

# Callback and Closure



Building Modern Web Applications - VSP2023

Karthik Pattabiraman  
Kumseok Jung  
Mohsen Salehi

# Function

1. **Function**
2. Callback Function
3. Closure Function



## Function: Recap

- JavaScript functions are **not typed**
- JavaScript functions are **first-class objects**
  - They can be **assigned to variables**
  - They can be **passed as arguments** into another function call
  - Functions can **return other functions**
- Function Declarations have the format:
  - `function functionName (arg, arg, ...) { /* body */ }`
- Function Expressions can create **anonymous functions**
  - `var x = function (arg, arg, ...) { /* body */ }`



## Function: Variadic Function

- JavaScript functions **cannot be overloaded**
- To emulate function overloading, we can define a **variadic function** using the special **arguments** object



## Function: Variadic Function

- JavaScript functions **cannot be overloaded**
- To emulate function overloading, we can define a **variadic function** using the special **arguments** object



```
1 function sayHi (firstName, lastName){
2     console.log("Hi " + firstName + " " + lastName);
3 };
4 function sayHi (firstName, middleName, lastName){
5     console.log("Hi " + firstName + " " + middleName + " " + lastName);
6 };
7
8 sayHi("Alice", "Brown"); // prints: Alice Brown undefined
```

## Function: Variadic Function

- JavaScript functions **cannot be overloaded**
- To emulate function overloading, we can define a **variadic function** using the special `arguments` object



```
1 function sayHi (){
2     if (arguments.length < 3)
3         console.log("Hi " + arguments[0] + " " + arguments[1]);
4     else
5         console.log("Hi " + arguments[0] + " " + arguments[1] + " " +
6 arguments[2]);
7 };
8 sayHi("Alice", "Brown"); // prints: Alice Brown
```

TRY IT!

## Function: Primitive vs Complex Objects

- The semantics of a function argument differs for primitive object versus complex objects
  - Primitive objects are passed by **value** - the function makes its own copy of the object passed as an argument
  - Complex objects are passed by **reference** - the function uses the same object passed as an argument



## Function: Primitive vs Complex Objects



- The semantics of a function argument differs for primitive object versus complex objects
  - Primitive objects are passed by **value** - the function makes its own copy of the object passed as an argument
  - Complex objects are passed by **reference** - the function uses the same object passed as an argument

```
1 function foo (x){  
2     x = 2;  
3 };  
4  
5 var y = 1;  
6 foo(y);  
7 console.log(y);    // prints: 1
```

**TRY IT!**



## Function: Primitive vs Complex Objects



- The semantics of a function argument differs for primitive object versus complex objects
  - Primitive objects are passed by **value** - the function makes its own copy of the object passed as an argument
  - Complex objects are passed by **reference** - the function uses the same object passed as an argument

```
1 function foo (x){  
2   x.z = 2;  
3 };  
4  
5 var y = { z: 1 };  
6 foo(y);  
7 console.log(y.z);    // prints: 2
```

**TRY IT!**

## Function: Nesting

- JavaScript functions can be nested arbitrarily



```
1 function alpha (x){
2   var i = x + x;
3   function bravo (y){
4     var j = y + i;
5     function charlie (z){
6       var k = z + j;
7       return k;
8     }
9     return charlie(j);
10  }
11  return bravo(i);
12 };
13
14 console.log(alpha(1)); // prints?
```

**TRY IT!**

## Function: Nesting

- JavaScript functions can be nested arbitrarily



```
1 function alpha (x){
2   var i = x + x;
3   function bravo (y){
4     var j = y + i;
5     function charlie (z){
6       var k = z + j;
7       return k;
8     }
9     return charlie(j);
10  }
11  return bravo(i);
12 };
13
14 console.log(alpha(1)); // prints: 8
```

**TRY IT!**

## Function: Scope

- Each function creates its own **scope** when invoked



```
1 function alpha (x){
2   var i = x + x;
3   function bravo (y){
4     var i = y + y;
5     console.log(i);    // prints: 4
6   }
7   bravo(i);
8   console.log(i);      // prints: 2
9 };
10
11 alpha(1);
12
13
14
```

TRY IT!

## Function: Scope

- A child function has access to its parent's **scope**



```
1 function alpha (x){
2   var i = x + x;
3   function bravo (y){
4     i = y + y;
5     console.log(i);    // prints: 4
6   }
7   bravo(i);
8   console.log(i);      // prints: 4
9 };
10
11 alpha(1);
12
13
14
```

TRY IT!

