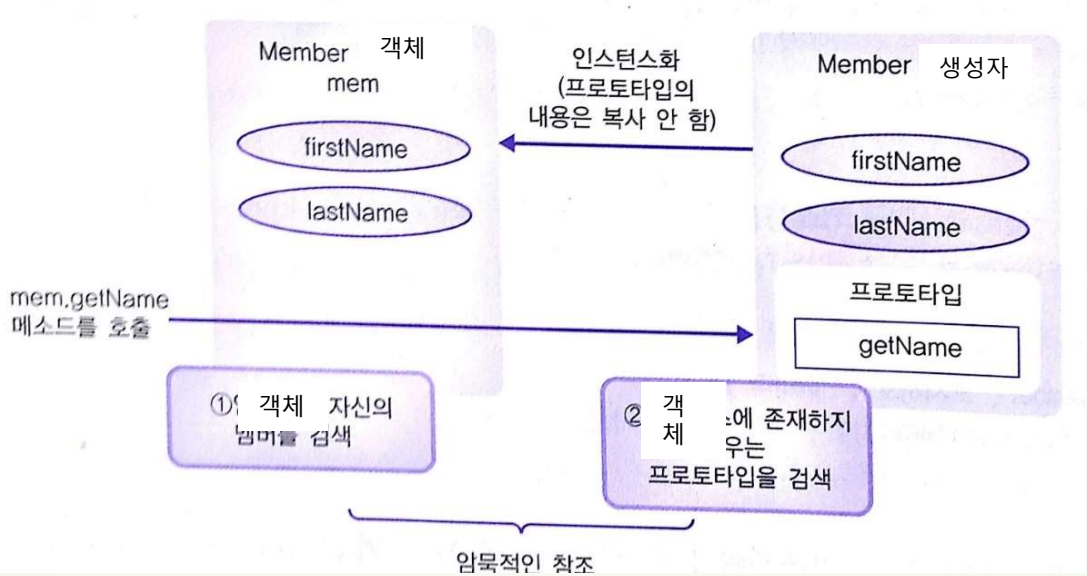
메소드는 프로토타입 객체에 추가한다

- 프로토타입 객체에 메소드 추가
  - 생성자의 prototype 객체에 대하여 암묵적인 참조를 함
  - prototype 객체에 추가된 메소드를 사용할 수 있음



