함수

함수란 어떤 작업을 수행하기 위해 필요한 문(statement)들의 집합을 정의한 코드 블록이다. 함수는 이름과 매개변수를 갖으며 필요한 때에 호출하여 코드 블록에 담긴 문들을 일괄적으로 실행할 수 있다.

```
// 함수의 정의(함수 선언문)
function square(number) {
  return number * number;
}
```

함수는 호출에 의해 실행되는데 한번만 호출할 수 있는 것이 아니라 여러번 호출할 수 있다.

```
// 함수의 정의(함수 선언문)
function square(number) {
  return number * number;
}

// 함수의 호출
square(2); // 4
```

동일한 작업을 반복적으로 수행해야 한다면 (동일한 구문을 계속해서 중복해서 작성하는 것이 아니라) 미리 정의된 함수를 재사용하는 것이 효율적이다. 이러한 특성은 코드의 재사용이라는 측면에서 매우 유용하다.

함수의 일반적 기능은 어떤 작업을 수행하는 문(statement)들의 집합을 정의하여 코드의 재사용에 목적이 있다. 이러한 일반적 기능 이외에 객체 생성, 객체의 행위 정의(메소드), 정보은닉, 클로저, 모듈화 등의 기능을 수행할 수 있다.

자바스크립트의 함수는 객체(일급 객체, First-class object)이다. 다른 객체와 구분될 수 있는 특징은 호출할 수 있다는 것이다. 함수도 객체이므로 다른 값들처럼 사용할 수 있다. 즉, 변수나 객체, 배열 등에 저장할 수 있고 다른 함수에 전달되는 인수로도 사용할 수 있으며 함수의 반환값이 될 수도 있다.

함수 정의

함수를 정의하는 방식은 3가지가 있다.

- 함수 선언문

- 함수 표현식

- Function 생성자 함수

함수 선언문

함수 선언문(Function declaration) 방식으로 정의한 함수는 `function` 키워드와 이하의 내용으로 구성된다.

함수명함수 선언문의 경우, 함수명은 생략할 수 없다. 함수명은 함수 몸체에서 자신을 재귀적(recursive) 호출하거나 자바스크립트 디버거가 해당 함수를 구분할 수 있는 식별자이다. **매개변수 목록**0개 이상의 목록으로 괄호로 감싸고 콤마로 분리한다. 다른 언어와의 차이점은 매개변수의 타입을 기술하지 않는다는 것이다. 이 때문에 함수 몸체 내에서 매개변수의 타입 체크가 필요할 수 있다.**함수 몸체**함수가 호출되었을 때 실행되는 문들의 집합이다. 중괄호({ })로 문들을 감싸고 `return` 문으로 결과값을 반환할 수 있다. 이를 반환값(return value)라 한다.

```
// 함수 선언문
function square(number) {
  return number * number;
}
```

함수 표현식

자바스크립트의 함수는 일급 객체이므로 아래와 같은 특징이 있다.

> 무명의 리터럴로 표현이 가능하다.변수나 자료 구조(객체, 배열...)에 저장할 수 있다.함수의 파라미터로 전달할 수 있다.반환값(return value)으로 사용할 수 있다.

함수의 일급객체 특성을 이용하여 함수 리터럴 방식으로 함수를 정의하고 변수에 할당할 수 있는데 이러한 방식을 함수 표현식(Function expression)이라 한다.

함수 선언문으로 정의한 함수 square()를 함수 표현식으로 정의하면 아래와 같다.

```
// 함수 표현식
var square = function(number) {
  return number * number;
};
```

함수 표현식 방식으로 정의한 함수는 함수명을 생략할 수 있다. 이러한 함수를 **익명 함수(anonymous function)**이라 한다. 함수 표현식에서는 함수명을 생략하는 것이 일반적이다.

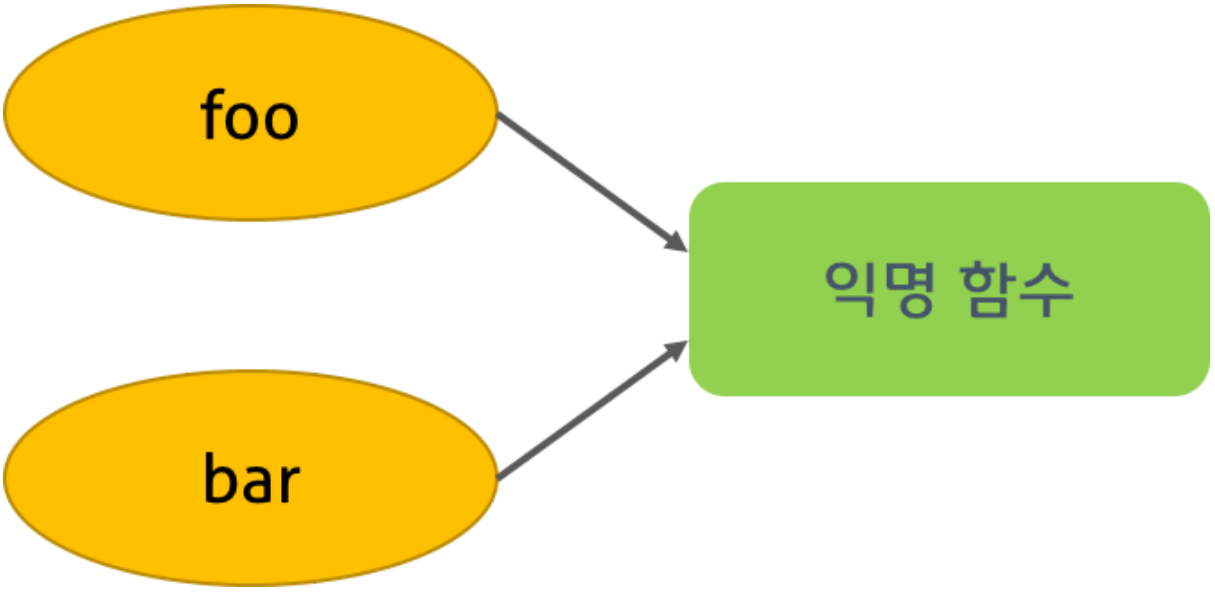
```
// 기명 함수 표현식(named function expression)
var foo = function multiply(a, b) {
  return a * b;
};

// 익명 함수 표현식(anonymous function expression)
var bar = function(a, b) {
  return a * b;
```

```
};

console.log(foo(10, 5)); // 50
console.log(multiply(10, 5)); // Uncaught ReferenceError: m
ultiply is not defined
```

