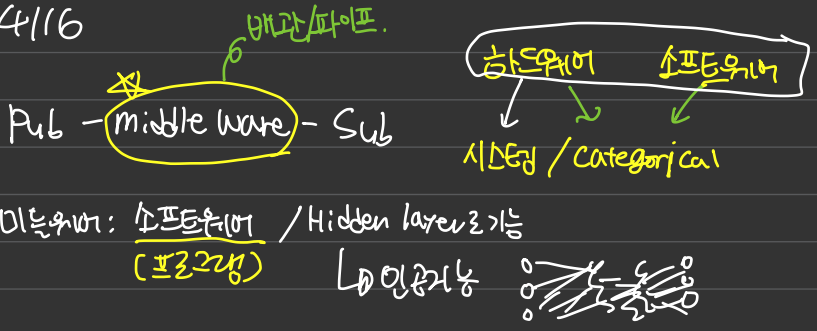
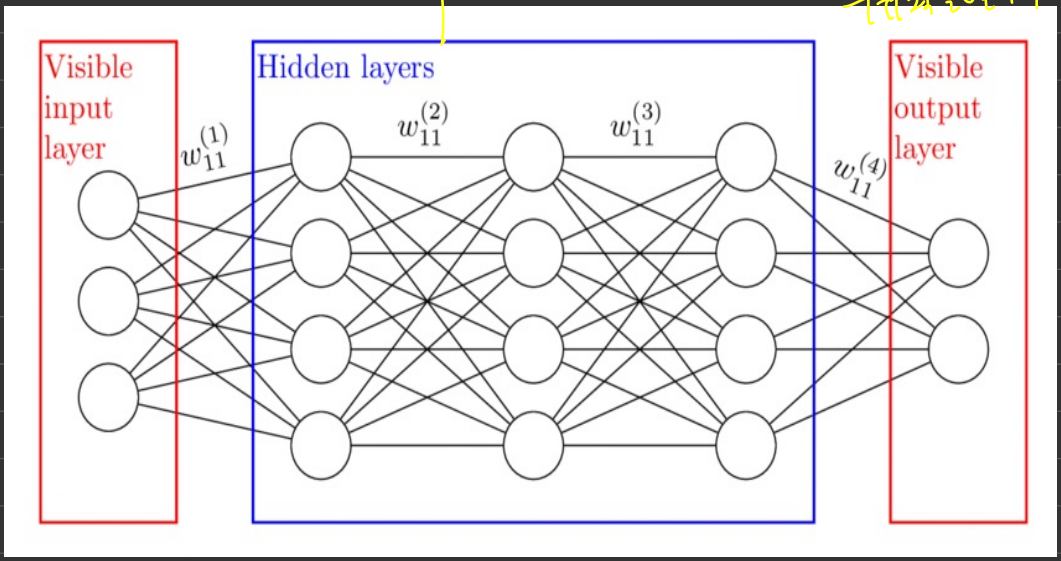
4/16



미들웨어는 운영 체제와 해당 운영 체제에서 실행되는 응용 프로그램 사이에 존재하는 소프트웨어입니다. 기본적으로 숨겨진 변환 계층으로 기능하는 미들웨어는 분산 응용 프로그램의 통신 및 데이터 관리를 가능하게 합니다. 데이터와 데이터베이스가 "파이프" 사이를 쉽게 통과할 수 있도록 두 가지 응용 프로그램을 함께 연결하기 때문에 배관이라고도 합니다. 미들웨어를 사용하면 사용자가 웹 브라우저에서 양식을 제출하거나 웹 서버가 사용자의 프로필을 기반으로 동적 웹 페이지를 반환하도록 요청할 수 있습니다.

