KYLE SIMPSON GETIFY@GMAIL.COM

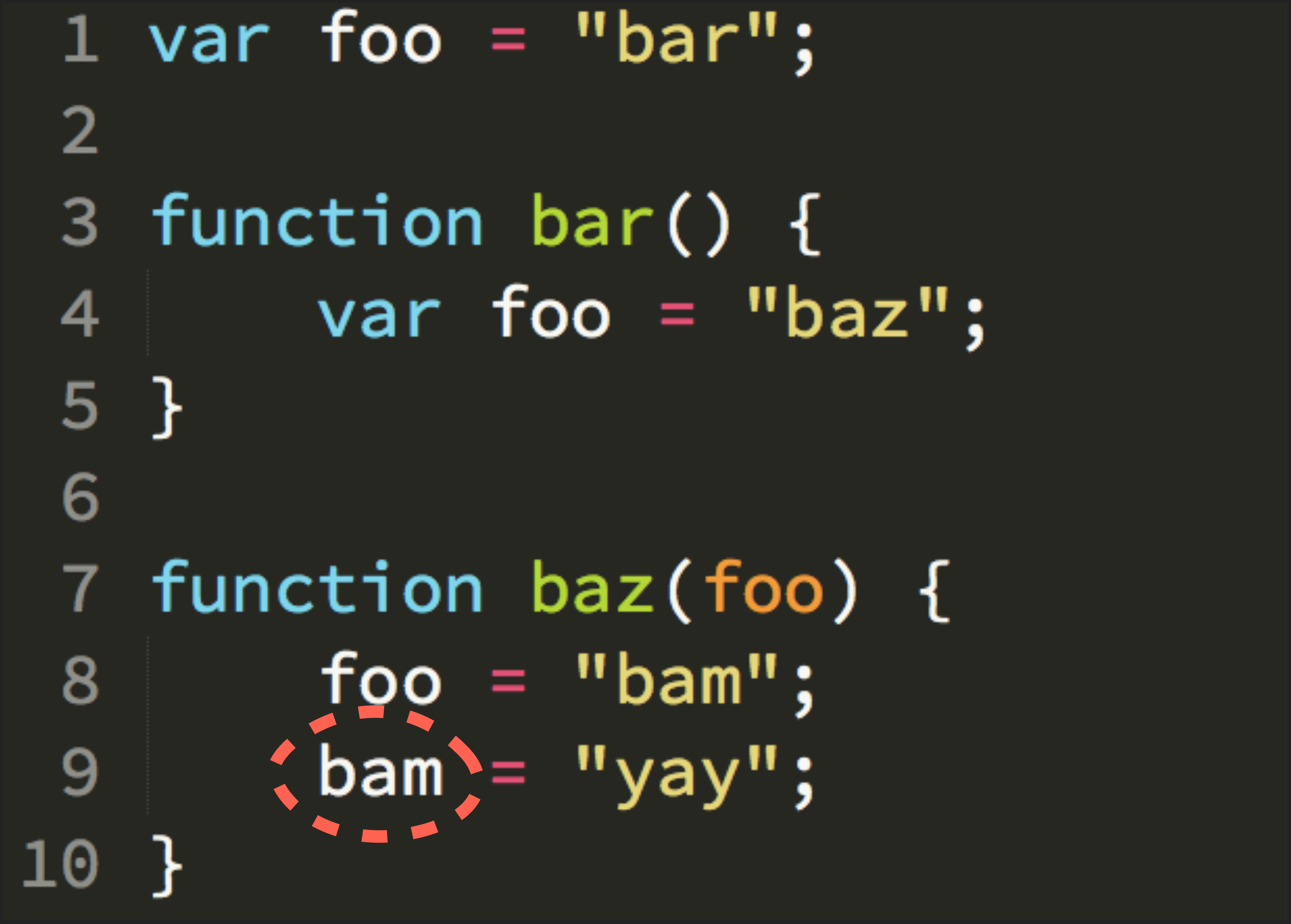
DEEP JS FOUNDATIONS

Scope, Closures

- Nested Scope
- Hoisting
- Closure
- Modules

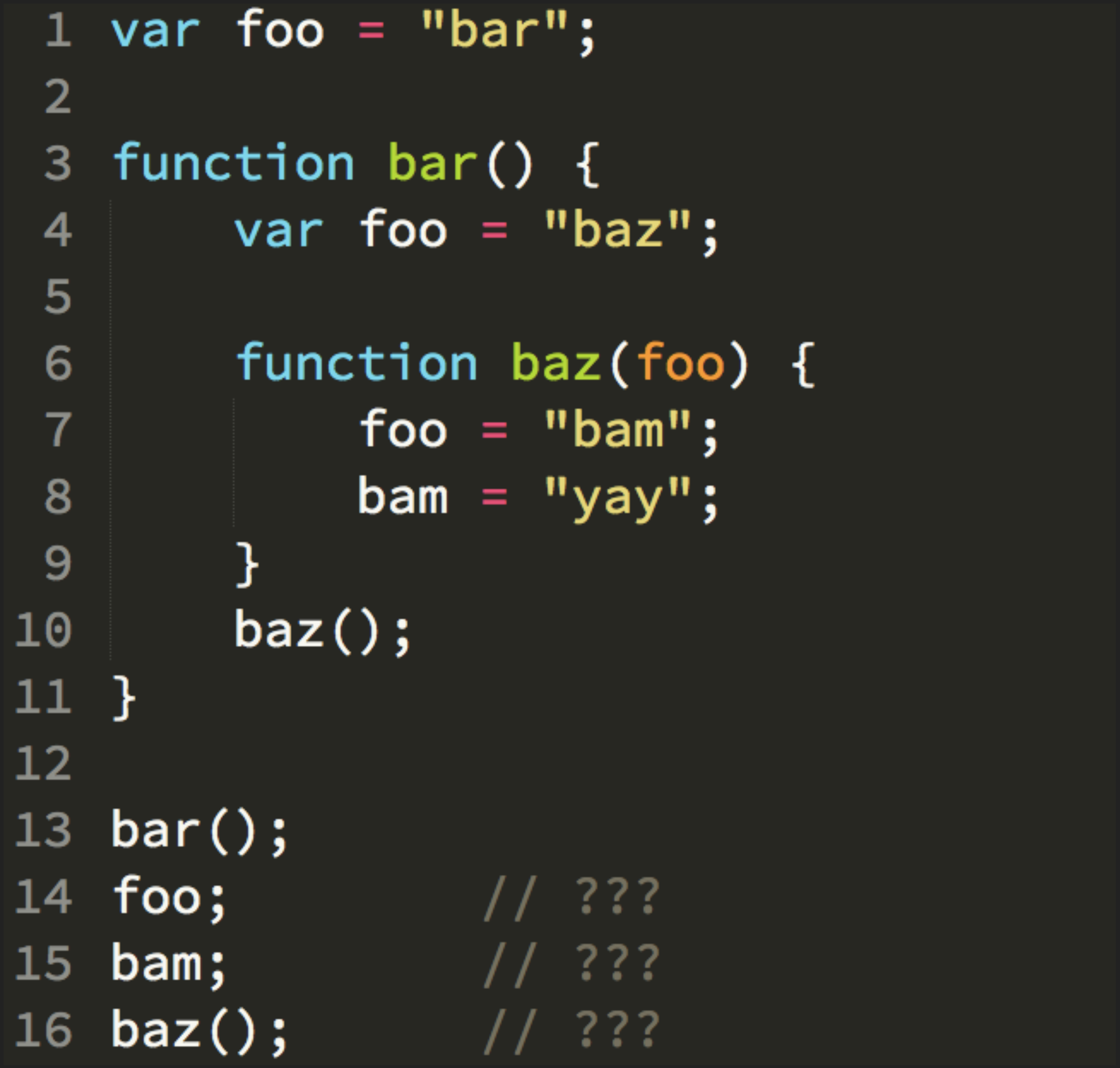
Scope: where to look for things

JavaScript has function scope only*



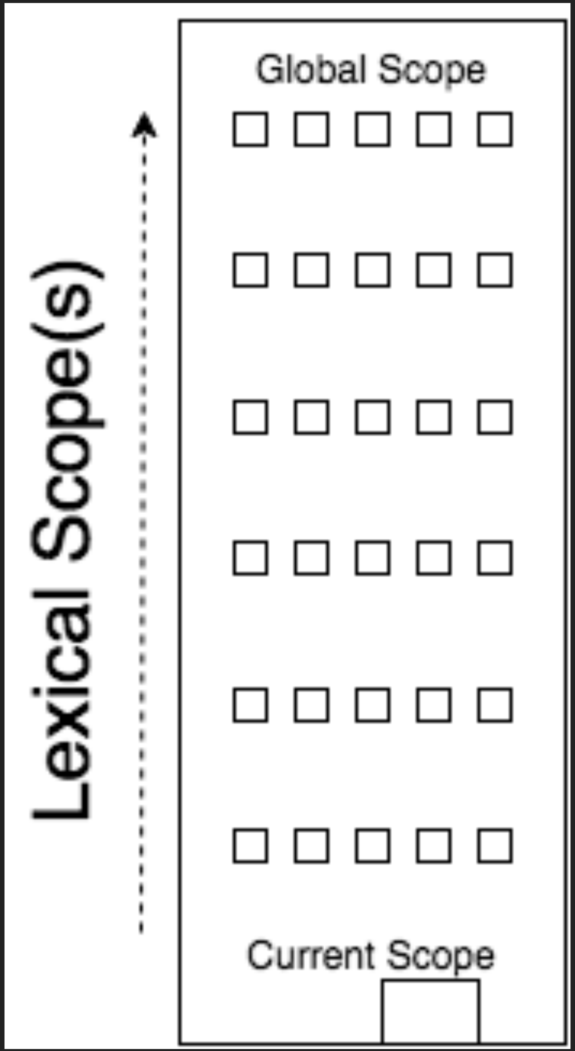
```javascript
var foo = "bar";

function bar() {
    var foo = "baz";
}

function baz(foo) {
    foo = "bam";
    bam = "yay";
}
```

Scope

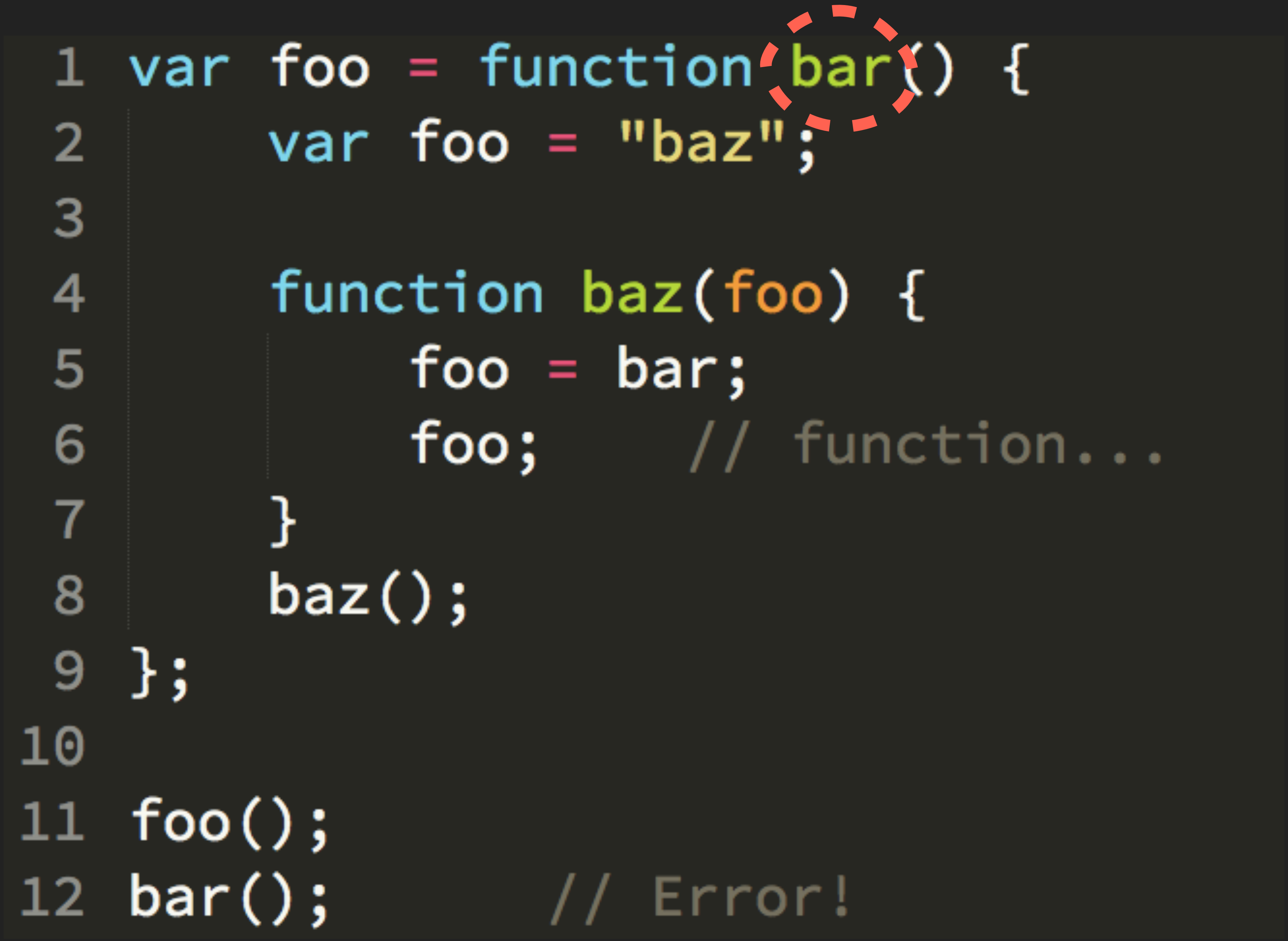


```
var foo = "bar";

function bar() {
    var foo = "baz";

    function baz(foo) {
        foo = "bam";
        bam = "yay";
    }
    baz();
}

bar();
foo;            // ???
bam;            // ???
baz();          // ???
```

Scope



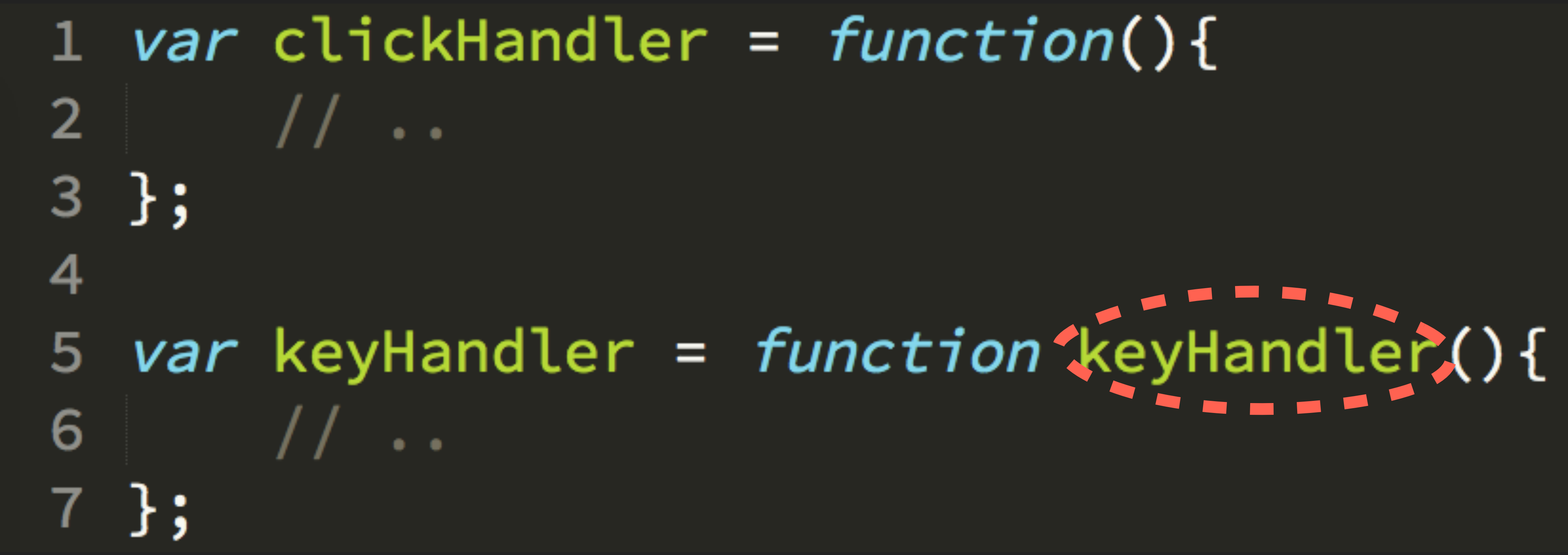
Scope



```
var foo = function bar() {
    var foo = "baz";

    function baz(foo) {
        foo = bar;
        foo;     // function...
    }
    baz();
};

foo();
bar();        // Error!
```

Scope: which scope?

Named Function Expressions



```javascript
var clickHandler = function(){
	// ..
};

var keyHandler = function keyHandler(){
	// ..
};
```

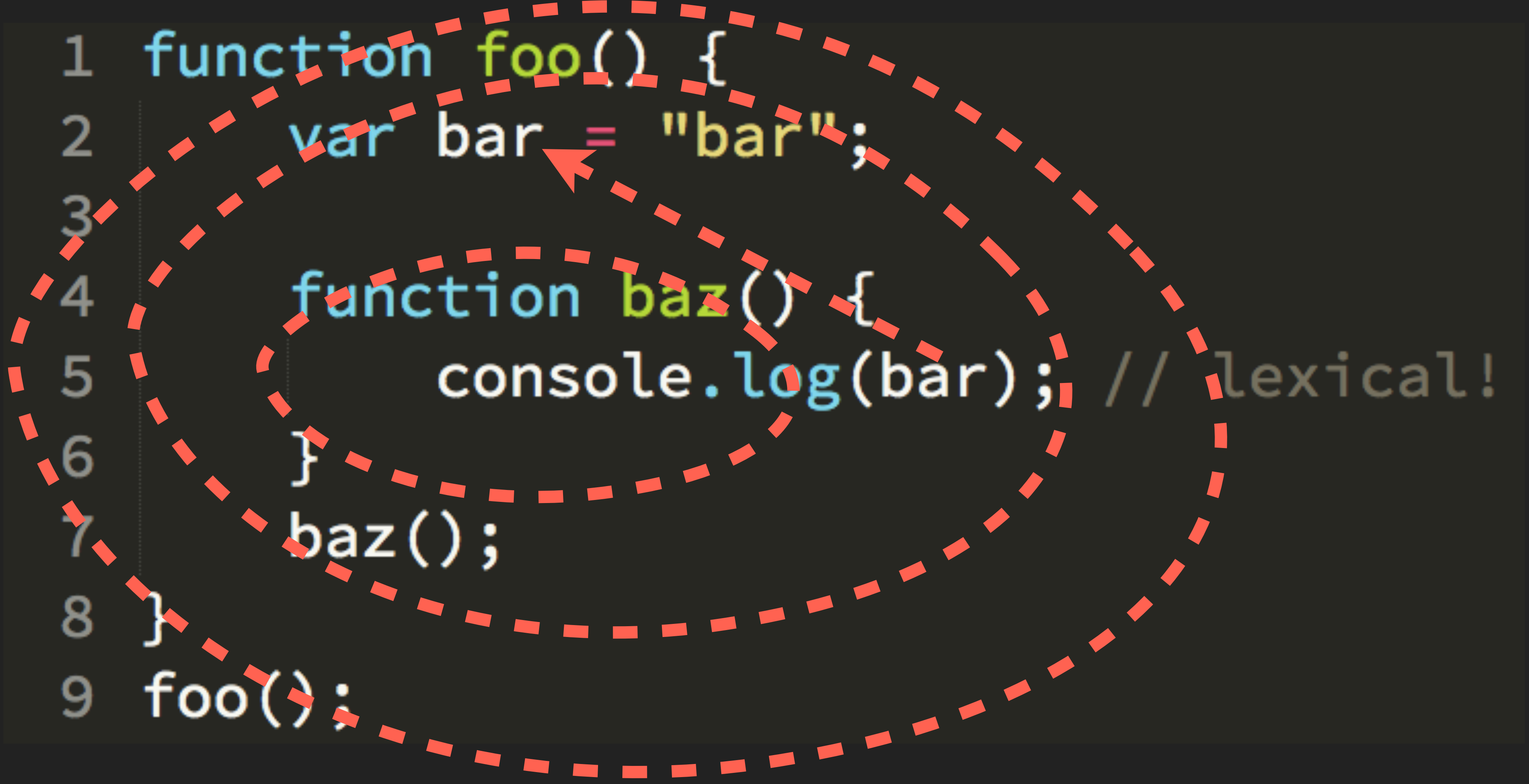
Named Function Expressions

1. Handy function self-reference
2. More debuggable stack traces
3. More self-documenting code

Named Function Expressions: Benefits

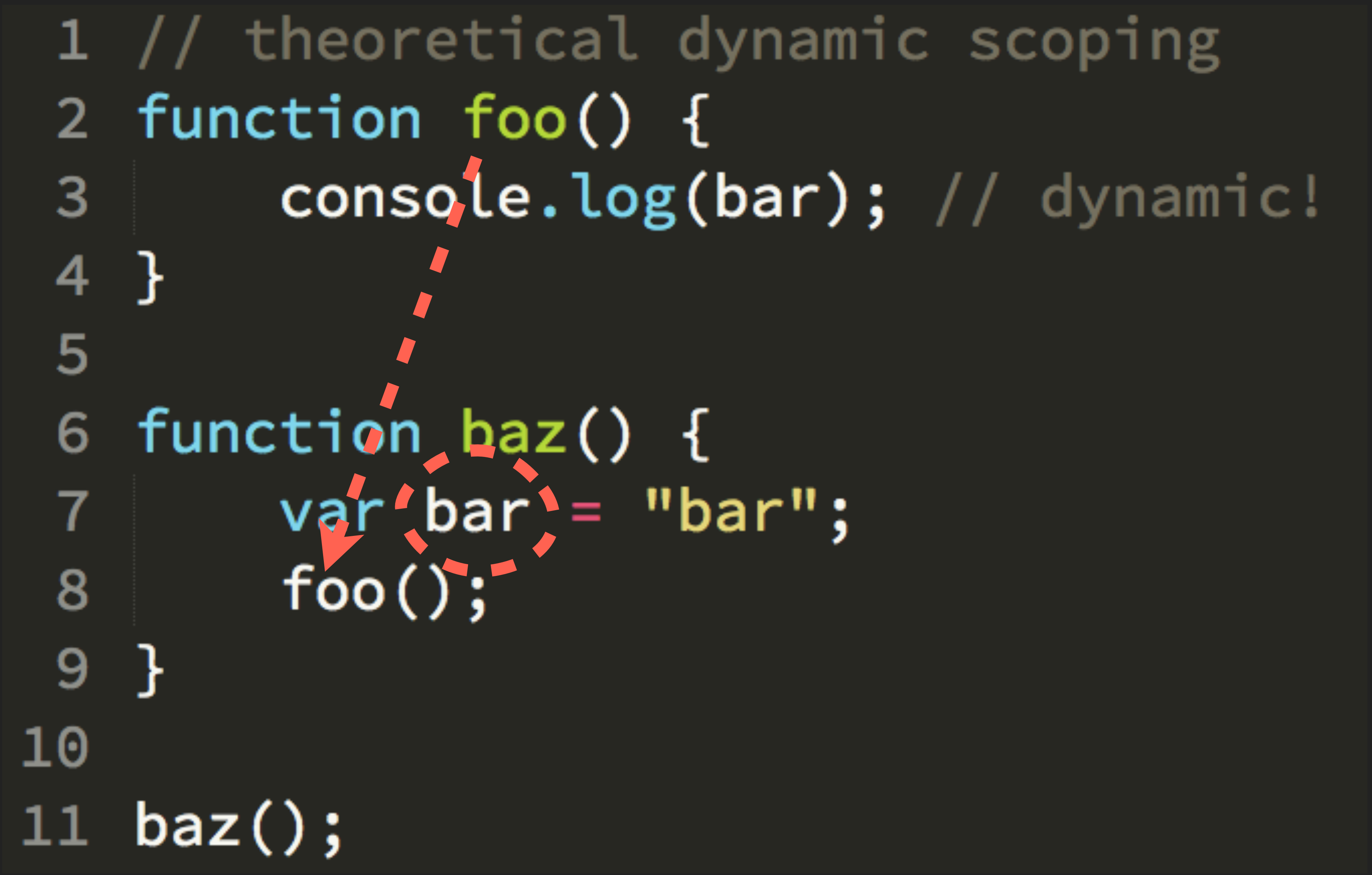
lexical scope

dynamic scope



```
function foo() {
    var bar = "bar";

    function baz() {
        console.log(bar); // lexical!
    }
    baz();
}
foo();
```

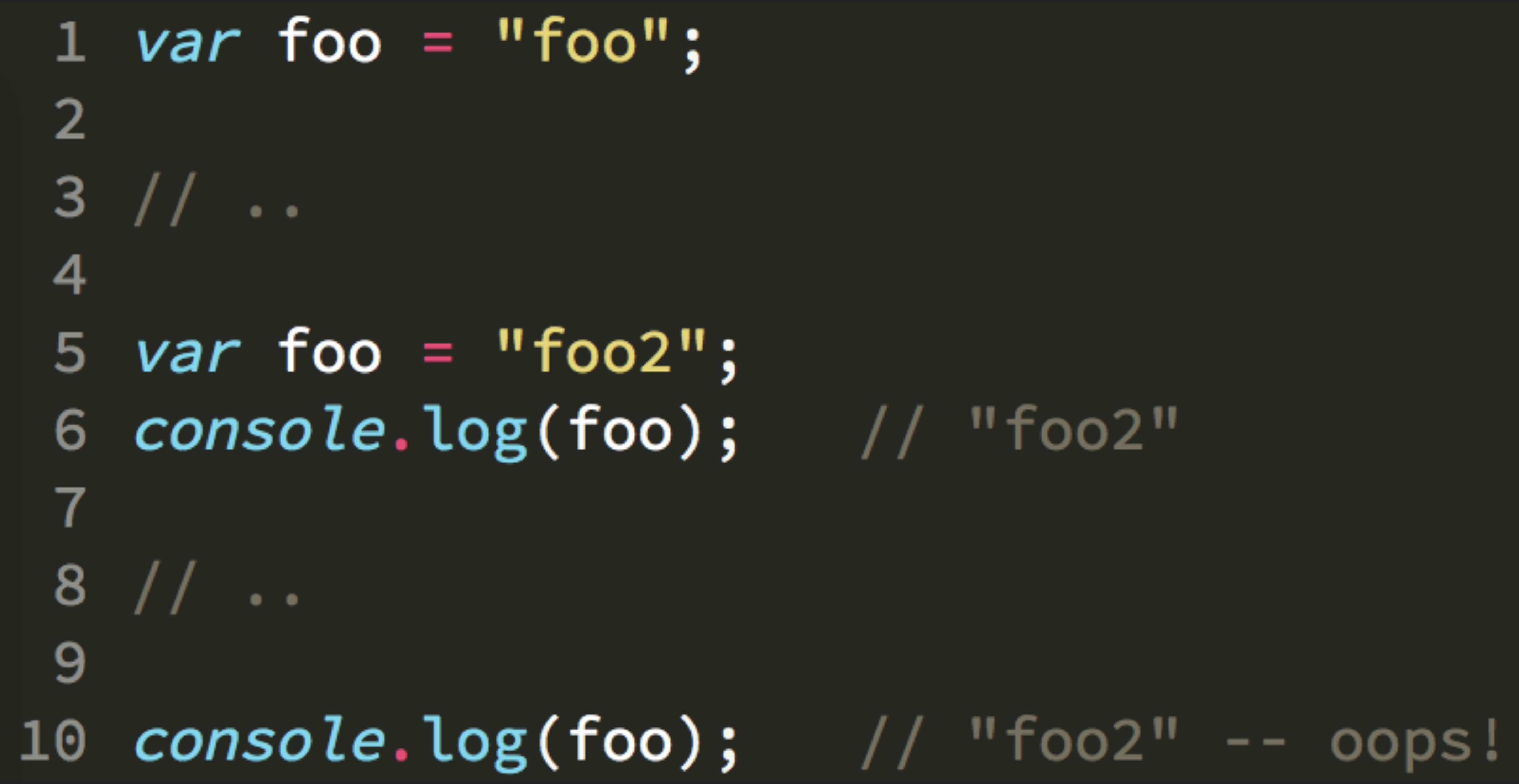
Scope: lexical



```
// theoretical dynamic scoping
function foo() {
    console.log(bar); // dynamic!
}

function baz() {
    var bar = "bar";
    foo();
}

baz();
```