함수는 일급객체이기 때문에 변수에 할당할 수 있는데 이 변수는 함수명이 아니라 할당된 함수를 가리키는 참조값을 저장하게 된다. 함수 호출시 함수명이 아니라 함수를 가리키는 변수명을 사용하여야 한다.

```
var foo = function(a, b) {
  return a * b;
};

var bar = foo;

console.log(foo(10, 10)); // 100
console.log(bar(10, 10)); // 100
```

변수 bar와 변수 foo는 동일한 익명 함수의 참조값을 갖는다.



함수가 할당된 변수를 사용해 함수를 호출하지 않고 기명 함수의 함수명을 사용해 호출하게 되면 에러가 발생한다. 이는 함수 표현식에서 사용한 함수명은 외부 코드에서 접근 불가능하기 때문이다. (사실은 함수 선언문의 경우도 마찬가지이다.)

함수 표현식과 함수 선언문에서 사용한 함수명은 함수 몸체에서 자신을 재귀적 호출(Recursive function call)하거나 자바스크립트 디버거가 해당 함수를 구분할 수 있는 식별자의 역할을 한다.

함수 선언문으로 정의한 함수 square의 경우, 함수명으로 호출할 수 있었는데 이는 자바스크립트 엔진에 의해 아래와 같은 함수 표현식으로 형태가 변경되었기 때문이다.

```
var square = function square(number) {
  return number * number;
};
```

함수명과 함수 참조값을 가진 변수명이 일치하므로 함수명으로 호출되는 듯 보이지만 사실은 변수명으로 호출된 것이다.

결국 함수 선언문도 함수 표현식과 동일하게 함수 리터럴 방식으로 정의되는 것이다.

Function 생성자 함수

함수 표현식으로 함수를 정의할 때 함수 리터럴 방식을 사용한다. 함수 선언문도 내부적으로 자바스크립트 엔진이 기명 함수 표현식으로 변환하므로 결국 함수 리터럴 방식을 사용한다.

따라서 **함수 선언문과 함수 표현식은 모두 함수 리터럴 방식으로 함수를 정의하는데 이것은 결국 내장 함수 Function 생성자 함수로 함수를 생성하는 것을 단순화시킨 short-hand(축약법)이다.**

Function 생성자 함수는 Function.prototype.constructor 프로퍼티로 접근할 수 있다.

Function 생성자 함수로 함수를 생성하는 문법은 다음과 같다.

```
new Function(arg1, arg2, ... argN, functionBody)
```

```
var square = new Function('number', 'return number * numbe
r');
console.log(square(10)); // 100
```

Function 생성자 함수로 함수를 생성하는 방식은 일반적으로 사용하지 않는다.

함수 호이스팅

3가지의 함수 정의 방식을 알아보았다. 정의 방식은 달라도 결국 Function 생성자 함수를 통해 함수를 생성하는 것까지 확인하였다. 그런데 이 3가지 함수 정의 방식은 동작 방식에 약간의 차이가 있다.

```
var res = square(5);

function square(number) {
  return number * number;
}
```

위 코드를 보면 함수 선언문으로 함수가 정의되기 이전에 함수 호출이 가능하다. 함수 선언문의 경우, 함수 선언의 위치와는 상관없이 코드 내 어느 곳에서든지 호출이 가능한데 이것을 함수 호이스팅(Function Hoisting)이라 한다.

자바스크립트는 ES6의 let, const를 포함하여 모든 선언(var, let, const, function, function*, class)을 호이스팅(Hoisting)한다.

호이스팅이란 var 선언문이나 function 선언문 등 모든 선언문이 해당 Scope의 선두로 옮겨진 것처럼 동작하는 특성을 말한다. 즉, 자바스크립트는 모든 선언문(var, let, const, function, function*, class)이 선언되기 이전에 참조 가능하다.

함수 선언문으로 정의된 함수는 자바스크립트 엔진이 스크립트가 로딩되는 시점에 바로 초기화하고 이를 VO(variable object)에 저장한다. 즉, **함수 선언, 초기화, 할당이 한번에 이루어진다.** 그렇기 때문에 함수 선언의 위치와는 상관없이 소스 내 어느 곳에서든지 호출이 가능하다.

다음은 함수 표현식으로 함수를 정의한 경우이다.

```
var res = square(5); // TypeError: square is not a function

var square = function(number) {
  return number * number;
}
```

함수 선언문의 경우와는 달리 TypeError가 발생하였다. **함수 표현식의 경우 함수 호이스팅이 아니라 변수 호이스팅이 발생한다.**

변수 호이스팅은 변수 생성 및 초기화와 할당이 분리되어 진행된다. 호이스팅된 변수는 undefined로 초기화 되고 실제값의 할당은 할당문에서 이루어진다.

함수 표현식은 함수 선언문과는 달리 스크립트 로딩 시점에 변수 객체(VO)에 함수를 할당하지 않고 runtime에 해석되고 실행되므로 이 두가지를 구분하는 것은 중요하다.

JavaScript: The Good Parts의 저자이며 자바스크립트의 권위자인 더글러스 크락포드(Douglas Crockford)는 이와 같은 문제 때문에 함수 표현식만을 사용할 것을 권고하고 있다. 함수 호이스팅이 함수 호출 전 반드시 함수를 선언하여야 한다는 규칙을 무시하므로 코드의 구조를 엉성하게 만들 수 있다고 지적한다.

또한 함수 선언문으로 함수를 정의하면 사용하기에 쉽지만 대규모 애플리케이션을 개발하는 경우 인터프리터가 너무 많은 코드를 변수 객체(VO)에 저장하므로 애플리케이션의 응답속도는 현저히 떨어질 수 있으므로 주의해야 할 필요가 있다.