## Function: Scope

- A parent function does not have access to its child's **scope**



```
1 function alpha (x){
2   var i = x + x;
3   function bravo (y){
4     var j = y + y;
5     console.log(i);    // prints: 2
6   }
7   bravo(i);
8   console.log(j);      // throws: ReferenceError: j is not defined
9 };
10
11 alpha(1);
12
13
14
```

**TRY IT!**

## Function: First-Class Objects

- Functions can be passed to other functions as arguments



```
1 function filter (list, f){
2   var arr = [];
3   for (var i = 0; i < list.length; i++){
4     if (f(list[i]) === true) arr.push(list[i]);
5   }
6   return arr;
7 };
8
9 var myList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
10 var filtered = filter(myList, function (item){
11   return (item < 5);
12 });
13 console.log(filtered);      // prints: 0, 1, 2, 3, 4
14
```

**TRY IT!**

## Function: Class Activity



[lectures/lecture-4/activity1.js](https://github.com/lectures/lecture-4/activity1.js)

- Implement the `map` function, which takes in an Array and a function `f` as arguments, and returns a new Array whose elements are the result of applying `f` on each of the items in the original Array



```
1 function map (list, f){
2   // to implement
3 };
4
5 var myList = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
6 var tList = map(myList, function (item){
7   return item + 5;
8 });
9 console.log(tList);      // prints: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
```



# Callback Function

1. Function
- 2. Callback Function**
3. Closure Function

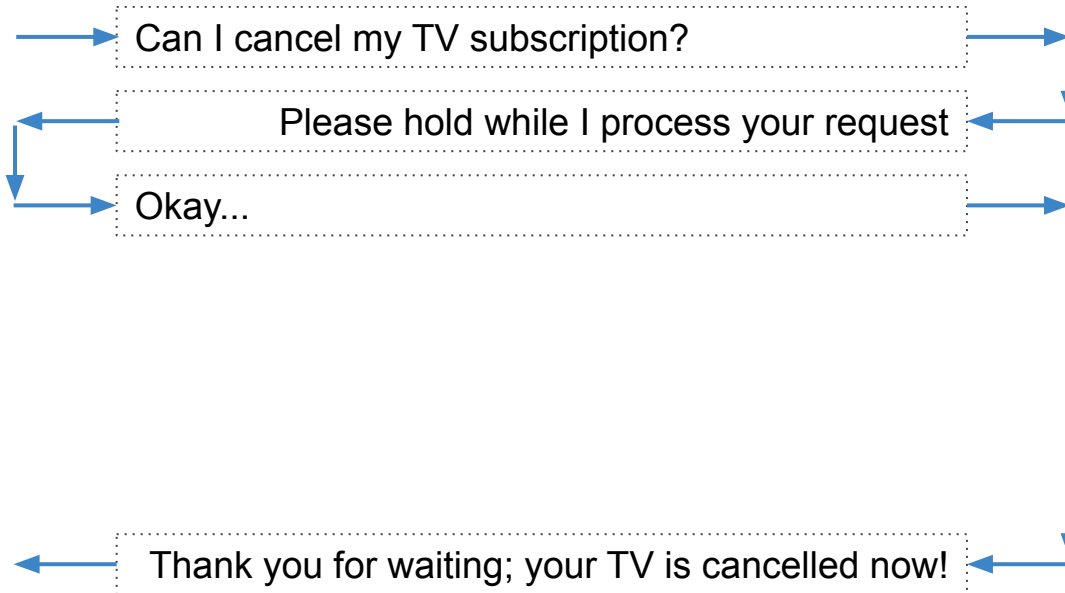


# Callback Function

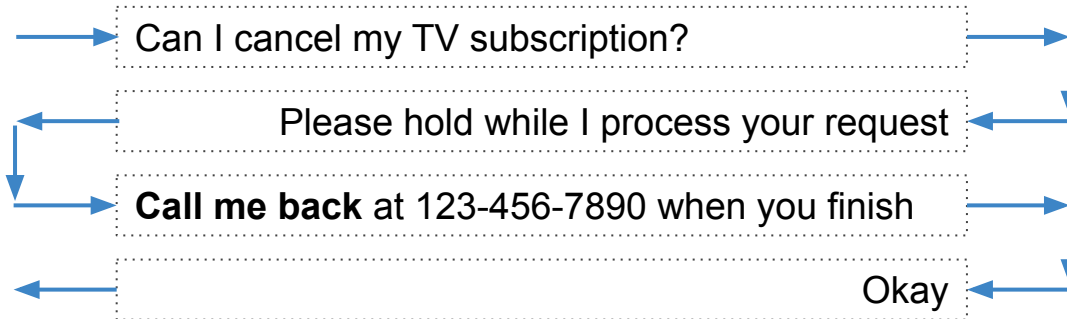
- Callback functions are just regular functions, used in a certain way
  - They are not some special function type
- Used for performing **asynchronous operations**
  - JavaScript applications are **full of asynchronous operations**, so callbacks are used very frequently
  - Most notable examples are **event listeners**
- Why use callbacks?
  - Some operations are **fundamentally asynchronous** (e.g., network requests)
  - We **don't want to wait for result indefinitely**. We would rather get a **call back** when something is done.