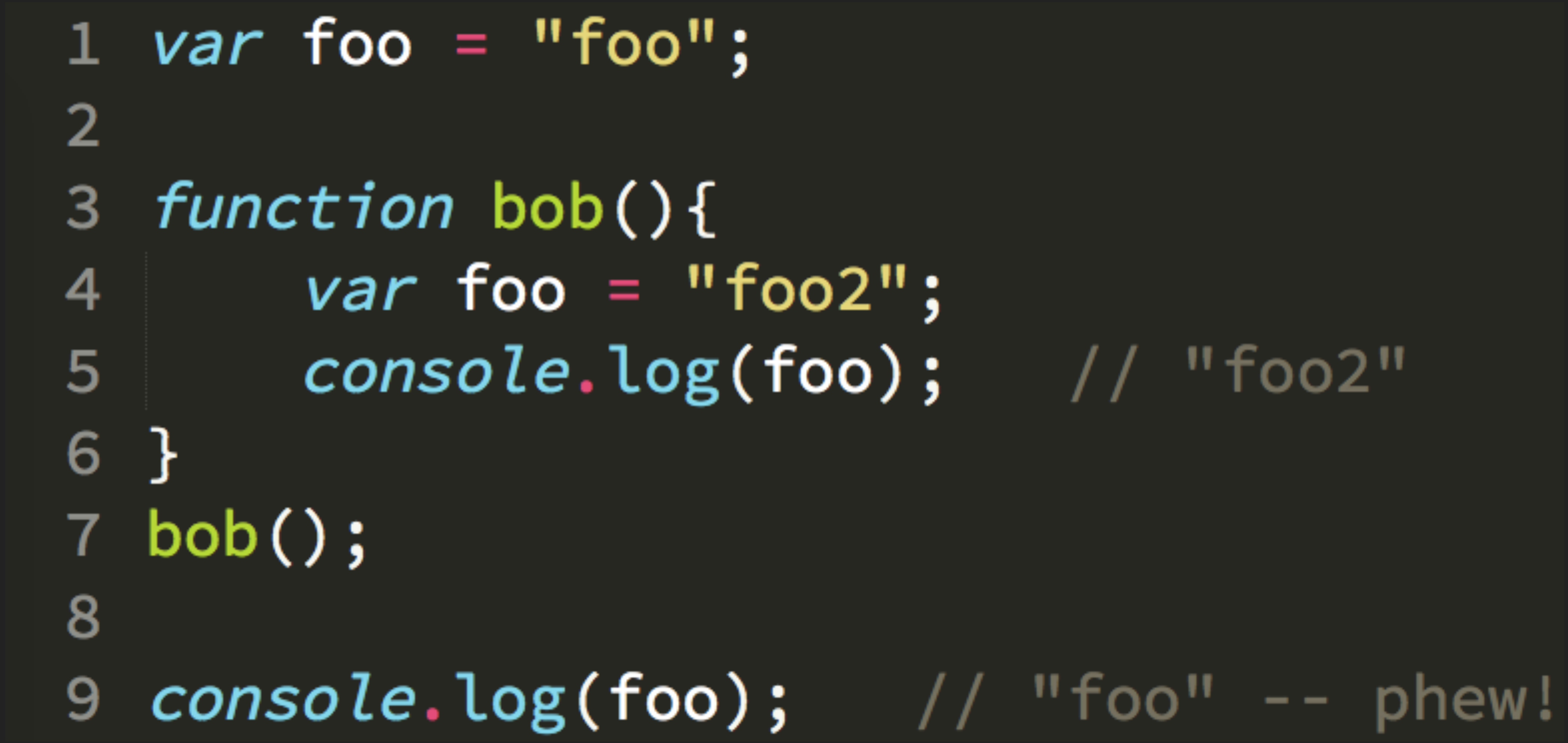
Scope: dynamic

Function Scoping



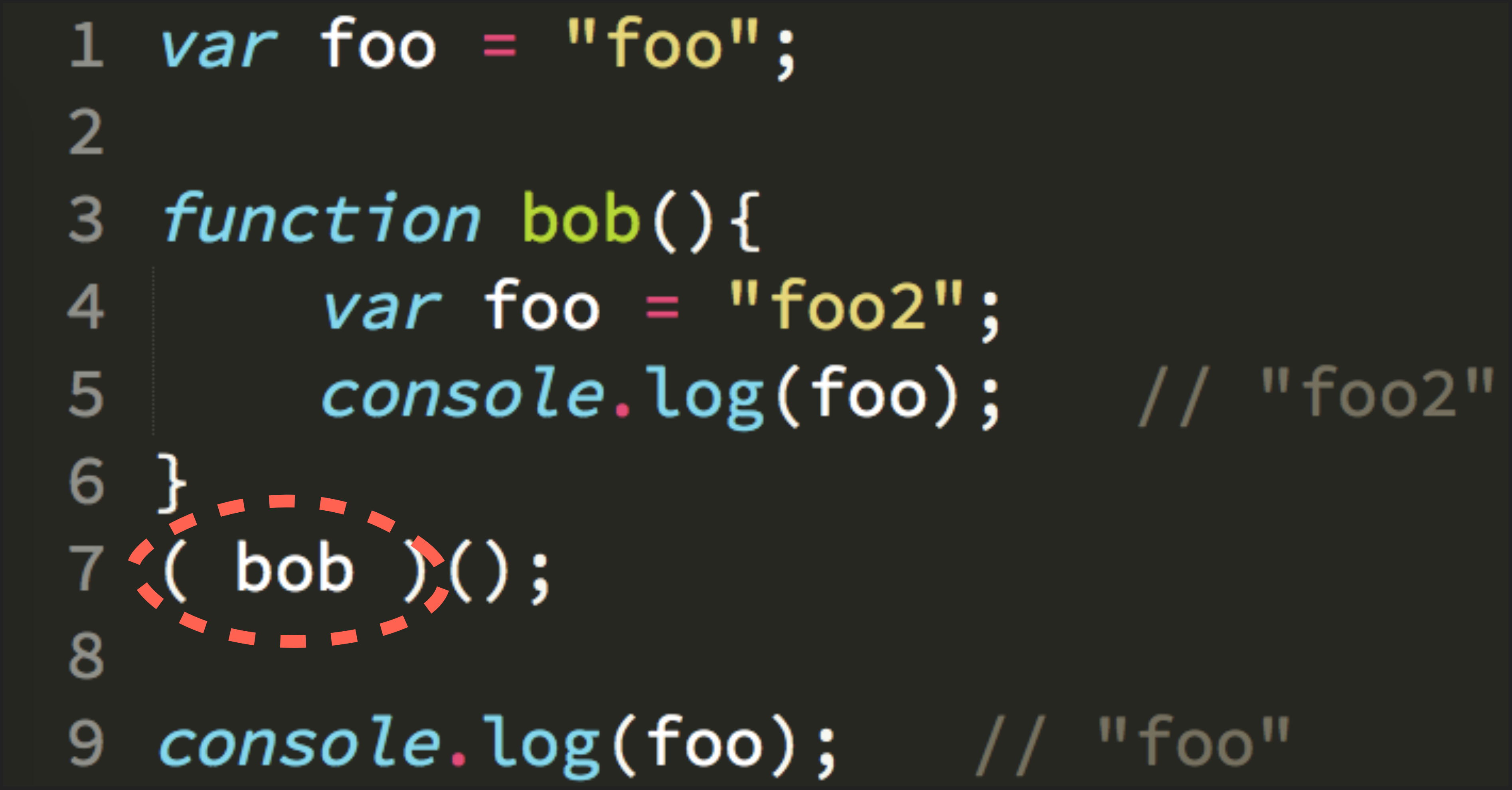
```javascript
var foo = "foo";

// ..

var foo = "foo2";
console.log(foo);      // "foo2"

// ..

console.log(foo);      // "foo2" -- oops!
```

Function Scoping



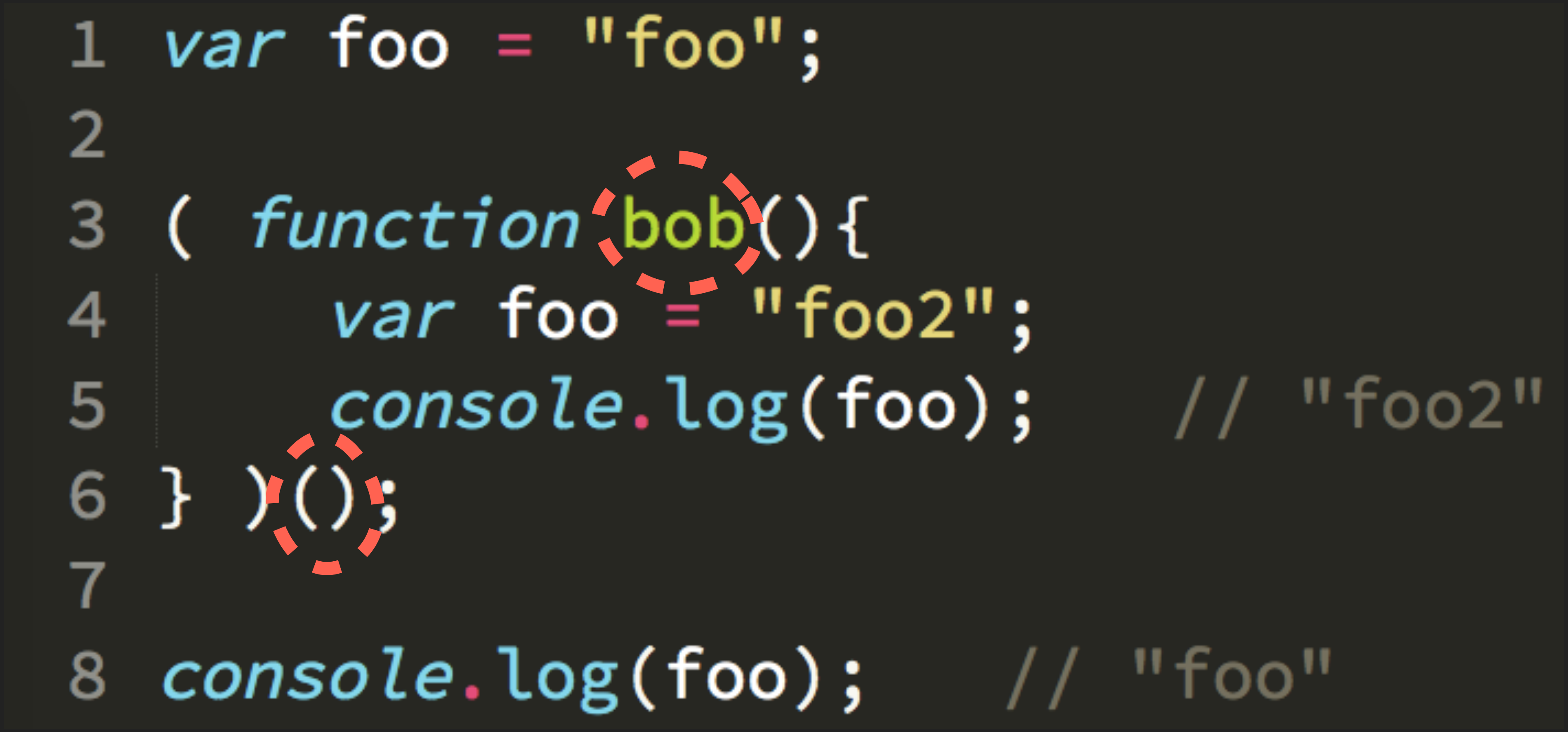
```
var foo = "foo";

function bob(){
    var foo = "foo2";
    console.log(foo);    // "foo2"
}
bob();

console.log(foo);    // "foo" -- phew!
```

Function Scoping



```
var foo = "foo";

function bob(){
      var foo = "foo2";
      console.log(foo);     // "foo2"
}
( bob )();

console.log(foo);     // "foo"
```

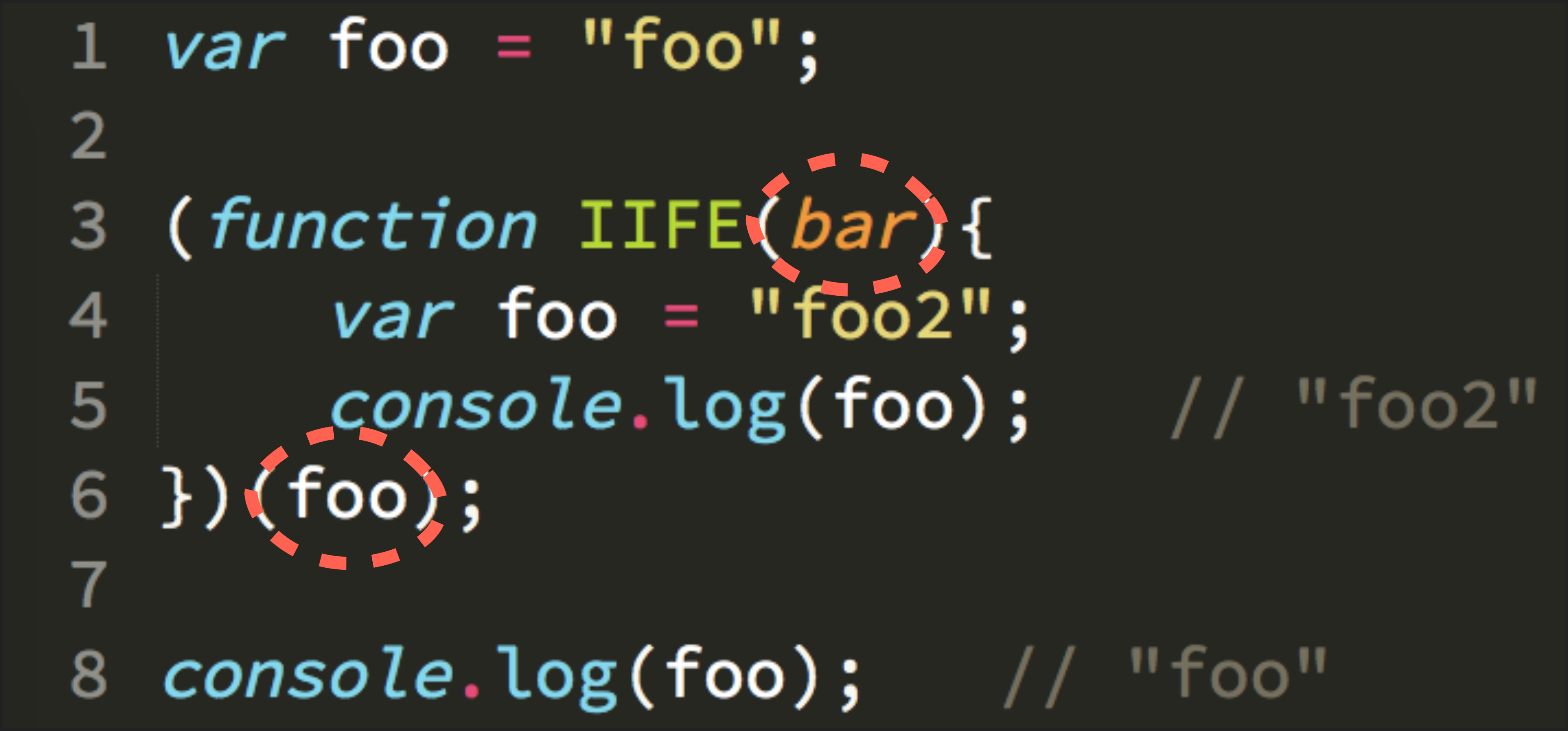
Function Scoping



```
var foo = "foo";

( function bob(){
      var foo = "foo2";
      console.log(foo);    // "foo2"
} )();

console.log(foo);    // "foo"
```

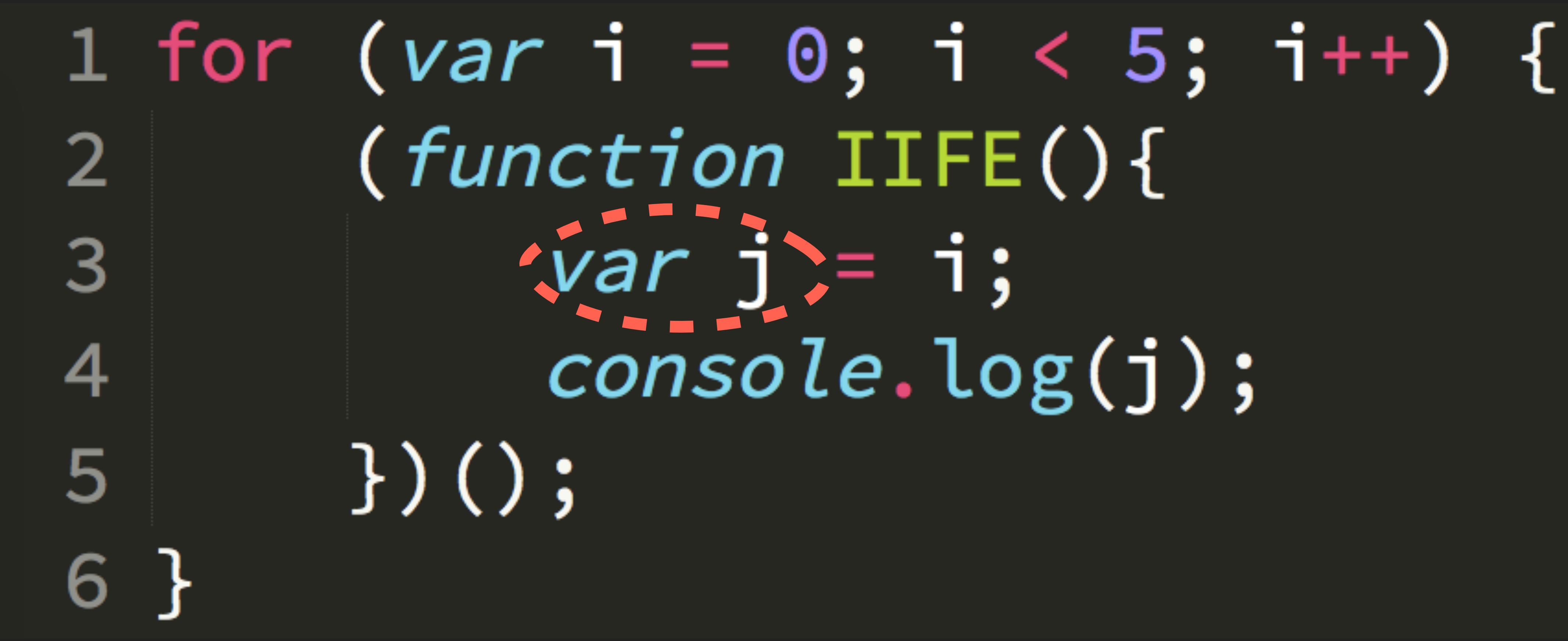
http://benalman.com/news/2010/11/immediately-invoked-function-expression/

Function Scoping: IIFE



```
var foo = "foo";

(function IIFE(bar){
    var foo = "foo2";
    console.log(foo);     // "foo2"
})(foo);

console.log(foo);     // "foo"
```

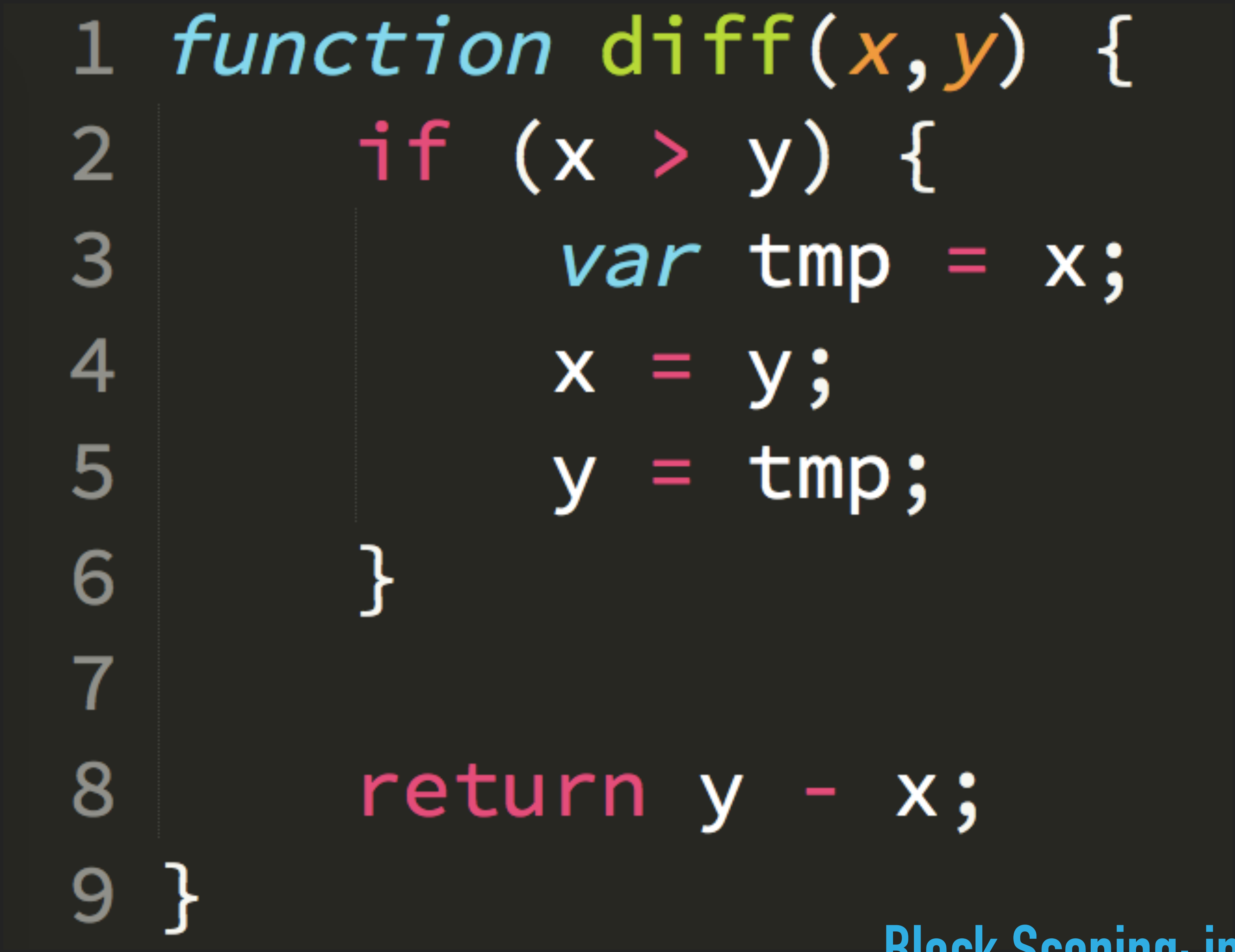
Function Scoping: IIFE



```
for (var i = 0; i < 5; i++) {
        (function IIFE(){
                var j = i;
                console.log(j);
        })();
}
```

