

# JavaScript

함수2 – 김근형 강사

# 스코프(Scope)

- 스코프(Scope)
  - 변수의 유효범위
    - 스코프는 참조 대상 식별자(identifier, 변수, 함수의 이름과 같이 어떤 대상을 다른 대상과 구분하여 식별할 수 있는 유일한 이름)를 찾아내기 위한 규칙이다. 자바스크립트는 이 규칙대로 식별자를 찾는다.
    - 프로그래밍은 변수를 선언하고 값을 할당하며 변수를 참조하는 기본적인 기능을 제공하며 이것으로 프로그램의 상태를 관리할 수 있다.
    - 변수는 전역 또는 코드 블록(if, for, while, try/catch 등)이나 함수 내에 선언하며 코드 블록이나 함수는 중첩될 수 있다.
    - 식별자는 자신이 어디에서 선언됐는지에 의해 자신이 유효한(다른 코드가 자신을 참조할 수 있는) 범위를 갖는다.

# 스코프(Scope)

- 스코프 구분(코드)

## 전역 스코프 (Global scope)

코드 어디에서든지 참조할 수 있다.

## 지역 스코프 (Local scope or Function-level scope)

함수 코드 블록이 만든 스코프로 함수 자신과 하위 함수에서만 참조할 수 있다.

# 스코프(Scope)

- 스코프 구분(변수)

## 전역 변수 (Global variable)

전역에서 선언된 변수이며 어디에든 참조할 수 있다.

## 지역 변수 (Local variable)

지역(함수) 내에서 선언된 변수이며 그 지역과 그 지역의 하부 지역에서만 참조할 수 있다.

# 스코프(Scope)

## ○ 스코프 레벨

- 보통 스코프가 어디에서 적용되는지를 적용하는 기준
- 일반적인 언어는 **블록 레벨 스코프**를 적용한다. 블록레벨 스코프는 브레이스({})라는 블록 기준으로 스코프를 나누는 것을 의미한다.
- 하지만 자바스크립트는 **함수 레벨 스코프**를 지향한다.
- 즉 자바스크립트는 함수 단위로 스코프를 지정하여 로직을 실행한다.
- 이후 ECMA6에서는 자바스크립트 또한 함수 단위가 아닌 블록단위 스코프를 지향하는 변수인 `let`과 `const`를 만든다.(그렇다고 함수 단위 레벨이 안되는 것은 아님)
- 스코프를 통해 해당 변수의 생존 범위를 알 수 있다.

# 스코프(Scope)

## ○ 스코프(Scope) 예제

```
var x = 'global';

function foo () {
  var x = 'function scope'
  console.log(x);
}

foo(); // function scope
console.log(x); // global
```

} 'function scope' 생존 범위

} 'global' 생존 범위

# 전역 스코프(Global Scope)

- 전역 스코프(Global Scope)
  - 전역에 변수를 선언하면 이 변수는 어디서든지 참조할 수 있는 전역 스코프를 갖는 전역 변수가 된다.
  - var 키워드로 선언한 전역 변수는 전역 객체(Global Object) window의 프로퍼티이다.
  - 전역 변수의 사용은 변수 이름이 중복될 수 있고, 의도치 않은 재할당에 의한 상태 변화로 코드를 예측하기 어렵게 만드므로 사용을 억제하여야 한다.

```
var global = 'global';
```

```
function foo() {  
    var local = 'local';  
    console.log(global);  
    console.log(local);  
}  
foo();
```

```
console.log(global);  
// Uncaught ReferenceError:  
// local is not defined  
console.log(local);
```

```
global  
local  
global  
✖ ▶ Uncaught ReferenceError:  
local is not defined
```

