Javascript III: Advanced Functions

Pau Fernández Rafa Genés Jose L. Muñoz

Universitat Politècnica de Catalunya (UPC)

Table of Contents

this

`this` in the Global context

Outside functions, `this` refers to the global object:

```
// In the browser
console.log(this === window);

a = 3;

console.log(window.a);
console.log(this.a);
```

`this` inside a function

Inside a function, `this` depends on how the function was called:

```
function f() { this.a = 3; }
```

No left object

```
f(); // this === window
```

Left object

```
let obj = { a: 0, f };
obj.f(); // this === obj
```

A function is a ”method”, it always has a `this` object...

this in strict mode

In strict mode, **this** is **undefined** when calling with no left object:

```
"use strict";

function f() {
  console.log(this);
}

f(); // --> undefined
```

Methods

The following two ways of defining methods are equivalent:

```
let obj = {
  name: 'Tim',
  sayHi: function () {
    console.log("Hi, I'm " + this.name);
  }
};
```

```
let obj = {
  name: 'Tim',
  sayHi() {
    console.log("Hi, I'm " + this.name);
  }
}
```

Both store a *function object* in the `sayHi` field.

Unbinding

We can extract a method from an object and keep a reference to it.

But the association with the object is lost.

```
let user = {
  name: "Tania",
  sayHi() {
    console.log("Hi, I'm " + this.name + "!")
  }
};

user.sayHi();  // --> Hi, I'm Tania!

let sayHi = user.sayHi;
sayHi();       // --> Hi, I'm undefined!
```

This is normal: there is no "left-object"!

Binding

We can produce a ”forced binding” to associate a function with an object:

```
let obj = {
  name: "Rose",
  sayHi() {
    console.log("Hi there, I'm " + this.name);
  }
}

let boundSayHi = obj.sayHi.bind(obj);
boundSayHi(); // --> Hi there, I'm Rose
```

`bind` returns a new function with a *permanent* binding of `this`.

Event handlers

```
let clickCounter = {
  numberOfClicks: 0,
  onClick() {
    this.numberOfClicks++;
  }
}

let elem = document.querySelector('.clickable');
elem.addEventListener(
  'click',
  clickCounter.onClick.bind(clickCounter)
);
```

Arrow functions don't have **this**

Arrow functions don't have a **this** variable.

But they take it from the lexical context.

```
// We put a field in the global object to see it later
this.yoohoo = true;

const showMe = () => {
  // 'this' is the global one, taken from the lexical scope
  console.log(this);
};

showMe();
```

this in event handlers

In event handlers, **this** is bound to the object that produced the event.

```
const button = document.querySelector('button');
button.addEventListener('click', function (e) {
  // -> 'this' is the button element!
  console.log(this);
  this.innerText = 'You did click!';
});
```

This behavior is consistent when using `addEventListener`, but not assigning to `.onclick`

This behavior is lost with arrow functions!

Table of Contents

Closures

Scope

A scope is the environment contained in a pair of braces ({}), in which you can declare new variables.

Arrow functions also define new scopes (even if they don't have braces).

A given piece of code can access all scopes that surround it. This is determined *statically*. *It would be very difficult to reason about programs otherwise.*

```
{
  /* 1 */
  let a = 1, b = 2, c;
  const f = (x) => {
    /* 2 */
    return () => /* 3 */ (x ? a : b);
  }
}
```

Typically, outer scopes *live longer* than inner scopes.

Outer Variables

A function has full access to outer variables:

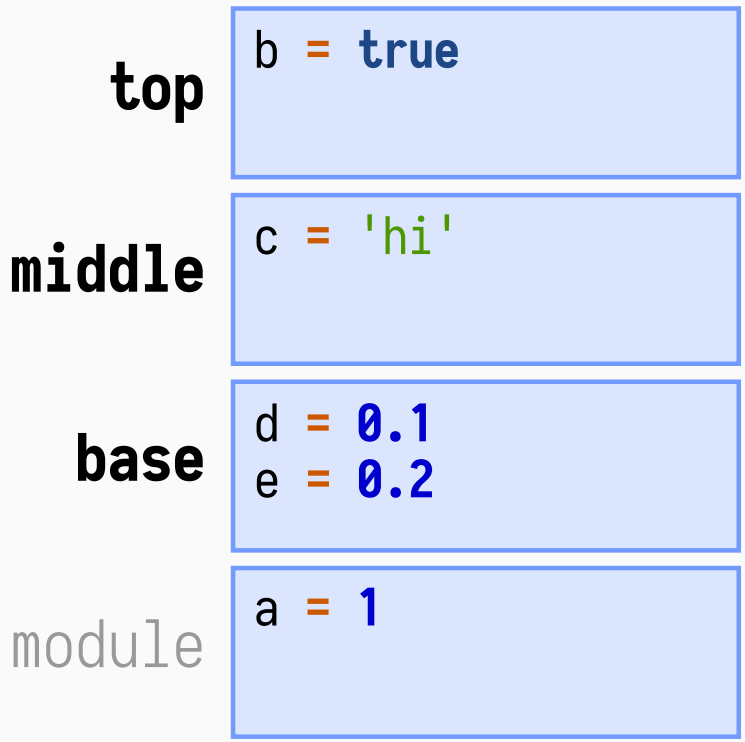
```
let messageCount = 0;

function showMessage(msg) {
  messageCount++;
  console.log(msg);
}

showMessage('meaow');
console.log(messageCount);
```

The Stack

```
let a = 1;
function top() {
  let b = true;
  console.log(a);
}
function middle(x) {
  let c = 'hi';
  top();
}
function base() {
  let d = 0.1, e = 0.2;
  middle(d);
}
base();
```



Stack vs Scope

```
let x = 10;

function f() {
  console.log(x);
}

function g() {
  let x = 15;
  f();
}

g(); // 10? 15?
```

What `x` will `f` refer to?

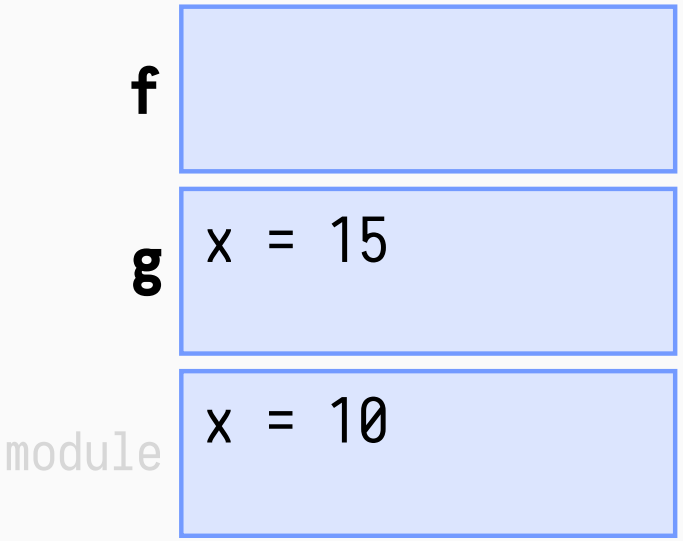
Stack vs Scope

```
let x = 10;

function f() {
  console.log(x);
}

function g() {
  let x = 15;
  f();
}

g(); // 10? 15?
```



Nested Functions

Functions can be defined inside other functions

```
function outer() {
  let a = 3, b = true;

  function inner() {
    let x = a + 4;
    let y = (b ? 'hi' : 'ho');
    return `${x}${y}`;
  }

  let result = inner();
  return result;
}
```

Inner functions **can reference variables outside their scope**.

Closures

A function can survive the scope in which it was created.