# Callback Function



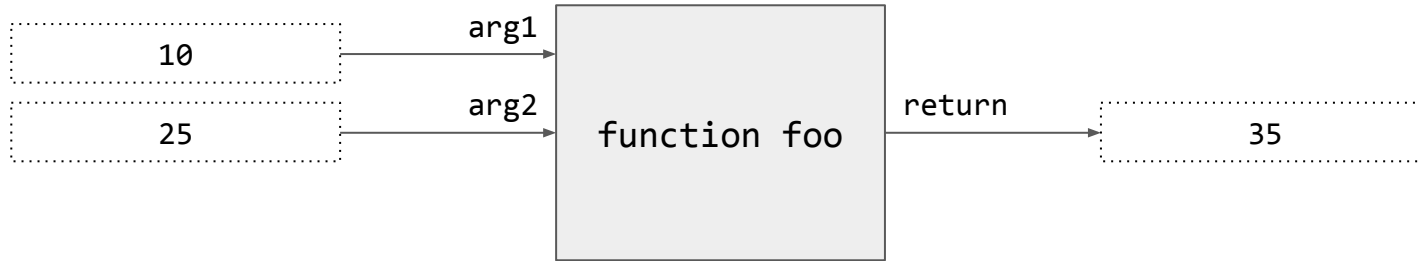
# Callback Function



Do other  
things

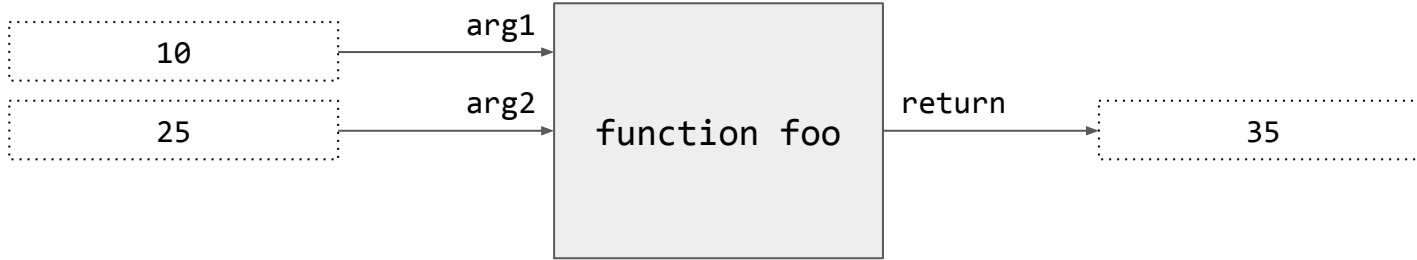


# Callback Function

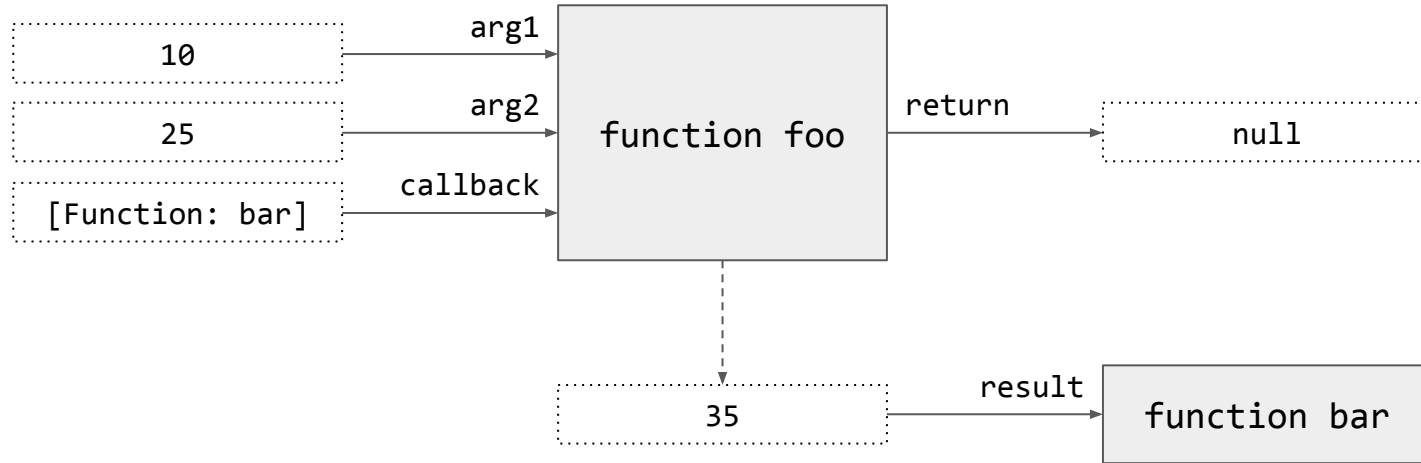


```
1 function foo (arg1, arg2){  
2   // ... do something ...  
3   return result;  
4 };  
5  
6 var result = foo(10, 25);  
7 console.log(result);
```