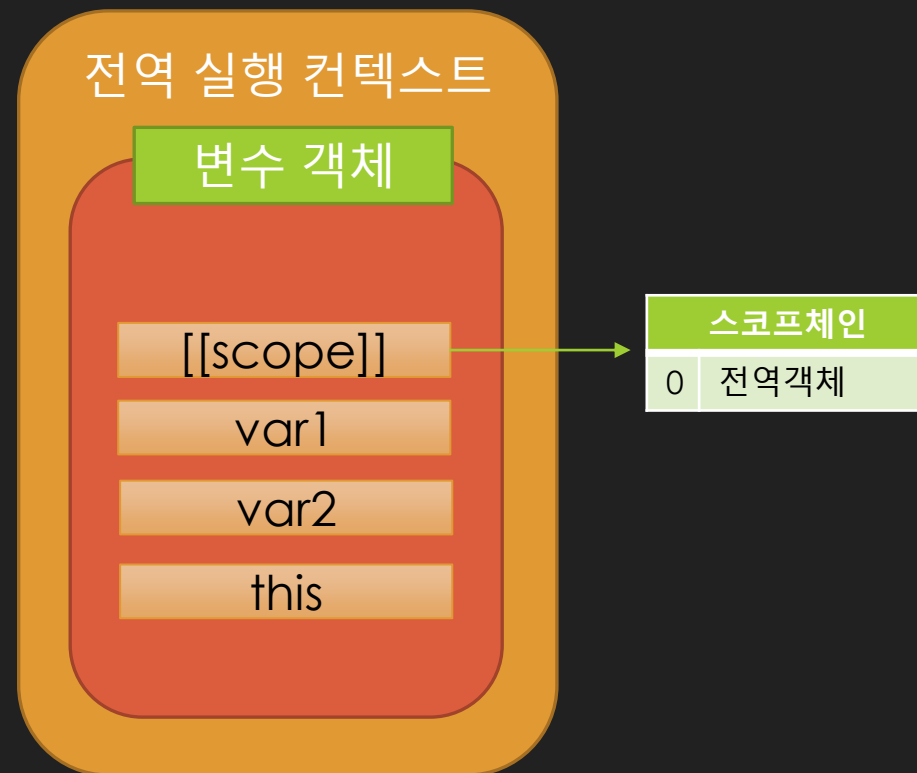
```
> console.dir(window)  
  
▼ Window ⓘ  
  ▶ alert: f alert()  
  ▶ atob: f atob()  
    global: "global"
```

# 전역 스코프(Global Scope)

- 전역 실행 컨텍스트 상에서의 전역 스코프 체인
  - 전역 실행 컨텍스트는 참조할 상위 컨텍스트가 존재하지 않는다.
  - 따라서 전역 객체의 스코프 체인은 자기 자신만을 가진다.

```
var var1 = 1;  
var var2 = 2;  
console.log(var1); // 1  
console.log(var2); // 2
```

```
▼ Window ⓘ  
  var1: 1  
  var2: 2
```





# 전역 스코프(Global Scope)

- 비 블록 레벨 스코프(Non block-level scope)
  - 자바스크립트는 블록 레벨 스코프를 사용하지 않으므로 함수 밖에서 선언된 변수는 코드 블록 내에서 선언되었다 할지라도 모두 전역 스코프를 갖게된다.

```
if (true) {  
  |   var x = 5;  
}  
console.log(x);
```

```
console.dir(window.x);  
5
```

# 함수 레벨 스코프(Function Level Scope)

- 함수 레벨 스코프(Function-level scope)
  - 자바스크립트는 함수 레벨 스코프를 사용한다.
  - 즉, 함수 내에서 선언된 매개변수와 변수는 함수 외부에서는 유효하지 않다.
  - 따라서 함수 내에 선언된 변수는 지역 변수이다.

```
<script>
  var a = 10;    // 전역변수

  function bbb() {
    var b = 20;  // 지역변수
    console.log(`global a : ${a}`); // 10
    console.log(`inner b : ${b}`); // 20
  }
  bbb();

  console.log(a); // 10
  console.log(b); // "b" is not defined
</script>
```

```
global a : 10
inner b : 20
10
✖ ▶ Uncaught ReferenceError: b is not defined
   at 05-FunctionLevelScope1.html:23
```