프로토타입 객체를 사용하는 이점

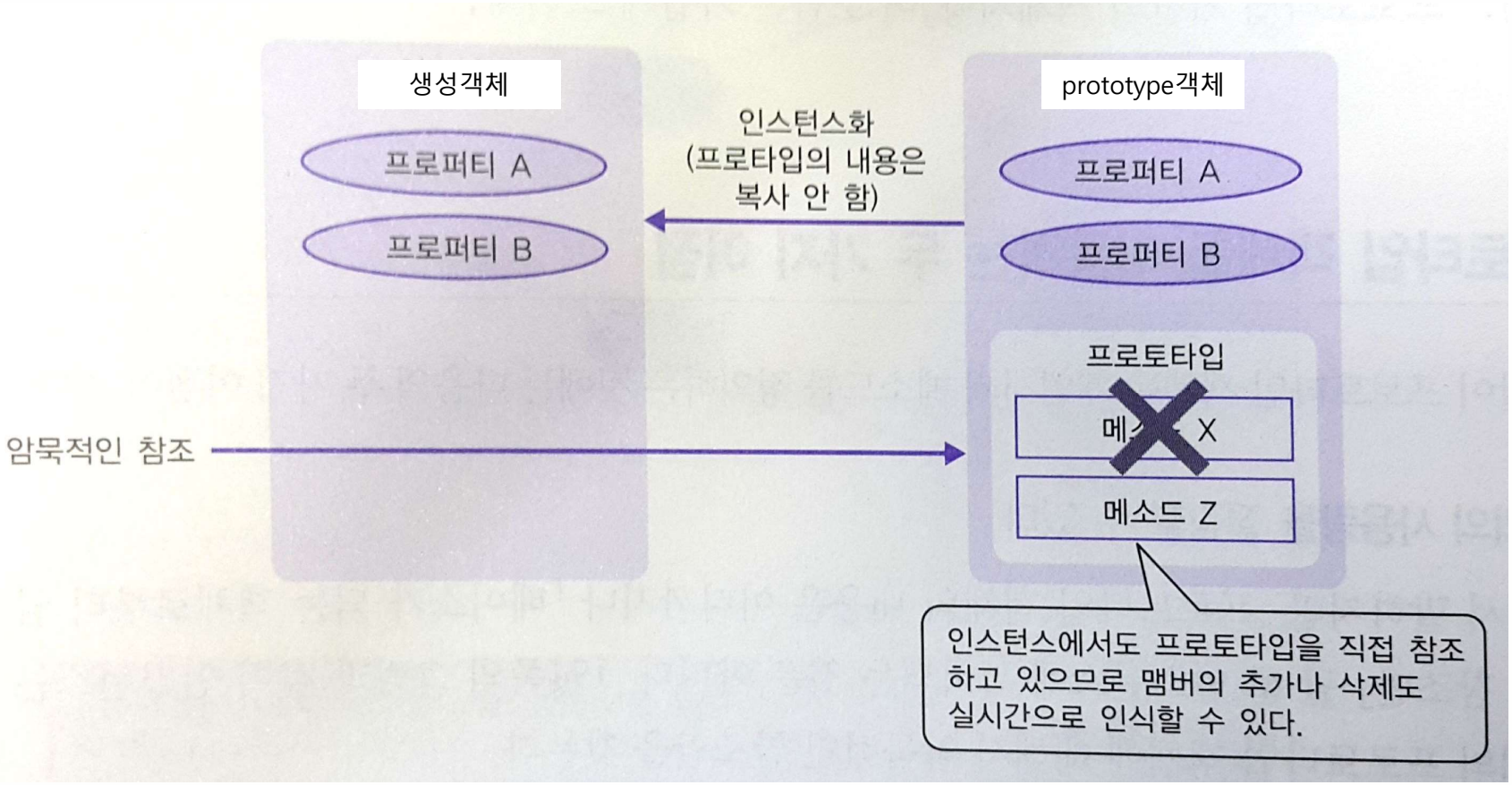
- 메모리의 사용량을 절감함
  - 프로토타입 객체의 내용은 객체로부터 암묵적으로 참조만 될 뿐 객체에 복사되는 것은 아님
  - 객체의 멤버가 호출되었을 때
    - 객체 측에 요구된 멤버가 존재하지 않는지를 확인
    - 존재하지 않는 경우는 암묵적인 참조를 통해 프로토타입 객체를 검색