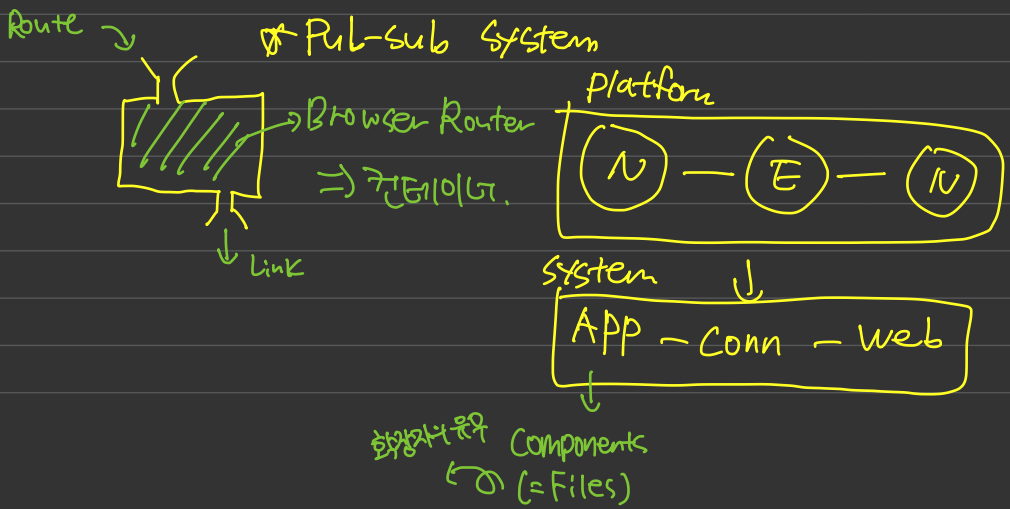
일반적인 미들웨어 예로는 데이터베이스 미들웨어, 애플리케이션 서버 미들웨어, 메시지 지향 미들웨어, 웹 미들웨어 및 트랜잭션 처리 모니터가 있습니다. 각 프로그램은 일반적으로 SOAP(Simple Object Access Protocol), 웹 서비스, REST(Representational State Transfer) 및 JSON(JavaScript Object Notation)과 같은 메시징 프레임워크를 사용하여 서로 다른 응용 프로그램이 통신할 수 있도록 메시지 서비스를 제공합니다. 모든 미들웨어가 통신 기능을 수행하지만 회사가 사용하기로 선택한 형식은 사용 중인 서비스와 통신해야 할 정보 형식에 따라 다릅니다. 여기에는 보안 인증, 트랜잭션 관리, 메시지 큐, 응용 프로그램 서버, 웹 서버 및 디렉터리가 포함될 수 있습니다. 미들웨어는 데이터를 앞뒤로 보내지 않고 실시간으로 발생하는 작업으로 분산 처리에도 사용할 수 있습니다

Spring Cloud Data Flow is a cloud-native programming and operating model for composable data microservices.
With Spring Cloud Data Flow, developers can create and orchestrate data pipelines for common use cases such as data ingest, real-time analytics, and data import/export.
This data pipelines come in two flavors, streaming and batch data pipelines.
In the first case, an unbounded amount of data is consumed or produced via messaging middleware. While in the second case the short-lived task processes a finite set of data and then terminate.



노드와 노드를 연결하는 개념이 엣지이다.
프로그래밍으로 들어와서 노드는 앱 혹은 웹이 되고 각종 path, connection 들은 엣지의 개념을 갖는다.
히든 레이어는 엣지에서 작동하는 미들웨어를 뜻하게 된다.

1. 미들웨어는 앱(웹) 사이에 존재하는 소프트웨어이다.
2. 미들웨어는 기본적으로 숨겨진 변환 계층으로 기능한다.



```
import {ArticleDetail, ArticleList, ArticleWrite,
ArticleUpdate} from 'article/index';
```

구조분해할당

(Observer Pattern)

JSON

MDN 정의에 따르면 배열이나 객체의 속성을 해체하여 그 값을 개별 변수에 담을 수 있게 하는 JavaScript 표현식
js 뿐만 아니라 C언어에도 있는 문법인 것 같다

component

Package

hidden layer.

옵서버 패턴(observer pattern)은 객체의 상태 변화를 관찰하는 관찰자들, 즉 옵저버들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴이다. 주로 분산 이벤트 핸들링 시스템을 구현하는 데 사용된다. 발행/구독 모델로 알려져 있기도 하다.

Pub-sub 모델

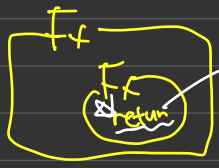
Cloud

이원호 : Observer pattern

클로저(closure)는 자바스크립트에서 중요한 개념 중 하나로 자바스크립트에 관심을 가지고 있다면 한번쯤은 들어보았을 내용이다. execution context에 대한 사전 지식이 있으면 이해하기 어렵지 않은 개념이다. 클로저는 자바스크립트 고유의 개념이 아니라 함수를 일급 객체로 취급하는 함수형 프로그래밍 언어(Functional Programming language: 얼랭(Erlnag), 스칼라(Scala), 하스켈(Haskell), 리스프(Lisp)…)에서 사용되는 중요한 특성이다.