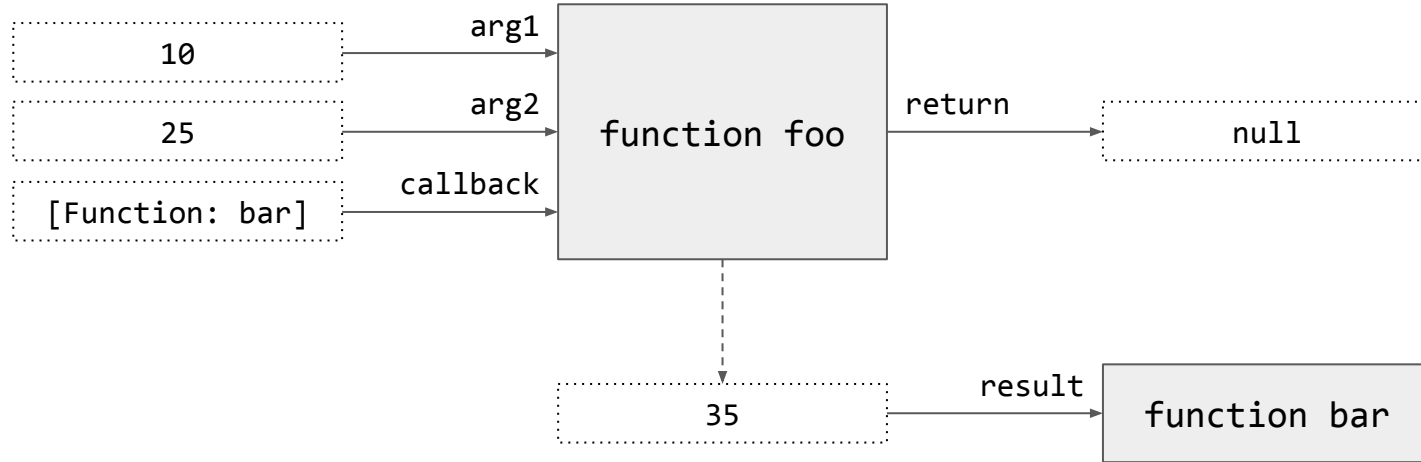
# Callback Function



# Callback Function



# Callback Function



“Here is a function, don’t call it now,  
but call it later when something happens”



# Callback Function

```
1 function foo (arg1, arg2, callback){
2   // ... do something ...
3   someAsync(function(){
4     // inside the callback of some asynchronous operation
5     callback(result);
6   });
7   return null;
8 };
9
10 foo(10, 25, function bar(result){
11   console.log(result);
12 });
```



## Callback Function



```
1 function asyncFunction (arg1, arg2, callback){
2     /*
3         do some asynchronous operations
4     */
5     callback(result);    // invoke callback when result is available
6     return null          // return immediately
7 };
8
9 asyncFunction(val1, val2, function(result){
10     /* do something with result */
11 });                      // this call returns null immediately
12
13 /* do other things */
14
```

## Callback Function: Class Activity



[lectures/lecture-4/activity2.js](https://github.com/lectures/lecture-4/activity2.js)

- Implement `whenBothFinish` function, which takes in 2 functions `fn1`, `fn2` and schedules them to execute after a random delay. It should invoke the 3rd argument `callback` when both have executed



```
1 function whenBothFinish (fn1, fn2, callback){
2   /*
3    to implement: use setTimeout(___, Math.floor(Math.random()*1000))
4    for scheduling fn1 and fn2
5   */
6 };
7
8 whenBothFinish(
9   function(){ console.log("fn1 finished!"); },
10  function(){ console.log("fn2 finished!"); },
11  function(){ console.log("Both functions finished!"); }
12 );
```

# Closure Function

1. Function
2. Callback Function
3. **Closure Function**



# Closure Function

- Closure functions are just regular functions, used in a certain way
  - They are not some special function type
- Closures are functions that carry references outside of their own scope
  - Used to hide objects while still providing the functionality
  - Used to create stateful functions