First-class object (일급 객체)

일급 객체(first-class object)란 생성, 대입, 연산, 인자 또는 반환값으로서의 전달 등 프로그래밍 언어의 기본적 조작을 제한없이 사용할 수 있는 대상을 의미한다.

다음 조건을 만족하면 일급 객체로 간주한다.

> 무명의 리터럴로 표현이 가능하다.변수나 자료 구조(객체, 배열 등)에 저장할 수 있다.함수의 매개변수에 전달할 수 있다.반환값으로 사용할 수 있다.

```
// 1. 무명의 리터럴로 표현이 가능하다.
// 2. 변수나 자료 구조에 저장할 수 있다.
var increase = function (num) {
  return ++num;
};

var decrease = function (num) {
  return --num;
};

var predicates = { increase, decrease };

// 3. 함수의 매개변수에 전달할 수 있다.
// 4. 반환값으로 사용할 수 있다.
function makeCounter(predicate) {
  var num = 0;

  return function () {
```

```
    num = predicate(num);
    return num;
  };
}

var increaser = makeCounter(predicates.increase);
console.log(increaser()); // 1
console.log(increaser()); // 2

var decreaser = makeCounter(predicates.decrease);
console.log(decreaser()); // -1
console.log(decreaser()); // -2
```

Javascript의 함수는 위의 조건을 모두 만족하므로 **Javascript의 함수는 일급객체이다.** 따라서 Javascript의 함수는 흡사 변수와 같이 사용할 수 있으며 코드의 어디에서든지 정의할 수 있다.

함수와 다른 객체를 구분짓는 특징은 호출할 수 있다는 것이다.

매개변수(Parameter, 인자)

함수의 작업 실행을 위해 추가적인 정보가 필요할 경우, 매개변수를 지정한다. 매개변수는 함수 내에서 변수와 동일하게 동작한다.

매개변수(parameter, 인자) vs 인수(argument)

매개변수는 함수 내에서 변수와 동일하게 메모리 공간을 확보하며 함수에 전달한 인수는 매개변수에 할당된다. 만약 인수를 전달하지 않으면 매개변수는 undefined로 초기화된다.

```
var foo = function (p1, p2) {
  console.log(p1, p2);
};

foo(1); // 1 undefined
```

Call-by-value

원시 타입 인수는 **Call-by-value**(값에 의한 호출)로 동작한다. 이는 함수 호출 시 원시 타입 인수를 함수에 매개변수로 전달할 때 매개변수에 값을 복사하여 함수로 전달하는 방식이다. 이때 함수 내에서 매개변수를 통해 값이 변경되어도 전달이 완료된 원시 타입 값은 변경되지 않는다.

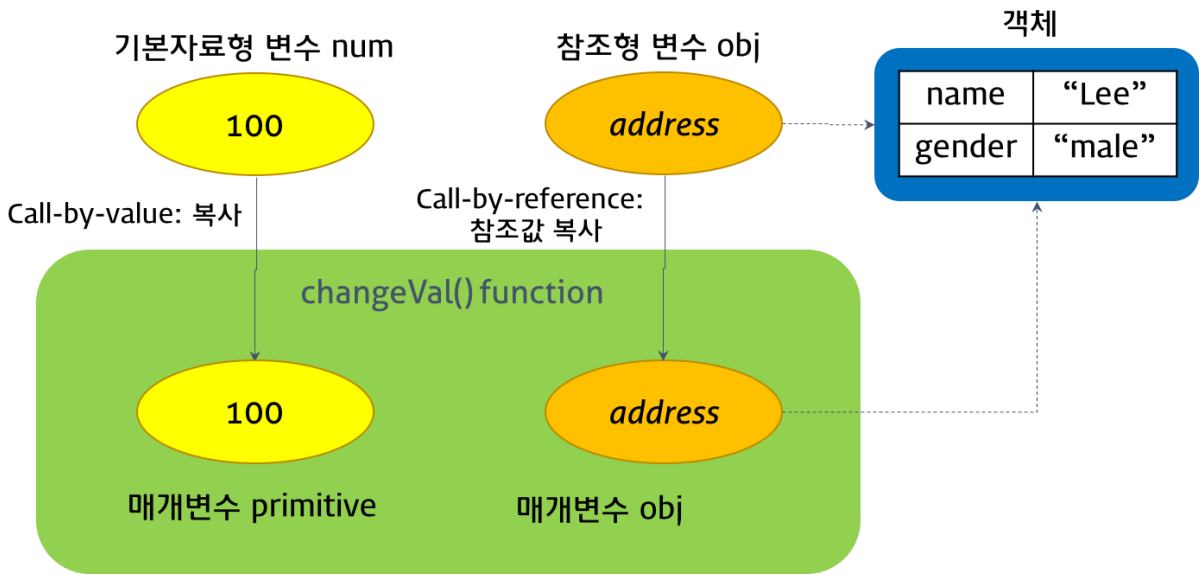
```
function foo(primitive) {
  primitive += 1;
  return primitive;
}

var x = 0;

console.log(foo(x)); // 1
console.log(x);      // 0
```

Call-by-reference

객체형(참조형) 인수는 **Call-by-reference**(참조에 의한 호출)로 동작한다. 이는 함수 호출 시 참조 타입 인수를 함수에 매개변수로 전달할 때 매개변수에 값이 복사되지 않고 객체의 참조값이 매개변수에 저장되어 함수로 전달되는 방식이다. 이때 함수 내에서 매개변수의 참조값이 이용하여 객체의 값을 변경했을 때 전달되어진 참조형의 인수값도 같이 변경된다.

```
function changeVal(primitive, obj) {
  primitive += 100;
  obj.name = 'Kim';
  obj.gender = 'female';
}

var num = 100;
var obj = {
  name: 'Lee',
  gender: 'male'
```

```
};

console.log(num); // 100
console.log(obj); // Object {name: 'Lee', gender: 'male'}

changeVal(num, obj);

console.log(num); // 100
console.log(obj); // Object {name: 'Kim', gender: 'female'}
```



changeVal 함수는 원시 타입과 객체 타입 인수를 전달 받아 함수 몸체에서 매개변수의 값을 변경하였다. 이때 원시 타입 인수는 값을 복사하여 매개변수에 전달하기 때문에 함수 몸체에서 그 값을 변경하여도 어떠한 부수 효과(side-effect)도 발생시키지 않는다.