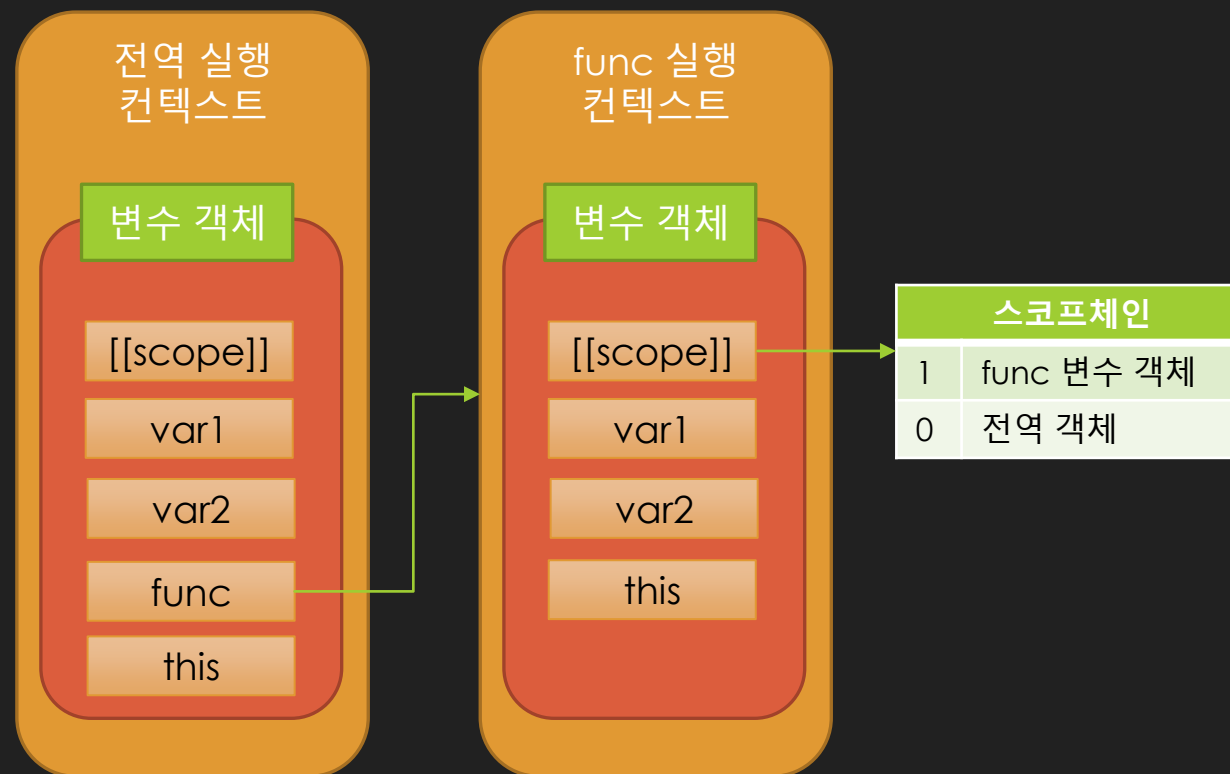
# 스코프 체인(Scope Chain)

- 스코프 체인(Scope Chain)
  - 각각의 함수에서 `[[Scope]]` 프로퍼티로 자신이 생성한 실행 컨텍스트 내부에서 참조되는 값
  - 함수가 실행 되면 해당 실행 컨텍스트는 실행된 함수의 `[[Scope]]` 프로퍼티를 기반으로 새로운 스코프 체인을 만든다.
  - `[[Scope]]`는 함수가 실행 될 때가 아니라 선언될 때 생성되며 `[[Scope]]`를 통해 함수가 어디를 참조하고 있는지 알 수 있다.