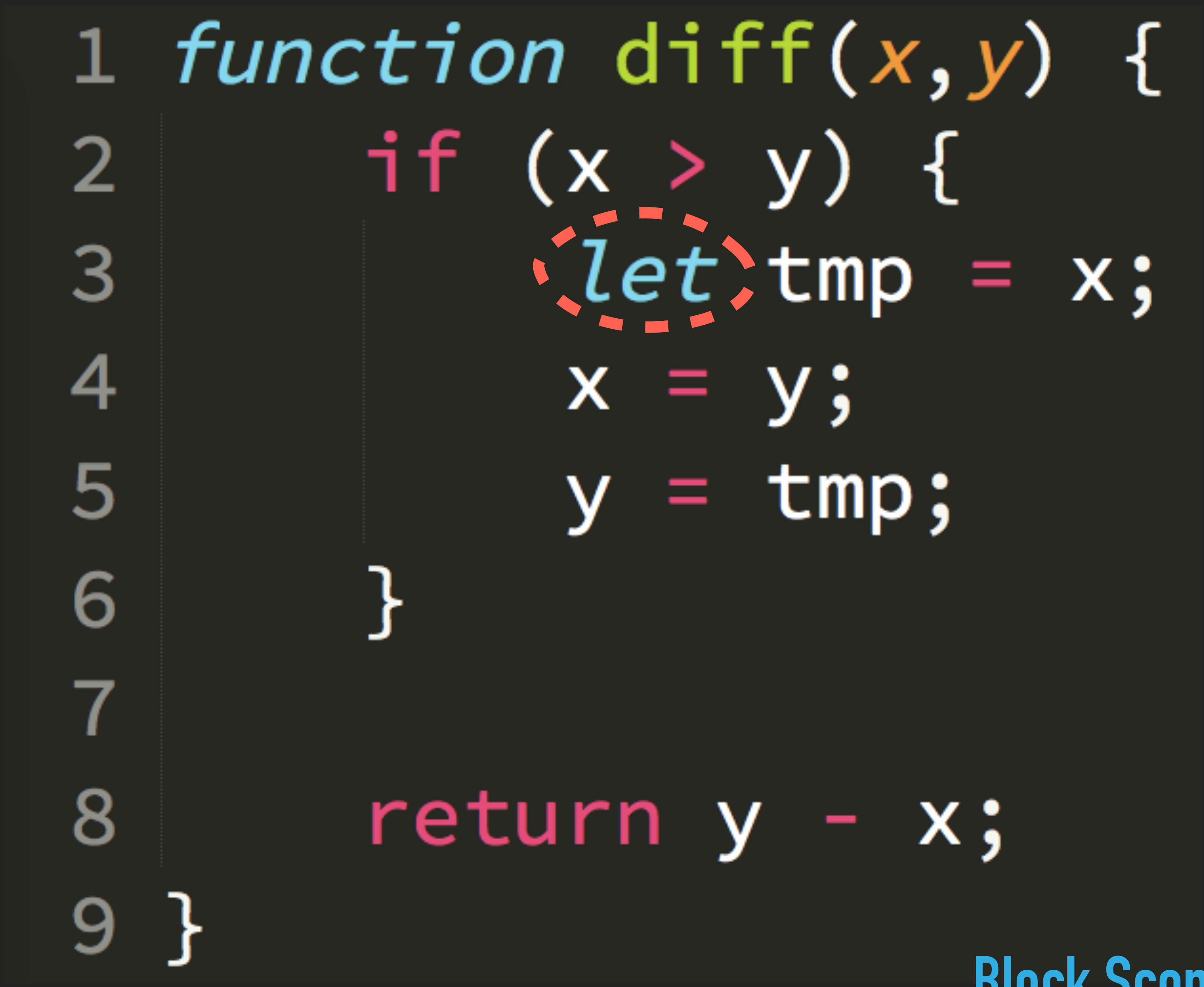
Function Scoping: IIFE

Block Scoping



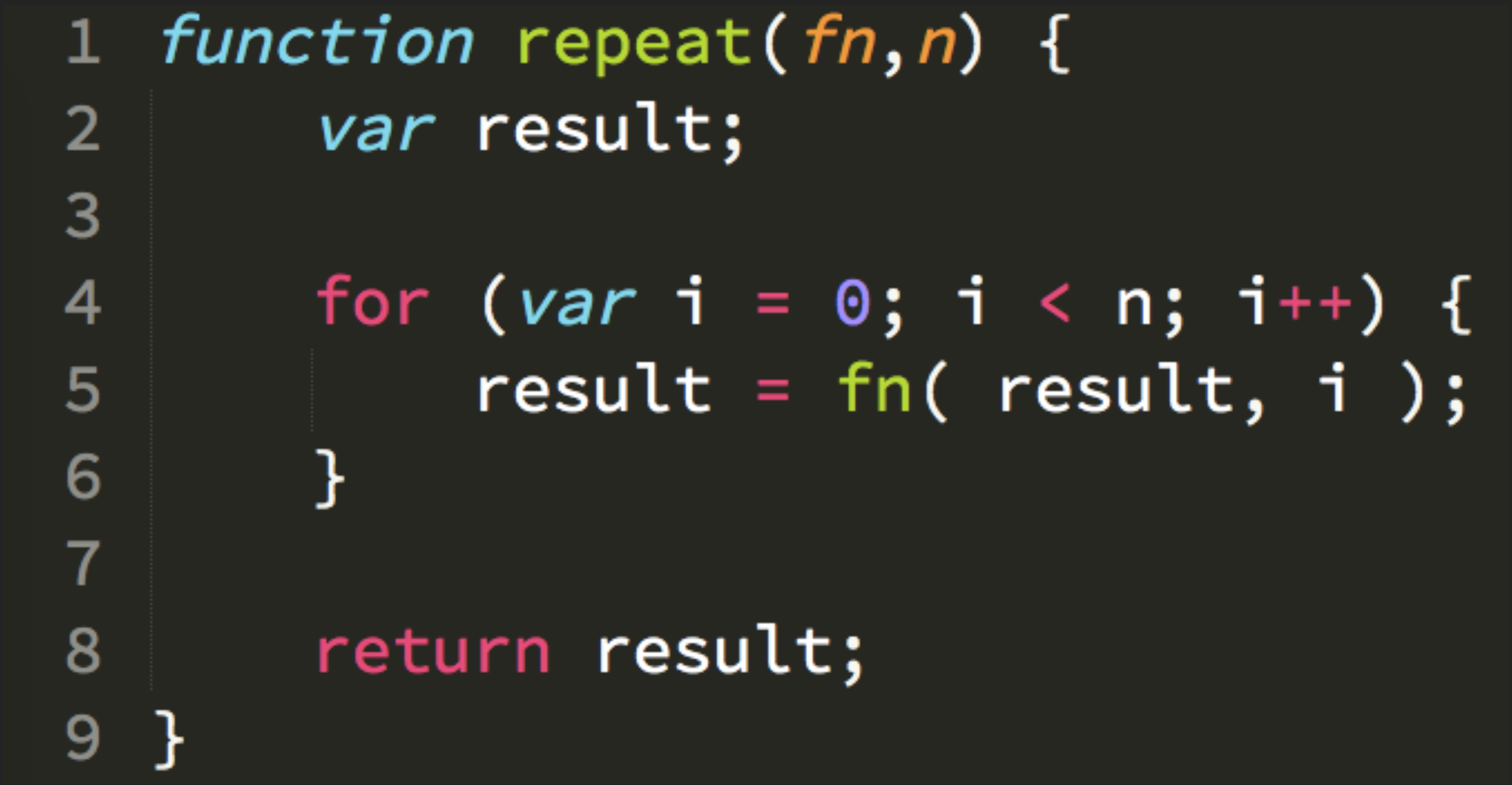
```
function diff(x,y) {
    if (x > y) {
        var tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Block Scoping: intent



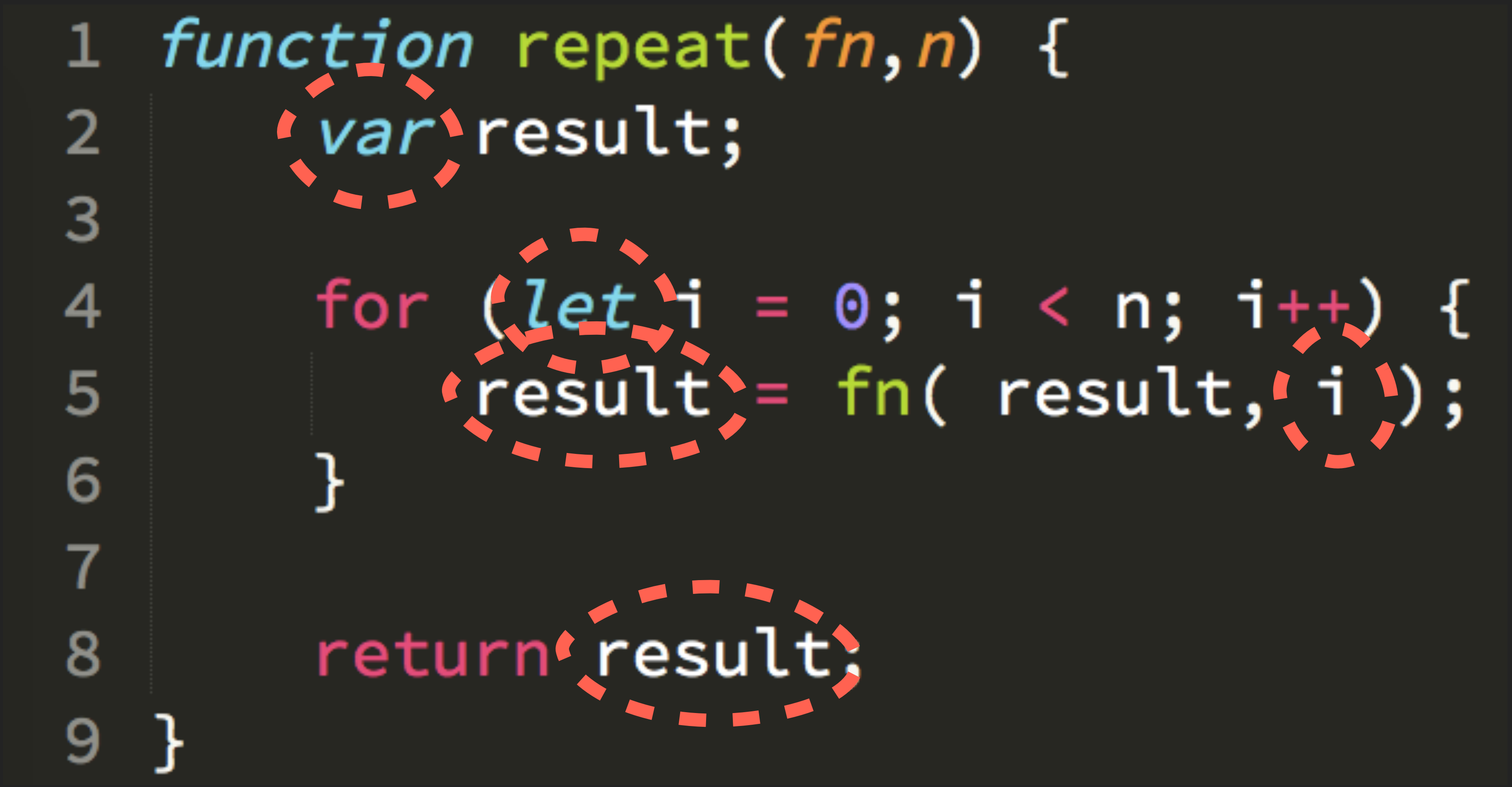
```
function diff(x,y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Block Scoping: let



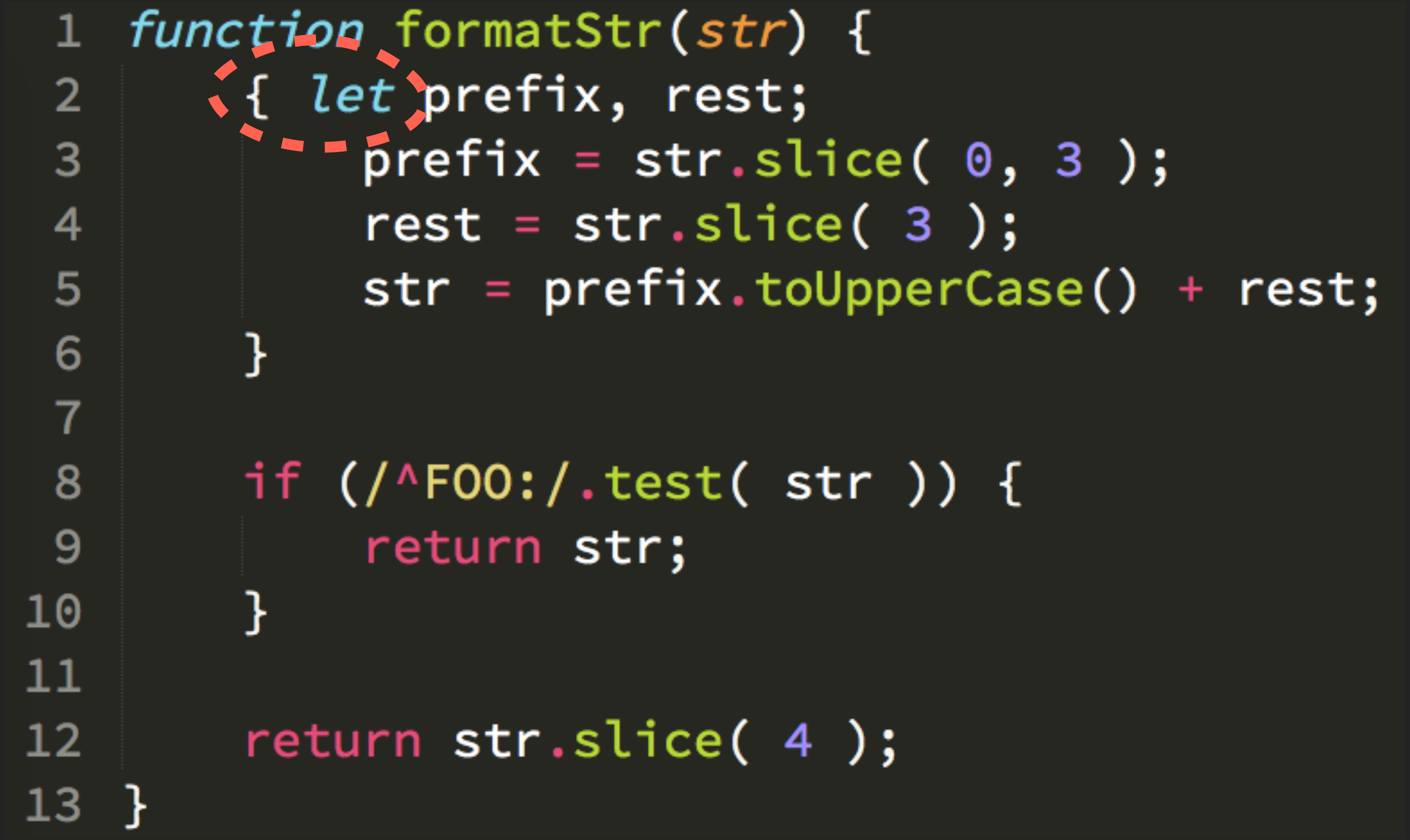
```
function repeat(fn,n) {
    var result;

    for (var i = 0; i < n; i++) {
        result = fn( result, i );
    }

    return result;
}
```

Block Scoping: "well, actually, not all vars..."



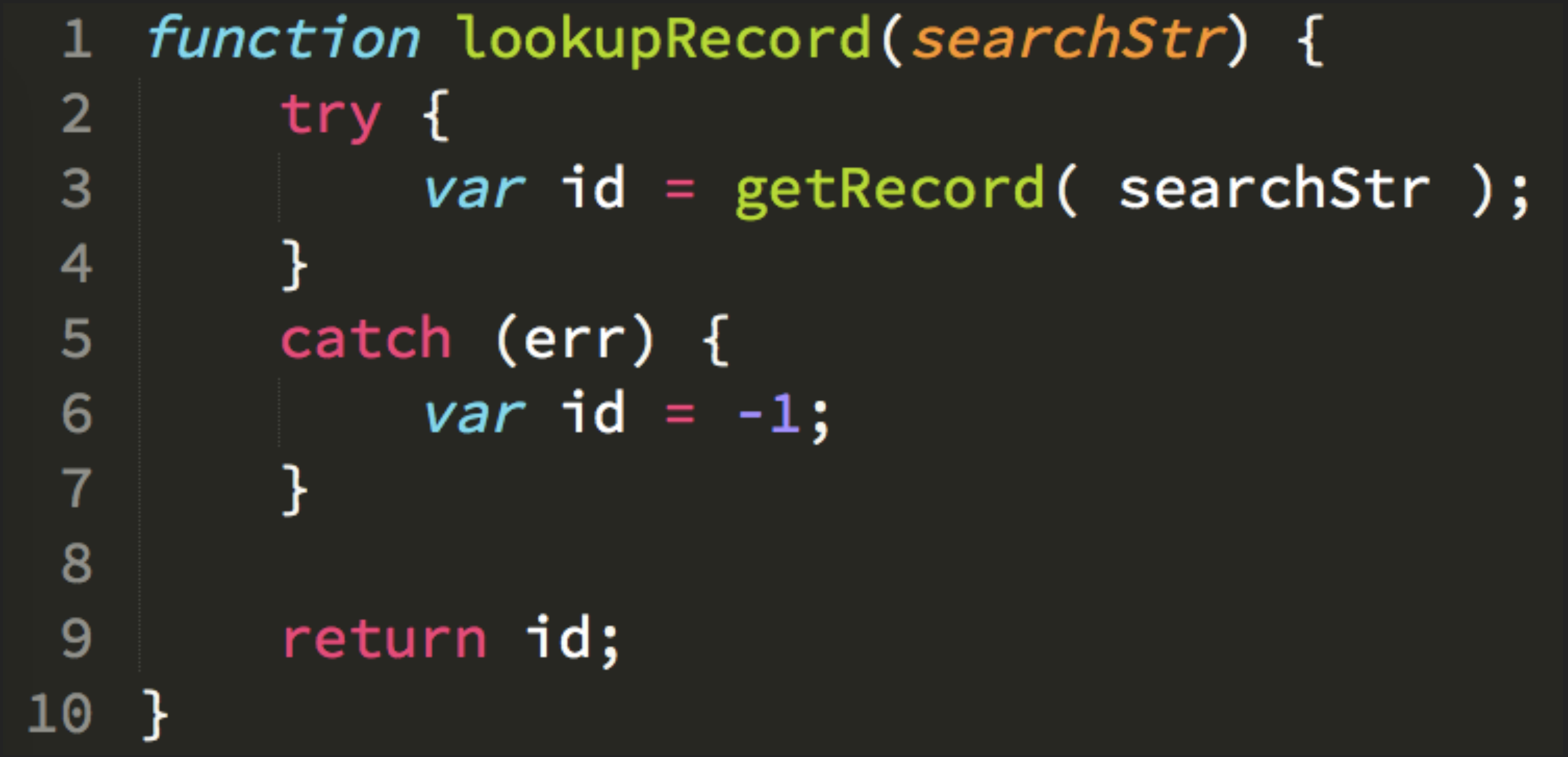
```
function repeat(fn,n) {
    var result;

    for (let i = 0; i < n; i++) {
        result = fn( result, i );
    }

    return result;
}
```

Block Scoping: let + var



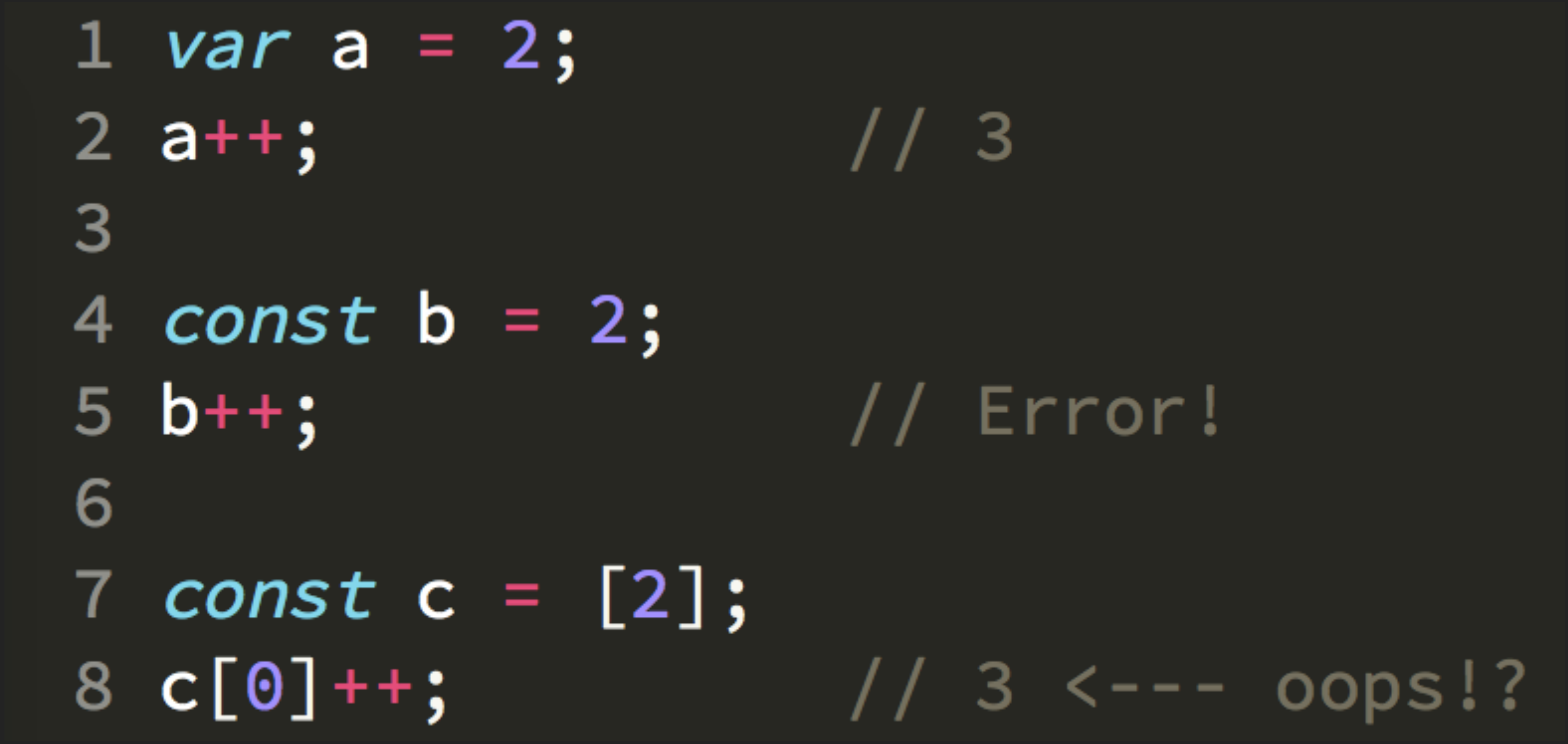
```javascript
function formatStr(str) {
    { let prefix, rest;
        prefix = str.slice( 0, 3 );
        rest = str.slice( 3 );
        str = prefix.toUpperCase() + rest;
    }

    if (/^FOO:/.test( str )) {
        return str;
    }

    return str.slice( 4 );
}
```

Block Scoping: explicit let block



```
function lookupRecord(searchStr) {
    try {
        var id = getRecord( searchStr );
    }
    catch (err) {
        var id = -1;
    }

    return id;
}
```

Block Scoping: sometimes var > let



```javascript
var a = 2;
a++;                // 3

const b = 2;
b++;                // Error!

const c = [2];
c[0]++;             // 3 <--- oops!?
```

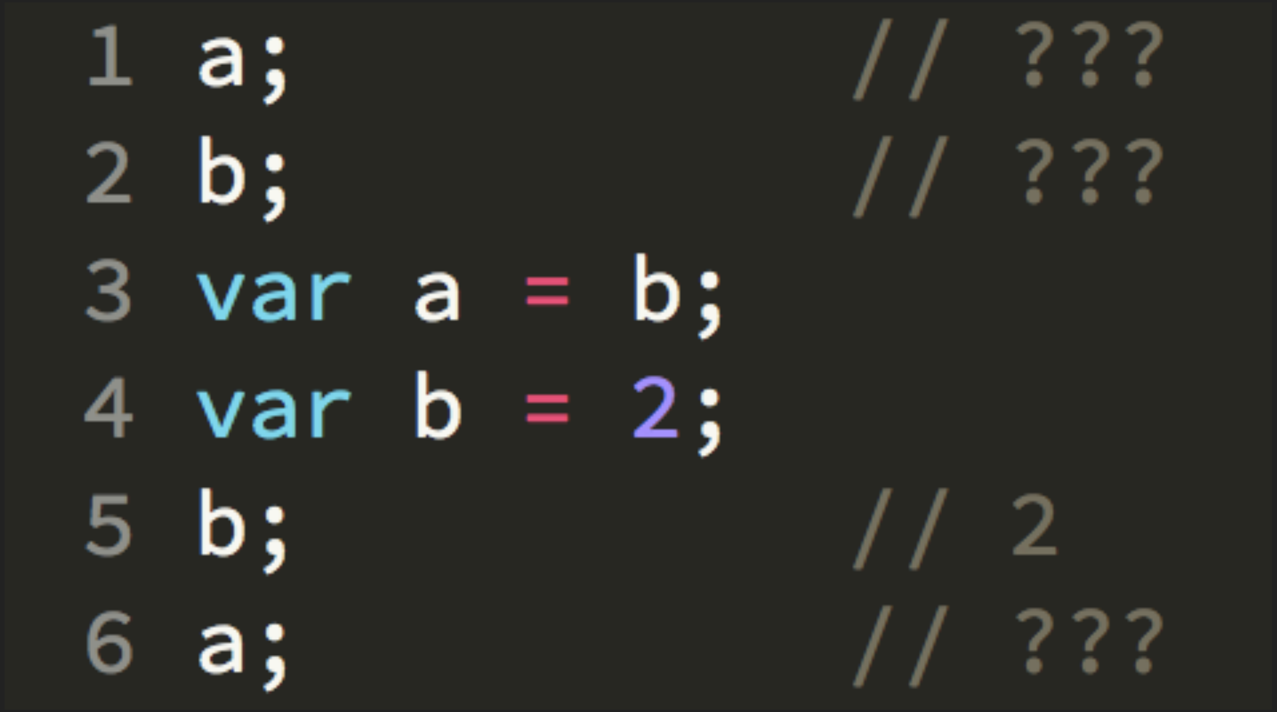
Block Scoping: const(antly confusing)

Quiz

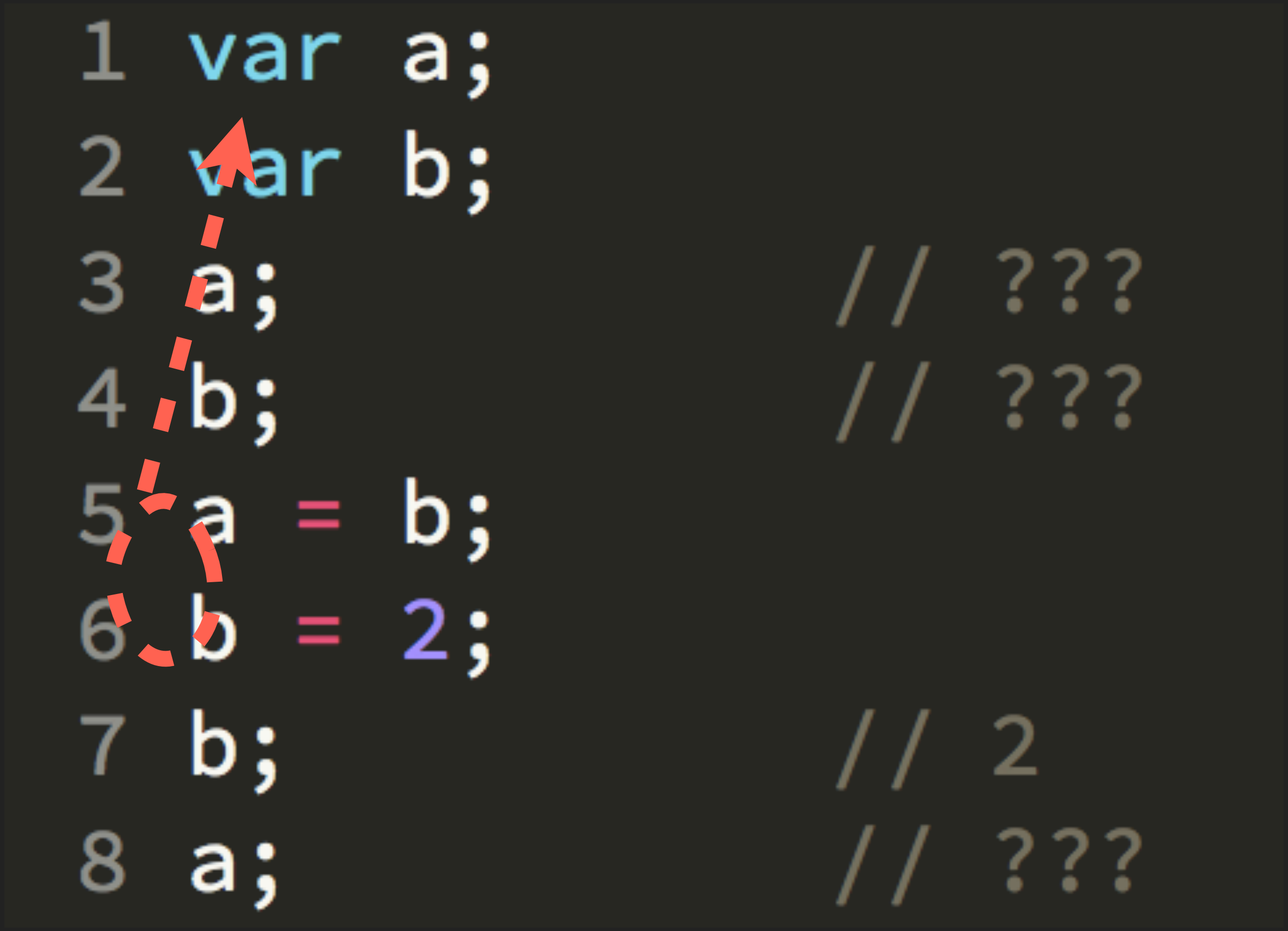
1. What type of scoping rule(s) does JavaScript have?
2. What are the 2 different ways you can create a new scope?
3. What's the difference between undeclared and undefined?

Hoisting

Scope: hoisting

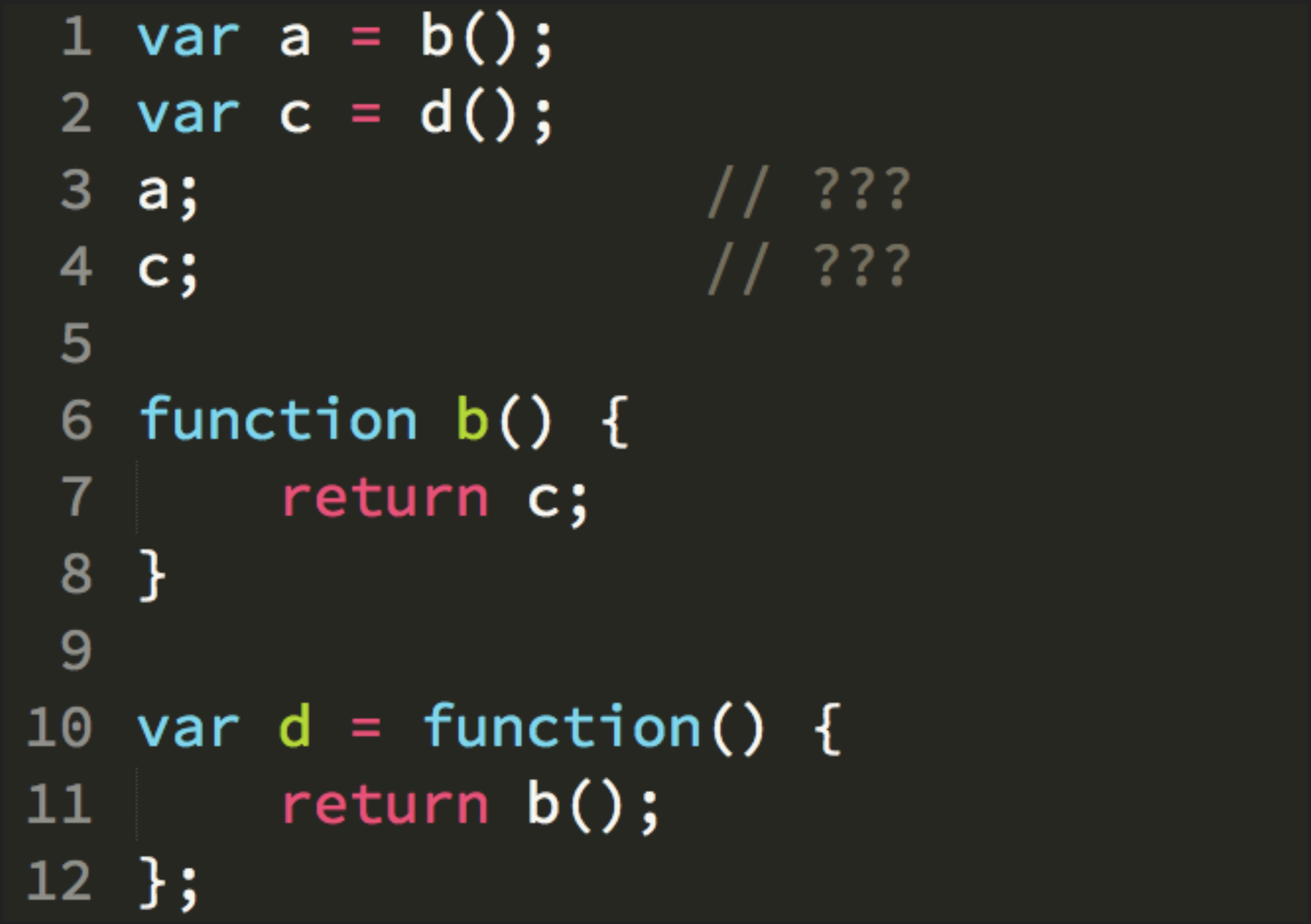


```javascript
a;            // ???
b;            // ???
var a = b;
var b = 2;
b;            // 2
a;            // ???
```