프로토타입 객체를 사용하는 이점

- 프로퍼티의 추가나 변경을 객체가 실시간으로 인식함

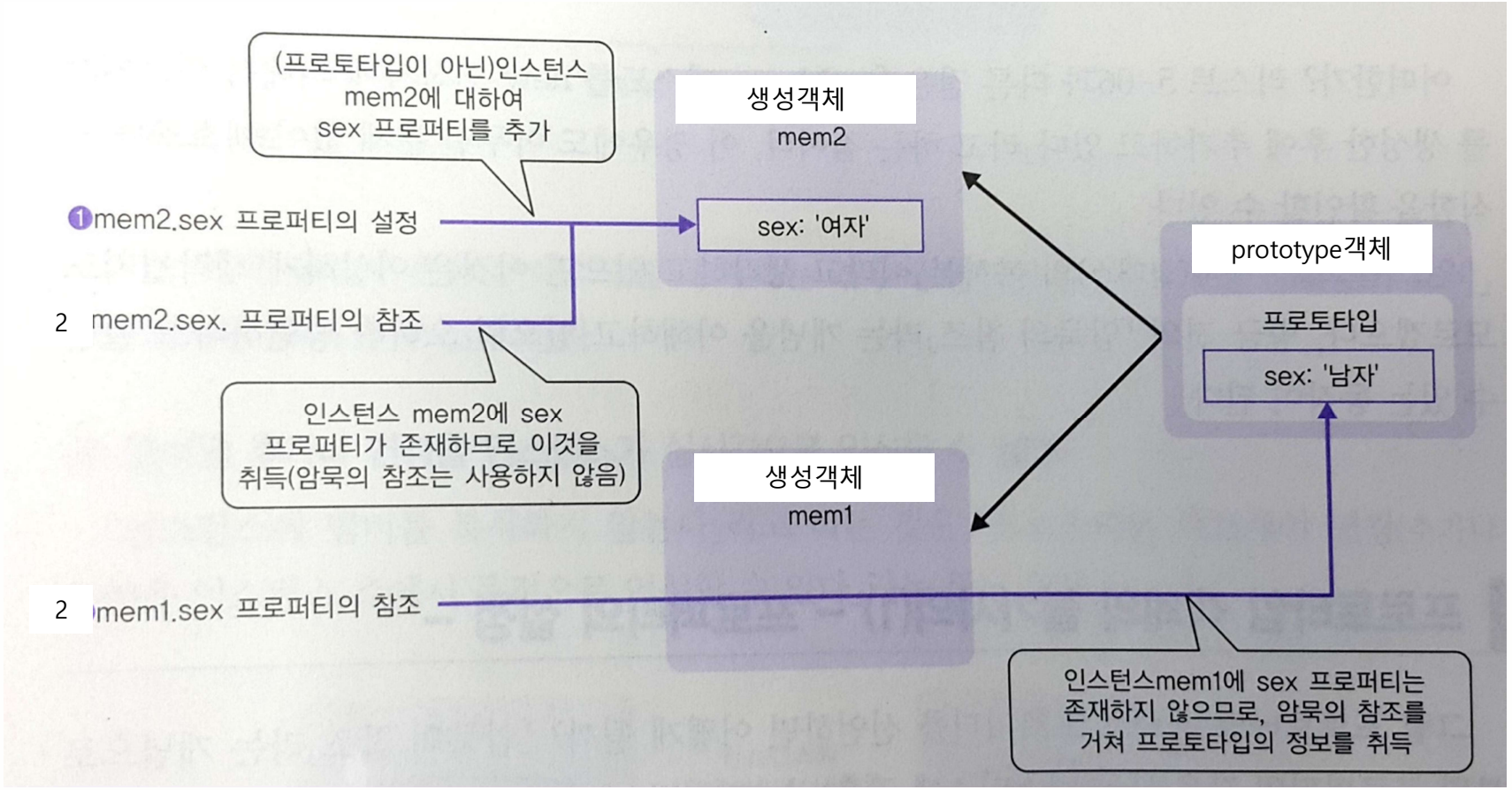
```
var Member = function(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
};

var mem = new Member('요시히로', '야마다');

Member.prototype.getName = function(){
        return this.lastName + ' ' + this.firstName;
};
document.writeln(mem.getName());  // 야마다 요시히로
```


생성객체
프로퍼티 A
프로퍼티 B
인스턴스화
(프로타입의 내용은
복사 안 함)
prototype객체
프로퍼티 A
프로퍼티 B
프로토타입
메소드 X
메소드 Z
암묵적인 참조
인스턴스에서도 프로토타입을 직접 참조
하고 있으므로 맴버의 추가나 삭제도
실시간으로 인식할 수 있다.

프로퍼티의 설정

- 프로토타입 객체의 사용
  - 값을 참조할 때 뿐임
- 객체에 대한 값의 설정은 해당 객체에 대하여 이루어짐
- 프로퍼티의 선언: 생성자
- 메소드의 선언: 프로토타입