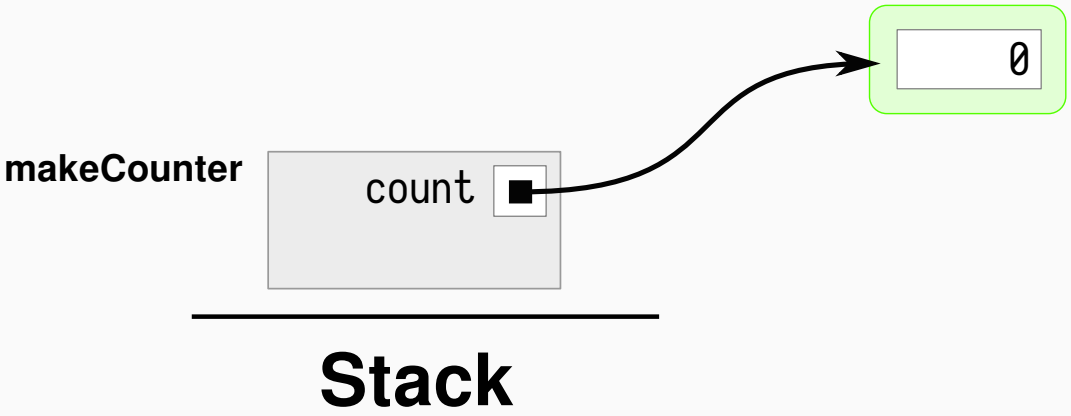
If it references variables in it, a closure has to be created. *(The environment of the outer function is put outside the stack so that it can last longer.)*

```
function makeCounter() {
  let count = 0;
  return function() {
    count++;
    return count;
  }
}

let c1 = makeCounter(), c2 = makeCounter();
console.log(c1(), c2(), c1());
```

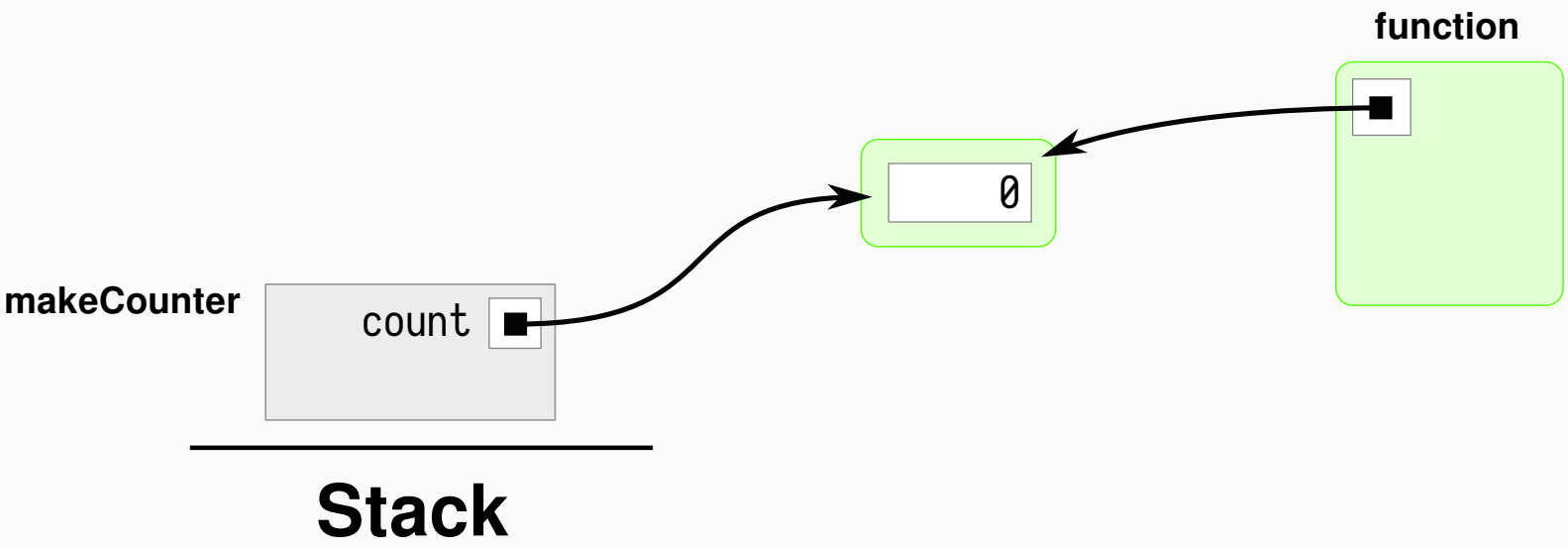
Closures Visualization (1)

The stack grows with `makeCounter` and it references `counter`.