# 스코프 체인(Scope Chain)

- 함수를 호출한 경우 생성되는 실행 컨텍스트의 스코프 체인

```
var var1 = 1;
var var2 = 2;
function func(){
  var var1 = 10;
  var var2 = 20;
  console.log(var1); // 10
  console.log(var2); // 20
}
func();
console.log(var1); // 1
console.log(var2); // 2
```



# 스코프 체인(Scope Chain)

- 함수를 호출한 경우 생성되는 실행 컨텍스트의 스코프 체인

스코프 체인 = 현재 실행 컨텍스트의 변수 객체 + 상위 컨텍스트의 **참조한** 스코프 체인

- 각 함수 객체는 `[[scope]]` 프로퍼티로 현재 컨텍스트의 스코프 체인을 참조한다.
- 한 함수가 실행되면 새로운 실행 컨텍스트가 만들어지며, 가장 기본적으로 전역 스코프가 같이 따라오게 된다.
- 참조하고자 하는 변수나 함수가 상위에 존재할 경우 해당 컨텍스트의 함수를 스코프로 연결해주지만 그렇지 않을 경우 해당 컨텍스트의 스코프는 생성하지 않는다.

# 스코프 체인(Scope Chain)

- 함수를 호출한 경우 생성되는 실행 컨텍스트의 스코프 체인

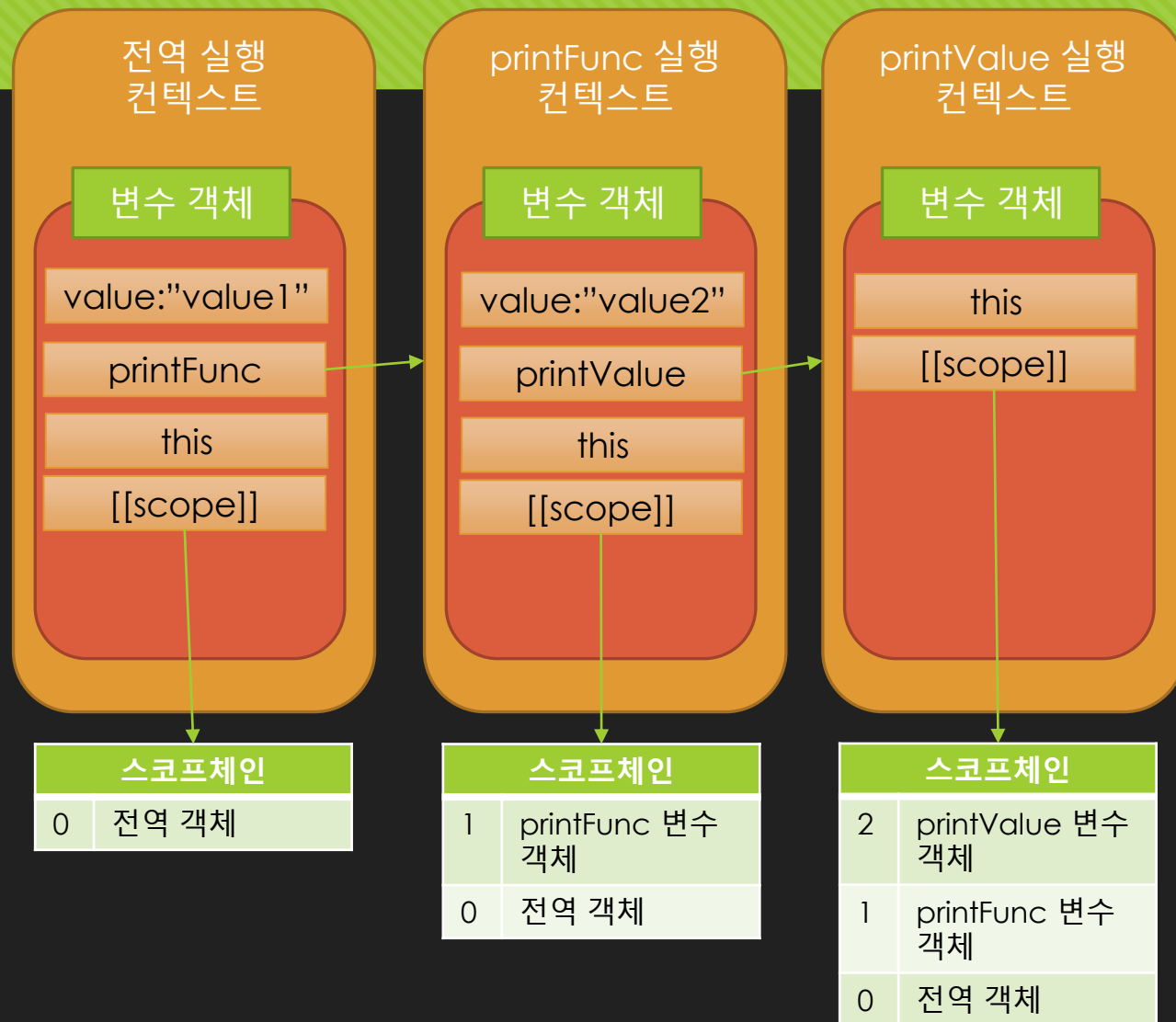
```
var value = "value1";

function printFunc(func) {
  var value = "value2";

  function printValue() {
    return value;
  }

  console.log(printValue());
}

printFunc();
```