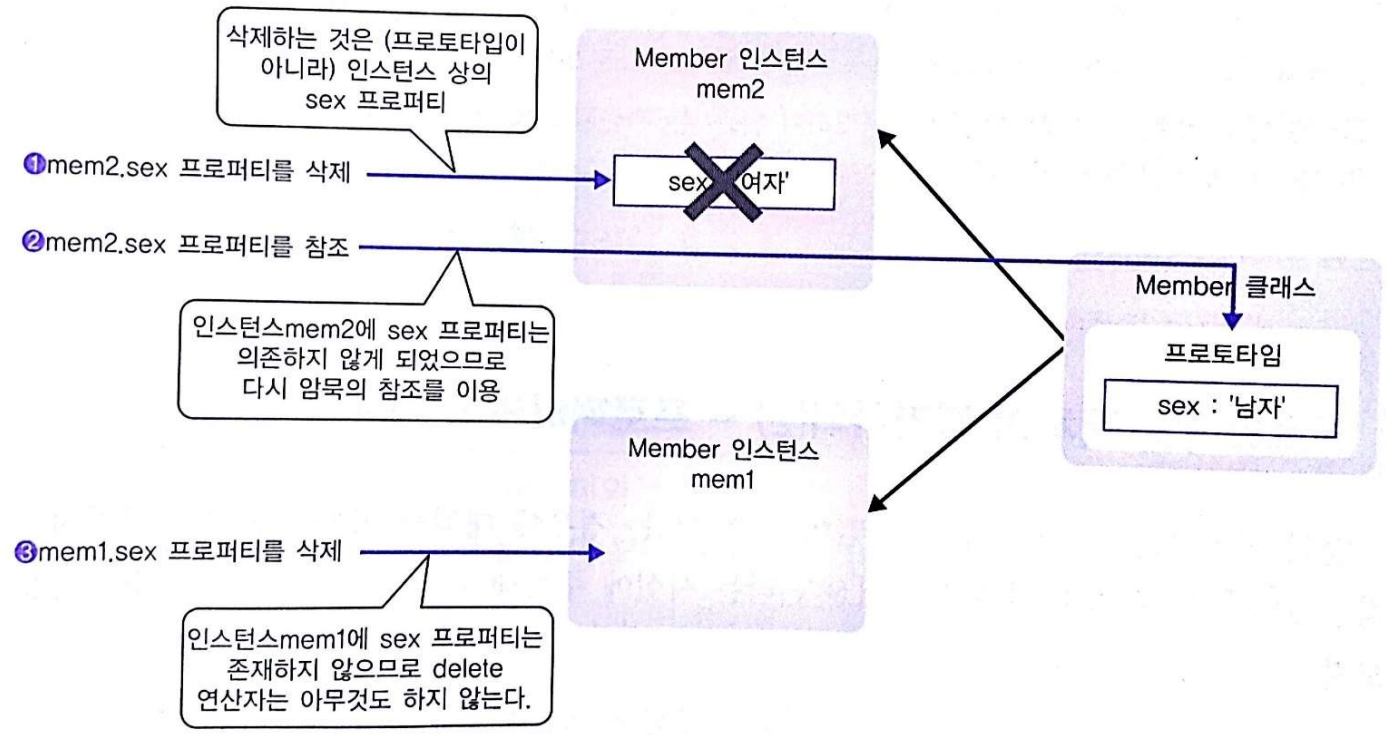
```
var Member = function(){ };
Member.prototype.sex = '남자';

var mem1 = new Member();
var mem2 = new Member();
document.writeln(mem1.sex + '|' + mem2.sex);     // 남자 | 남자
mem2.sex = '여자';   // ← 1
document.writeln(mem1.sex + '|' + mem2.sex);     // 남자 | 여자  ← 2
```


(프로토타입이 아닌)인스턴스
mem2에 대하여
sex 프로퍼티를 추가
생성객체
mem2
1 mem2.sex 프로퍼티의 설정
sex: '여자'
prototype객체
프로토타입
sex: '남자'
2 mem2.sex. 프로퍼티의 참조
인스턴스 mem2에 sex
프로퍼티가 존재하므로 이것을
취득(암묵의 참조는 사용하지 않음)
생성객체
mem1
2 mem1.sex 프로퍼티의 참조
인스턴스mem1에 sex 프로퍼티는
존재하지 않으므로, 암묵의 참조를
거쳐 프로토타입의 정보를 취득