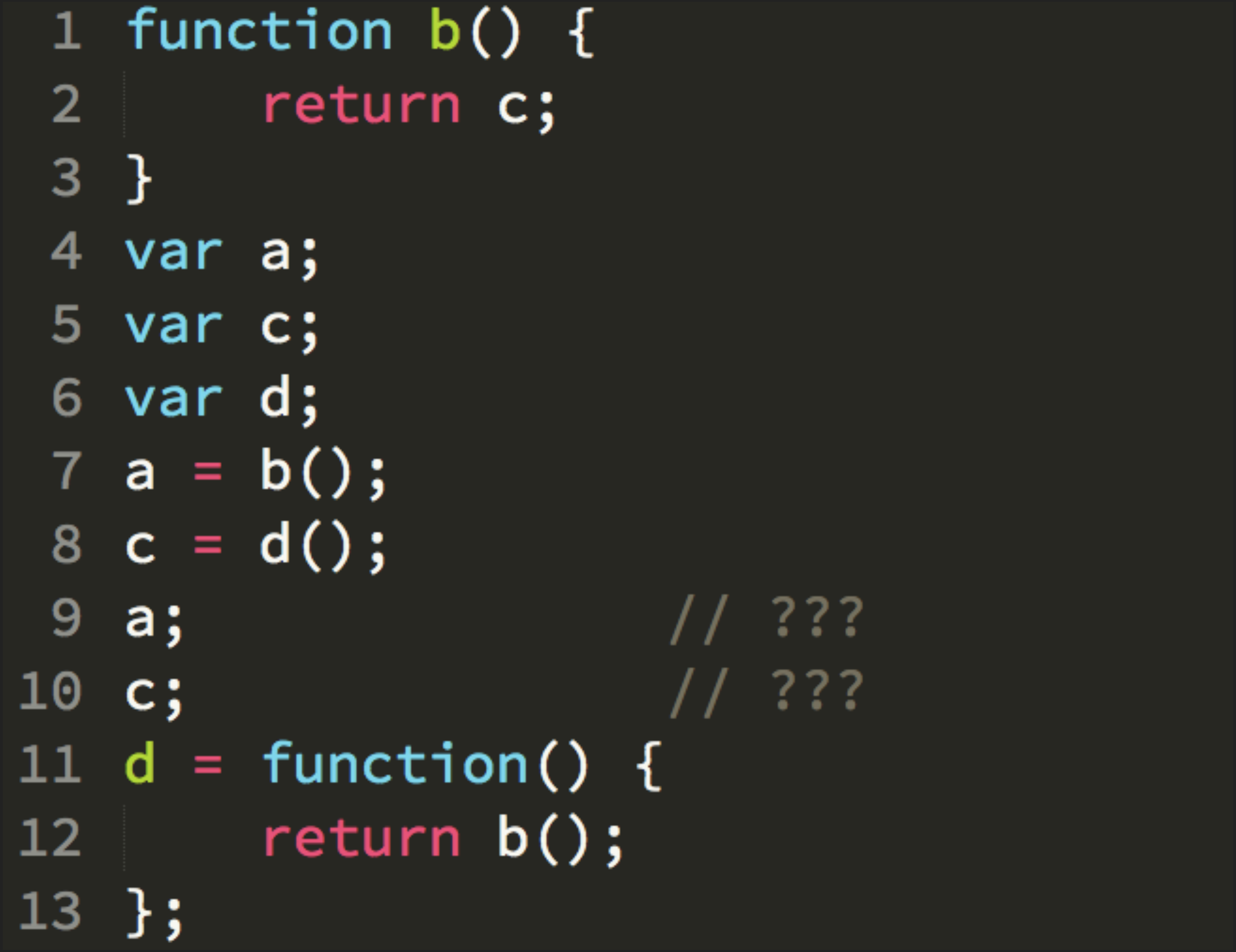
```
var a;
var b;
a;                  // ???
b;                  // ???
a = b;
b = 2;
b;                  // 2
a;                  // ???
```

Scope: hoisting



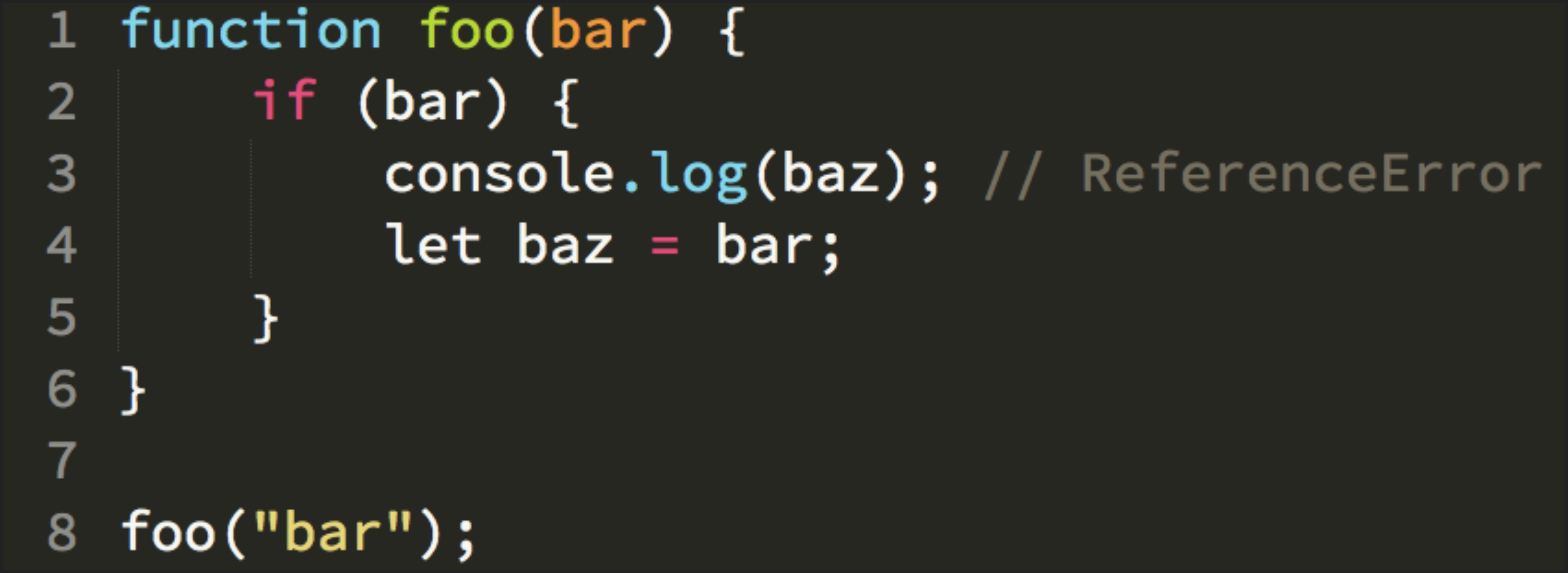
```javascript
var a = b();
var c = d();
a;                    // ???
c;                    // ???

function b() {
    return c;
}

var d = function() {
    return b();
};
```

Scope: hoisting



```
function b() {
    return c;
}
var a;
var c;
var d;
a = b();
c = d();
a;                  // ???
c;                  // ???
d = function() {
    return b();
};
```

Scope: hoisting

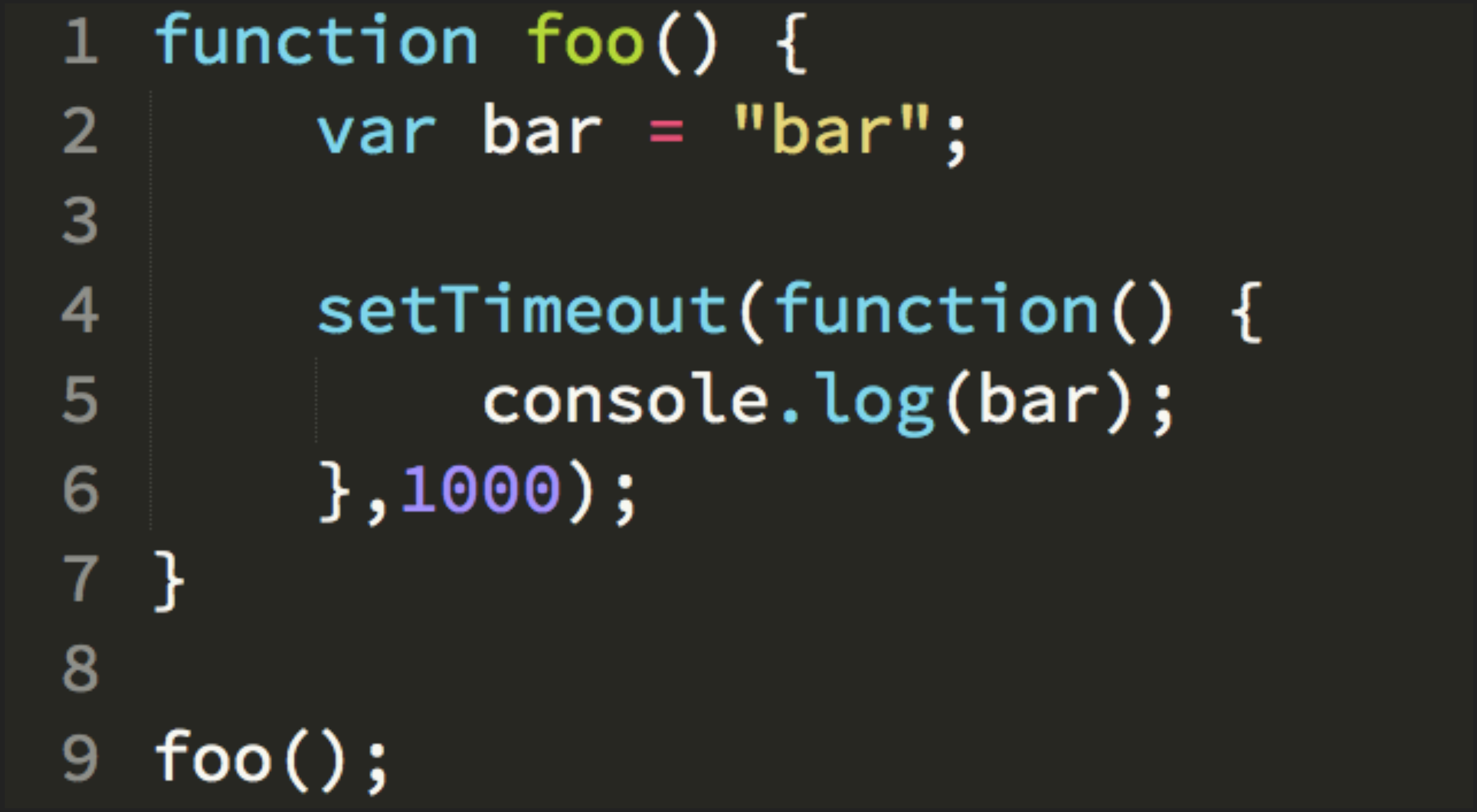


```javascript
function foo(bar) {
    if (bar) {
        console.log(baz); // ReferenceError
        let baz = bar;
    }
}

foo("bar");
```

Hoisting: let gotcha

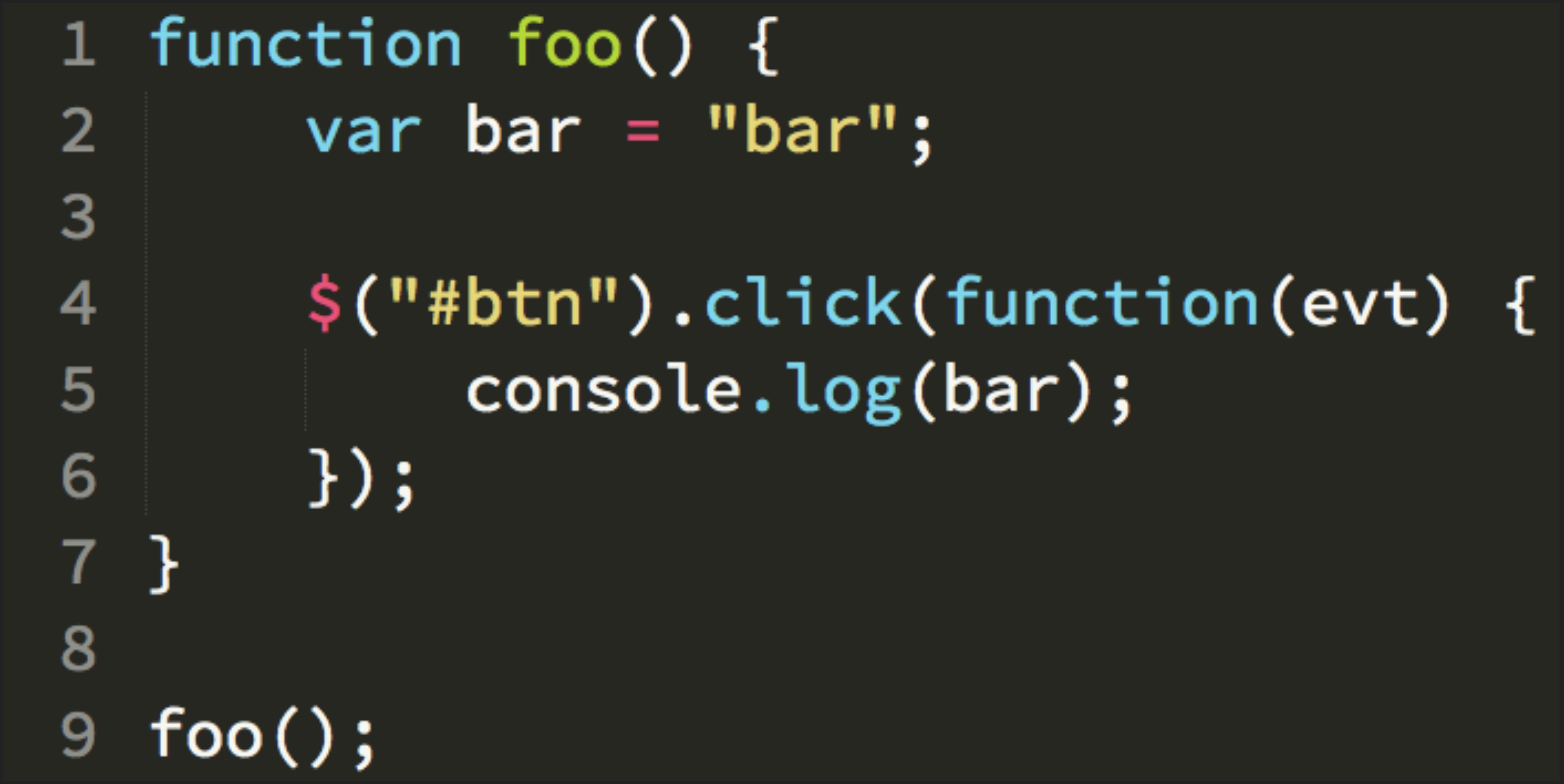
Closure

Closure is when a function “remembers” its lexical scope even when the function is executed outside that lexical scope.



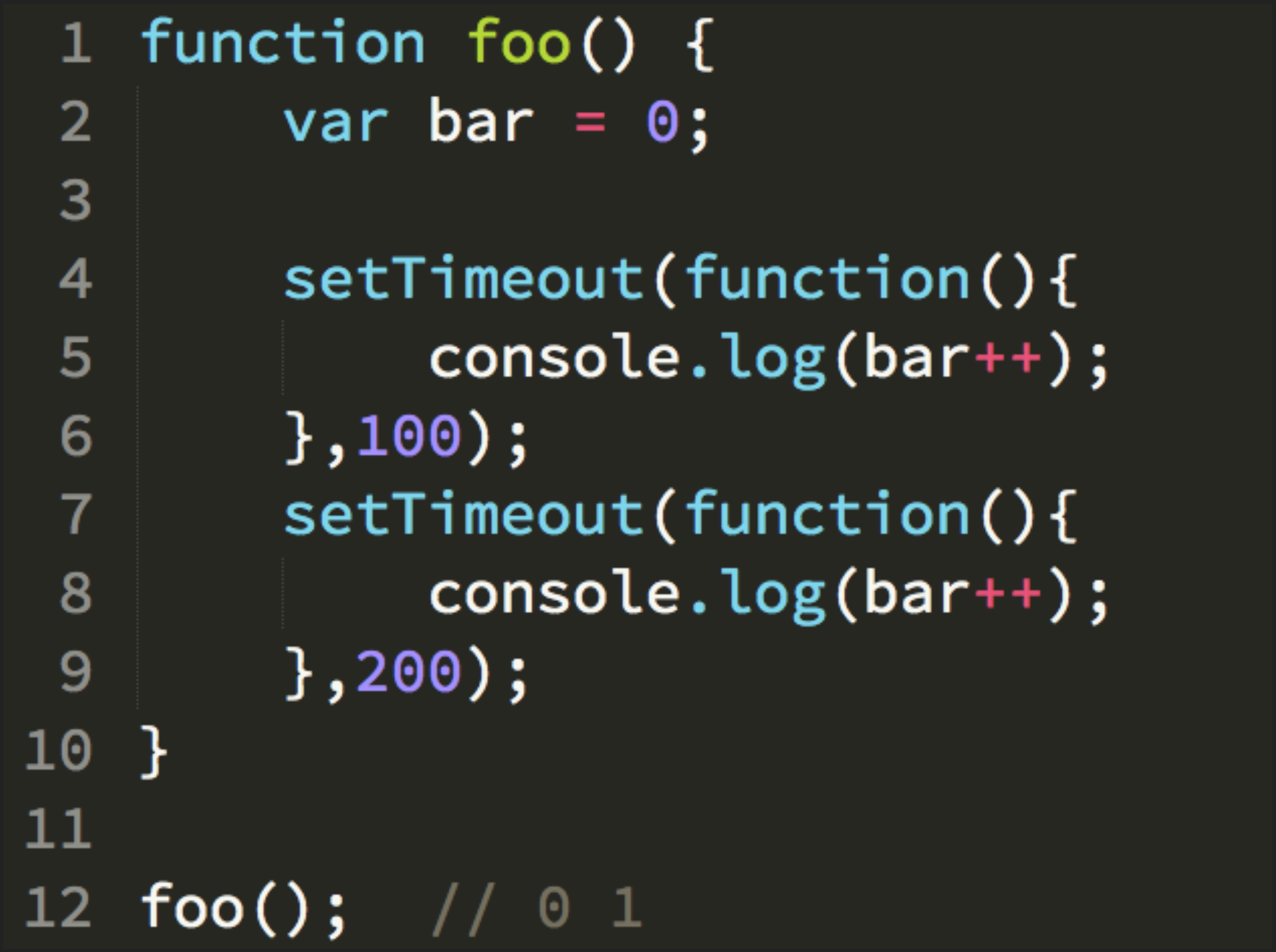
```
function foo() {
    var bar = "bar";

    setTimeout(function() {
        console.log(bar);
    },1000);
}

foo();
```

Closure



```
function foo() {
    var bar = "bar";

    $("#btn").click(function(evt) {
        console.log(bar);
    });
}

foo();
```

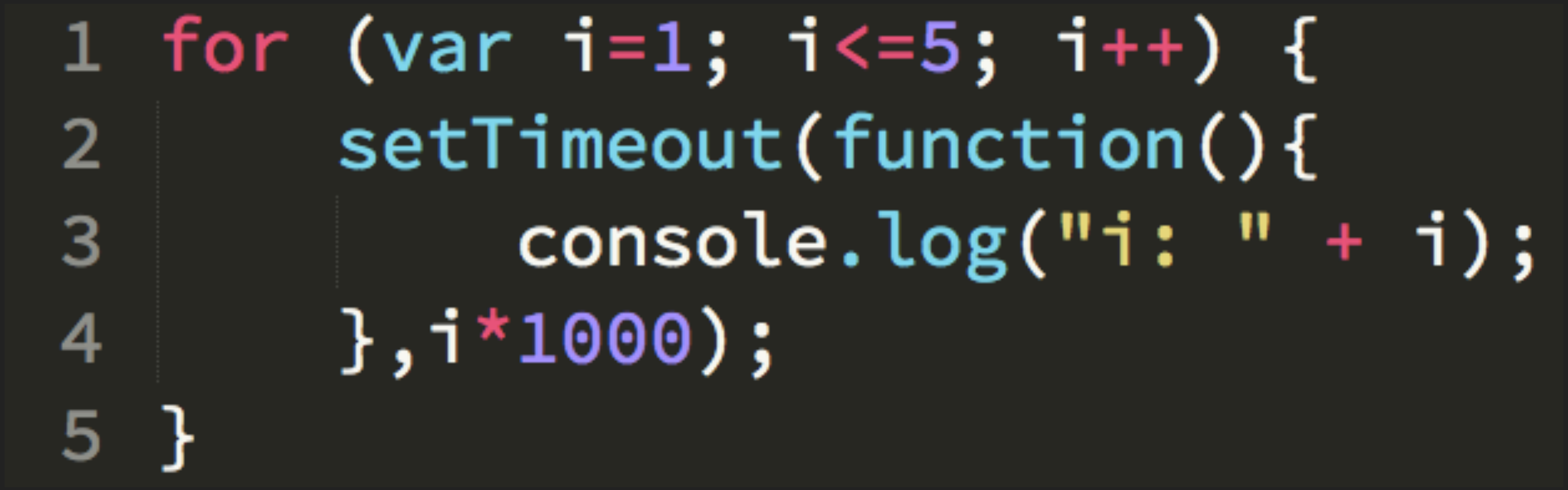
Closure



```
function foo() {
    var bar = 0;

    setTimeout(function(){
        console.log(bar++);
    },100);
    setTimeout(function(){
        console.log(bar++);
    },200);
}

foo();  // 0 1
```

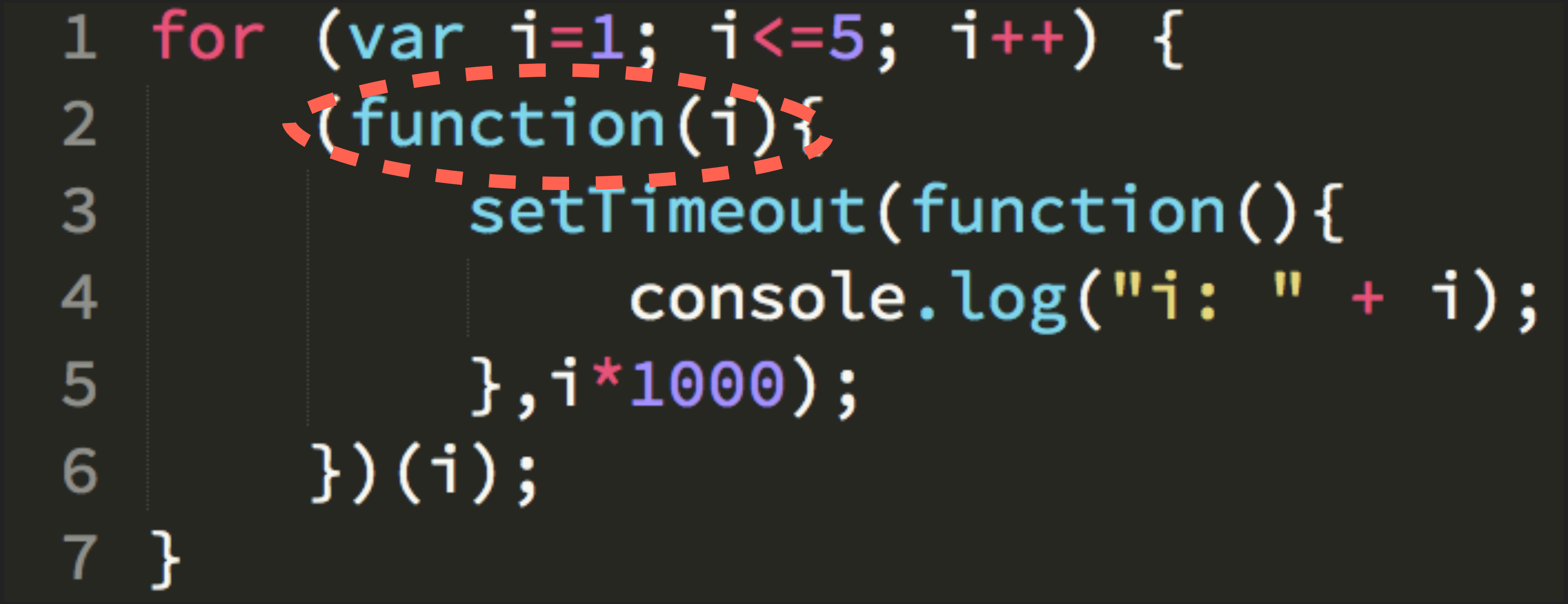
Closure: shared scope

Code Me



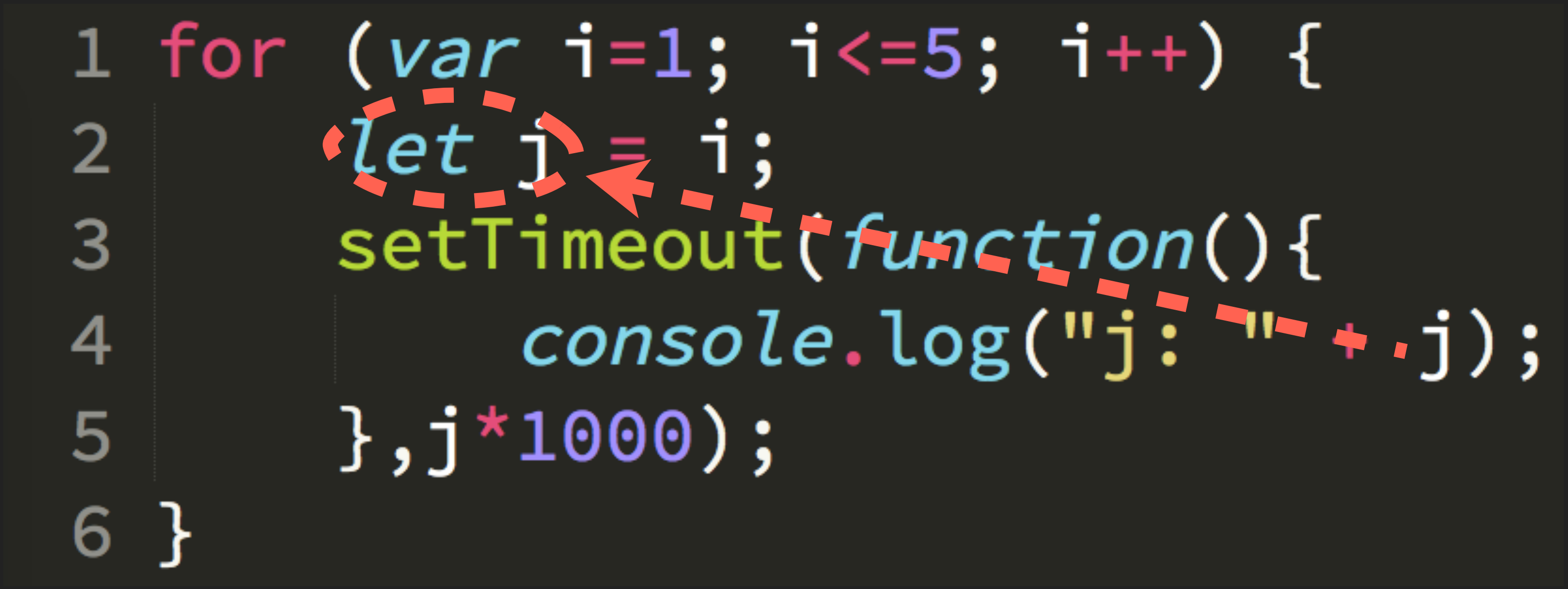
```
for (var i=1; i<=5; i++) {
    setTimeout(function(){
        console.log("i: " + i);
    },i*1000);
}
```

Closure: loops



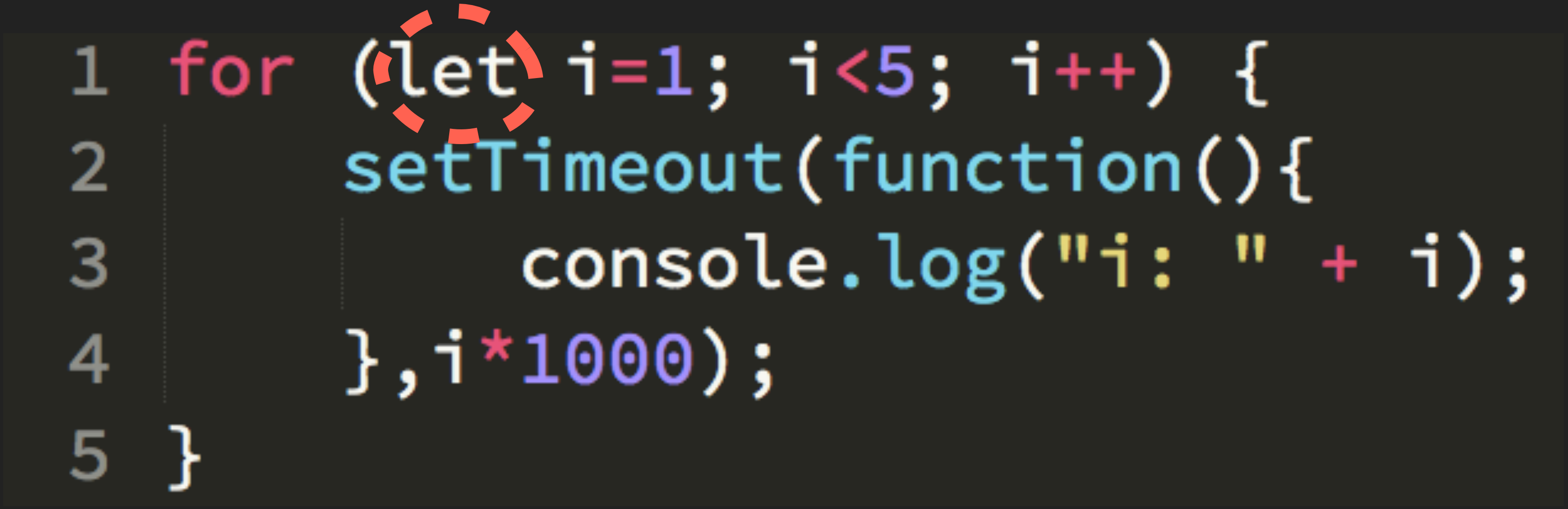
```
for (var i=1; i<=5; i++) {
    (function(i){
        setTimeout(function(){
            console.log("i: " + i);
        },i*1000);
    })(i);
}
```