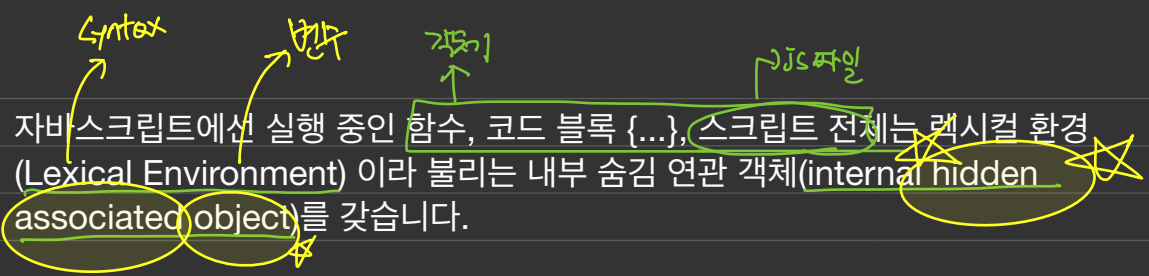
> “A closure is the combination of a function and the lexical environment within which that function was declared.”
> 클로저는 함수와 그 함수가 선언됐을 때의 렉시컬 환경(Lexical environment)과의 조합이다.

환경이다

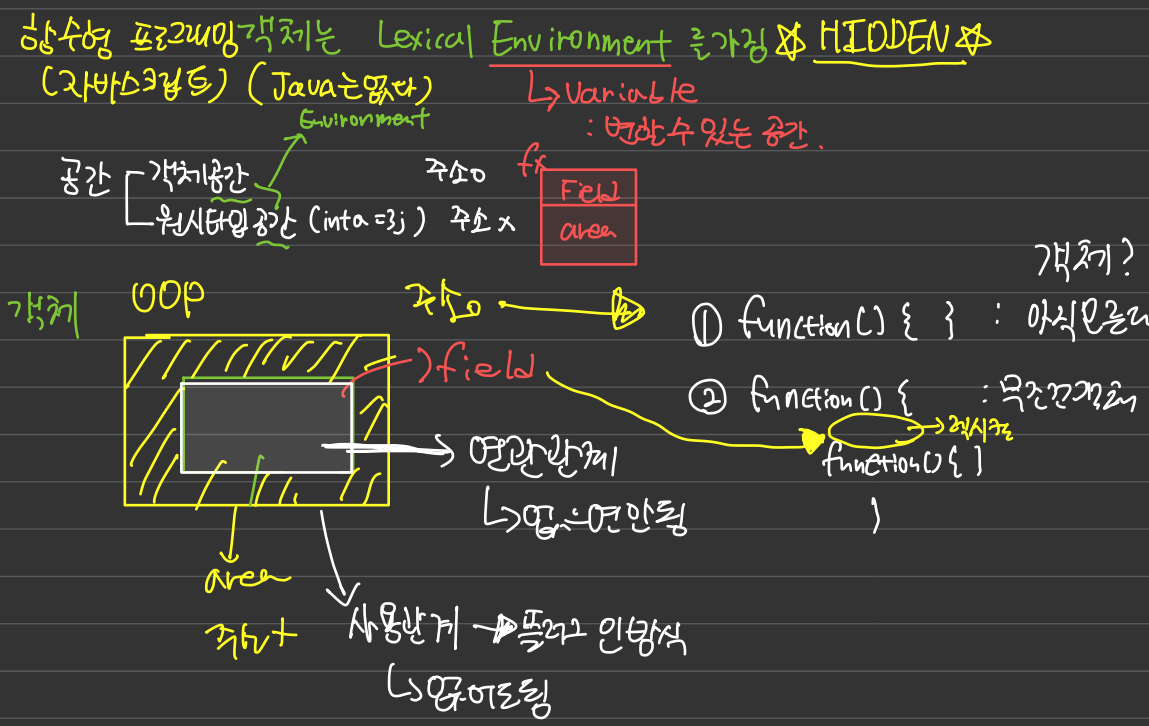


여기의 환경

⇒ 반환해서 const xx = () => { return3



자바스크립트에선 실행 중인 함수, 코드 블록 {...}, 스크립트 전체는 렉시컬 환경(Lexical Environment) 이라 불리는 내부 숨김 연관 객체(internal hidden associated object)를 갖습니다.