# 스코프 체인(Scope Chain)

- 함수를 호출한 경우 생성되는 실행 컨텍스트의 스코프 체인
  - console.dir을 이용한 각 함수의 [[Scope]] 확인

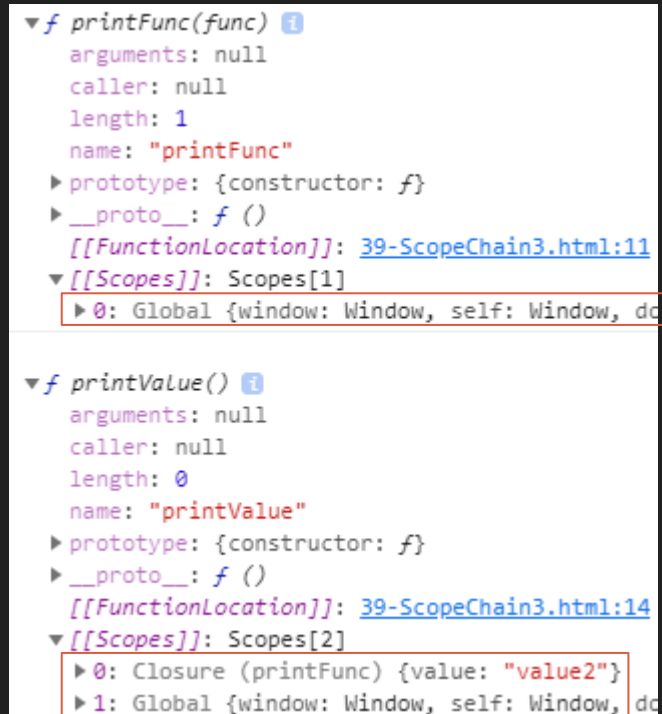
```
var value = "value1";

function printFunc(func) {
  var value = "value2";

  function printValue() {
    return value;
  }

  console.dir(printValue);
  console.log(printValue());
}

console.dir(printFunc);
printFunc();
```