하지만 객체형 인수는 참조값을 매개변수에 전달하기 때문에 함수 몸체에서 그 값을 변경할 경우 원본 객체가 변경되는 부수 효과(side-effect)가 발생한다. 이와 같이 부수 효과를 발생시키는 비순수 함수(Impure function)는 복잡성을 증가시킨다. 비순수 함수를 최대한 줄이는 것은 부수 효과를 최대한 억제하는 것과 같다. 이것은 디버깅을 쉽게 만든다.

어떤 외부 상태도 변경하지 않는 함수를 순수함수(Pure function), 외부 상태도 변경시켜는 부수 효과(side-effect)가 발생시키는 함수를 비순수 함수(Impure function)라 한다.

반환값

함수는 자신을 호출한 코드에게 수행한 결과를 반환(return)할 수 있다. 이때 반환된 값을 반환값(return value)이라 한다.

- `return` 키워드는 함수를 호출한 코드(caller)에게 값을 반환할 때 사용한다.
- 함수는 배열 등을 이용하여 한 번에 여러 개의 값을 리턴할 수 있다.
- 함수는 반환을 생략할 수 있다. 이때 함수는 암묵적으로 undefined를 반환한다.
- 자바스크립트 해석기는 `return` 키워드를 만나면 함수의 실행을 중단한 후, 함수를 호출한 코드로 되돌아간다. 만일 `return` 키워드 이후에 다른 구문이 존재하면 그 구문은 실행되지 않는다.

```
function calculateArea(width, height) {
  var area = width * height;
  return area; // 단일 값의 반환
}
console.log(calculateArea(3, 5)); // 15
console.log(calculateArea(8, 5)); // 40

function getSize(width, height, depth) {
  var area = width * height;
  var volume = width * height * depth;
  return [area, volume]; // 복수 값의 반환
}

console.log('area is ' + getSize(3, 2, 3)[0]);   // area is
6
console.log('volume is ' + getSize(3, 2, 3)[1]); // volume
is 18
```

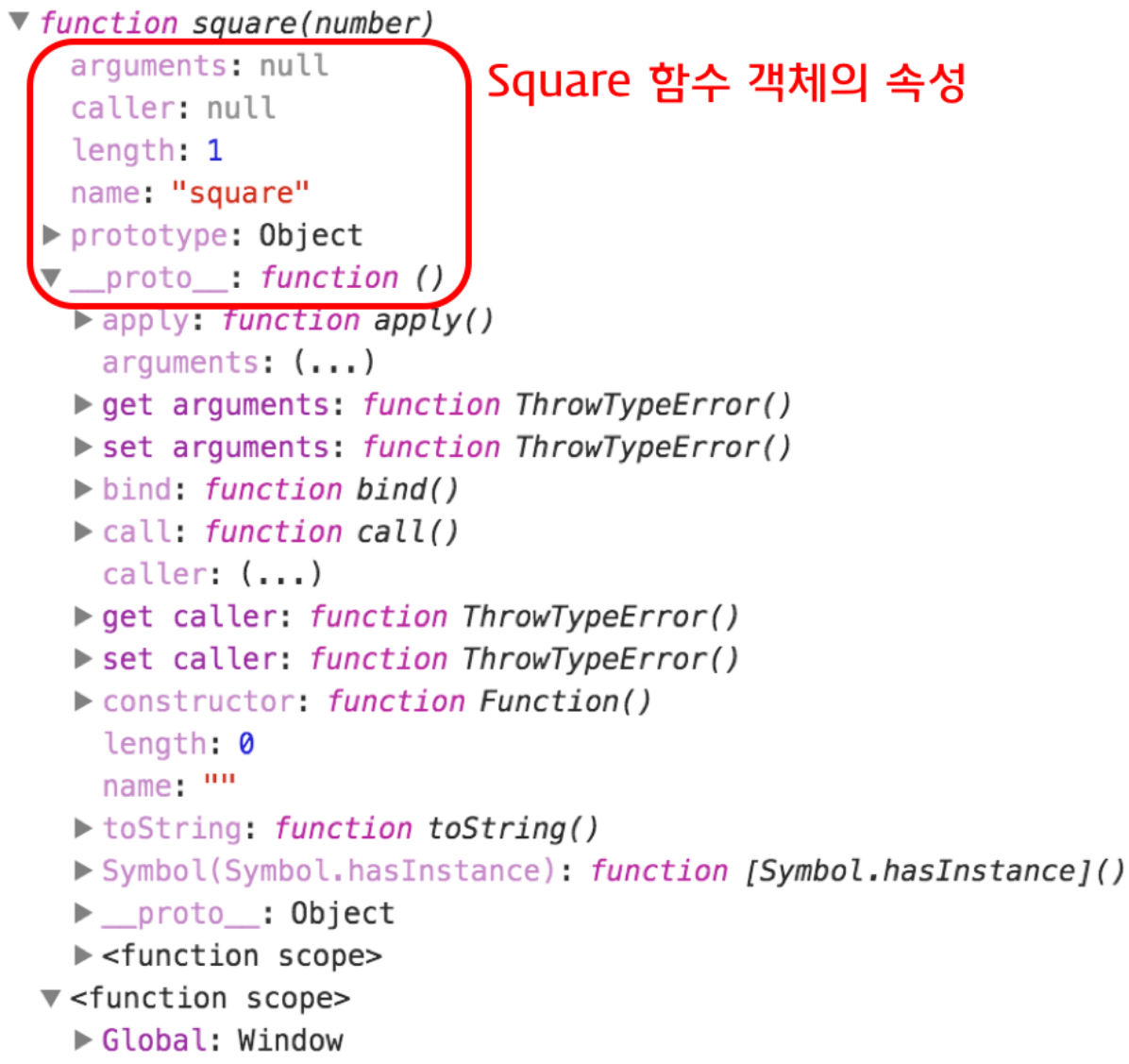
함수 객체의 프로퍼티

함수는 객체이다. 따라서 함수도 프로퍼티를 가질 수 있다.

```
function square(number) {
  return number * number;
}

square.x = 10;
square.y = 20;

console.log(square.x, square.y);
```