스코프는 함수를 호출할 때가 아니라 함수를 어디에 선언하였는지에 따라 결정된다. 이를 **렉시컬 스코핑(Lexical scoping)**라 한다. 위 예제의 함수 innerFunc는 함수 outerFunc의 내부에서 선언되었기 때문에 함수 innerFunc의 상위 스코프는 함수 outerFunc이다. 함수 innerFunc가 전역에 선언되었다면 함수 innerFunc의 상위 스코프는 전역 스코프가 된다.

함수 innerFunc가 함수 outerFunc의 내부에 선언된 내부함수이므로 함수 innerFunc는 자신이 속한 렉시컬 스코프(전역, 함수 outerFunc, 자신의 스코프)를 참조할 수 있다. 이것을 실행 컨텍스트의 관점에서 설명해보자.
내부함수 innerFunc가 호출되면 자신의 실행 컨텍스트가 실행 컨텍스트 스택에 쌓이고 변수 객체(Variable Object)와 스코프 체인(Scope chain) 그리고 this에 바인딩할 객체가 결정된다. 이때 스코프 체인은 전역 스코프를 가리키는 전역 객체와 함수 outerFunc의 스코프를 가리키는 함수 outerFunc의 활성 객체(Activation object) 그리고 함수 자신의 스코프를 가리키는 활성 객체를 순차적으로 바인딩한다. 스코프 체인이 바인딩한 객체가 바로 렉시컬 스코프의 실체이다.

```
function outerFunc() {
  var x = 10;
  var innerFunc = function () { console.log(x); };
  return innerFunc;
}

/**
 * 함수 outerFunc를 호출하면 내부 함수 innerFunc가 반환된다.
 * 그리고 함수 outerFunc의 실행 컨텍스트는 소멸한다.
 */
var inner = outerFunc();
inner(); // 10
```

* Getter / Setter

	input	output
Setter : Consumer	O	X
Getter : Supplier	X	O

실행 컨텍스트

ECMAScript 스펙에 따르면 실행 컨텍스트를 **실행 가능한 코드를 형상화하고 구분하는 추상적인 개념**이라고 정의한다. 좀 더 쉽게 말하자면 **실행 컨텍스트는 실행 가능한 코드가 실행되기 위해 필요한 환경**이라고 말할 수 있겠다.

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

실행 컨텍스트는 실행 가능한 코드를 형상화하고 구분하는 추상적인 개념이지만 물리적으로는 객체의 형태를 가지며 아래의 3가지 프로퍼티를 소유한다.

Variable Object

실행 컨텍스트가 생성되면 자바스크립트 엔진은 실행에 필요한 여러 정보들을 담을 객체를 생성한다. 이를 Variable Object(VO / 변수 객체)라고 한다. Variable Object는 코드가 실행될 때 엔진에 의해 참조되며 코드에서는 접근할 수 없다.
Variable Object는 아래의 정보를 담는 객체이다.

- 변수
- 매개변수(parameter)와 인수 정보(arguments)
- 함수 선언(함수 표현식은 제외)

Variable Object는 실행 컨텍스트의 프로퍼티이기 때문에 값을 갖는데 이 값은 다른 객체를 가리킨다. 그런데 전역 코드 실행시 생성되는 전역 컨텍스트의 경우와 함수를 실행할 때 생성되는 함수 컨텍스트의 경우, 가리키는 객체가 다르다. 이는 전역 코드와 함수의 내용이 다르기 때문이다. 예를 들어 전역 코드에는 매개변수가 없지만 함수에는 매개변수가 있다.
Variable Object가 가리키는 객체는 아래와 같다.

전역 컨텍스트의 경우

Variable Object는 유일하며 최상위에 위치하고 모든 전역 변수, 전역 함수 등을 포함하는 **전역 객체(Global Object / GO)**를 가리킨다. 전역 객체는 전역에 선언된 전역 변수와 전역 함수를 프로퍼티로 소유한다.