The screenshot shows the Chrome DevTools console with two function objects expanded. The first function is `printFunc`, and the second is `printValue`. Both functions have a `[[Scopes]]` property. For `printFunc`, the scope chain is `Scopes[1]` pointing to the Global object. For `printValue`, the scope chain is `Scopes[2]`, which includes the closure for `printFunc` (containing `value: "value2"`) and the Global object.

```
▼ f printFunc(func) ⓘ
  arguments: null
  caller: null
  length: 1
  name: "printFunc"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: 39-ScopeChain3.html:11
  ▼ [[Scopes]]: Scopes[1]
    ▶ 0: Global {window: Window, self: Window, do

▼ f printValue() ⓘ
  arguments: null
  caller: null
  length: 0
  name: "printValue"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: 39-ScopeChain3.html:14
  ▼ [[Scopes]]: Scopes[2]
    ▶ 0: Closure (printFunc) {value: "value2"}
    ▶ 1: Global {window: Window, self: Window, do
```

# 스코프 체인(Scope Chain)

- 중간 컨텍스트의 자원을 참조하지 않는 경우의 Scope 변화

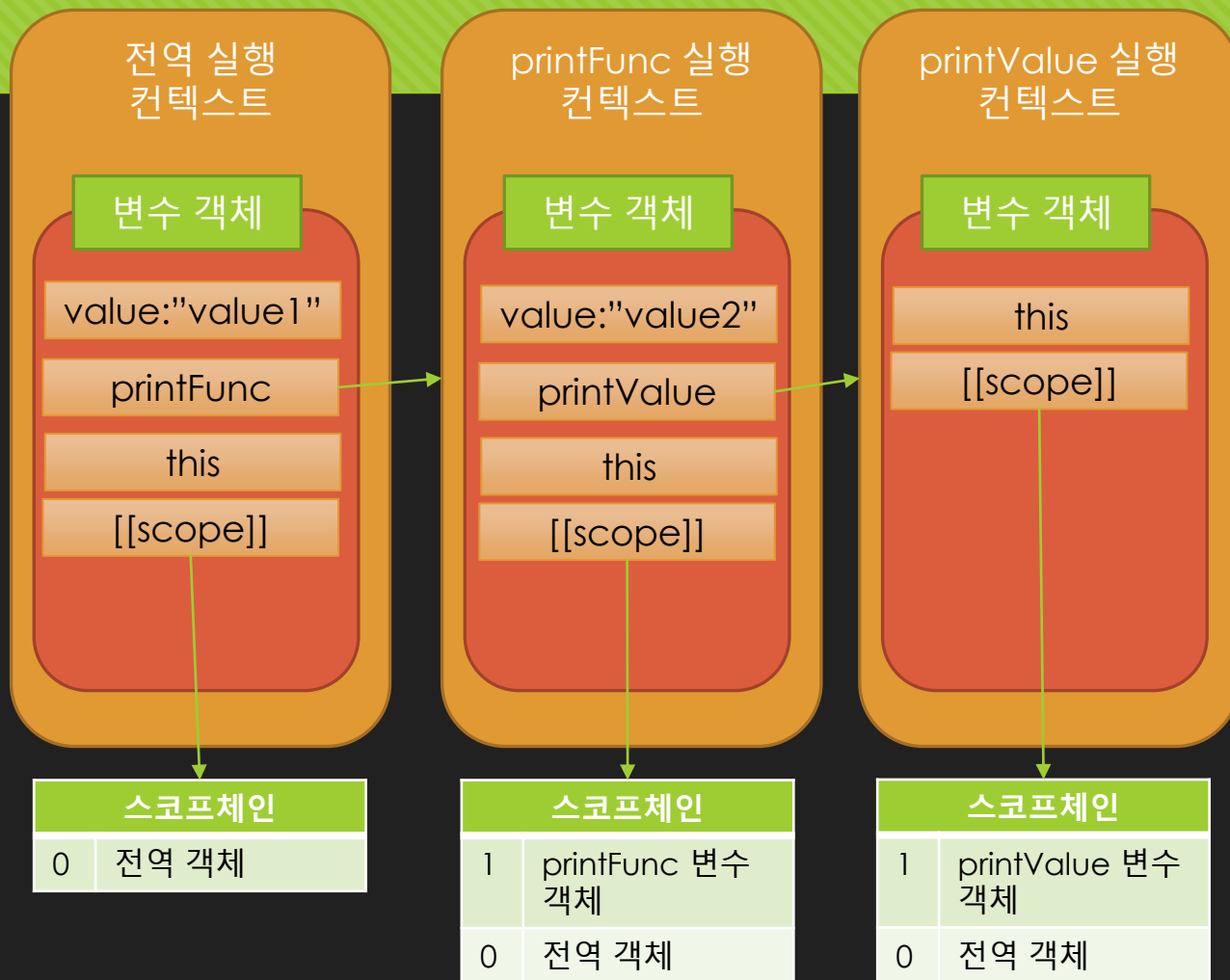
```
var value1 = "value1";

function printFunc(func) {
  var value2 = "value2";

  function printValue() {
    return value1;
  }

  console.dir(printValue);
  console.log(printValue());
}

console.dir(printFunc);
printFunc();
```