함수는 일반 객체와는 다른 함수만의 프로퍼티를 갖는다.

```
function square(number) {
  return number * number;
}
console.dir(square);
```



arguments 프로퍼티

arguments 객체는 함수 호출 시 전달된 인수(argument)들의 정보를 담고 있는 순회가능한(iterable) 유사 배열 객체(array-like object)이며 함수 내부에서 지역변수처럼 사용된다. 즉, 함수 외부에서는 사용할 수 없다.

arguments 프로퍼티는 현재 일부 브라우저에서 지원하고 있지만 ES3부터 표준에서 deprecated 되었다. Function.arguments와 같은 사용 방법은 권장되지 않으며 함수 내부에서 지역변수처럼 사용할 수 있는 arguments 객체를 참조하도록 한다.

자바스크립트는 함수 호출 시 함수 정의에 따라 인수를 전달하지 않아도 에러가 발생하지 않는다.

```
function multiply(x, y) {
  console.log(arguments);
  return x * y;
}

multiply();         // {}
multiply(1);        // { '0': 1 }
multiply(1, 2);     // { '0': 1, '1': 2 }
multiply(1, 2, 3); // { '0': 1, '1': 2, '2': 3 }
```

매개변수(parameter)는 인수(argument)로 초기화된다.

- 매개변수의 갯수보다 인수를 적게 전달했을 때(multiply(), multiply(1)) 인수가 전달되지 않은 매개변수는 `undefined`으로 초기화된다.
- 매개변수의 갯수보다 인수를 더 많이 전달한 경우, 초과된 인수는 무시된다.

이러한 자바스크립트의 특성때문에 런타임 시에 호출된 함수의 인자 갯수를 확인하고 이에 따라 동작을 달리 정의할 필요가 있을 수 있다. 이때 유용하게 사용되는 것이 arguments 객체이다.

arguments 객체는 매개변수 갯수가 확정되지 않은 **가변 인자 함수**를 구현할 때 유용하게 사용된다.

```
function sum() {
  var res = 0;

  for (var i = 0; i < arguments.length; i++) {
    res += arguments[i];
  }
```

```
  return res;
}

console.log(sum());        // 0
console.log(sum(1, 2));    // 3
console.log(sum(1, 2, 3)); // 6
```

자바스크립트는 함수를 호출할 때 인수들과 함께 암묵적으로 arguments 객체가 함수 내부로 전달된다. arguments 객체는 배열의 형태로 인자값 정보를 담고 있지만 실제 배열이 아닌 **유사배열객체(array-like object)**이다.

유사배열객체란 length 프로퍼티를 가진 객체를 말한다. 유사배열객체는 배열이 아니므로 배열 메소드를 사용하는 경우 에러가 발생하게 된다. 따라서 배열 메소드를 사용하려면 Function.prototype.call, Function.prototype.apply를 사용하여야 하는 번거로움이 있다.

```
function sum() {
  if (!arguments.length) return 0;

  // arguments 객체를 배열로 변환
  var array = Array.prototype.slice.call(arguments);
  return array.reduce(function (pre, cur) {
    return pre + cur;
  });
}

// ES6
// function sum(...args) {
//   if (!args.length) return 0;
//   return args.reduce((pre, cur) => pre + cur);
// }

console.log(sum(1, 2, 3, 4, 5)); // 15
```

caller 프로퍼티

caller 프로퍼티는 자신을 호출한 함수를 의미한다.

```
function foo(func) {
  var res = func();
  return res;
}

function bar() {
  return 'caller : ' + bar.caller;
}

console.log(foo(bar)); // caller : function foo(func) {...}
console.log(bar());    // null (browser에서의 실행 결과)
```

length 프로퍼티

length 프로퍼티는 함수 정의 시 작성된 매개변수 갯수를 의미한다.

```
function foo() {}
console.log(foo.length); // 0

function bar(x) {
  return x;
}
console.log(bar.length); // 1

function baz(x, y) {
  return x * y;
}
console.log(baz.length); // 2
```

arguments.length의 값과는 다를 수 있으므로 주의하여야 한다. arguments.length는 함수 호출시 인자의 갯수이다.

name 프로퍼티

함수명을 나타낸다. 기명함수의 경우 함수명을 값으로 갖고 익명함수의 경우 빈문자열을 값으로 갖는다.

```
// 기명 함수 표현식(named function expression)
var namedFunc = function multiply(a, b) {
  return a * b;
};
// 익명 함수 표현식(anonymous function expression)
var anonymousFunc = function(a, b) {
  return a * b;
};

console.log(namedFunc.name);      // multiply
console.log(anonymousFunc.name); // ''
```

__proto__ 접근자 프로퍼티

모든 객체는 [[Prototype]]이라는 내부 슬롯이 있다. [[Prototype]] 내부 슬롯은 프로토타입 객체를 가리킨다. 프로토타입 객체란 프로토타입 기반 객체 지향 프로그래밍의 근간을 이루는 객체로서 객체간의 상속(Inheritance)을 구현하기 위해 사용된다. 즉, 프로토타입 객체는 다른 객체에 공유 프로퍼티를 제공하는 객체를 말한다.