프로퍼티의 삭제

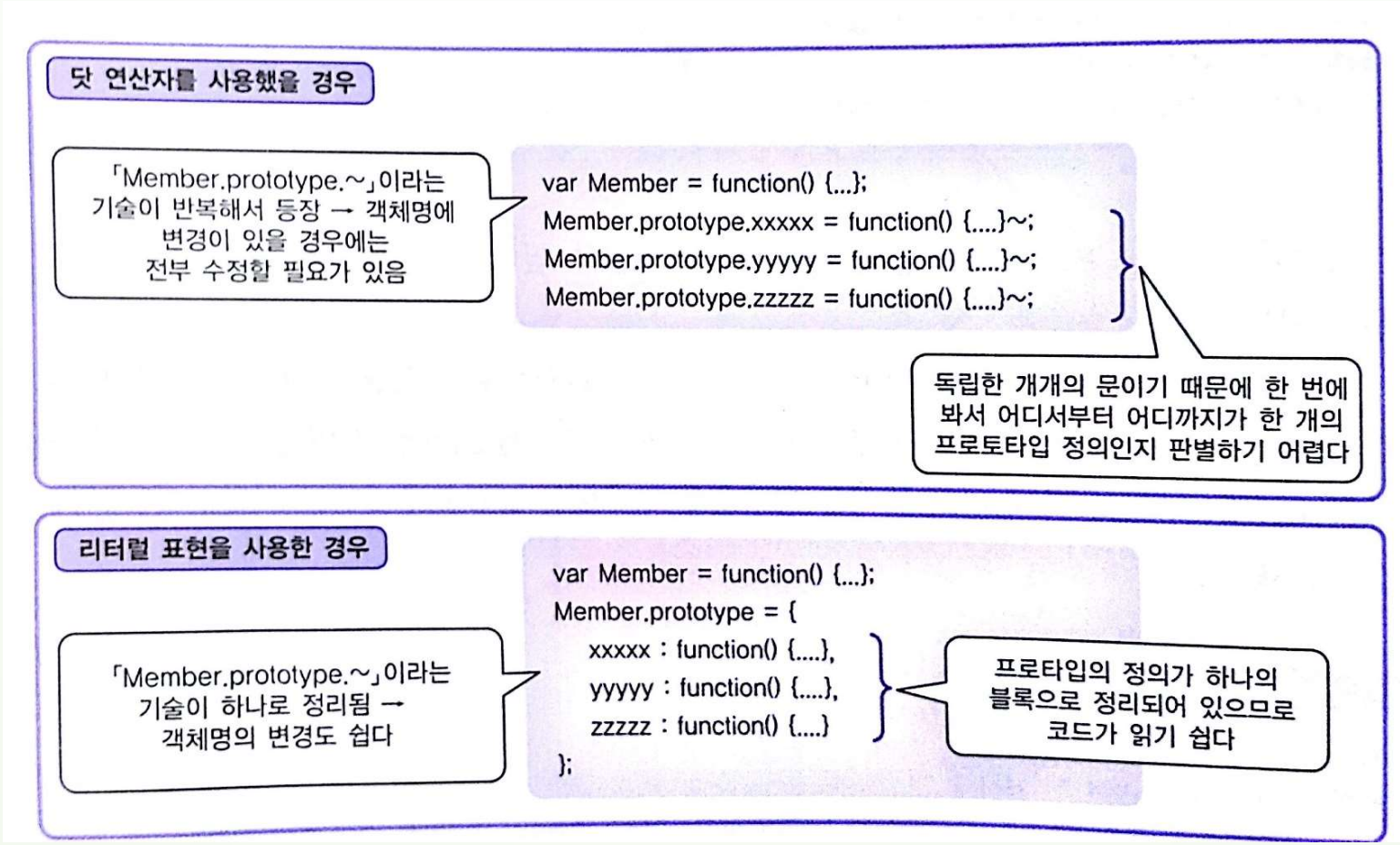
- 프로퍼티의 삭제는 객체 단위로 이루어짐

```
var Member = function(){ };

Member.prototype.sex = '남자';

var mem1 = new Member();
var mem2 = new Member();
document.writeln(mem1.sex + '|' + mem2.sex);    // 남자 | 남자
mem2.sex = '여자';
document.writeln(mem1.sex + '|' + mem2.sex);    // 남자 | 여자
delete mem1.sex   ← 1
delete mem2.sex   ← 2
document.writeln(mem1.sex + '|' + mem2.sex);    // 남자 | 남자  ← 3
```



⊙ 암묵의 참조(프로퍼티의 삭제)