# Closure Function



```
1 function makeCounter (initial, increment){
2   var count = initial;
3   return function next(){
4     count += increment;
5     return count;
6   }
7 };
8 var counter = makeCounter(3, 1);
9 console.log(counter());      // prints: 4
10 console.log(counter());     // prints: 5
11 console.log(counter());     // prints: 6
12 console.log(count);         // prints: undefined
```

**TRY IT!**

# Closure Function



```
1 function makeCounter (initial, increment){
2   var count = initial;
3   return function next(){
4     count += increment;
5     return count;
6   }
7 };
8 var counter1 = makeCounter(3, 1);
9 var counter2 = makeCounter(5, 5);
10 console.log(counter1());      // prints: 4
11 console.log(counter2());      // prints: 10
12 console.log(counter1());      // prints: 5
13 console.log(counter2());      // prints: 15
```

**TRY IT!**

# Closure Function

window
makeCounter: [Fn]



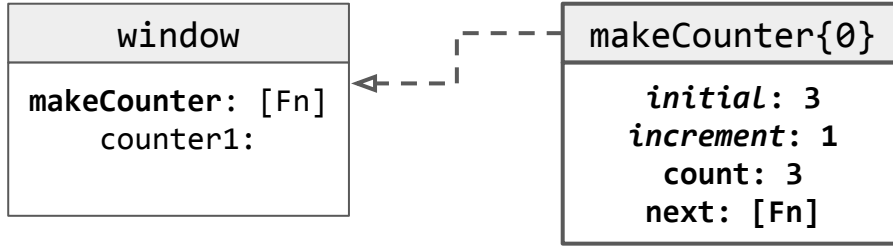
## Legend

- ▶ Direct Reference
- - - ▶ Access to Scope

```
1 function makeCounter (initial,  
2 increment){  
3     /* makeCounter code */  
4 };  
5  
6
```



# Closure Function

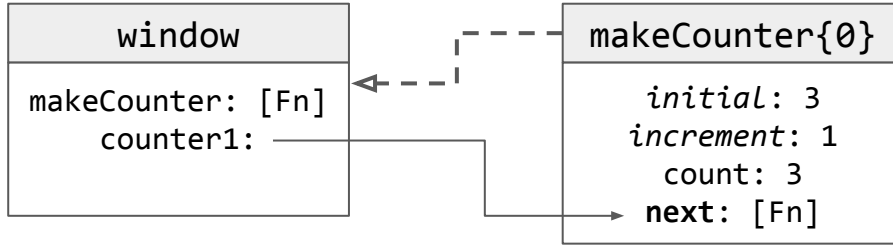


## Legend

- ▶ Direct Reference
- - - ▶ Access to Scope

```
1 var counter1 = makeCounter(3, 1);
2
3
4
5
6
```

# Closure Function

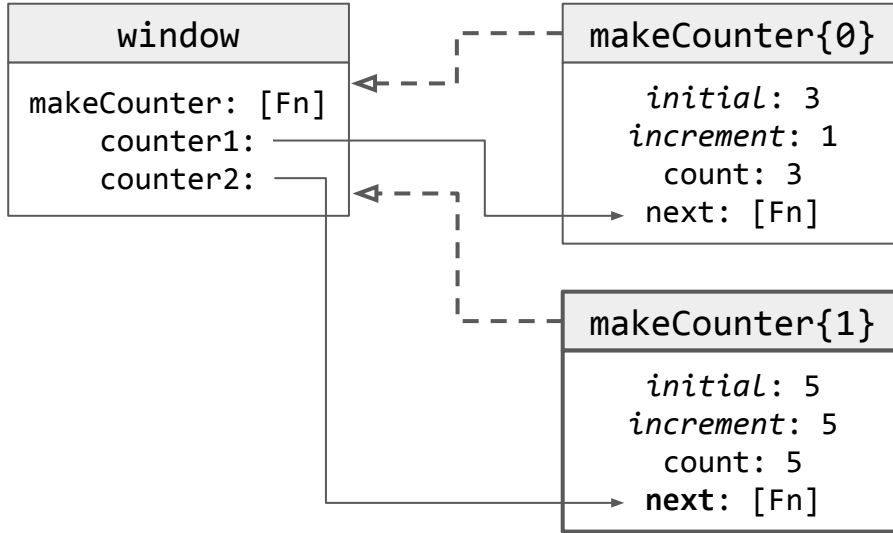


## Legend

- ▶ Direct Reference
- - - ▶ Access to Scope

```
1 var counter1 = makeCounter(3, 1);
2
3
4
5
6
```