# 스코프 체인(Scope Chain)

- 중간 컨텍스트의 자원을 참조하지 않는 경우의 Scope 변화 – (Log 참조)

```
var value1 = "value1";

function printFunc(func) {
  var value2 = "value2";

  function printValue() {
    return value1;
  }

  console.dir(printValue);
  console.log(printValue());
}

console.dir(printFunc);
printFunc();
```

The screenshot shows the Chrome DevTools console with two function objects expanded. The top function is `printFunc`, and the bottom function is `printValue`. Both functions have their `[[Scopes]]` property expanded, showing a single entry: `0: Global {window: Window, self: ...}`. This indicates that both functions share the same global scope, as `printValue` does not access any variables from its parent scope (`printFunc`).

```
▼ f printFunc(func) ⓘ
  arguments: null
  caller: null
  length: 1
  name: "printFunc"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: 08-FunctionSc
  ▼ [[Scopes]]: Scopes[1]
    ▶ 0: Global {window: Window, self: ...}

▼ f printValue() ⓘ
  arguments: null
  caller: null
  length: 0
  name: "printValue"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: 08-FunctionSc
  ▼ [[Scopes]]: Scopes[1]
    ▶ 0: Global {window: Window, self: ...}
```

# 렉시컬 스코프(Lexical Scope)

- 스코프 방식의 종류

- 동적 스코프(Dynamic scope) : 함수를 어디서 호출하였는지에 따라 상위 스코프를 결정하는 방식
- 렉시컬 스코프(Lexical scope) : 함수를 어디서 선언하였는지에 따라 상위 스코프를 결정하는 방식
- 자바스크립트는 렉시컬 스코프 방식으로 움직인다.