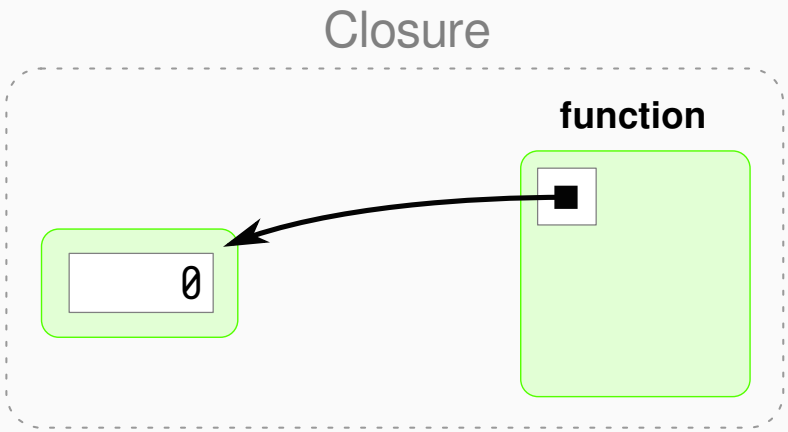
Closures Visualization (2)

A new function is created which references `counter`



Closures Visualization (3)

The stack shrinks but the function still references `counter`, a *closure* is created.



Stack

Table of Contents

Higher-order Functions

Higher-order functions

Higher-Order Functions (HOFs) are functions that either receive functions as parameters, or return other functions.

(`map`, `filter`, and `reduce` are higher-order functions.)

Wrapping

HOFs also can alter parameters or results:

```
function logger(func) {
  return (...args) => {
    console.log("Calling with", args);
    let result = func(...args);
    console.log("=>", result);
    return result;
  }
}
```

With `logger` we can now convert any function and observe what parameters it receives and what result it returns:

```
const inc = logger(x => x + 1);
inc(4);
// Calling with [ 4 ]
// => 5
```

Memoization

Memoization is the caching of already computed values for a pure function.
We can implement memoization using a closure:

```
function memoize(f) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {     // if the result is in the map
      return cache.get(x);  // return it
    }
    let result = f(x);      // otherwise call f
    cache.set(x, result);   // and cache the result
    return result;
  };
}
```

```
function isPrimeSlow(n) { ... } // compute if a number is prime
let isPrimeFast = memoize(isPrimeSlow);
```

Currying

A function `f(a, b)` can be *curried* into `g(a)(b)`, that does the same.

```
const add = (a, b) => a + b;
const addC = a => b => a + b;

console.log(add(5, 6));
console.log(addC(5)(6));

const add10 = addC(10);
console.log(add10(5));
```

This lets us ”delay” the computation, and keep intermediate parameters.

Currying as ”function configuration” (1)

Using currying, we can ”configure” returned functions:

```
const classify = (thres1, thres2) => a => {
  if (a >= thres1) {
    return 'high';
  } else if (a >= thres2) {
    return 'middle';
  } else {
    return 'low';
  }
}

let array = [5, -1, 3, 20, -7];
array.map(classify(7, 4));
```

Currying as ”function configuration” (2)

```
const greaterThan = n => (x => x > n);
const lengthIs = n => (x => x.length === n);
```

Now we have a two functions that produce function comparators with a fixed lower bound or length.

```
[10, 11, 9, 12, 15, 8, 7].every(greaterThan(10)); // -> false
["a", "good", "place"].filter(lengthIs(1));       // -> ["a"]
```

Currying in event handlers

```
const div = document.querySelector('div');

const toggleClass = _class =>
  function(event) {
    this.classList.toggle(_class);
  };

div.addEventListener("click", toggleClass("selected"));
```

Partial Application

The `bind` method not only can associate the `this` object, but also partially fill in some parameters. This is called *partial application*.

```
function exp(base, exponent) {
  let result = 1;
  for (let i = 0; i < exponent; i++) {
    result *= base;
  }
  return result;
}

let exp10 = exp.bind(null, 10); // base = 10
let exp2  = exp.bind(null, 2);  // base = 2

console.log(exp10(4)); // -> 10000
console.log(exp2(5));  // -> 32
```

Composition

We can even write a HOF that will return a function which is the functional composition of a sequence of functions:

```
const compose = (...functions) =>
    args => functions.reduceRight((arg, fn) => fn(arg), args);

const plus1 = x => x+1;
const mul2  = y => y*2;

const A = [1, 2, 3, 4, 5];
A.map(plus1).map(mul2) // -> [4, 6, 8, 10, 12]
A.map(compose(mul2, plus1)) // -> [4, 6, 8, 10, 12]
```