# Closure Function

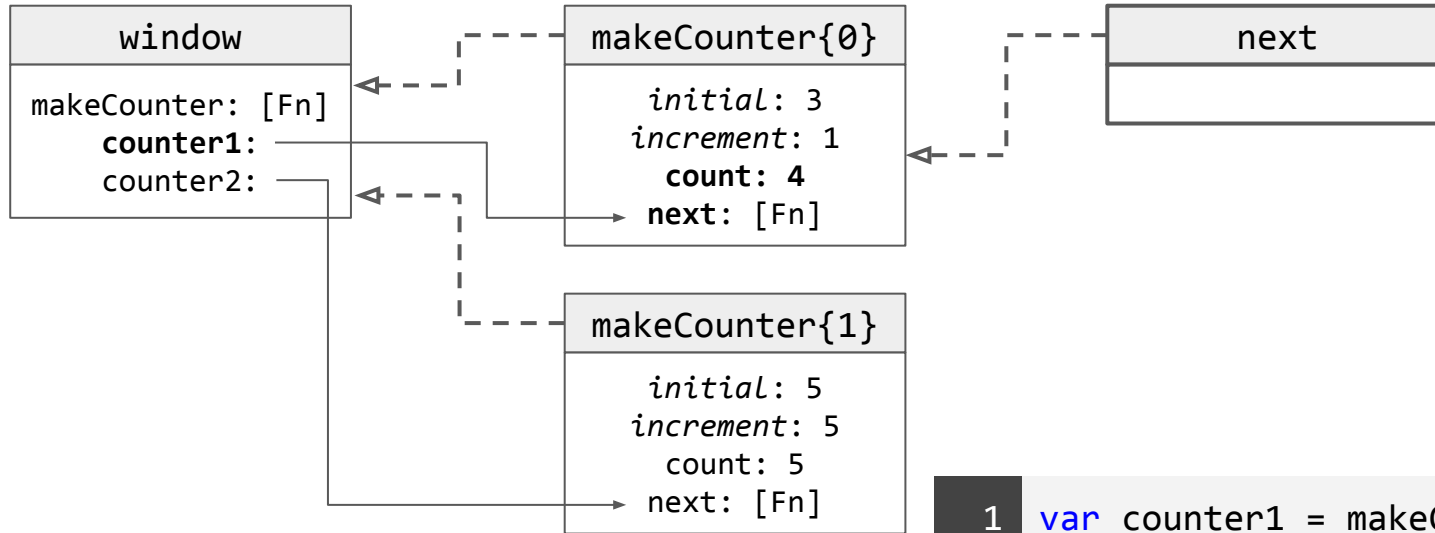


## Legend

- ▶ Direct Reference
- - - ▶ Access to Scope

```
1 var counter1 = makeCounter(3, 1);
2 var counter2 = makeCounter(5, 5);
3
4
5
6
```