__proto__ 프로퍼티는 [[Prototype]] 내부 슬롯이 가리키는 프로토타입 객체에 접근하기 위해 사용하는 접근자 프로퍼티이다. 내부 슬롯에는 직접 접근할 수 없고 간접적인 접근 방법을 제공하는 경우에 한하여 접근할 수 있다. [[Prototype]] 내부 슬롯에도 직접 접근할 수 없으며 __proto__ 접근자 프로퍼티를 통해 간접적으로 프로토타입 객체에 접근할 수 있다.

```
// __proto__ 접근자 프로퍼티를 통해 자신의 프로토타입 객체에 접근할 수 있다.
// 객체 리터럴로 생성한 객체의 프로토타입 객체는 Object.prototype이다.
console.log({}.__proto__ === Object.prototype); // true
```

__proto__ 프로퍼티는 객체가 직접 소유하는 프로퍼티가 아니라 모든 객체의 프로토타입 객체인 Object.prototype 객체의 프로퍼티이다. 모든 객체는 상속을 통해 __proto__ 접근자 프로퍼티는 사용할 수 있다.

```
// 객체는 __proto__ 프로퍼티를 소유하지 않는다.
console.log(Object.getOwnPropertyDescriptor({}, '__proto__'));
// undefined

// __proto__ 프로퍼티는 모든 객체의 프로토타입 객체인 Object.prototype의 접근자 프로퍼티이다.
console.log(Object.getOwnPropertyDescriptor(Object.prototype, '__proto__'));
// {get: ƒ, set: ƒ, enumerable: false, configurable: true}

// 모든 객체는 Object.prototype의 접근자 프로퍼티 __proto__를 상속받아 사용할 수 있다.
console.log({}.__proto__ === Object.prototype); // true
```

함수도 객체이므로 __proto__ 접근자 프로퍼티를 통해 프로토타입 객체에 접근할 수 있다.

```
// 함수 객체의 프로토타입 객체는 Function.prototype이다.
console.log((function() {}).__proto__ === Function.prototype); // true
```

prototype 프로퍼티

prototype 프로퍼티는 함수 객체만이 소유하는 프로퍼티이다. 즉 일반 객체에는 prototype 프로퍼티가 없다.

```
// 함수 객체는 prototype 프로퍼티를 소유한다.
console.log(Object.getOwnPropertyDescriptor(function() {},
'prototype'));
// {value: {…}, writable: true, enumerable: false, configur
able: false}

// 일반 객체는 prototype 프로퍼티를 소유하지 않는다.
console.log(Object.getOwnPropertyDescriptor({}, 'prototyp
e'));
// undefined
```

prototype 프로퍼티는 함수가 객체를 생성하는 생성자 함수로 사용될 때, 생성자 함수가 생성한 인스턴스의 프로토타입 객체를 가리킨다.

함수의 다양한 형태

즉시 실행 함수

함수의 정의와 동시에 실행되는 함수를 즉시 실행 함수(IIFE, Immediately Invoke Function Expression)라고 한다. 최초 한번만 호출되며 다시 호출할 수는 없다. 이러한 특징을 이용하여 최초 한번만 실행이 필요한 초기화 처리등에 사용할 수 있다.

```
// 기명 즉시 실행 함수(named immediately-invoked function expr
ession)
(function myFunction() {
  var a = 3;
  var b = 5;
  return a * b;
}());

// 익명 즉시 실행 함수(immediately-invoked function expressio
```

```
n)
(function () {
  var a = 3;
  var b = 5;
  return a * b;
}());

// SyntaxError: Unexpected token (
// 함수선언문은 자바스크립트 엔진에 의해 함수 몸체를 닫는 중괄호 뒤에 ;
가 자동 추가된다.
function () {
  // ...
}(); // => };();

// 따라서 즉시 실행 함수는 소괄호로 감싸준다.
(function () {
  // ...
}());

(function () {
  // ...
})();
```

자바스크립트에서 가장 큰 문제점 중의 하나는 파일이 분리되어 있다하여도 글로벌 스코프가 하나이며 글로벌 스코프에 선언된 변수나 함수는 코드 내의 어디서든지 접근이 가능하다는 것이다.

따라서 다른 스크립트 파일 내에서 동일한 이름으로 명명된 변수나 함수가 같은 스코프 내에 존재할 경우 원치 않는 결과를 가져올 수 있다.

즉시 실행 함수 내에 처리 로직을 모아 두면 혹시 있을 수도 있는 변수명 또는 함수명의 충돌을 방지할 수 있어 이를 위한 목적으로 즉시실행함수를 사용되기도 한다.

특히 jQuery와 같은 라이브러리의 경우, 코드를 즉시 실행 함수 내에 정의해 두면 라이브러리의 변수들이 독립된 영역 내에 있게 되므로 여러 라이브러리들은 동시에 사용하더라도 변수명 충돌과 같은 문제를 방지할 수 있다.

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

var foo = 100;
console.log(foo);
```

내부 함수

함수 내부에 정의된 함수를 내부함수(Inner function)라 한다.

아래 예제의 내부함수 child는 자신을 포함하고 있는 부모함수 parent의 변수에 접근할 수 있다. 하지만 부모함수는 자식함수(내부함수)의 변수에 접근할 수 없다.

```
function parent(param) {
  var parentVar = param;
  function child() {
    var childVar = 'lee';
    console.log(parentVar + ' ' + childVar); // Hello lee
  }
  child();
  console.log(parentVar + ' ' + childVar);
  // Uncaught ReferenceError: childVar is not defined
}
parent('Hello');
```

또한 내부함수는 부모함수의 외부에서 접근할 수 없다.