객체 리터럴로 프로토타입 객체 정의하기



객체 리터럴로 프로토타입 객체 정의하기

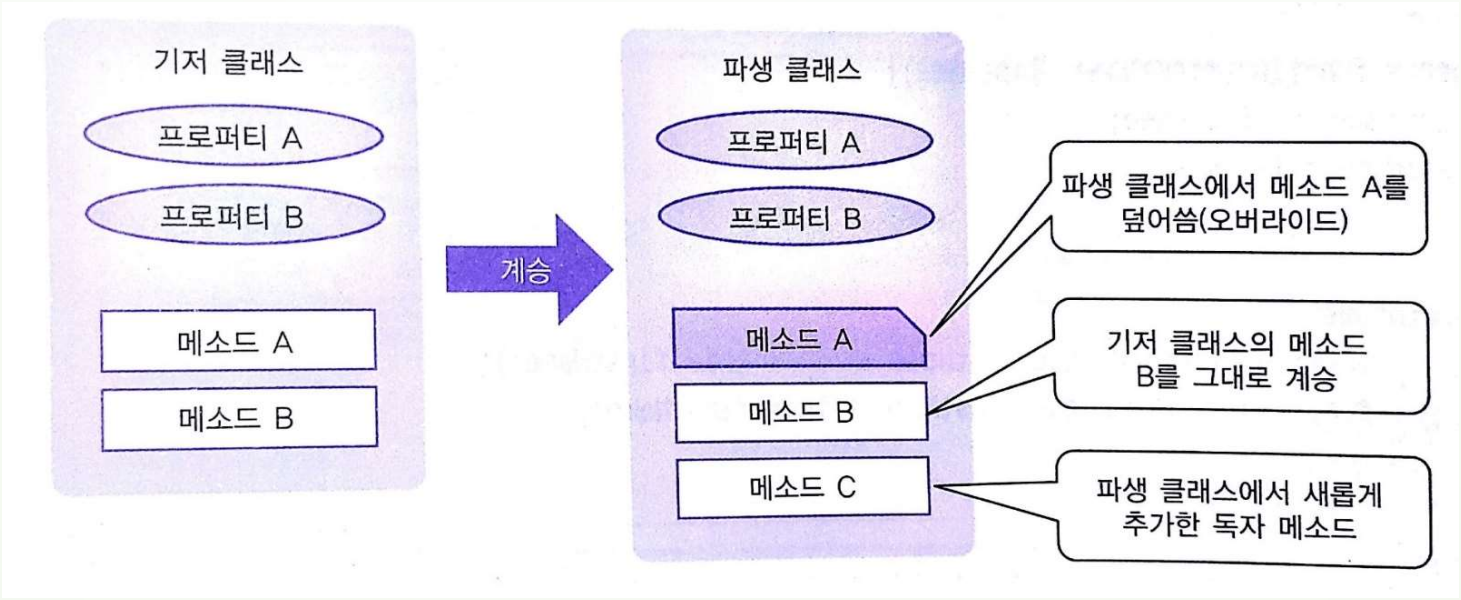
```
var Member = function(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
};

Member.prototype.getName = function(){
        return this.lastName + ' ' + this.firstName;
};

Member.prototype.toString = function(){
        return this.lastName + this.firstName;
};
```

```
var Member = function(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
};

Member.prototype = {
        getName : function(){return this.lastName + ' ' + this.firstName;},
        toString : function(){return this.lastName + this.firstName;}
};
```