# Closure Function

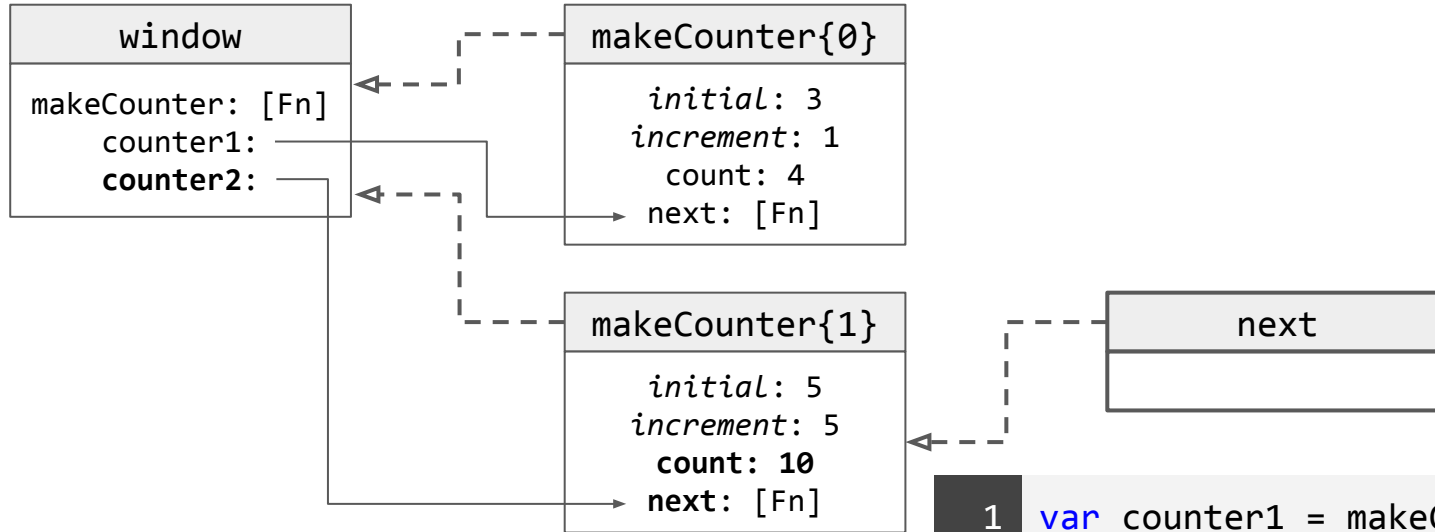


## Legend

- ▶ Direct Reference
- - - ▶ Access to Scope

```
1 var counter1 = makeCounter(3, 1);  
2 var counter2 = makeCounter(5, 5);  
3 console.log(counter1());  
4  
5  
6
```