```
function sayHello(name){
  var text = 'Hello ' + name;
```

```
  var logHello = function(){ console.log(text); }
  logHello();
}

sayHello('lee');  // Hello lee
logHello('lee');  // logHello is not defined
```

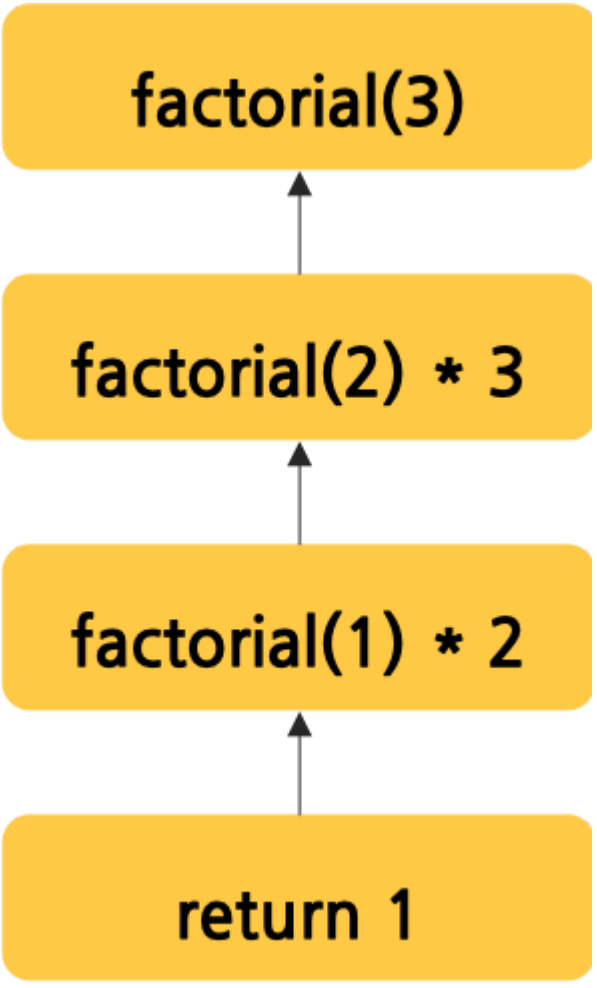
재귀 함수

재귀 함수(Recusive function)는 자기 자신을 호출하는 함수를 말한다.

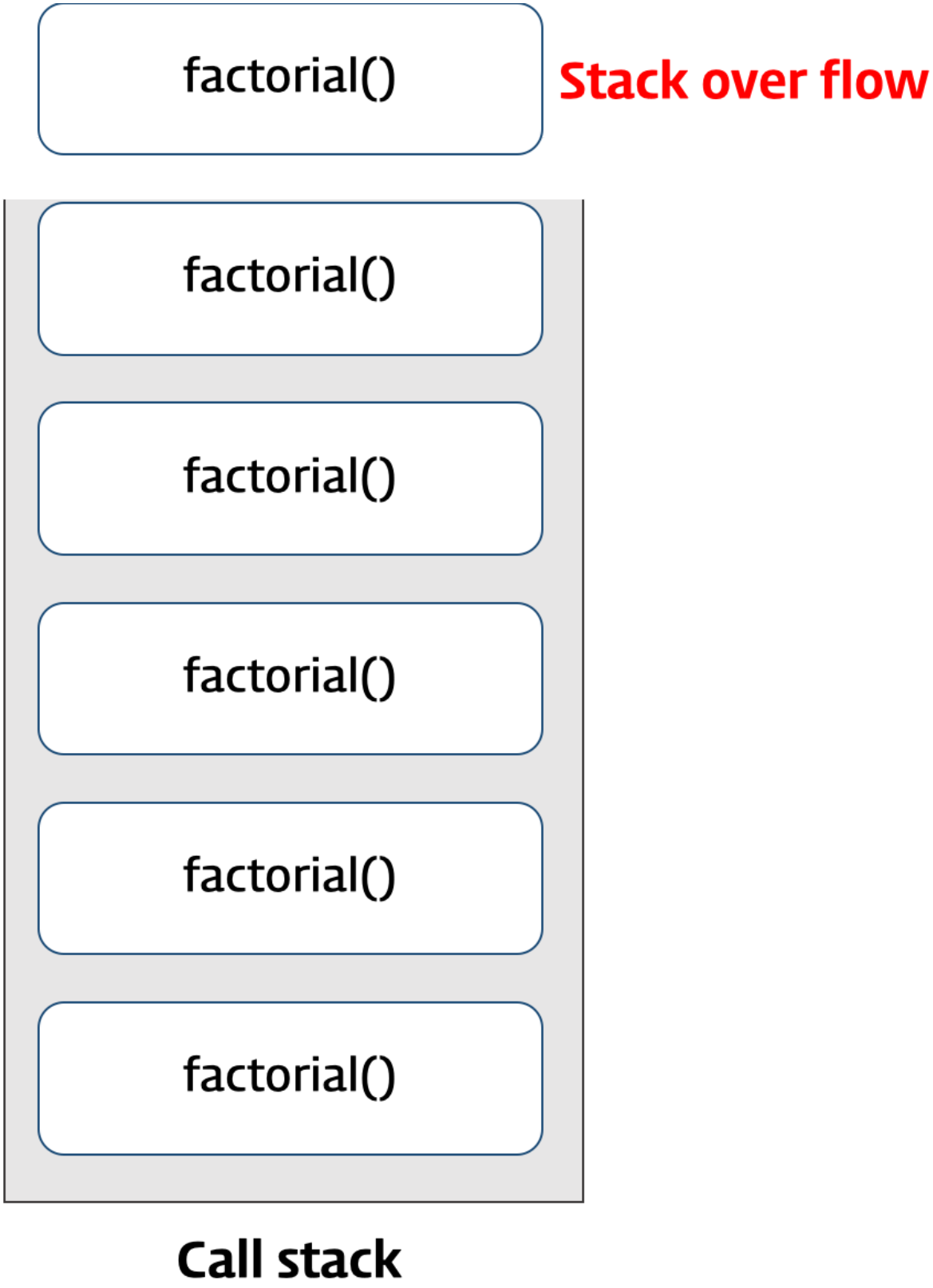
```
// 피보나치 수열
// 피보나치 수는 0과 1로 시작하며, 다음 피보나치 수는 바로 앞의 두 피보나치 수의 합이 된다.
// 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...
function fibonacci(n) {
  if (n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(0)); // 0
console.log(fibonacci(1)); // 1
console.log(fibonacci(2)); // 1
console.log(fibonacci(3)); // 2
console.log(fibonacci(4)); // 3
console.log(fibonacci(5)); // 5
console.log(fibonacci(6)); // 8

// 팩토리얼
// 팩토리얼(계승)은 1부터 자신까지의 모든 양의 정수의 곱이다.
// n! = 1 * 2 * ... * (n-1) * n
function factorial(n) {
  if (n < 2) return 1;
  return factorial(n - 1) * n;
}
```