Closure: loops



```
for (var i=1; i<=5; i++) {
    let j = i;
    setTimeout(function(){
        console.log("j: " + j);
    },j*1000);
}
```

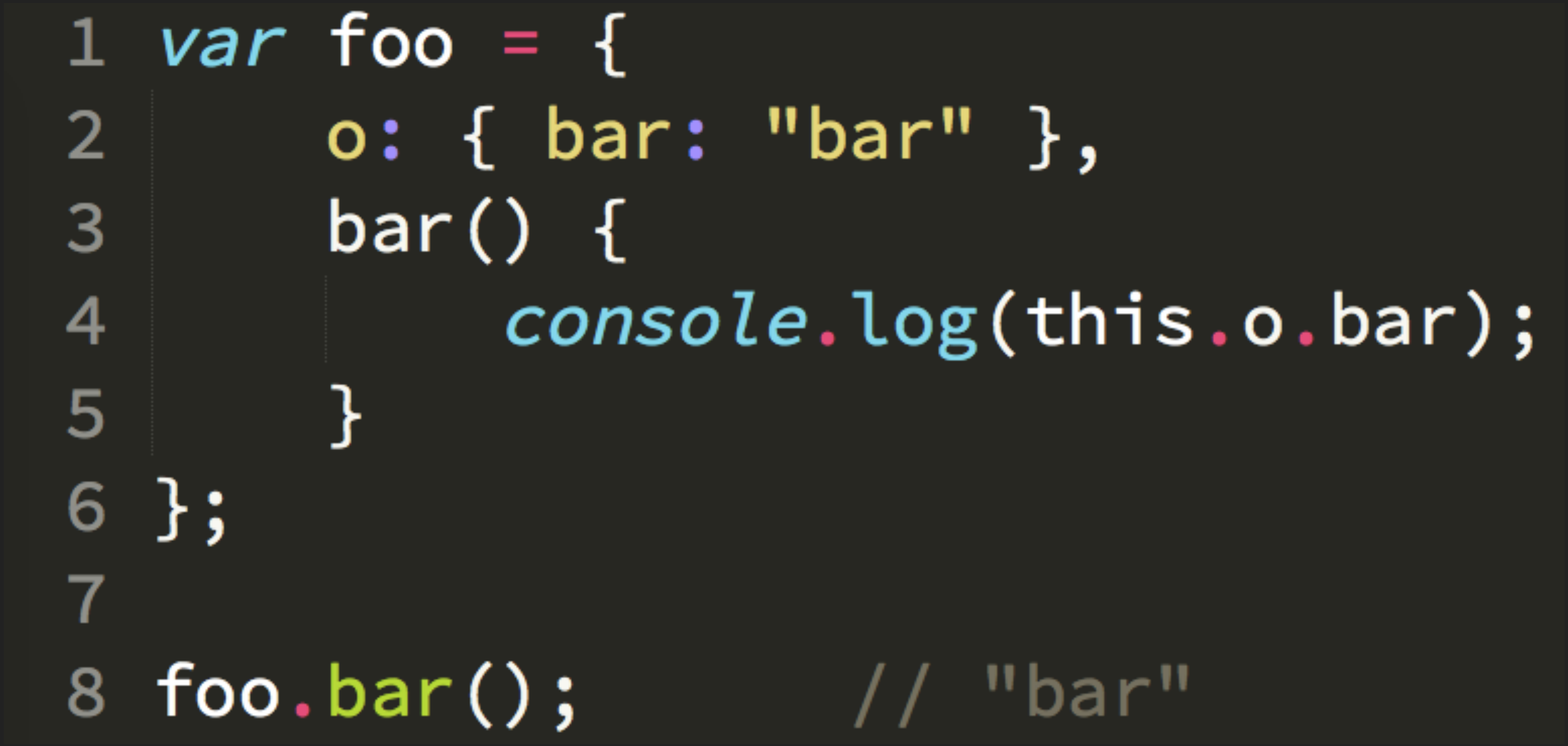
Closure: loops + block scope



```
for (let i=1; i<5; i++) {
    setTimeout(function(){
        console.log("i: " + i);
    },i*1000);
}
```

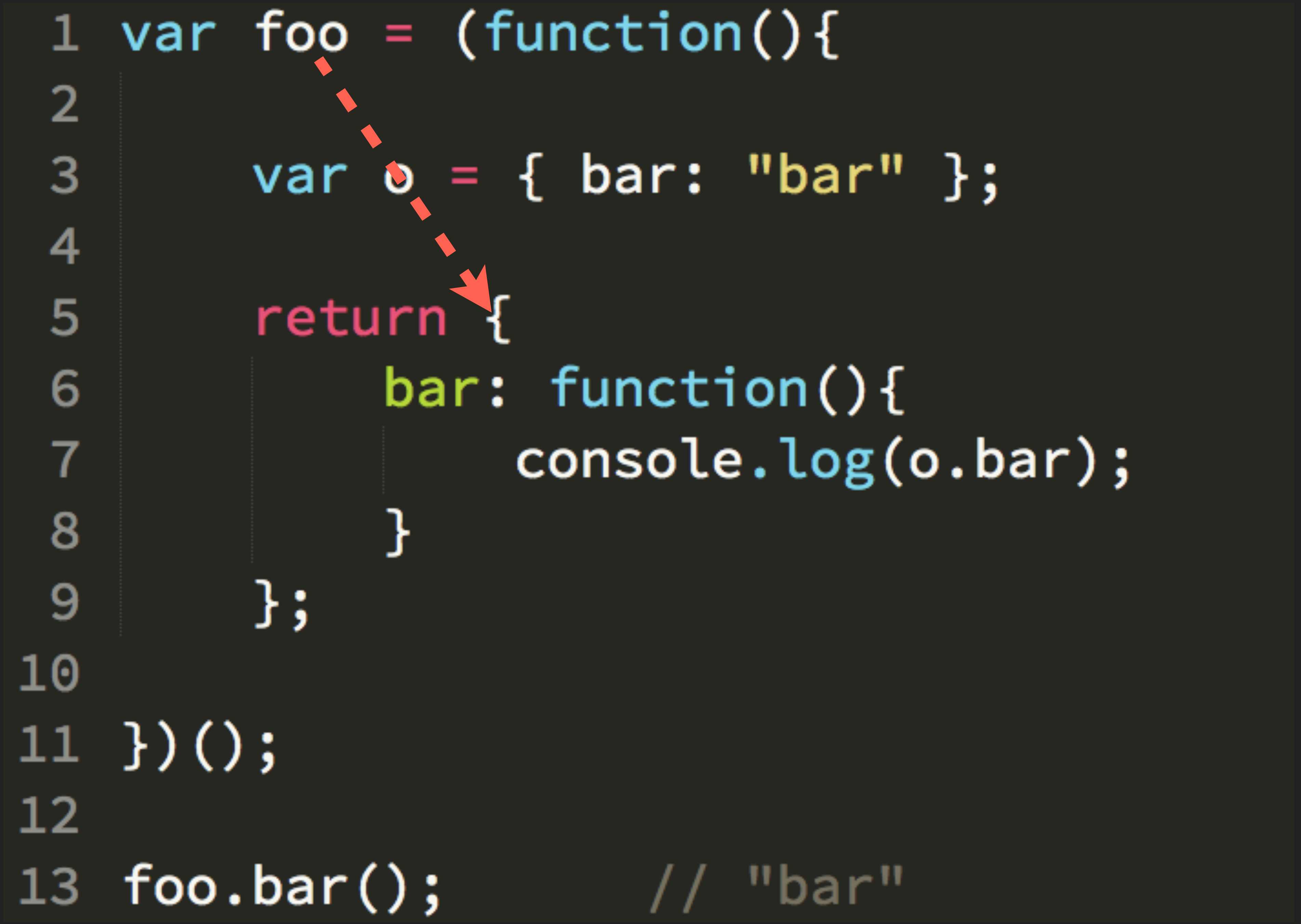
Closure: loops + block scope

Modules



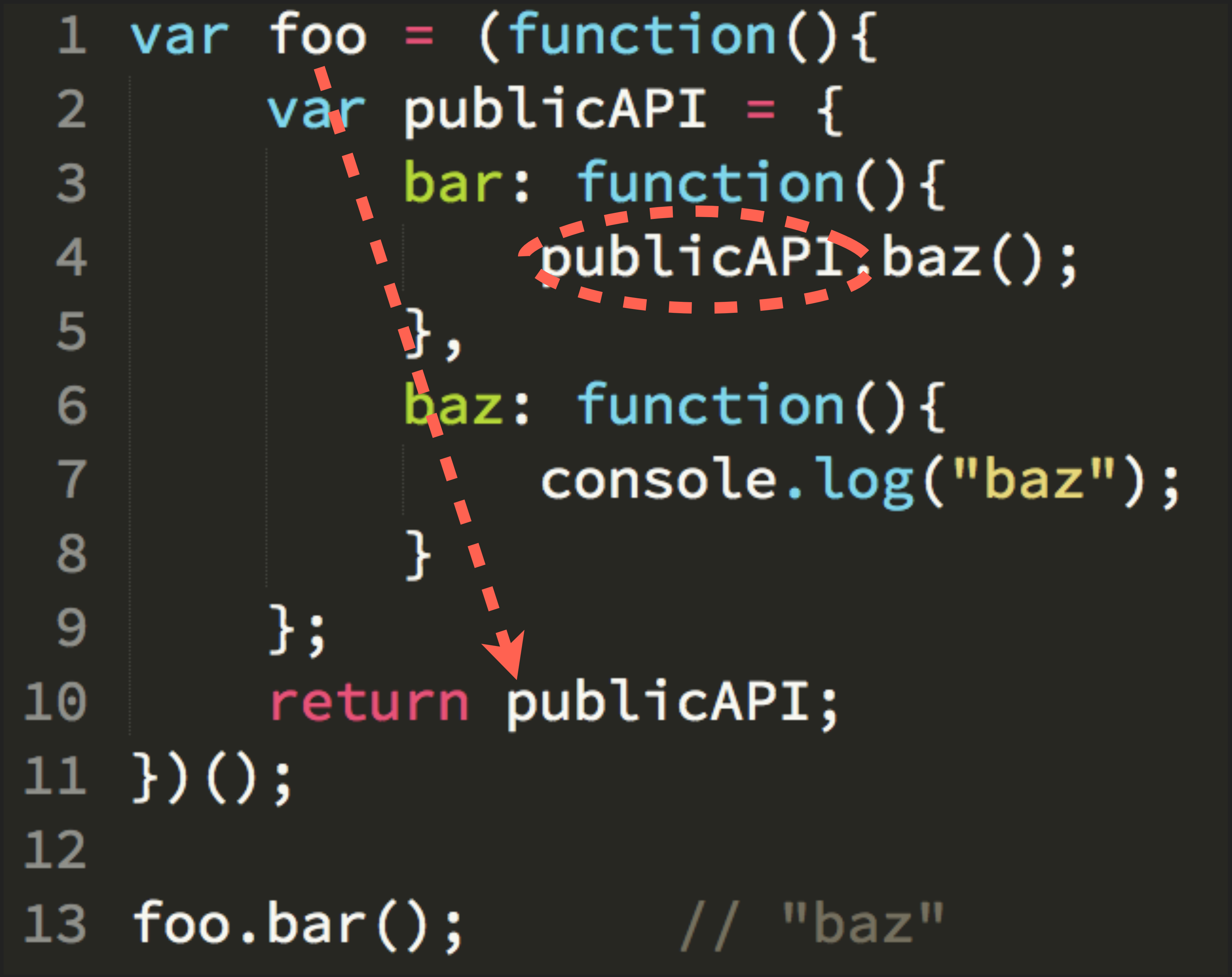
```
var foo = {
    o: { bar: "bar" },
    bar() {
        console.log(this.o.bar);
    }
};

foo.bar();          // "bar"
```

Not a module



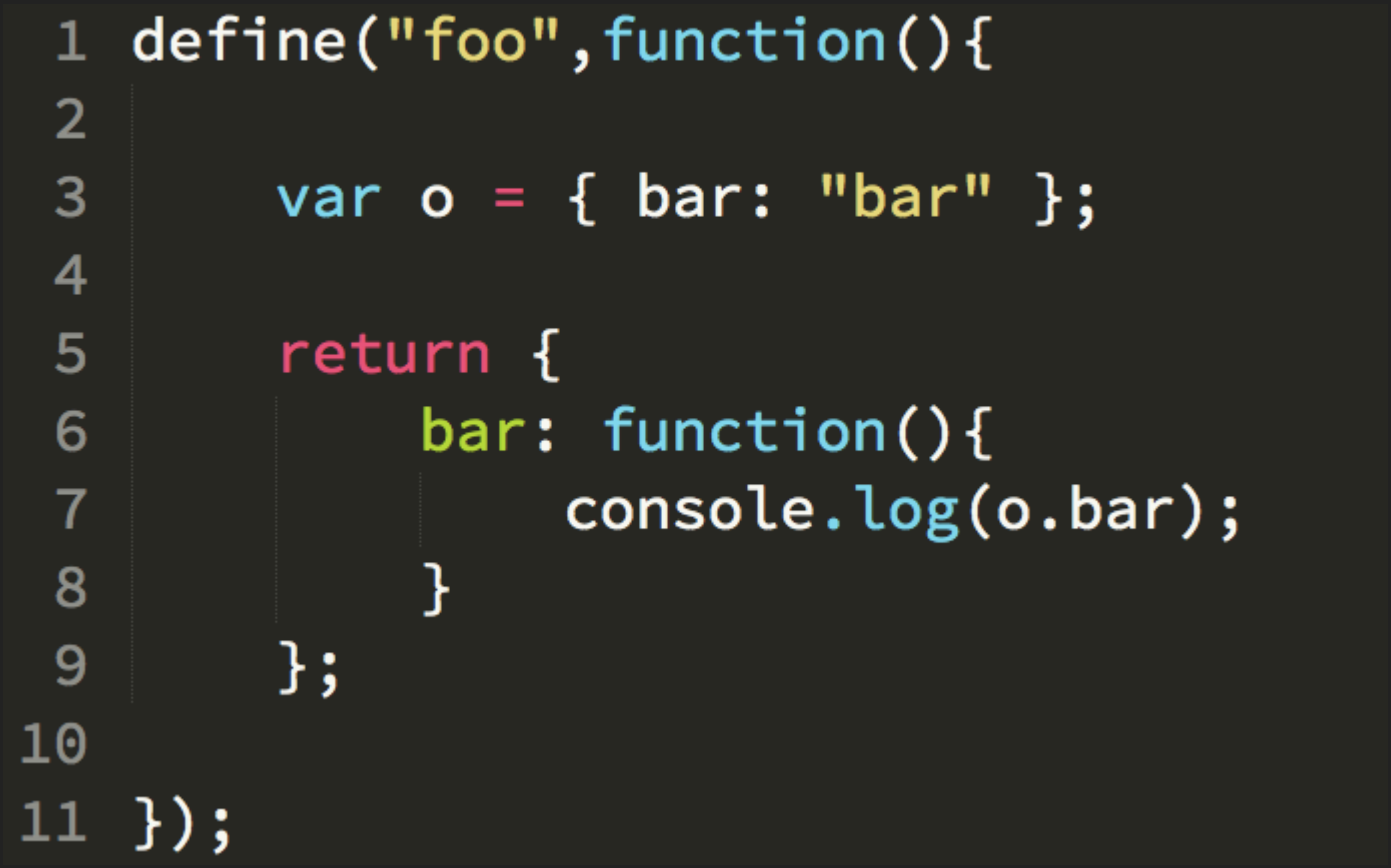
```
var foo = (function(){

    var o = { bar: "bar" };

    return {
        bar: function(){
            console.log(o.bar);
        }
    };

})();

foo.bar();          // "bar"
```

Classic module pattern



```
var foo = (function(){
    var publicAPI = {
        bar: function(){
            publicAPI.baz();
        },
        baz: function(){
            console.log("baz");
        }
    };
    return publicAPI;
})();

foo.bar();        // "baz"
```

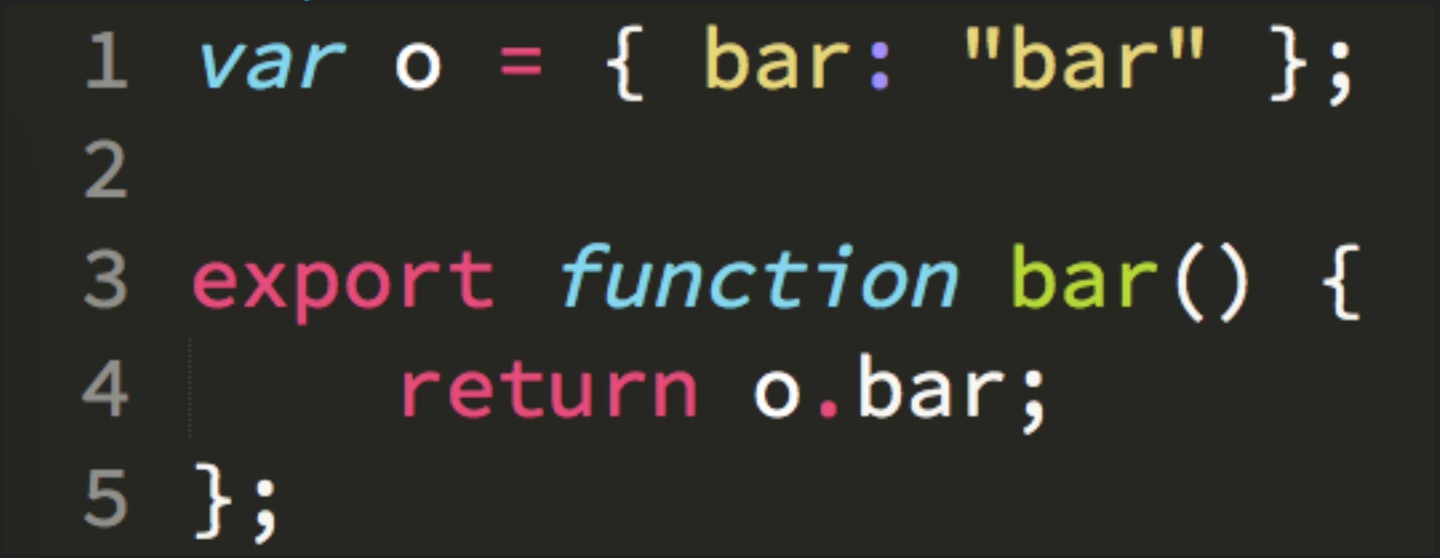
Classic module pattern: modified



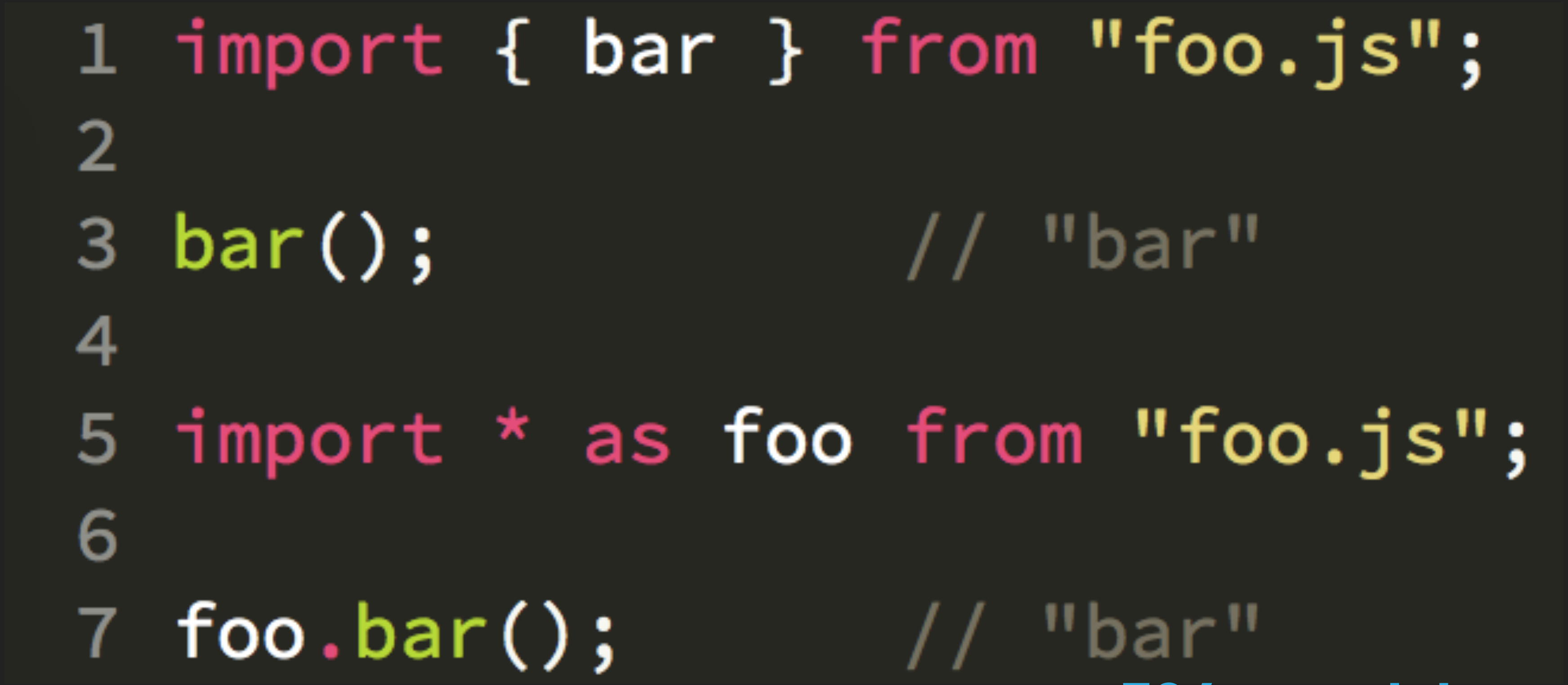
```
define("foo",function(){

    var o = { bar: "bar" };

    return {
        bar: function(){
            console.log(o.bar);
        }
    };

});
```

Modern module pattern

foo.js:



```
var o = { bar: "bar" };

export function bar() {
        return o.bar;
};
```



```
import { bar } from "foo.js";

bar();              // "bar"

import * as foo from "foo.js";

foo.bar();          // "bar"
```

ES6+ module pattern

Quiz

1. What is a closure and how is it created?
2. How long does its scope stay around?
3. Why doesn't a function callback inside a loop behave as expected? How do we fix it?
4. How do you use a closure to create an encapsulated module? What's the benefits of that approach?

(exercise #5: 20min)

Object-Orienting

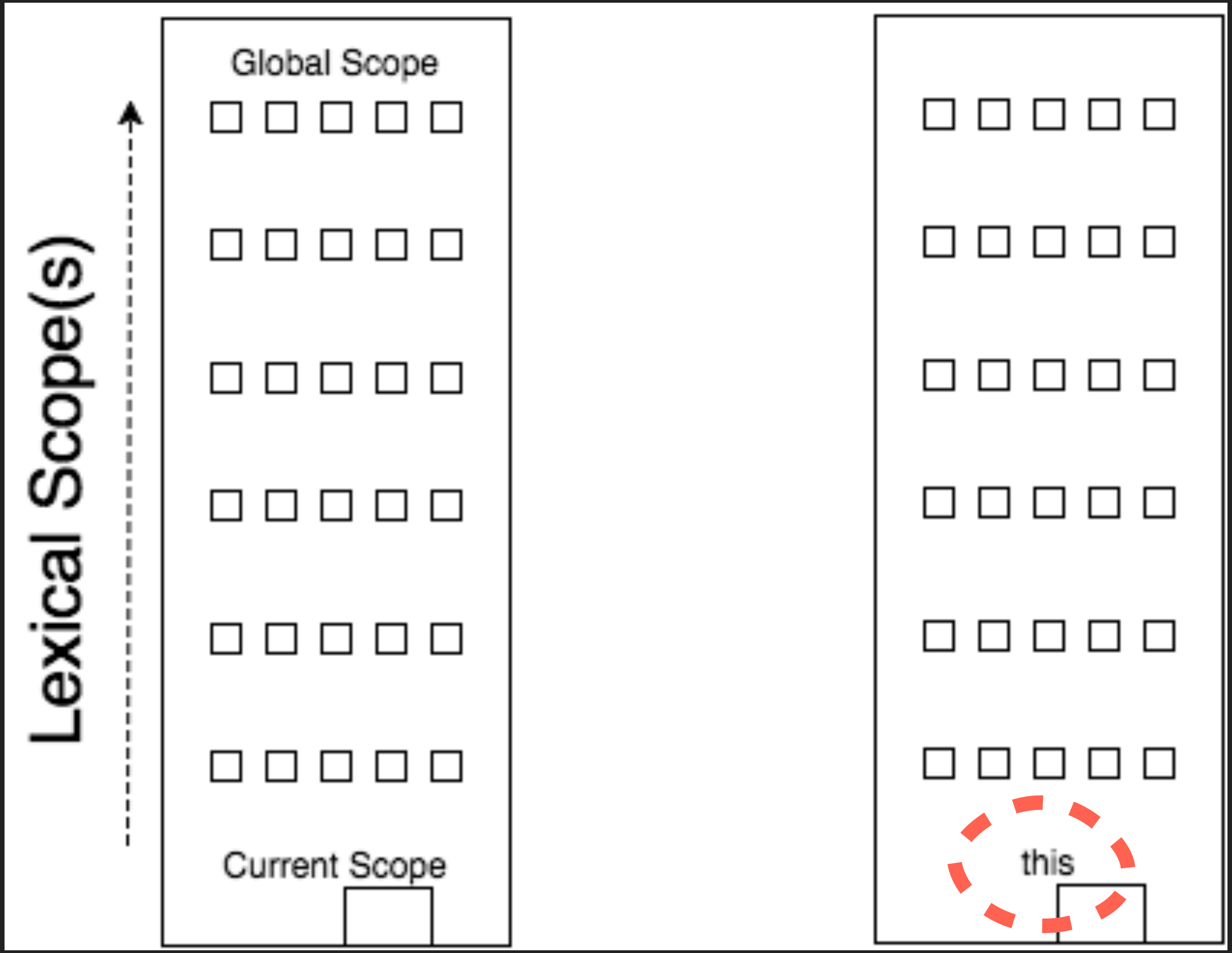
- this
- Prototypes
- class { }
- “Inheritance” vs. “Behavior Delegation”
  (OO vs. OLOO)

this

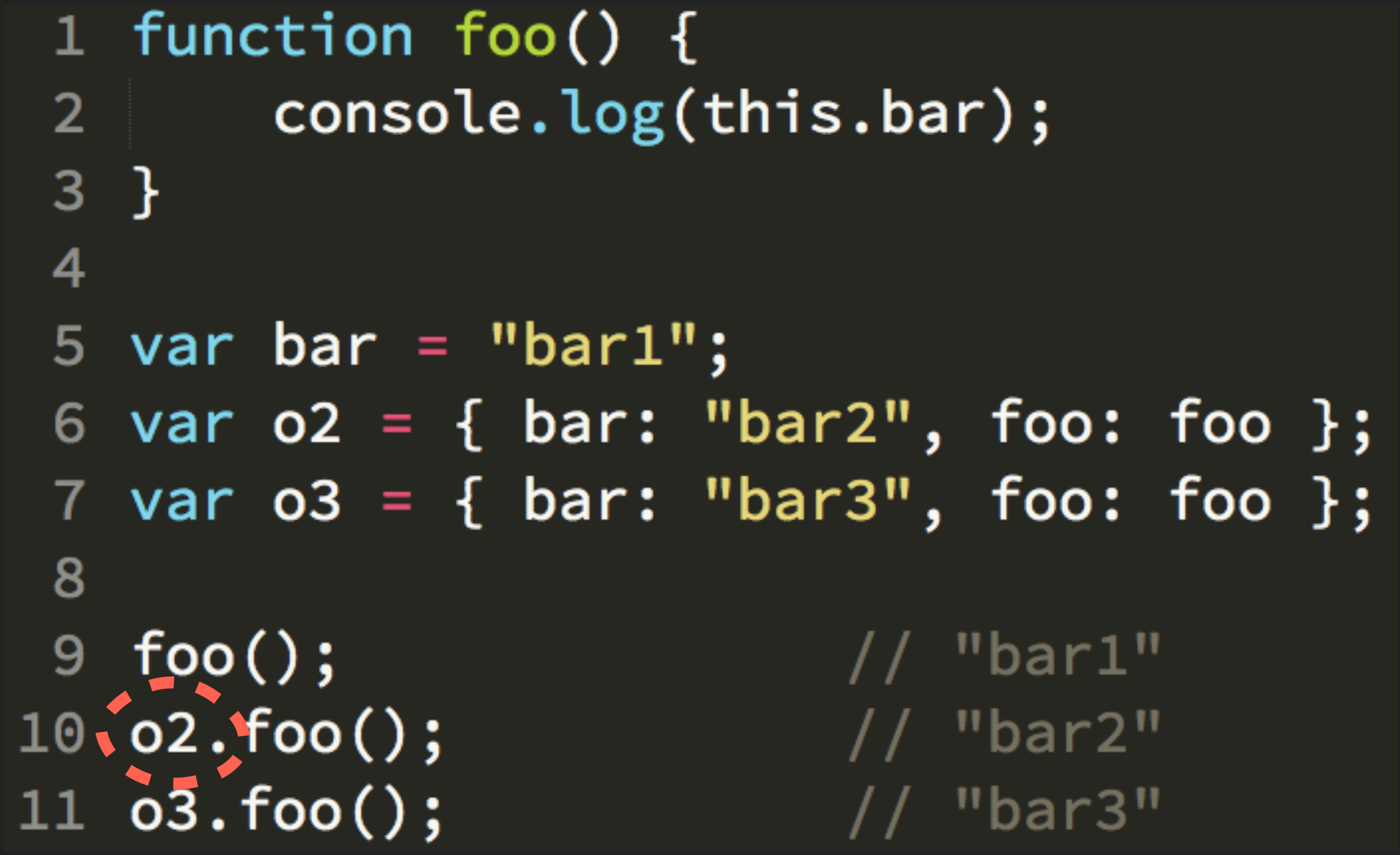
Every* function, while executing, has a reference to its current execution context, called this.