# Closure Function

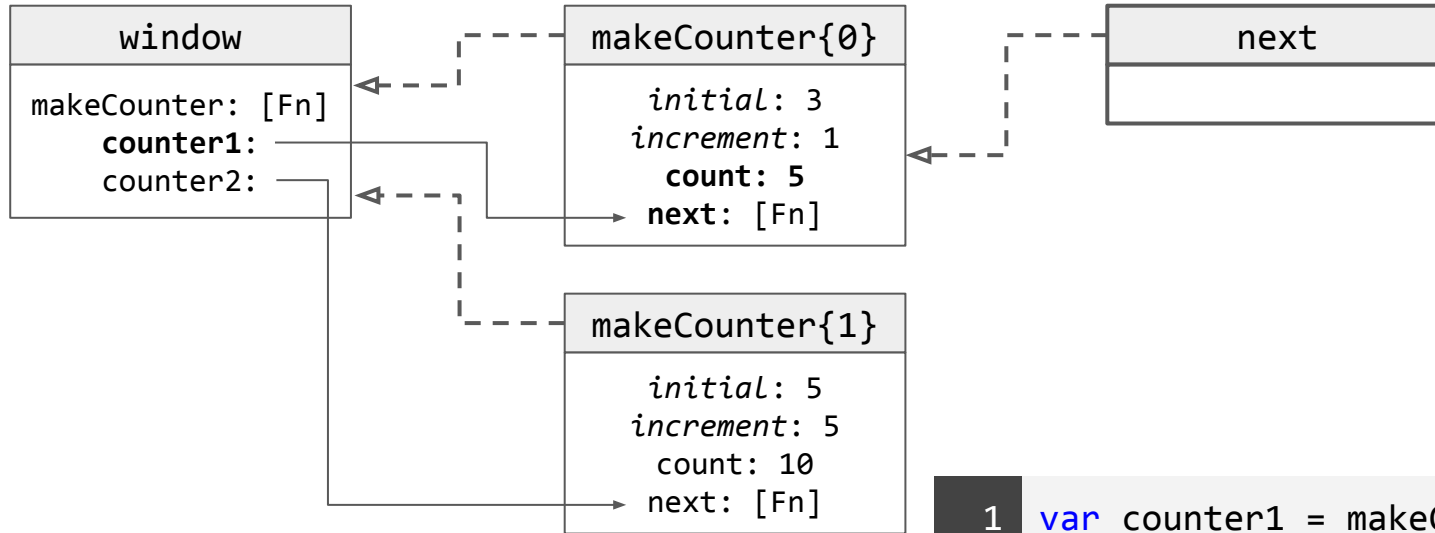


Legend

- ▶ Direct Reference
- - - ▶ Access to Scope

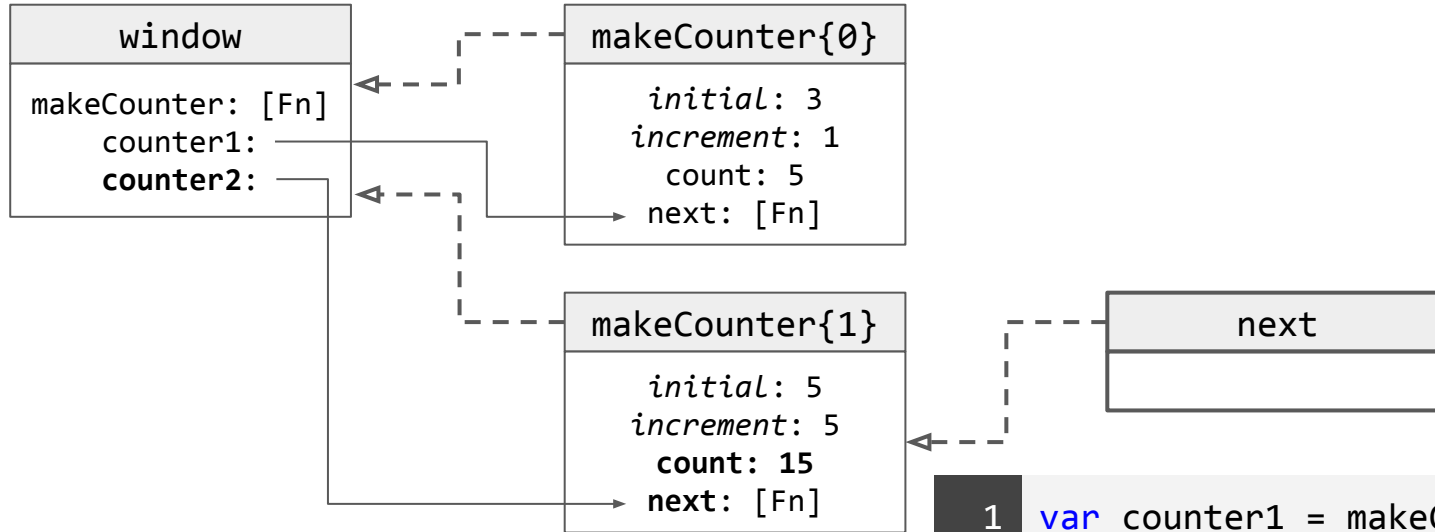
```
1 var counter1 = makeCounter(3, 1);
2 var counter2 = makeCounter(5, 5);
3 console.log(counter1());
4 console.log(counter2());
5
6
```

# Closure Function



```
1 var counter1 = makeCounter(3, 1);
2 var counter2 = makeCounter(5, 5);
3 console.log(counter1());
4 console.log(counter2());
5 console.log(counter1());
6
```

# Closure Function



```
1 var counter1 = makeCounter(3, 1);
2 var counter2 = makeCounter(5, 5);
3 console.log(counter1());
4 console.log(counter2());
5 console.log(counter1());
6 console.log(counter2());
```

# Closure Function



```
1 function makeCounter (initial, increment){
2     var count = initial;
3     return function next(){
4         count += increment;
5         return count;
6     }
7 };
8 var counter1 = makeCounter(3, 1);
9 var counter2 = makeCounter(5, 5);
10 console.log(counter1());           // prints: 4
11 console.log(counter2());           // prints: 10
12 console.log(counter1());           // prints: 5
13 console.log(counter2());           // prints: 15
```

**TRY IT!**



## Closure Function: Class Activity



[lectures/lecture-4/activity3.js](https://github.com/lectures/lecture-4/activity3.js)



- Implement the `makeAccount` function below
  - It should keep a variable `balance`, initially assigned the value of `initial`
  - It should return an object with 3 properties: `deposit`, `withdraw`, `getBalance`
    - `deposit`: a function that accepts an argument `amount` and adds it to `balance`
    - `withdraw`: a function that accepts an argument `amount` and subtracts it from `balance`
    - `getBalance`: a function that returns the `balance`

```
1 function makeAccount (initial){ /* your implementation */ };
2
3 var alice = makeAccount(300);
4 alice.deposit(100);
5 alice.withdraw(50);
6 console.log(alice.getBalance());           // prints: 350
```

# Closure Function

```
1 function makeCounters (n){
2   var counts = [];
3   var counters = [];
4   for (var i = 0; i < n; i++){
5     counts[i] = 0;
6     counters[i] = function next(){
7       counts[i] ++;
8       return counts[i];
9     };
10  }
11  return counters;
12 };
13
14 var cs = makeCounters(10);
15 console.log( cs[0]() );      // prints?
16 console.log( cs[4]() );      // prints?
```



# Closure Function

```
1 function makeCounters (n){
2   var counts = [];
3   var counters = [];
4   for (var i = 0; i < n; i++){
5     counts[i] = 0;
6     counters[i] = function next(){
7       counts[i] ++;
8       return counts[i];
9     };
10  }
11  return counters;
12 };
13
14 var cs = makeCounters(10);
15 console.log( cs[0]() );      // prints: NaN
16 console.log( cs[4]() );      // prints: NaN
```



# Closure Function

```
1 function makeCounters (n){
2   var counts = [];
3   var counters = [];
4   for (var i = 0; i < n; i++){
5     counts[i] = 0;
6     counters[i] = (function (j){
7       return function next(){
8         counts[j] ++;
9         return counts[j];
10      };
11    })(i);
12  }
13  return counters;
14 };
15
16 var cs = makeCounters(10);
```



# Closure Function

1. Function
2. Callback Function
3. Closure Function