객체의 상속-프로토타입 체인

상속

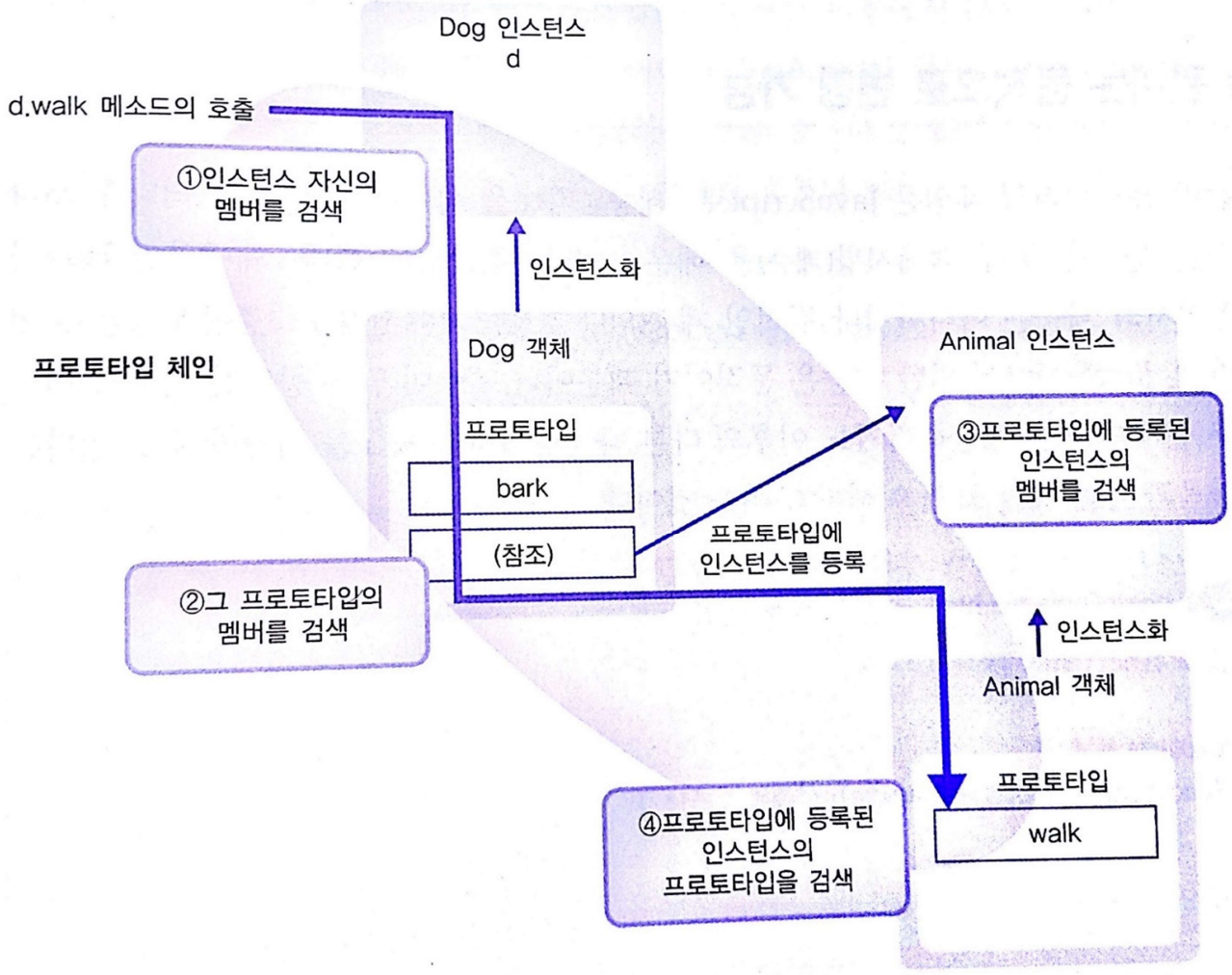
- 상속
  - 베이스가 되는 객체(클래스)의 기능을 계승하여 새로운 클래스를 정의하는 기능