# 렉시컬 스코프(Lexical Scope)

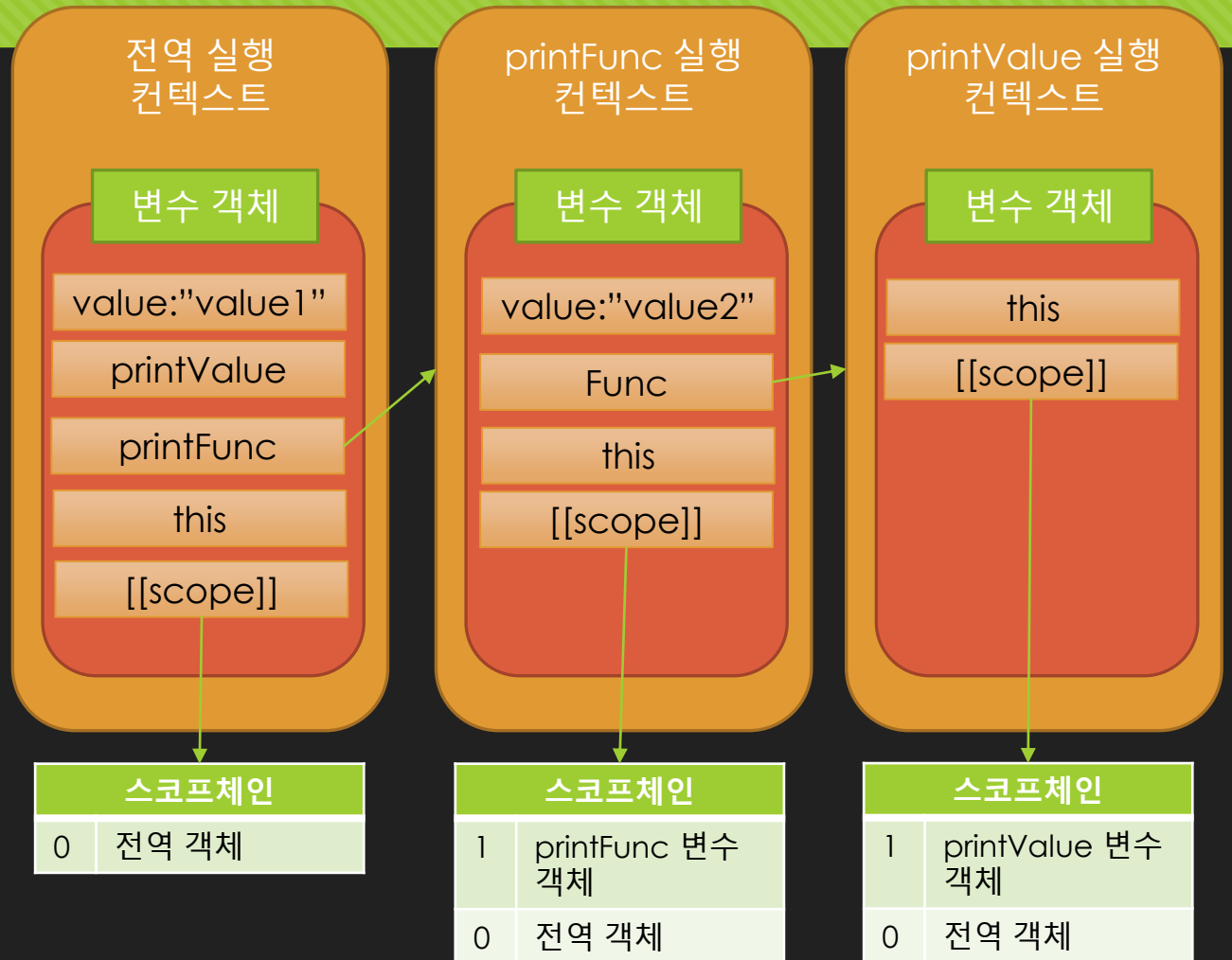
- 렉시컬 스코프를 활용한 스코프 체인 예제

```
var value = "value1";

function printValue() {
  return value;
}

function printFunc(func) {
  var value = "value2";
  console.log(func());
}

printFunc(printValue);
```



# 렉시컬 스코프(Lexical Scope)

- 렉시컬 스코프를 활용한 스코프 체인 예제 결과

```
var value = "value1";

function printValue() {
  return value;
}

function printFunc(func) {
  var value = "value2";
  console.log(func());
}

printFunc(printValue);
console.dir(printFunc);
console.dir(printValue);
```

```
value1

▼ f printFunc(func) ⓘ
  arguments: null
  caller: null
  length: 1
  name: "printFunc"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: 09-LexicalScopeChain
  ▼ [[Scopes]]: Scopes[1]
    ▶ 0: Global {window: Window, self: Window, ...}

▼ f printValue() ⓘ
  arguments: null
  caller: null
  length: 0
  name: "printValue"
  ▶ prototype: {constructor: f}
  ▶ __proto__: f ()
  [[FunctionLocation]]: 09-LexicalScopeChain
  ▼ [[Scopes]]: Scopes[1]
    ▶ 0: Global {window: Window, self: Window, ...}
```