Remember lexical scope vs. dynamic scope?

JavaScript's version of "dynamic scope" is this.

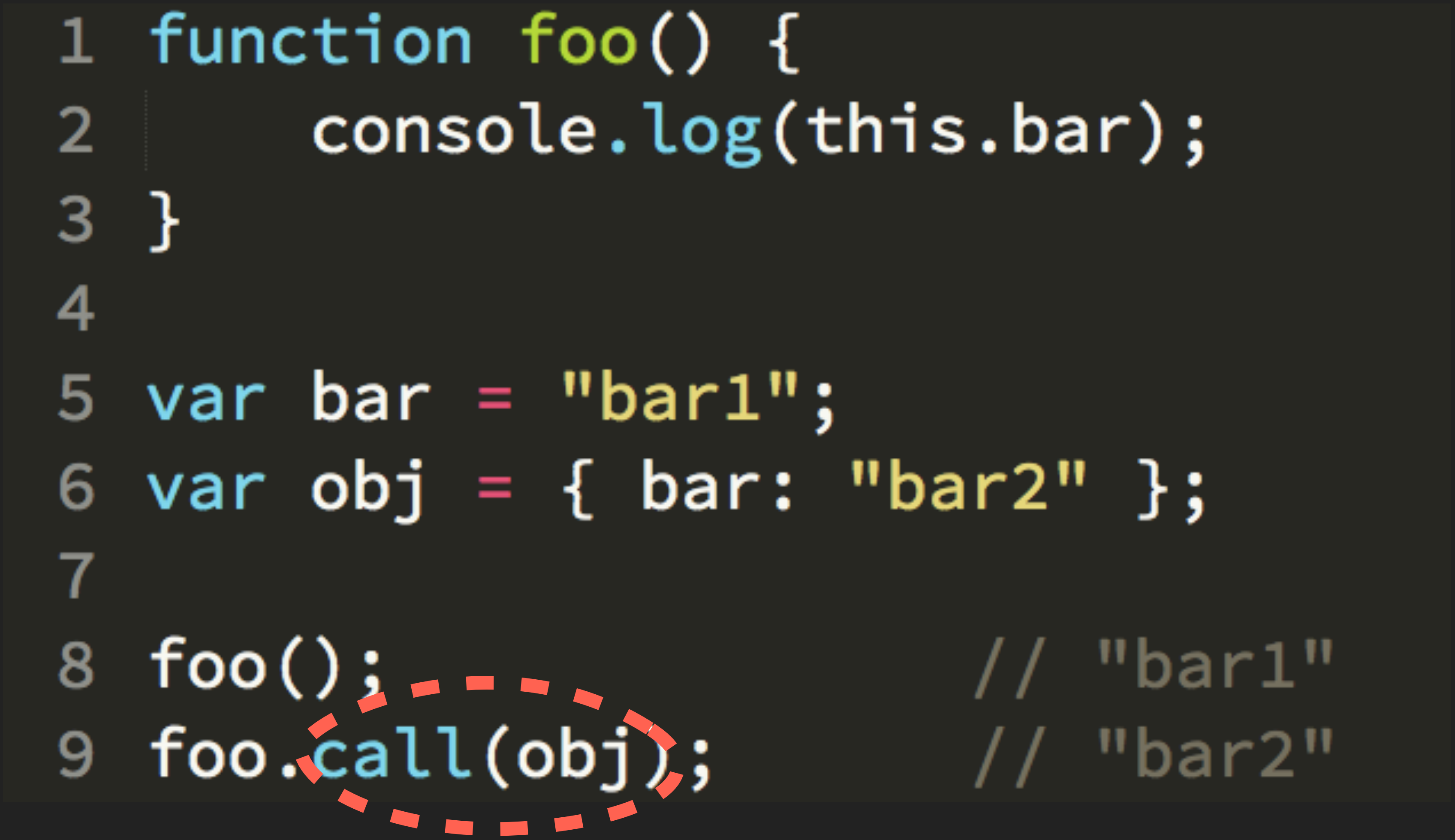


Scope



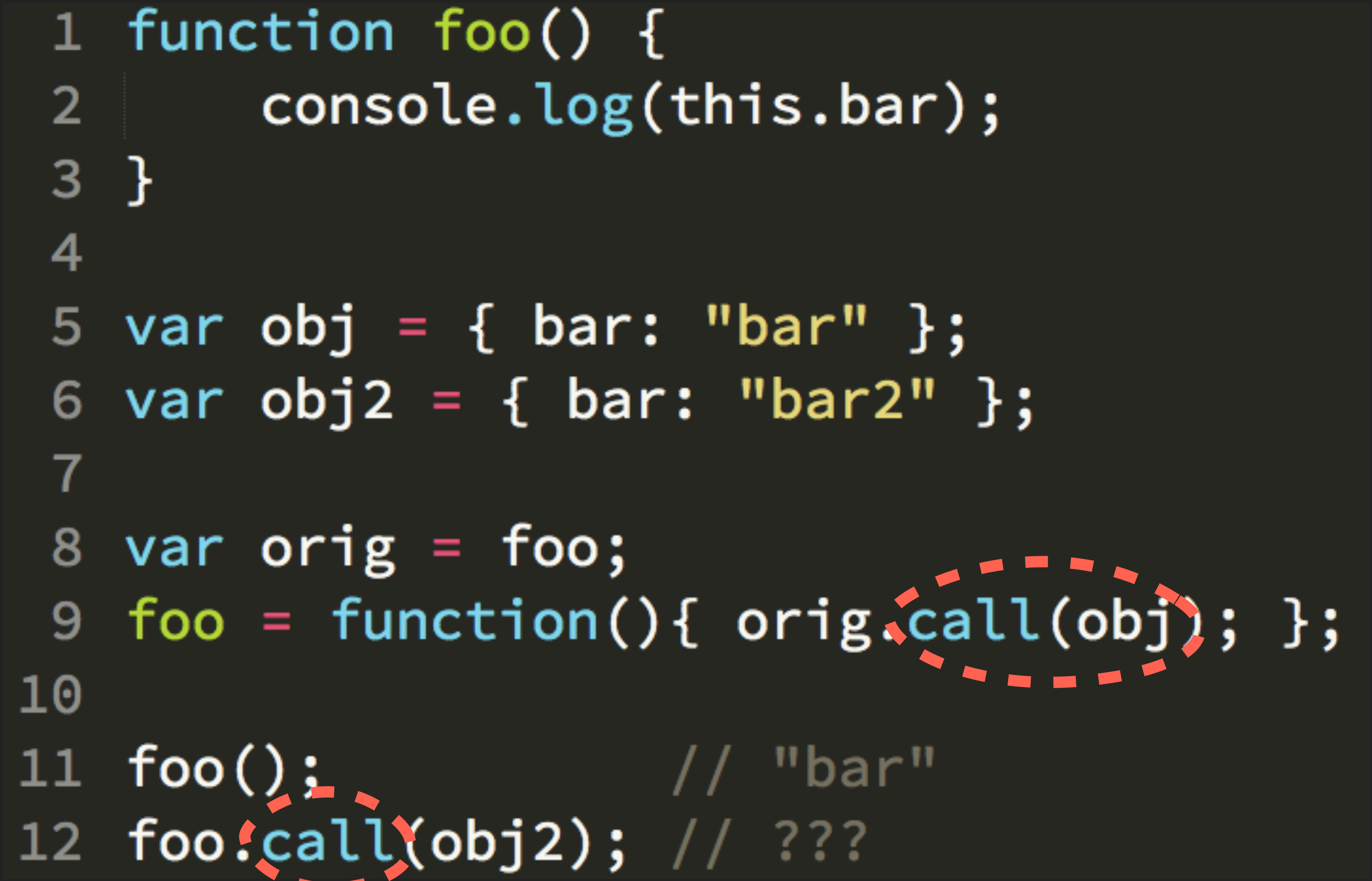
```
function foo() {
    console.log(this.bar);
}

var bar = "bar1";
var o2 = { bar: "bar2", foo: foo };
var o3 = { bar: "bar3", foo: foo };

foo();                  // "bar1"
o2.foo();               // "bar2"
o3.foo();               // "bar3"
```

this: implicit & default binding



```javascript
function foo() {
    console.log(this.bar);
}

var bar = "bar1";
var obj = { bar: "bar2" };

foo();                  // "bar1"
foo.call(obj);          // "bar2"
```

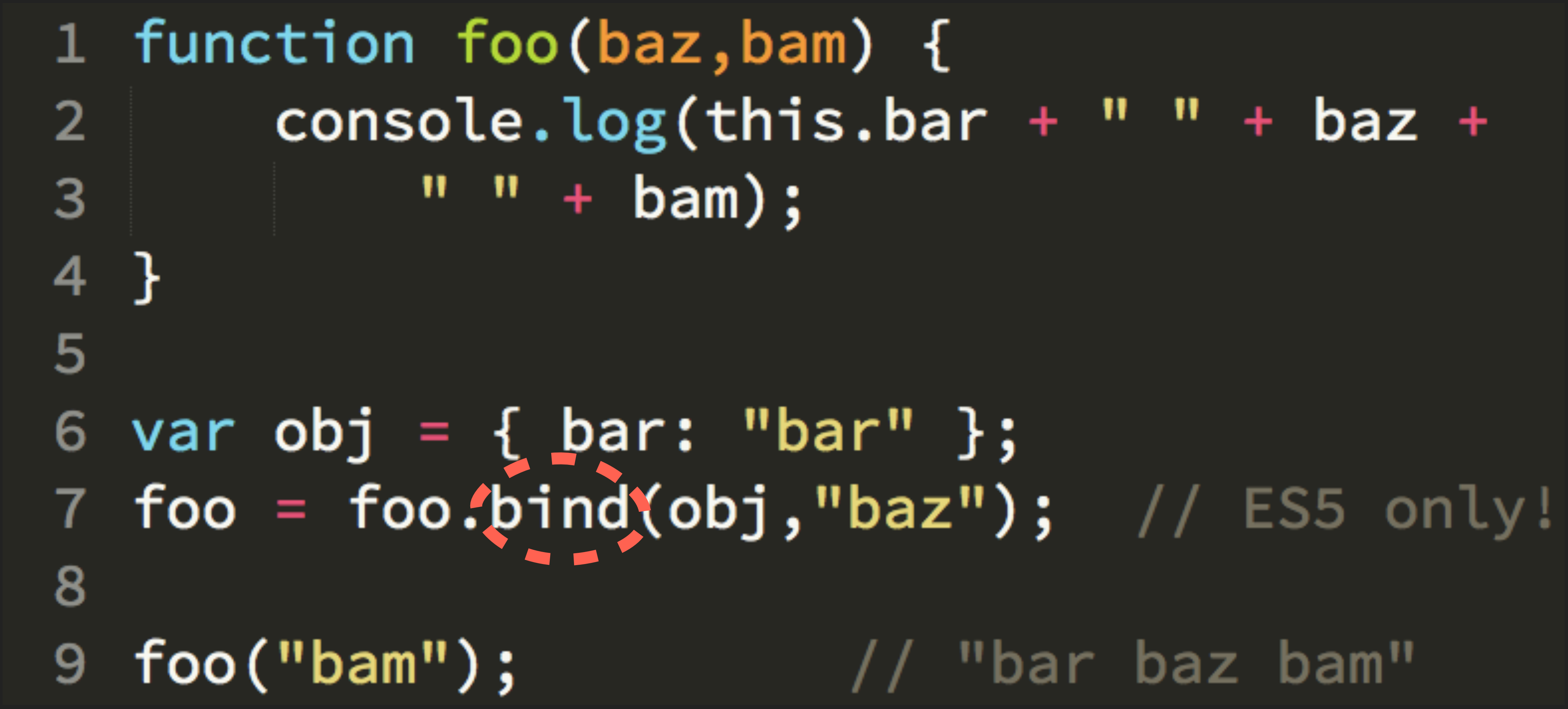
this: explicit binding



```javascript
function foo() {
    console.log(this.bar);
}

var obj = { bar: "bar" };
var obj2 = { bar: "bar2" };

var orig = foo;
foo = function(){ orig.call(obj); };

foo();          // "bar"
foo.call(obj2); // ???
```

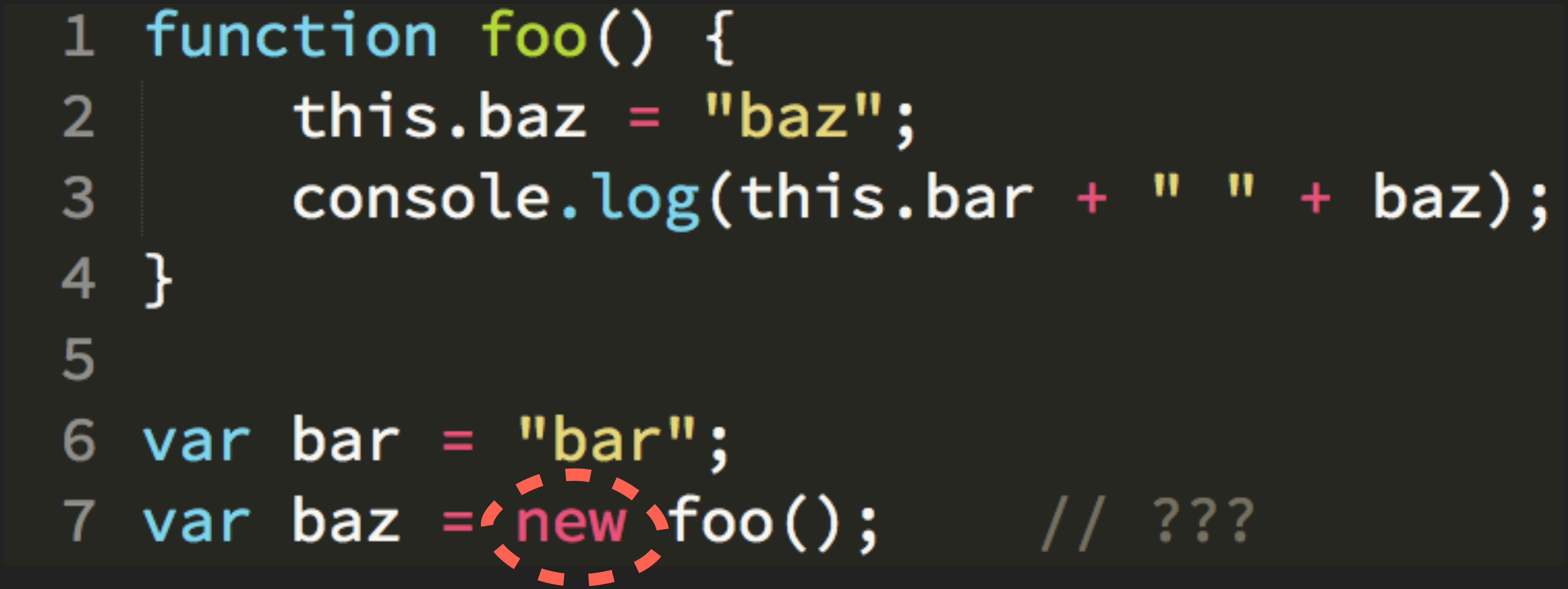
this: hard binding

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/bind



```javascript
function foo(baz,bam) {
    console.log(this.bar + " " + baz +
        " " + bam);
}

var obj = { bar: "bar" };
foo = foo.bind(obj,"baz");   // ES5 only!

foo("bam");               // "bar baz bam"
```

this: hard binding



```javascript
function foo() {
      this.baz = "baz";
      console.log(this.bar + " " + baz);
}

var bar = "bar";
var baz = new foo();      // ???
```

AKA: "constructor call"

this: new binding

1. Is the function called by new?
2. Is the function called by call() or apply()?

   Note: bind() effectively uses apply()

3. Is the function called on a context object?
4. DEFAULT: global object (except strict mode)

this: determination

Quiz

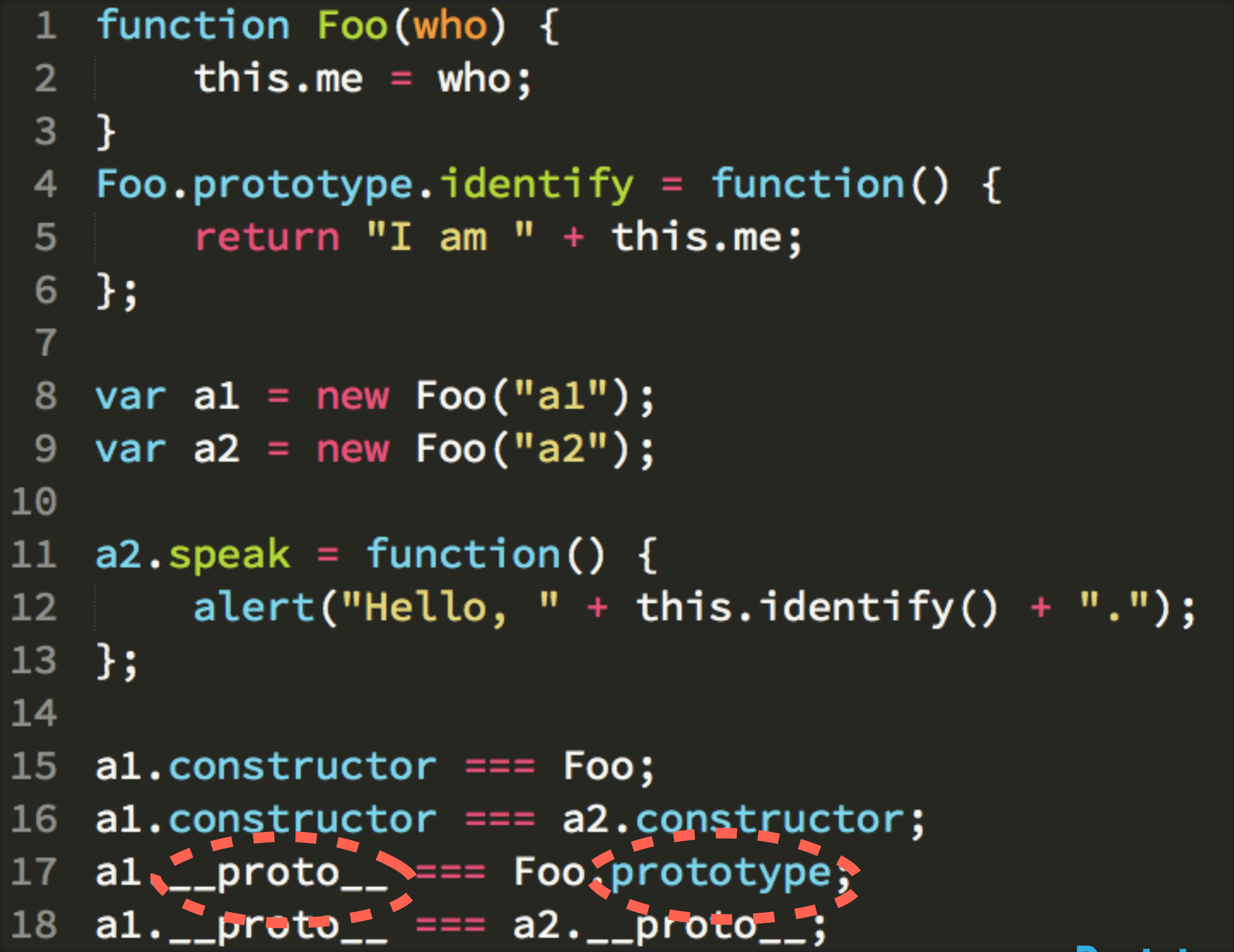
1. How do you “borrow” a function and implicitly set this?
2. How do you explicitly set this for the function call?
3. How can you lock a specific this to a function? Why do that? Why not?
4. How do you create a new this for the function call?

Prototypes

Objects are built by constructor calls

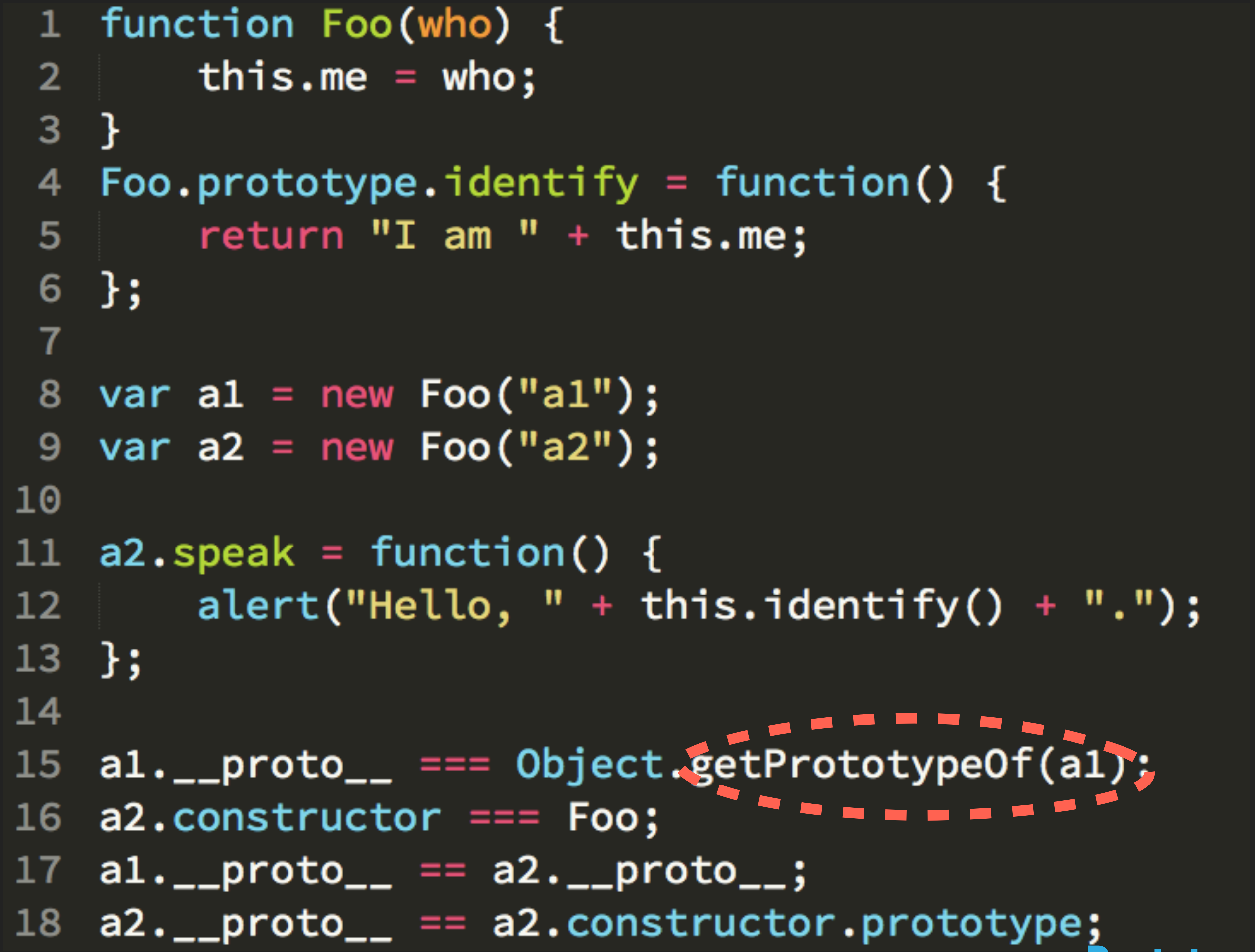
A constructor makes an object "~~based on~~" its own prototype

A constructor makes an object linked to its own prototype



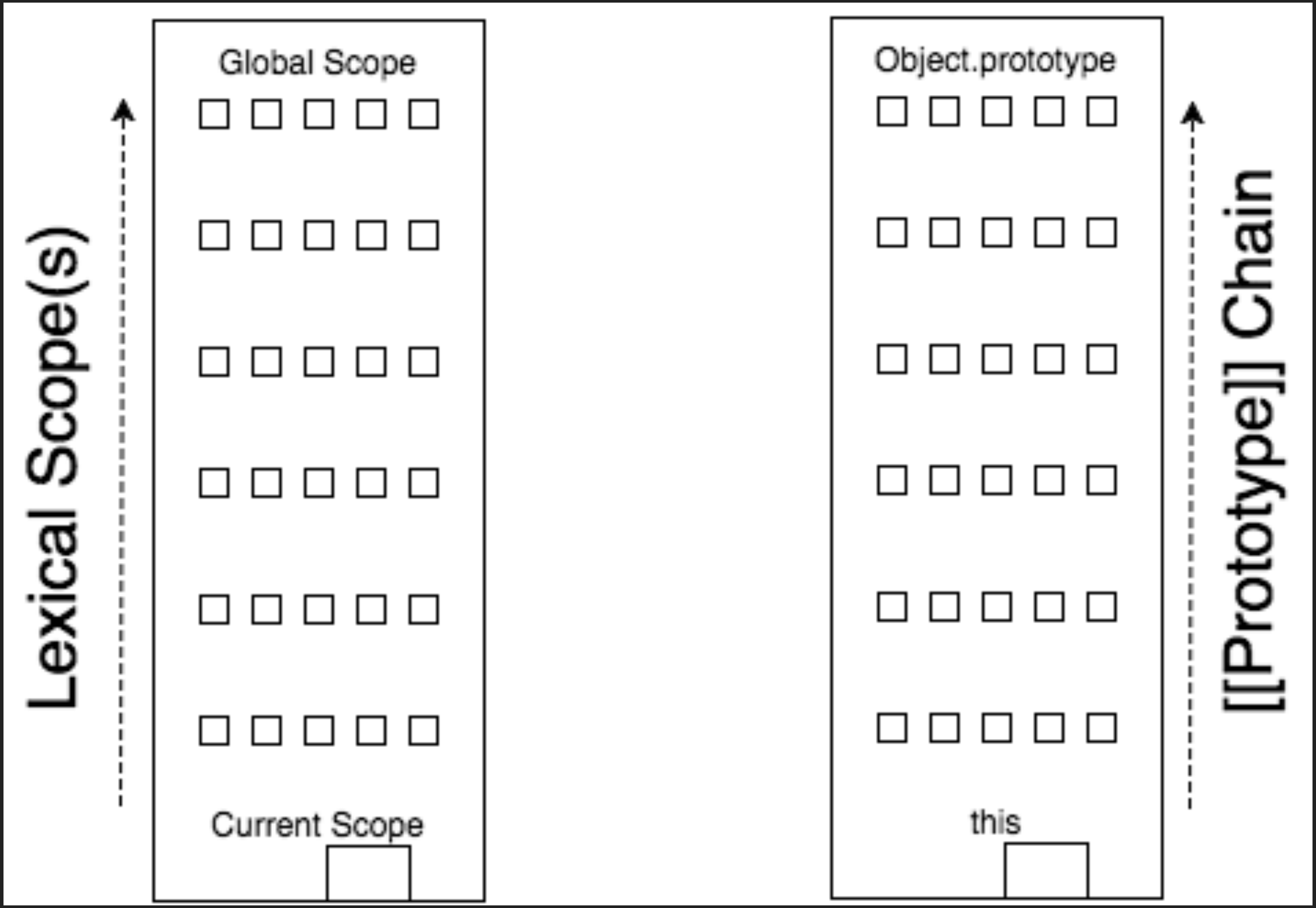
```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

var a1 = new Foo("a1");
var a2 = new Foo("a2");

a2.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

a1.constructor === Foo;
a1.constructor === a2.constructor;
a1.__proto__ === Foo.prototype;
a1.__proto__ === a2.__proto__;
```