함수 컨텍 . 스트의 경우

Variable Object는 **Activation Object(AO / 활성 객체)**를 가리키며 매개변수와 인수들의 정보를 배열의 형태로 담고 있는 객체인 arguments object가 추가된다.

스코프 체인(Scope Chain)은 일종의 리스트로서 전역 객체와 중첩된 함수의 스코프의 레퍼런스를 차례로 저장하고 있다. 다시 말해, 스코프 체인은 해당 전역 또는 함수가 참조할 수 있는 변수, 함수 선언 등의 정보를 담고 있는 전역 객체(GO) 또는 활성 객체(AO)의 리스트를 가리킨다.
현재 실행 컨텍스트의 활성 객체(AO)를 선두로 하여 순차적으로 상위 컨텍스트의 활성 객체(AO)를 가리키며 마지막 리스트는 전역 객체(GO)를 가리킨다.

> **스코프 체인은 식별자 중에서 객체(전역 객체 제외)의 프로퍼티가 아닌 식별자, .즉 변수를 검색하는 메커니즘이다.**
> 식별자 중에서 변수가 아닌 객체의 프로퍼티(물론 메소드도 포함된다)를 검색하는 메커니즘은 **프로토타입 체인(Prototype Chain)**이다.

엔진은 스코프 체인을 통해 렉시컬 스코프를 파악한다. 함수가 중첩 상태일 때 하위함수 내에서 상위함수의 스코프와 전역 스코프까지 참조할 수 있는데 이것은 스코프 체인을 검색을 통해 가능하다. 함수가 중첩되어 있으면 중첩될 때마다 부모 함수의 Scope가 자식 함수의 스코프 체인에 포함된다. 함수 실행중에 변수를 만나면 그 변수를 우선 현재 Scope, 즉 Activation Object에서 검색해보고, 만약 검색에 실패하면 스코프 체인에 담겨진 순서대로 그 검색을 이어가게 되는 것이다. 이것이 스코프 체인이라고 불리는 이유이다.
예를 들어 함수 내의 코드에서 변수를 참조하면 엔진은 스코프 체인의 첫번째 리스트가 가리키는 AO에 접근하여 변수를 검색한다. 만일 검색에 실패하면 다음 리스트가 가리키는 Activation Object(또는 전역 객체)를 검색한다. 이와 같이 순차적으로 스코프 체인에서 변수를 검색하는데 결국 검색에 실패하면 정의되지 않은 변수에 접근하는 것으로 판단하여 Reference 에러를 발생시킨다. 스코프 체인은 함수의 감추인 프로퍼티인 [[Scope]]로 참조할 수 있다

this 프로퍼티에는 this 값이 할당된다. this에 할당되는 값은 함수 호출 패턴에 의해 결정된다.

실행 컨텍스트가 생성된 이후 가장 먼저 스코프 체인의 생성과 초기화가 실행된다. 이때 스코프 체인은 전역 객체의 레퍼런스를 포함하는 리스트가 된다.

스코프 체인의 생성과 초기화

변수 객체화 → 공간생성.

스코프 체인의 생성과 초기화가 종료하면 변수 객체화(Variable Instantiation)가 실행된다. Variable Instantiation은 Variable Object에 프로퍼티와 값을 추가하는 것을 의미한다. 변수 객체화라고 번역하기도 하는데 이는 변수, 매개변수와 인수 정보(arguments), 함수 선언을 Variable Object에 추가하여 객체화하기 때문이다.
전역 코드의 경우, Variable Object는 Global Object를 가리킨다.

Variable Instantiation(변수 객체화): VO와 GO의 연결

Variable Instantiation(변수 객체화)는 아래의 순서로 Variable Object에 프로퍼티와 값을 set한다. (반드시 1→2→3 순서로 실행된다.)