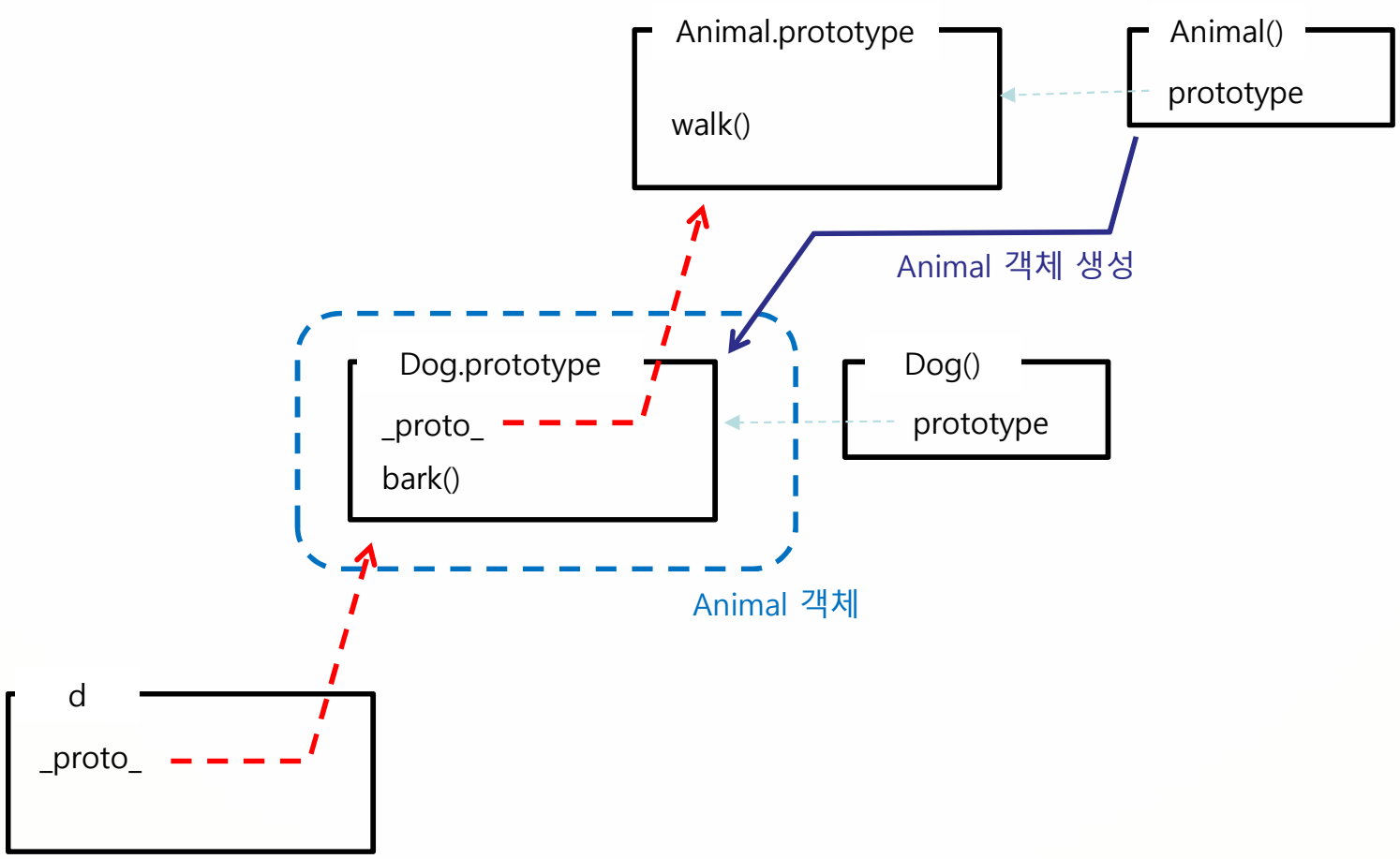


프로토타입 체인의 기초

- 프로토타입 체인
  - prototype 프로퍼티에 객체를 설정
  - 암묵적인 참조를 이용하여 상속 관계를 가질 수 있음

```
var Animal = function() {}

Animal.prototype = {
        walk : function() { document.writeln('종종...'); }
};

var Dog = function() {};
Dog.prototype = new Animal();

Dog.prototype.bark = function() { document.writeln('멍멍 ! '); }

var d = new Dog();
d.walk();      // 종종 ...
d.bark();      // 멍멍
```


Dog 인스턴스
d
d.walk 메소드의 호출
①인스턴스 자신의
멤버를 검색
인스턴스화
Dog 객체
프로토타입 체인
프로토타입
bark
(참조)
②그 프로토타입의
멤버를 검색
Animal 인스턴스
③프로토타입에 등록된
인스턴스의
멤버를 검색
프로토타입에
인스턴스를 등록
인스턴스화
Animal 객체
프로토타입
walk
④프로토타입에 등록된
인스턴스의
프로토타입을 검색


Animal.prototype
walk()
Animal()
prototype
Animal 객체 생성
Dog.prototype
proto
bark()
Dog()
prototype
Animal 객체
d
proto

상속 관계는 동적으로 변경 가능

- 프로토타입 체인
  - 생성자 프로토타입은 동적으로 변경 가능함
  - 변경 후에 생성된 객체
    - 변경된 프로토타입 체인을 만듬
  - 변경 전에 생성된 객체
    - 생성될 때 프로토타입 체인을 유지함

Person 프로토타입을 만들고 객체 생성하기

- **Person이라는 이름의 함수를 만들고 프로토타입 객체로 사용**
- **new 연산자를 사용하는 시점에 생성자 함수로 동작**

ch03_test17.js

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.walk = function(speed) {
    console.log(speed + 'km 속도로 걸어갑니다.');
}

var person01 = new Person('소녀시대', 20);

person01.walk(10);
```

Person 안에 자동으로 만들어지는 prototype 속성

- 함수 객체를 만들 때 자동으로 생성됨
- 다른 함수를 prototype 객체에 추가하면 new 연산자를 이용해 만든 객체에서 공용으로 사용