Prototypes

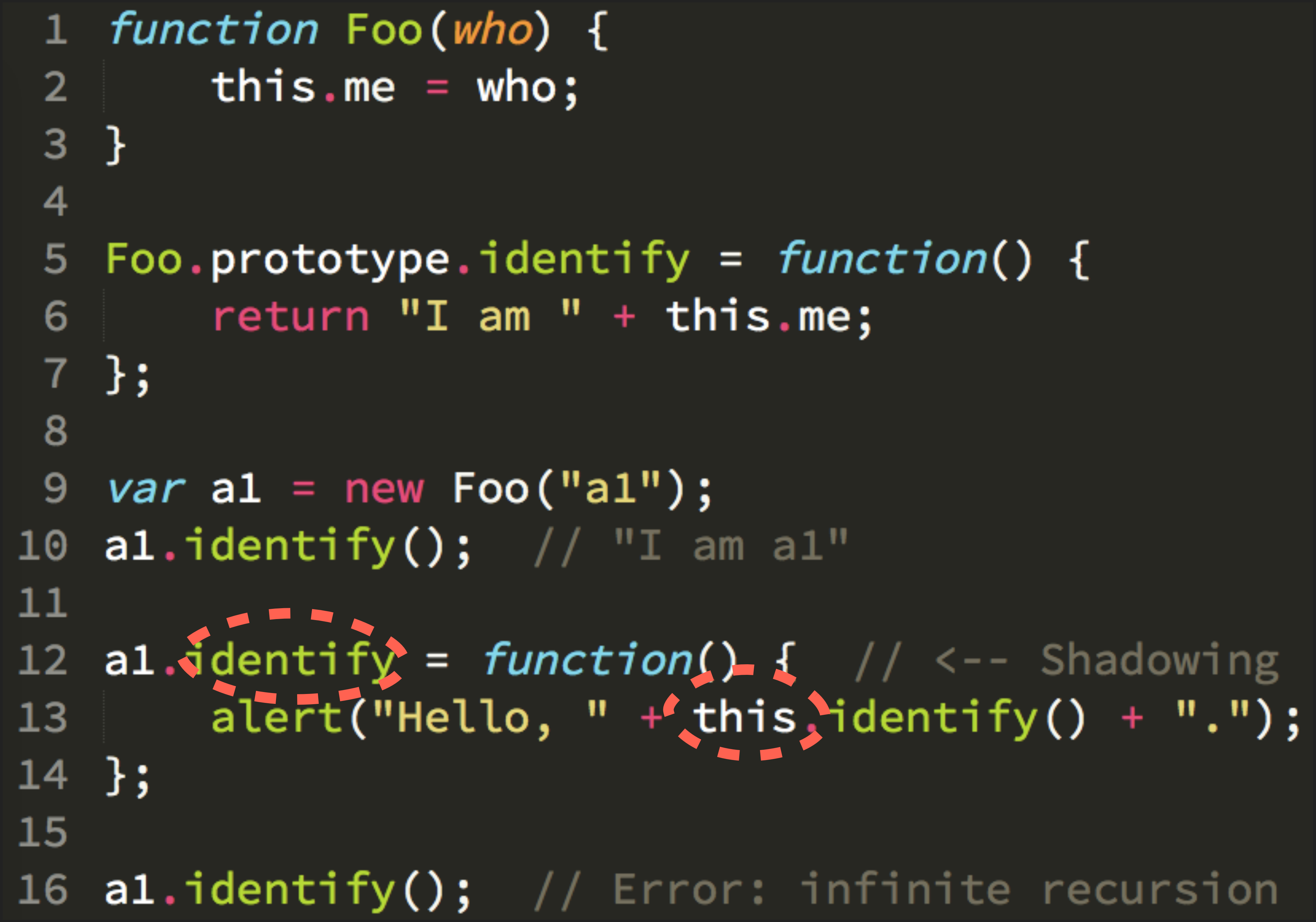


```javascript
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

var a1 = new Foo("a1");
var a2 = new Foo("a2");

a2.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

a1.__proto__ === Object.getPrototypeOf(a1);
a2.constructor === Foo;
a1.__proto__ == a2.__proto__;
a2.__proto__ == a2.constructor.prototype;
```

Prototypes


Lexical Scope(s)
Global Scope
Current Scope
Object.prototype
this
[[Prototype]] Chain

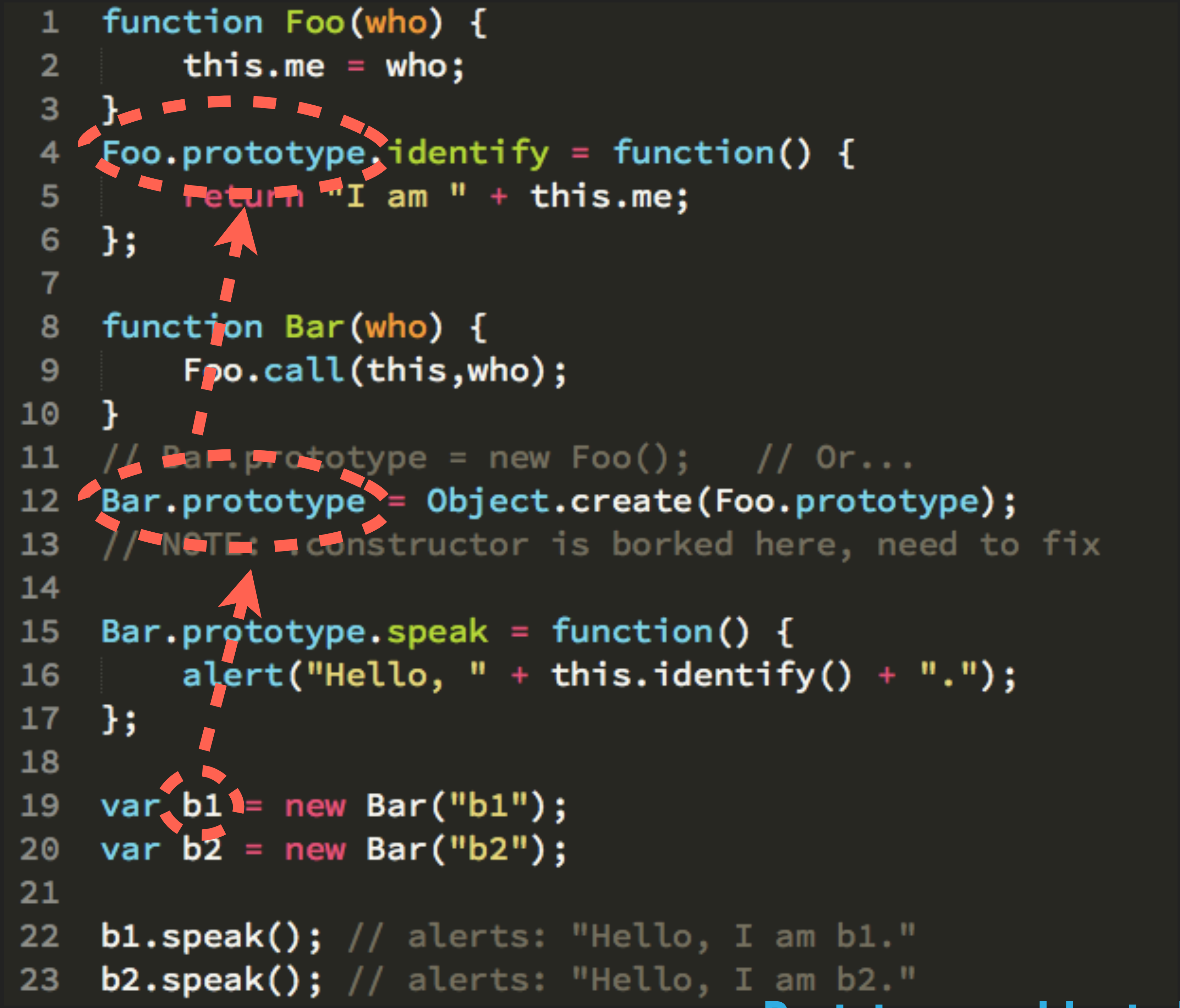
Prototypes



```javascript
function Foo(who) {
    this.me = who;
}

Foo.prototype.identify = function() {
    return "I am " + this.me;
};

var a1 = new Foo("a1");
a1.identify();  // "I am a1"

a1.identify = function() {  // <-- Shadowing
    alert("Hello, " + this.identify() + ".");
};

a1.identify();  // Error: infinite recursion
```

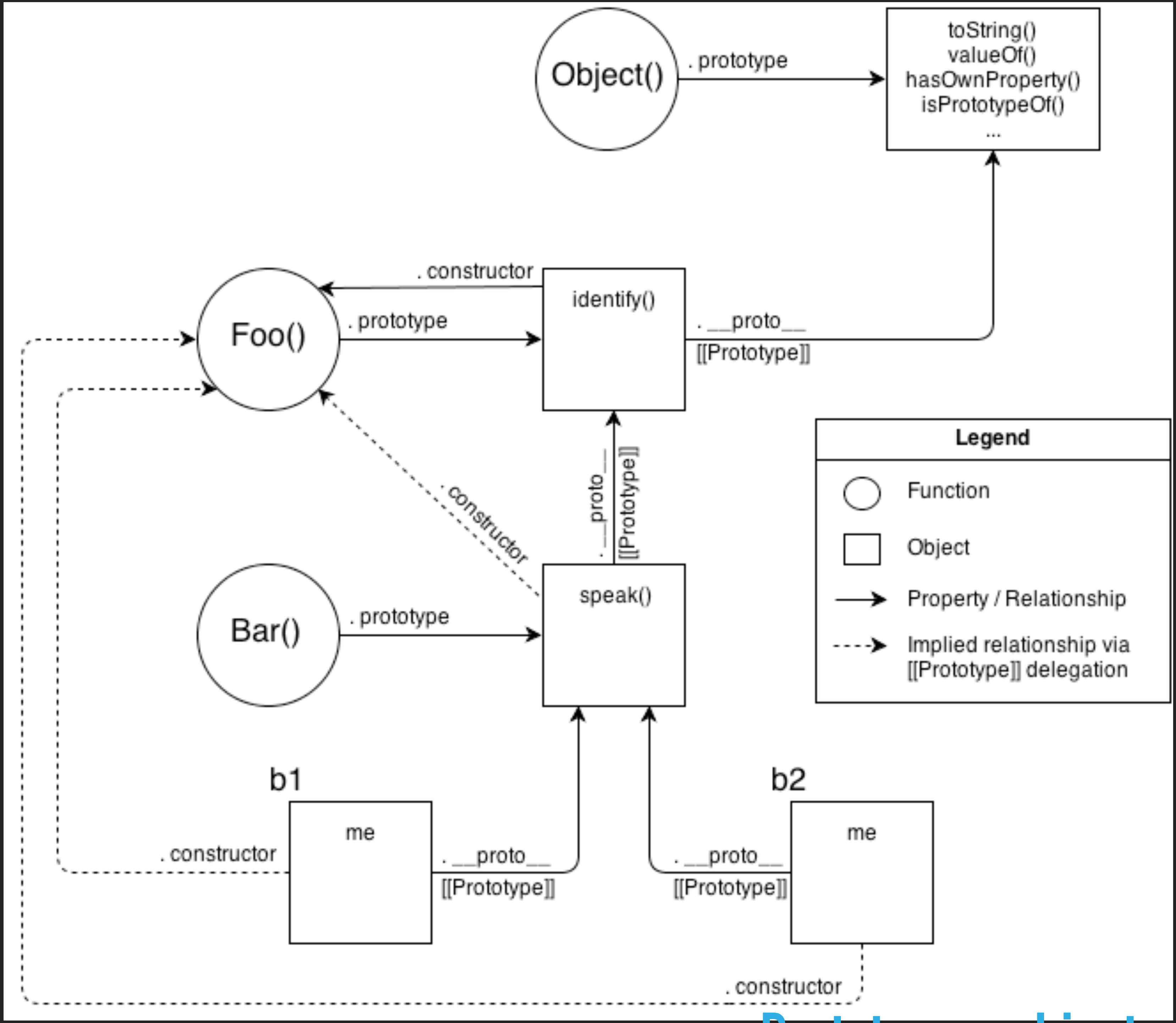
Prototypes: shadowing

“Inheritance”

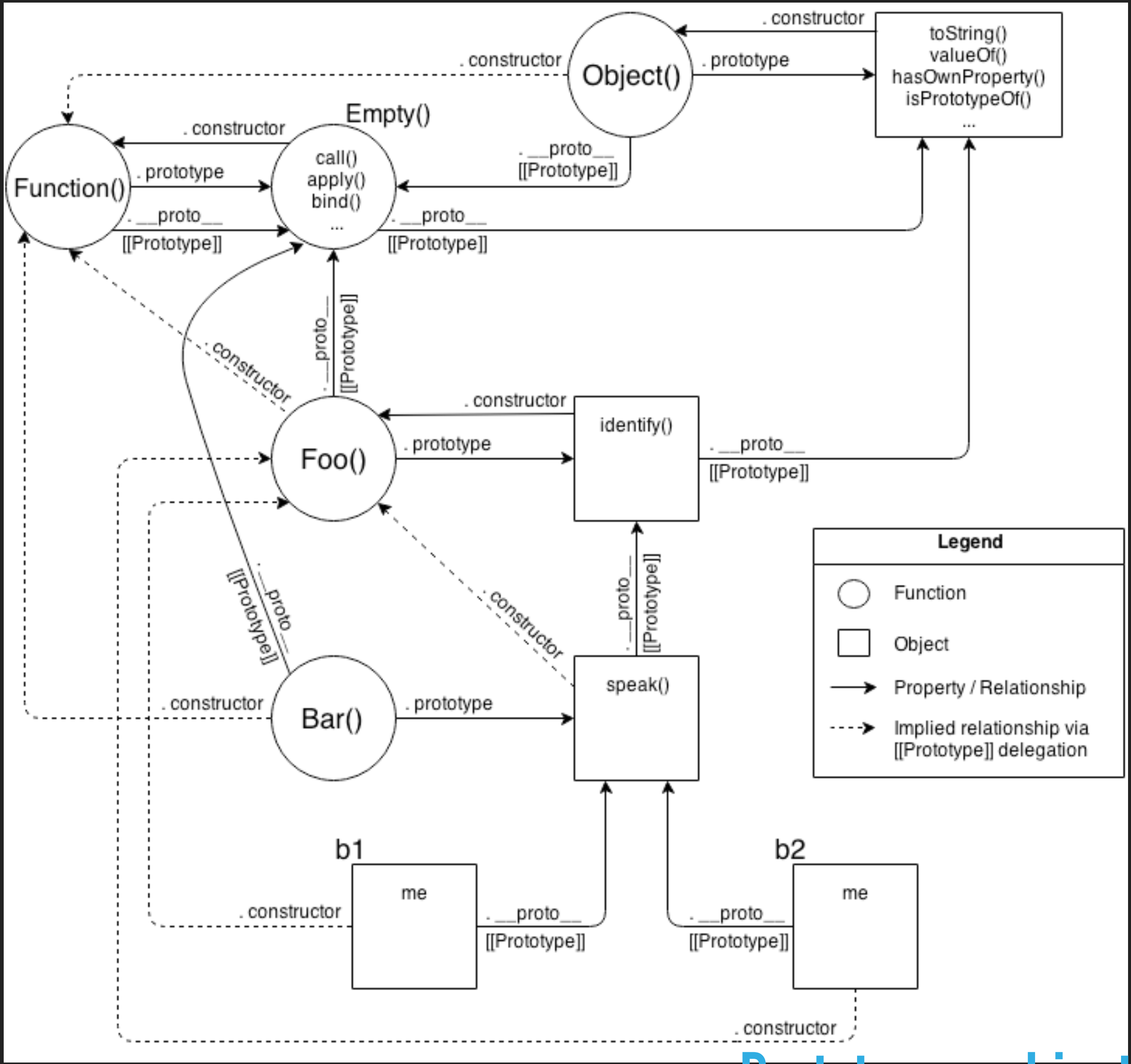


```javascript
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call(this,who);
}
// Bar.prototype = new Foo();    // Or...
Bar.prototype = Object.create(Foo.prototype);
// NOTE: .constructor is borked here, need to fix

Bar.prototype.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

var b1 = new Bar("b1");
var b2 = new Bar("b2");

b1.speak(); // alerts: "Hello, I am b1."
b2.speak(); // alerts: "Hello, I am b2."
```

Prototypes: objects linked



Prototypes: objects linked



Prototypes: objects linked

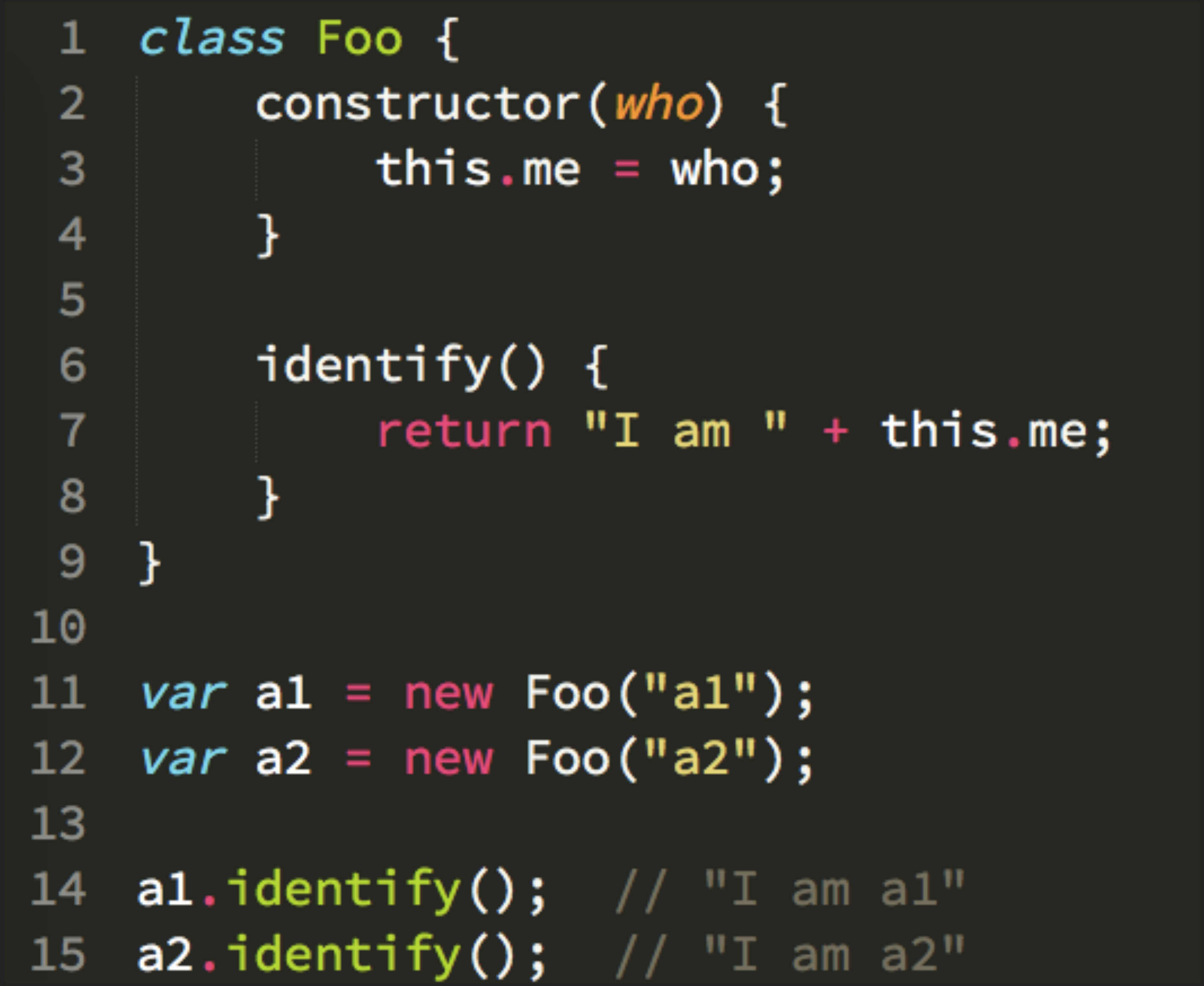
Quiz

1. What is a constructor call?
2. What is [[Prototype]] and where does it come from?
3. How does [[Prototype]] affect the behavior of an object?
4. How do we find out where an object's [[Prototype]] points to (3 ways)?

(exercise #7: 10min)

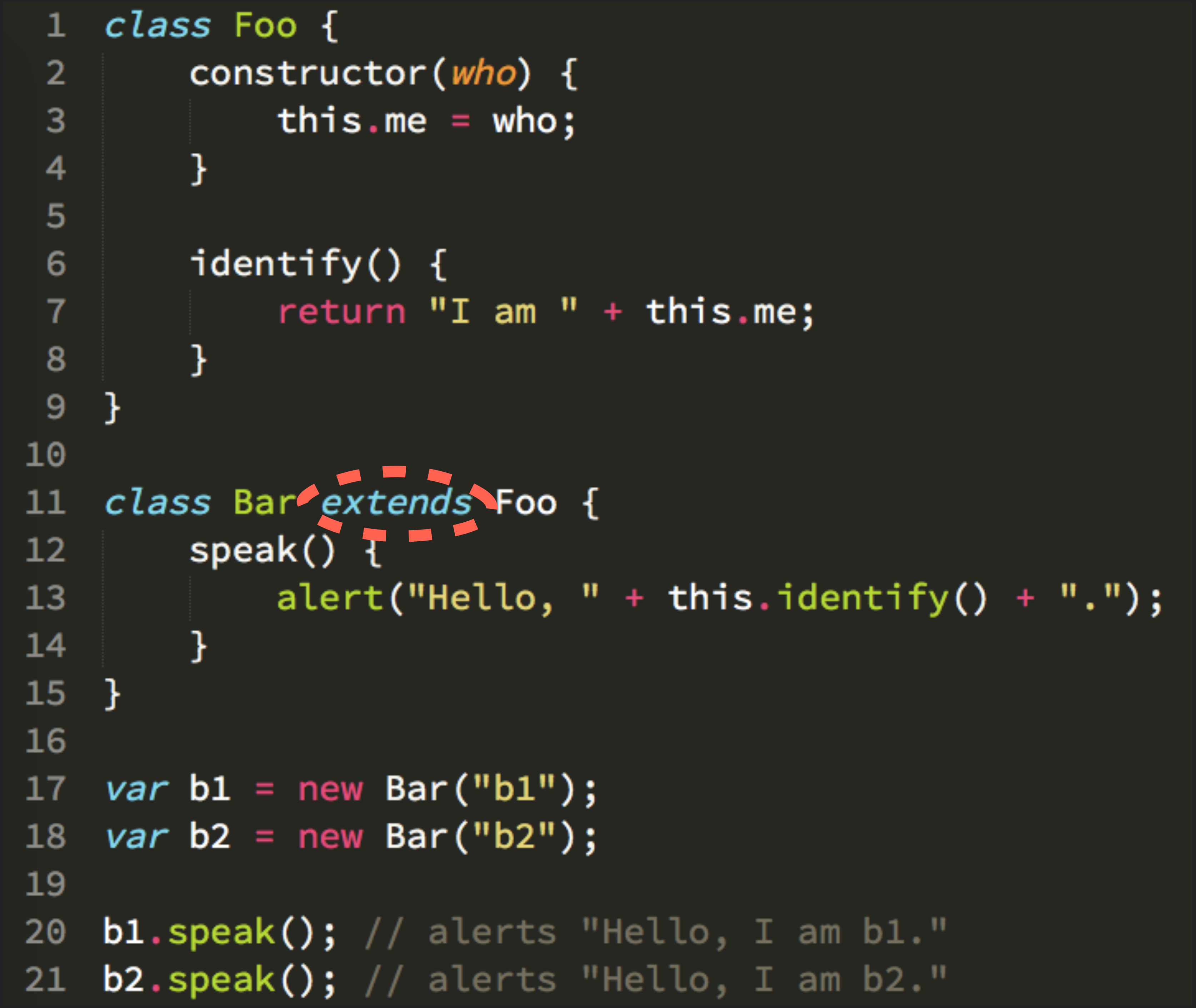
class { }

ES6



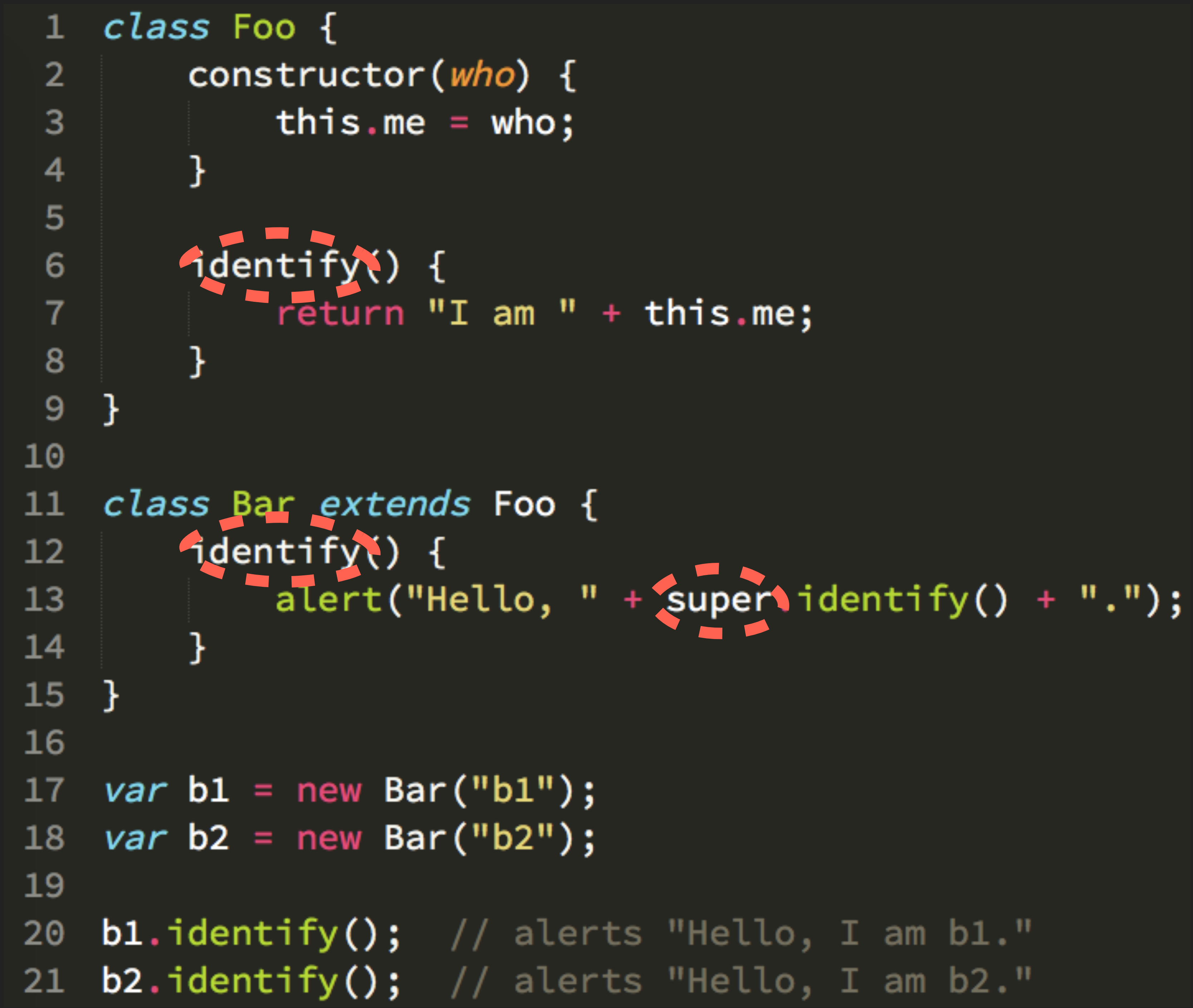
```
class Foo {
	constructor(who) {
		this.me = who;
	}

	identify() {
		return "I am " + this.me;
	}
}

var a1 = new Foo("a1");
var a2 = new Foo("a2");

a1.identify();  // "I am a1"
a2.identify();  // "I am a2"
```