1. (Function Code인 경우) **매개변수(parameter)**가 Variable Object의 프로퍼티로, 인수(argument)가 값으로 설정된다.
2. 대상 코드 내의 **함수** 선언(함수 표현식 제외)을 대상으로 함수명이 Variable Object의 프로퍼티로, 생성된 함수 객체가 값으로 설정된다.(**함수 호이스팅**)
3. 대상 코드 내의 **변수** 선언을 대상으로 변수명이 Variable Object의 프로퍼티로, undefined가 값으로 설정된다.(**변수 호이스팅**)

위 예제 코드를 보면 전역 코드에 변수 x와 함수 foo(매개변수 없음)가 선언되었다. Variable Instantiation의 실행 순서 상, 우선 2. 함수 foo의 선언이 처리되고(함수 코드가 아닌 전역 코드이기 때문에 1. 매개변수 처리는 실행되지 않는다.) 그 후 3. 변수 x의 선언이 처리된다.
#

3.1.2.1 함수 foo의 선언 처리

함수 선언은 Variable Instantiation 실행 순서 2.와 같이 선언된 함수명 foo가 Variable Object(전역 코드인 경우 Global Object)의 프로퍼티로, 생성된 함수 객체가 값으로 설정된다.

함수 foo의 선언 처리

생성된 함수 객체는 [[Scopes]] 프로퍼티를 가지게 된다. [[Scopes]] 프로퍼티는 함수 객체만이 소유하는 내부 프로퍼티(Internal Property)로서 **함수 객체가 실행되는 환경**을 가리킨다. 따라서 현재 실행 컨텍스트의 스코프 체인이 참조하고 있는 객체를 값으로 설정한다. 내부 함수의 [[Scopes]] 프로퍼티는 자신의 실행 환경(Lexical Enviroment)과 자신을 포함하는 외부 함수의 실행 환경과 전역 객체를 가리키는데 이때 자신을 포함하는 외부 함수의 실행 컨텍스트가 소멸하여도 [[Scopes]] 프로퍼티가 가리키는 외부 함수의 실행 환경(Activation object)은 소멸하지 않고 참조할 수 있다. 이것이 **클로저**이다.

함수 foo의 [[Scopes]]

지금까지 살펴본 실행 컨텍스트는 아직 코드가 실행되기 이전이다. 하지만 스코프 체인이 가리키는 변수 객체(VO)에 이미 함수가 등록되어 있으므로 이후 코드를 실행할 때 함수선언식 이전에 함수를 호출할 수 있게 되었다.
이때 알 수 있는 것은 함수선언식의 경우, 변수 객체(VO)에 함수표현식과 동일하게 함수명을 프로퍼티로 함수 객체를 할당한다는 것이다. 단, 함수선언식은 변수 객체(VO)에 함수명을 프로퍼티로 추가하고 즉시 함수 객체를 즉시 할당하지만 함수 표현식은 일반 변수의 방식을 따른다. 따라서 함수선언식의 경우, 선언문 이전에 함수를 호출할 수 있다. 이러한 현상을 **함수 호이스팅(Function Hoisting)**이라 한다.

변수 선언은 Variable Instantiation 실행 순서 3.과 같이 선언된 변수명(x)이 Variable Object의 프로퍼티로, undefined가 값으로 설정된다. 이것을 좀더 세분화 해보면 아래와 같다.
선언 단계(Declaration phase)
변수 객체(Variable Object)에 변수를 등록한다. 이 변수 객체는 스코프가 참조할 수 있는 대상이 된다.
초기화 단계(Initialization phase)
변수 객체(Variable Object)에 등록된 변수를 메모리에 할당한다. 이 단계에서 변수는 undefined로 초기화된다.
할당 단계(Assignment phase)
undefined로 초기화된 변수에 실제값을 할당한다.
var 키워드로 선언된 변수는 선언 단계와 초기화 단계가 한번에 이루어진다. 다시 말해 스코프 체인이 가리키는 변수 객체에 변수가 등록되고 변수는 undefined로 초기화된다. 따라서 변수 선언문 이전에 변수에 접근하여도 Variable Object에 변수가 존재하기 때문에 에러가 발생하지 않는다. 다만 undefined를 반환한다. 이러한 현상을 **변수 호이스팅(Variable Hoisting)**이라 한다.
아직 변수 x는 'xxx'로 초기화되지 않았다. 이후 변수 할당문에 도달하면 비로소 값의 할당이 이루어진다.

변수 선언 처리가 끝나면 다음은 this value가 결정된다. **this value가 결정되기 이전에 this는 전역 객체를 가리키고 있다가 함수 호출 패턴에 의해 this에 할당되는 값이 결정된다.** 전역 코드의 경우, this는 전역 객체를 가리킨다.

this value 결정
전역 컨텍스트(전역 코드)의 경우, Variable Object, 스코프 체인, this 값은 언제나 전역 객체이다.