```
console.log(factorial(0)); // 1
console.log(factorial(1)); // 1
console.log(factorial(2)); // 2
console.log(factorial(3)); // 6
console.log(factorial(4)); // 24
console.log(factorial(5)); // 120
console.log(factorial(6)); // 720
```

재귀 함수는 자신을 무한히 연쇄 호출하므로 호출을 멈출 수 있는 탈출 조건을 반드시 만들어야 한다. 탈출 조건이 없는 경우, 함수가 무한 호출되어 stackoverflow 에러가 발생한다. 위의 두개의 예제 모두 조건식을 통해 재귀 호출을 중지하고 있다.



재귀 함수는 반복 연산을 간단히 구현할 수 있다는 장점이 있지만 무한 반복에 빠질 수 있고, stackoverflow 에러를 발생시킬 수 있으므로 주의하여야 한다.



대부분의 재귀 함수는 for나 while 문으로 구현이 가능하다. 반복문보다 재귀 함수를 통해 보다 직관적으로 이해하기 쉬운 구현이 가능한 경우에만 한정적으로 적용하는 것이 바람직

하다.

콜백 함수

콜백 함수(Callback function)는 함수를 명시적으로 호출하는 방식이 아니라 특정 이벤트가 발생했을 때 시스템에 의해 호출되는 함수를 말한다.

콜백 함수가 자주 사용되는 대표적인 예는 이벤트 핸들러 처리이다.

```
<!DOCTYPE html>
<html>
<body>
  <button id="myButton">Click me</button>
  <script>
    var button = document.getElementById('myButton');
    button.addEventListener('click', function() {
      console.log('button clicked!');
    });
  </script>
</body>
</html>
```

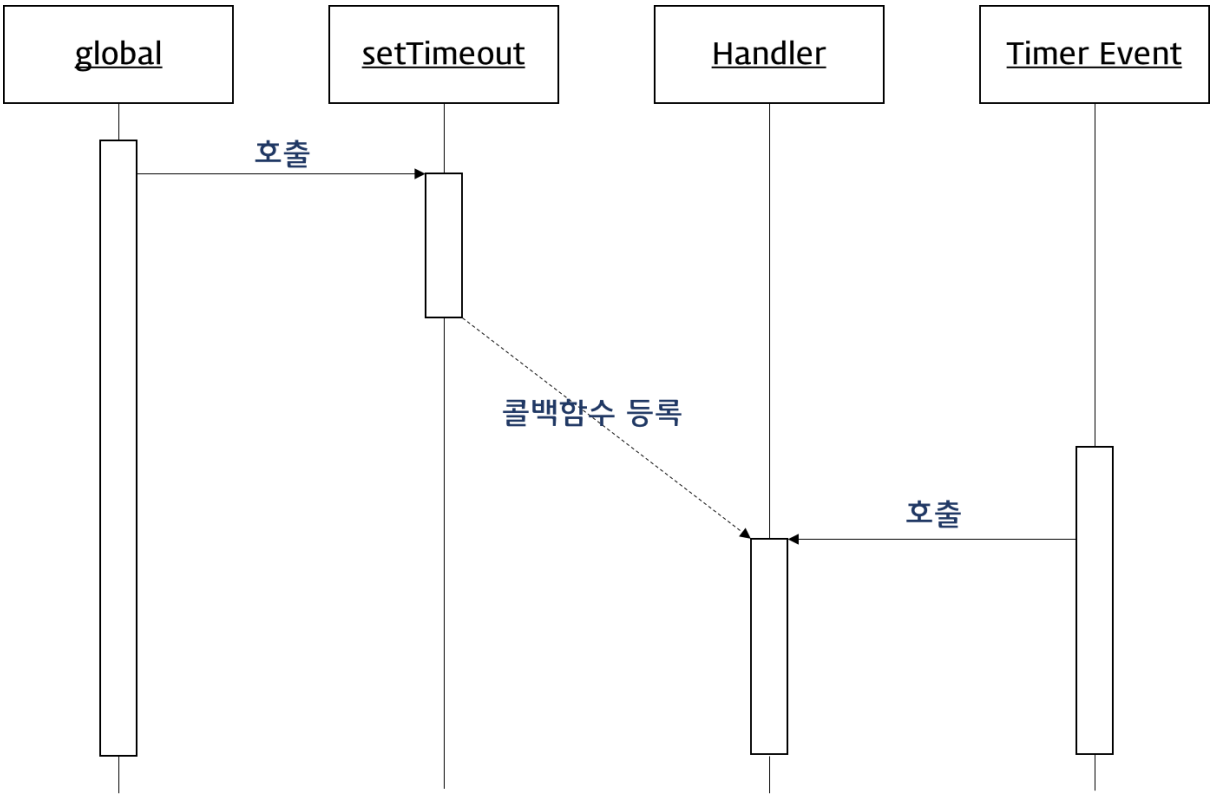
Javascript의 함수는 일급객체이다. 따라서 Javascript의 함수는 흡사 변수와 같이 사용될 수 있다.

콜백 함수는 매개변수를 통해 전달되고 전달받은 함수의 내부에서 **어느 특정시점**에 실행된다.

setTimeout()의 콜백 함수를 살펴보자. 두번째 매개변수에 전달된 시간이 경과되면 첫번째 매개변수에 전달한 콜백 함수가 호출된다.

```
setTimeout(function () {
  console.log('1초 후 출력된다.');
```

```
}, 1000);
```



콜백 함수는 주로 비동기식 처리 모델(Asynchronous processing model)에 사용된다. 비동기식 처리 모델이란 처리가 종료하면 호출될 함수(콜백함수)를 미리 매개변수에 전달하고 처리가 종료하면 콜백함수를 호출하는 것이다.

콜백함수는 콜백 큐에 들어가 있다가 해당 이벤트가 발생하면 호출된다. 콜백 함수는 클로저이므로 콜백 큐에 단독으로 존재하다가 호출되어도 콜백함수를 전달받은 함수의 변수에 접근할 수 있다.

```
function doSomething() {
  var name = 'Lee';

  setTimeout(function () {
    console.log('My name is ' + name);
  }, 100);
```

```
}

doSomething(); // My name is Lee
```