ES6 class



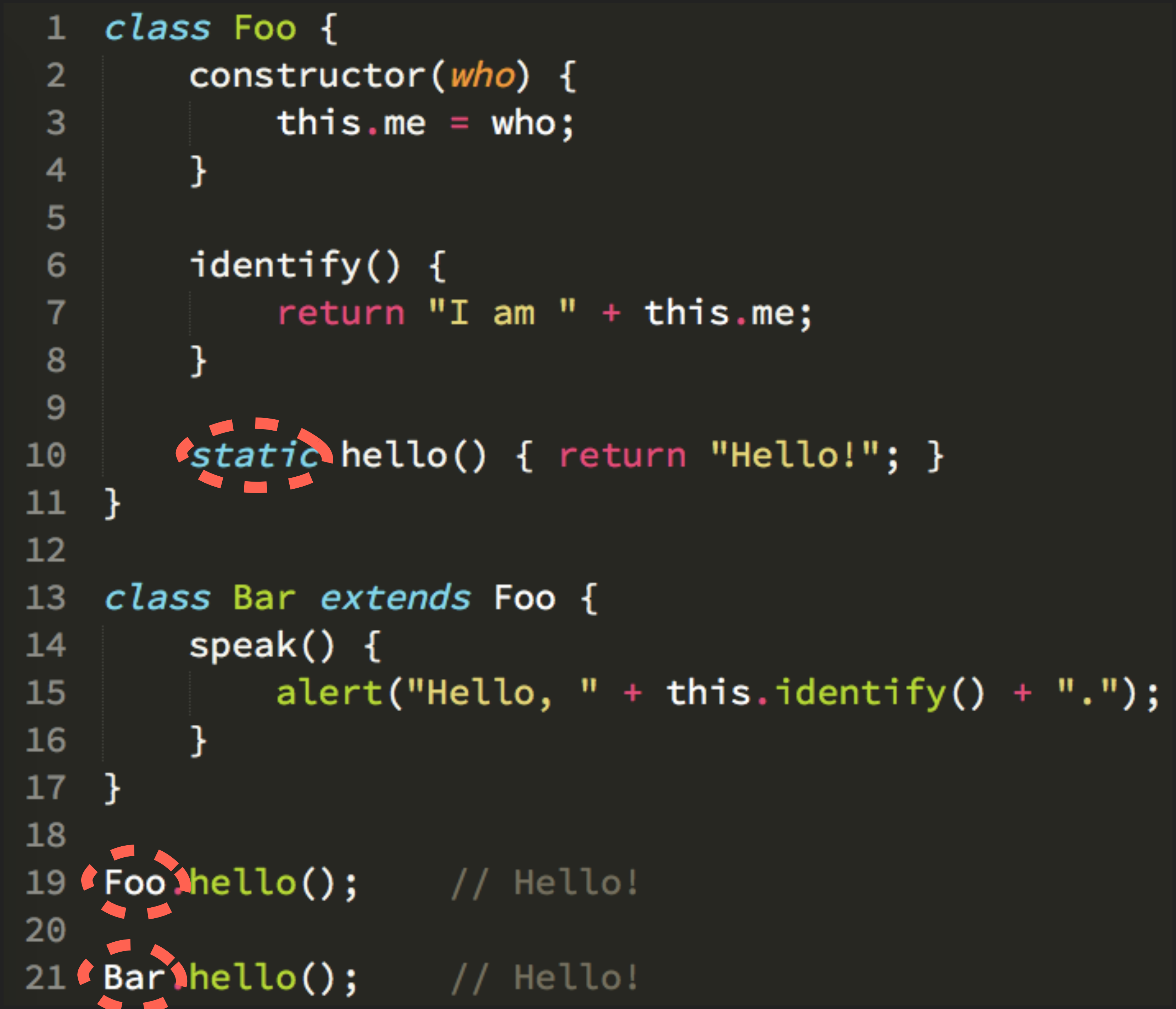
```
class Foo {
    constructor(who) {
        this.me = who;
    }

    identify() {
        return "I am " + this.me;
    }
}

class Bar extends Foo {
    speak() {
        alert("Hello, " + this.identify() + ".");
    }
}

var b1 = new Bar("b1");
var b2 = new Bar("b2");

b1.speak(); // alerts "Hello, I am b1."
b2.speak(); // alerts "Hello, I am b2."
```

ES6 class: extends (inheritance)



```
class Foo {
    constructor(who) {
        this.me = who;
    }

    identify() {
        return "I am " + this.me;
    }
}

class Bar extends Foo {
    identify() {
        alert("Hello, " + super.identify() + ".");
    }
}

var b1 = new Bar("b1");
var b2 = new Bar("b2");

b1.identify();  // alerts "Hello, I am b1."
b2.identify();  // alerts "Hello, I am b2."
```

ES6 class: super (relative polymorphism)



```
class Foo {
    constructor(who) {
        this.me = who;
    }

    identify() {
        return "I am " + this.me;
    }

    static hello() { return "Hello!"; }
}

class Bar extends Foo {
    speak() {
        alert("Hello, " + this.identify() + ".");
    }
}

Foo.hello();     // Hello!

Bar.hello();     // Hello!
```

ES6 class: static (constructor inheritance)

Clearing Up Inheritance

$$\begin{tikzcd}
\text{Foo} \arrow[r] \arrow[dr] \arrow[dd] & \text{a1} \\
 & \text{a2} \\
\text{Bar} \arrow[r] \arrow[dr] & \text{b1} \\
 & \text{b2}
\end{tikzcd}$$

OO: classical inheritance

$$\begin{tikzcd}
\text{Foo.prototype} & \text{a1} \arrow[l] \\
& \text{a2} \arrow[ul] \\
\text{Bar.prototype} \arrow[uu] & \text{b1} \arrow[l] \\
& \text{b2} \arrow[ul]
\end{tikzcd}$$

(another design pattern)

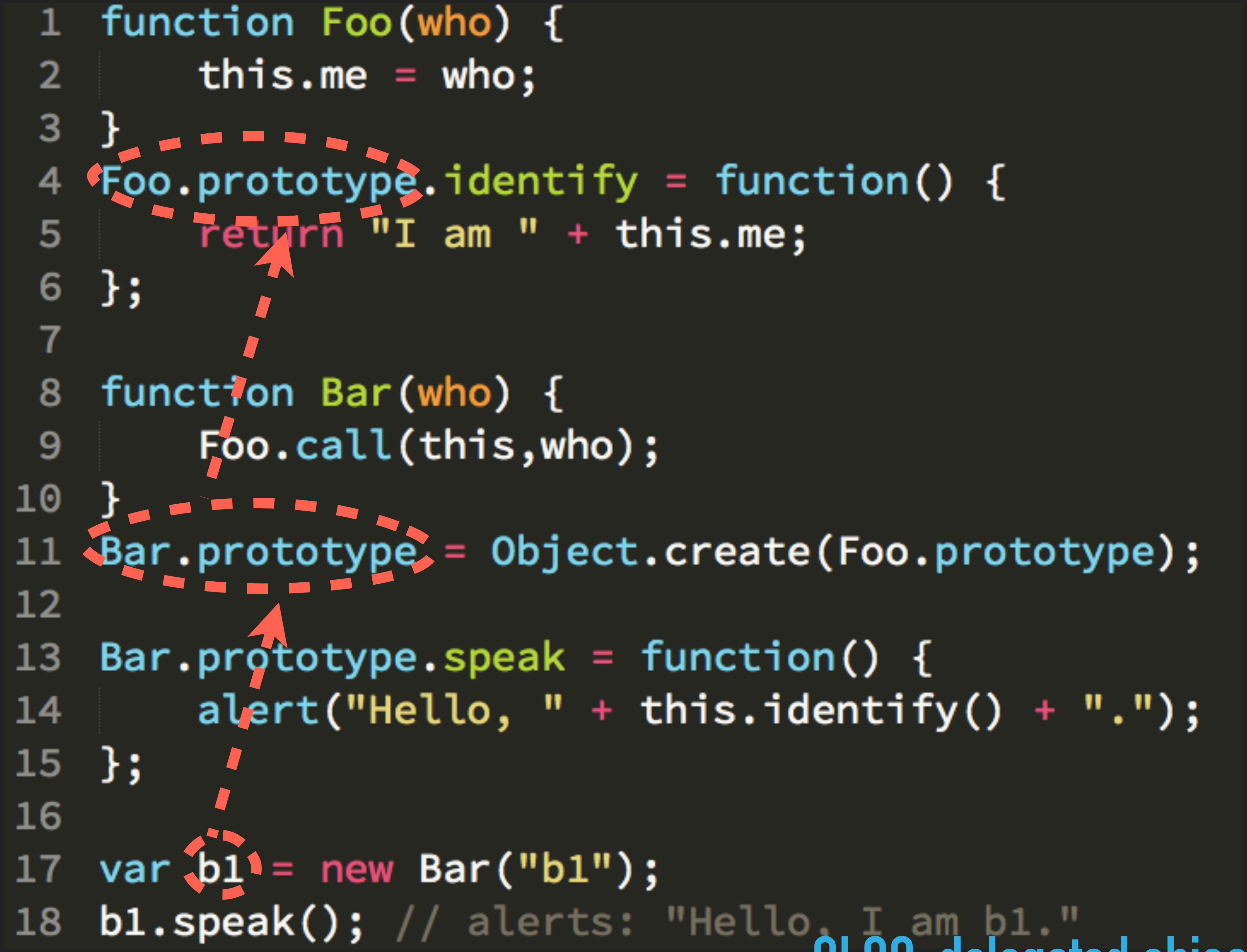
OO: “prototypal inheritance”

JavaScript ~~“Inheritance”~~
“Behavior Delegation”

Let's Simplify!

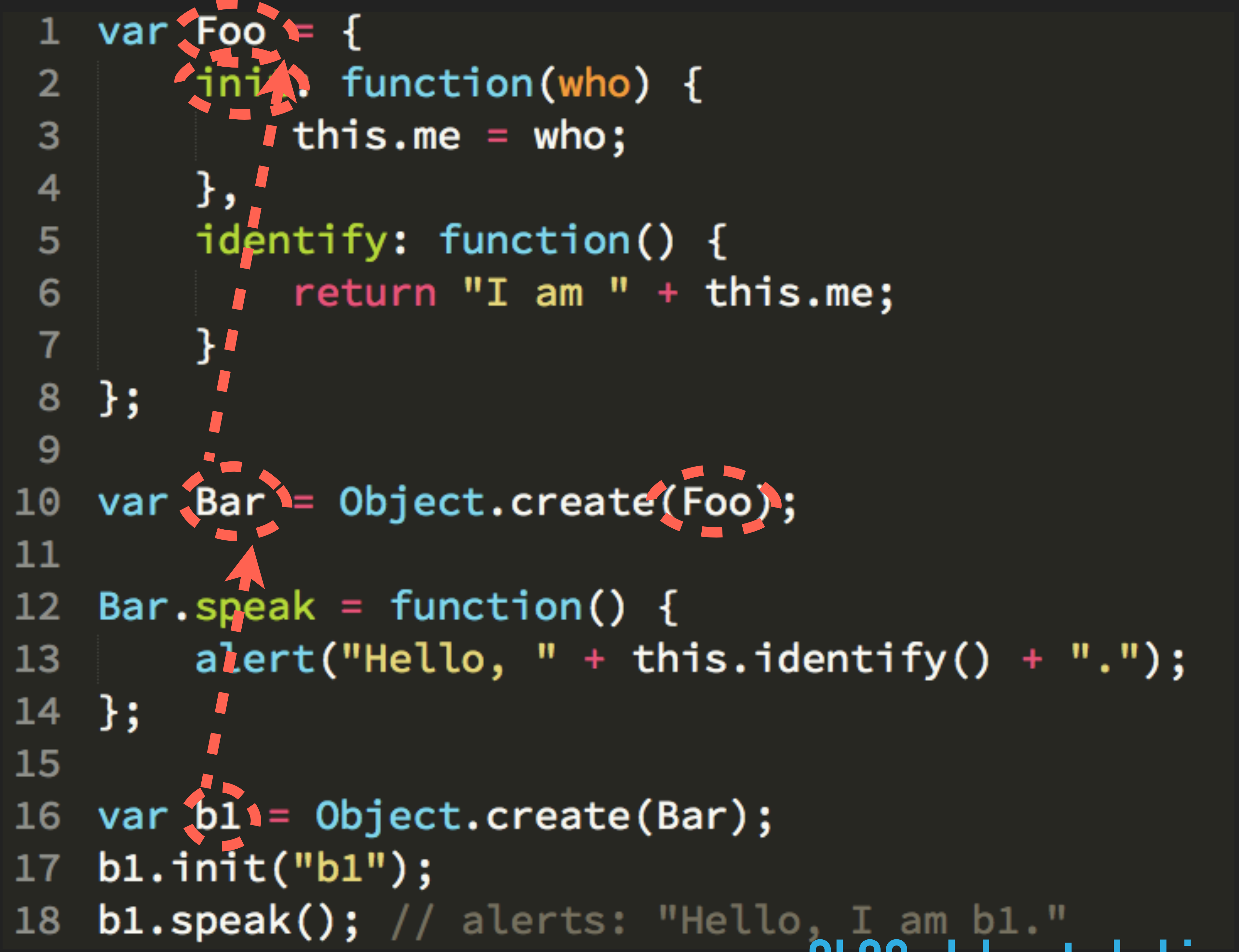
OLOO:

Objects Linked to Other Objects



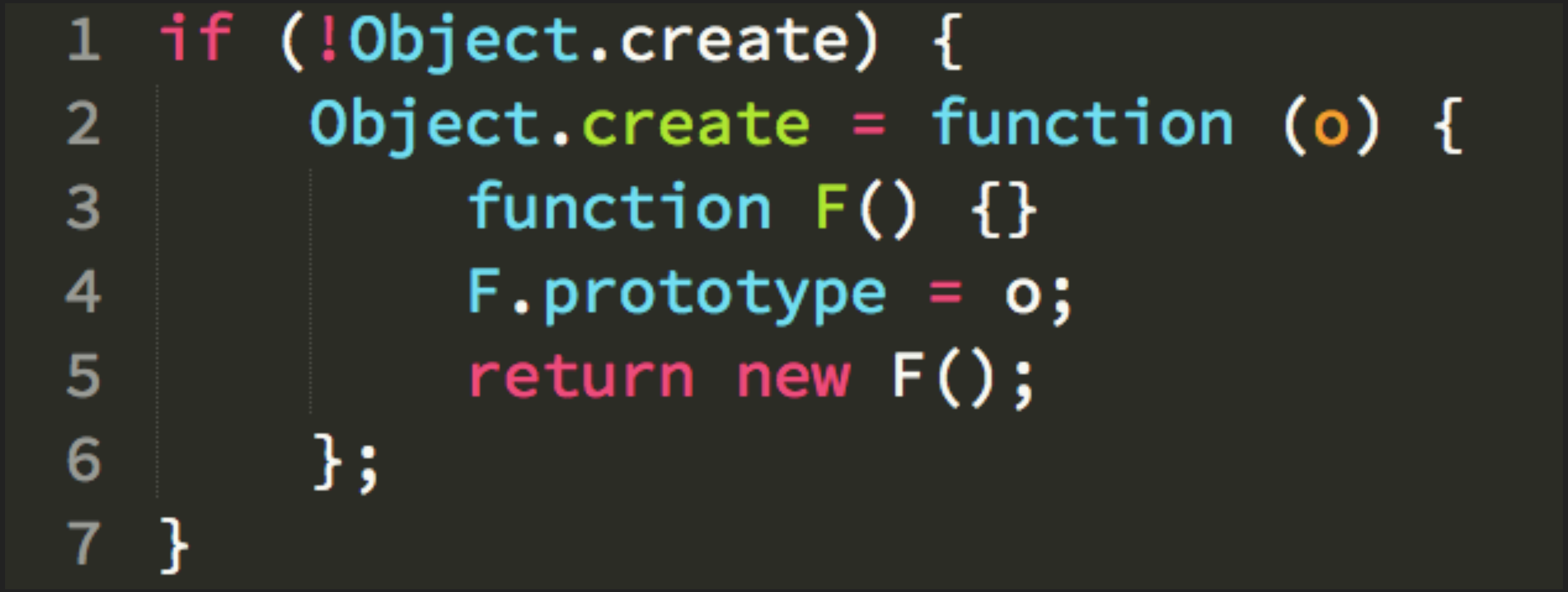
```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call(this,who);
}
Bar.prototype = Object.create(Foo.prototype);

Bar.prototype.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

var b1 = new Bar("b1");
b1.speak(); // alerts: "Hello, I am b1."
```

OLOO: delegated objects



```
var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create(Foo);

Bar.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

var b1 = Object.create(Bar);
b1.init("b1");
b1.speak(); // alerts: "Hello, I am b1."
```

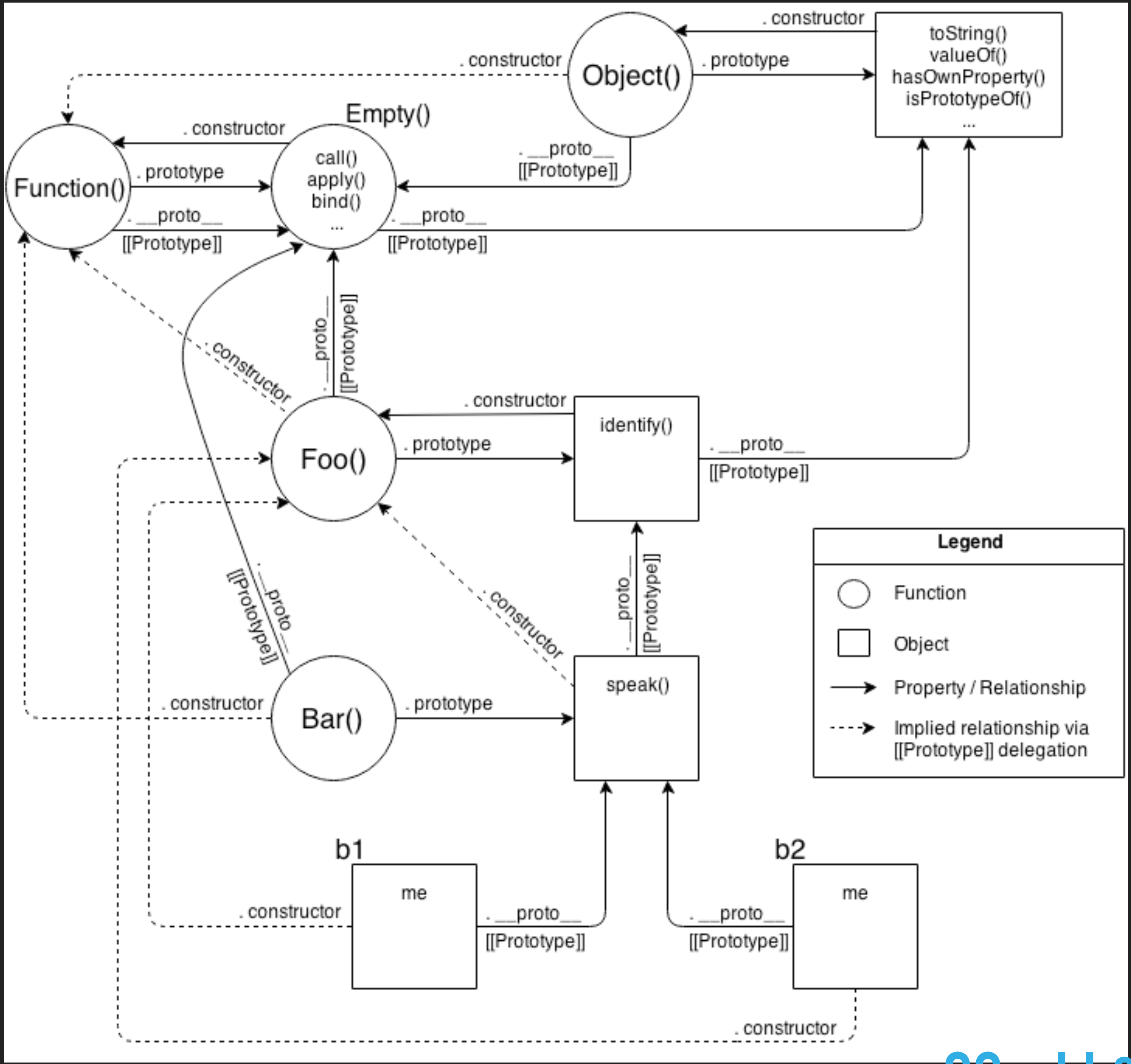
OLOO: delegated objects



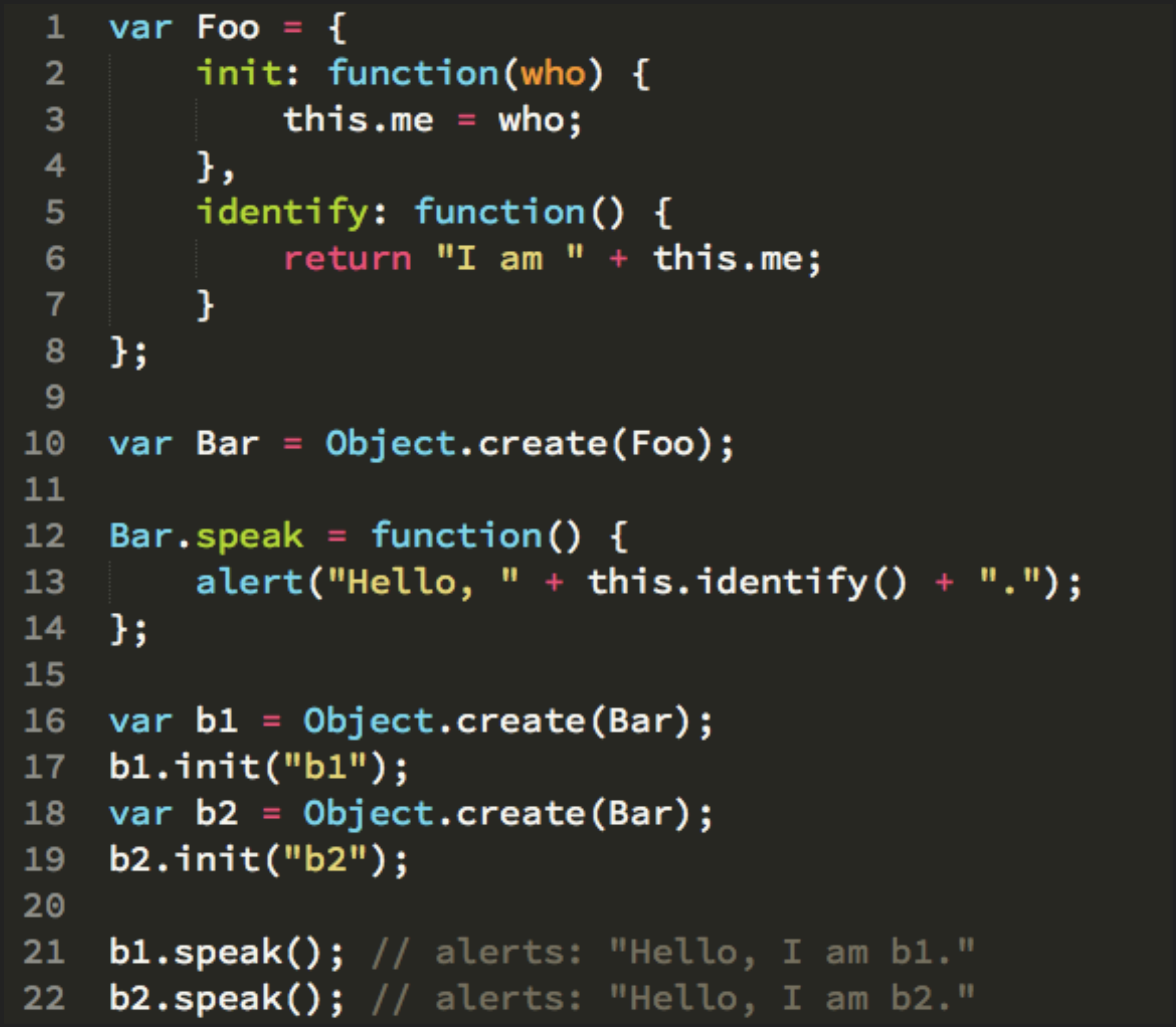
```
if (!Object.create) {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

OLOO: Object.create()

Mental Models

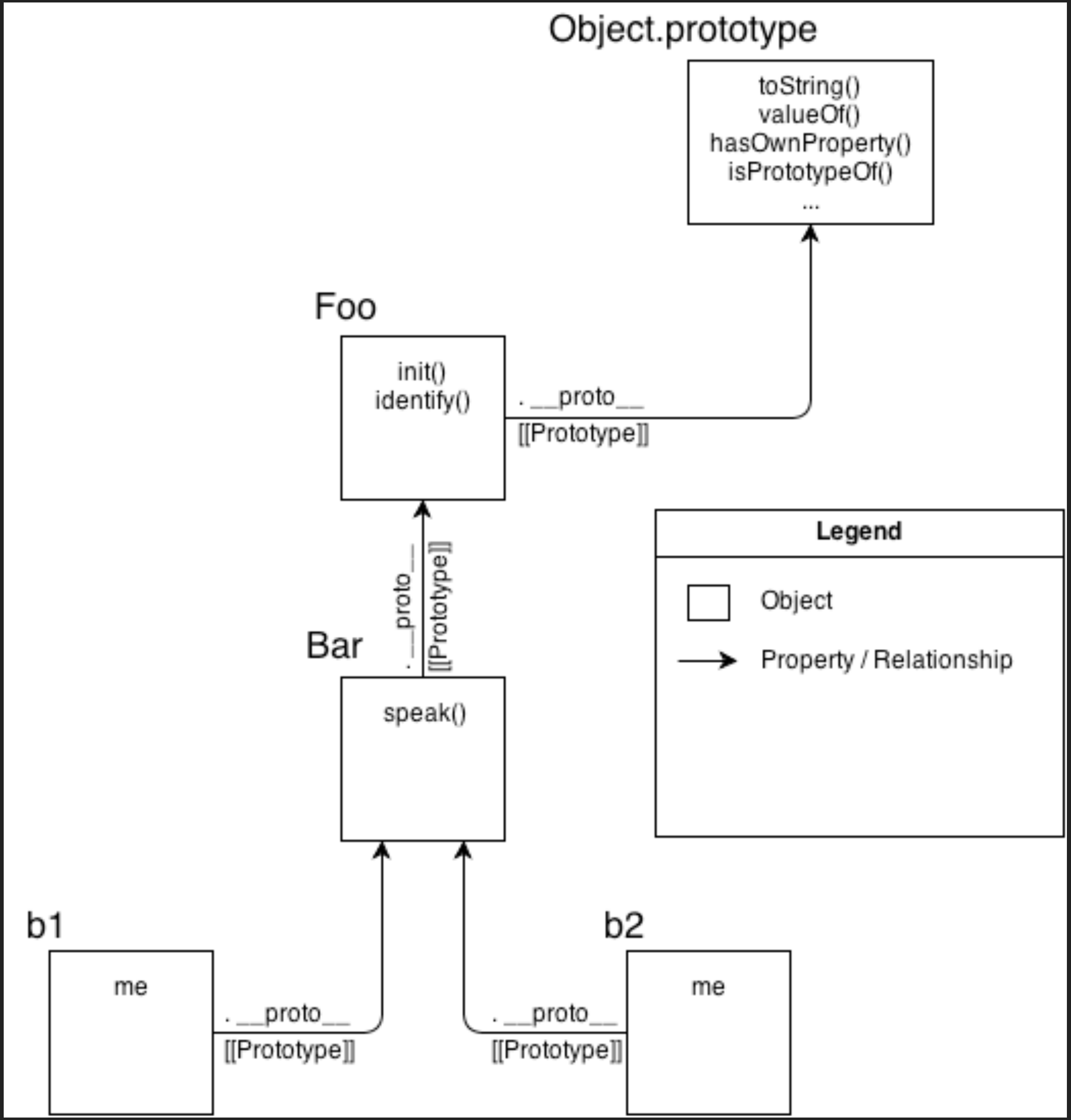


OO: old & busted



```javascript
var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create(Foo);

Bar.speak = function() {
    alert("Hello, " + this.identify() + ".");
};

var b1 = Object.create(Bar);
b1.init("b1");
var b2 = Object.create(Bar);
b2.init("b2");

b1.speak(); // alerts: "Hello, I am b1."
b2.speak(); // alerts: "Hello, I am b2."
```

OLOO: new hotness

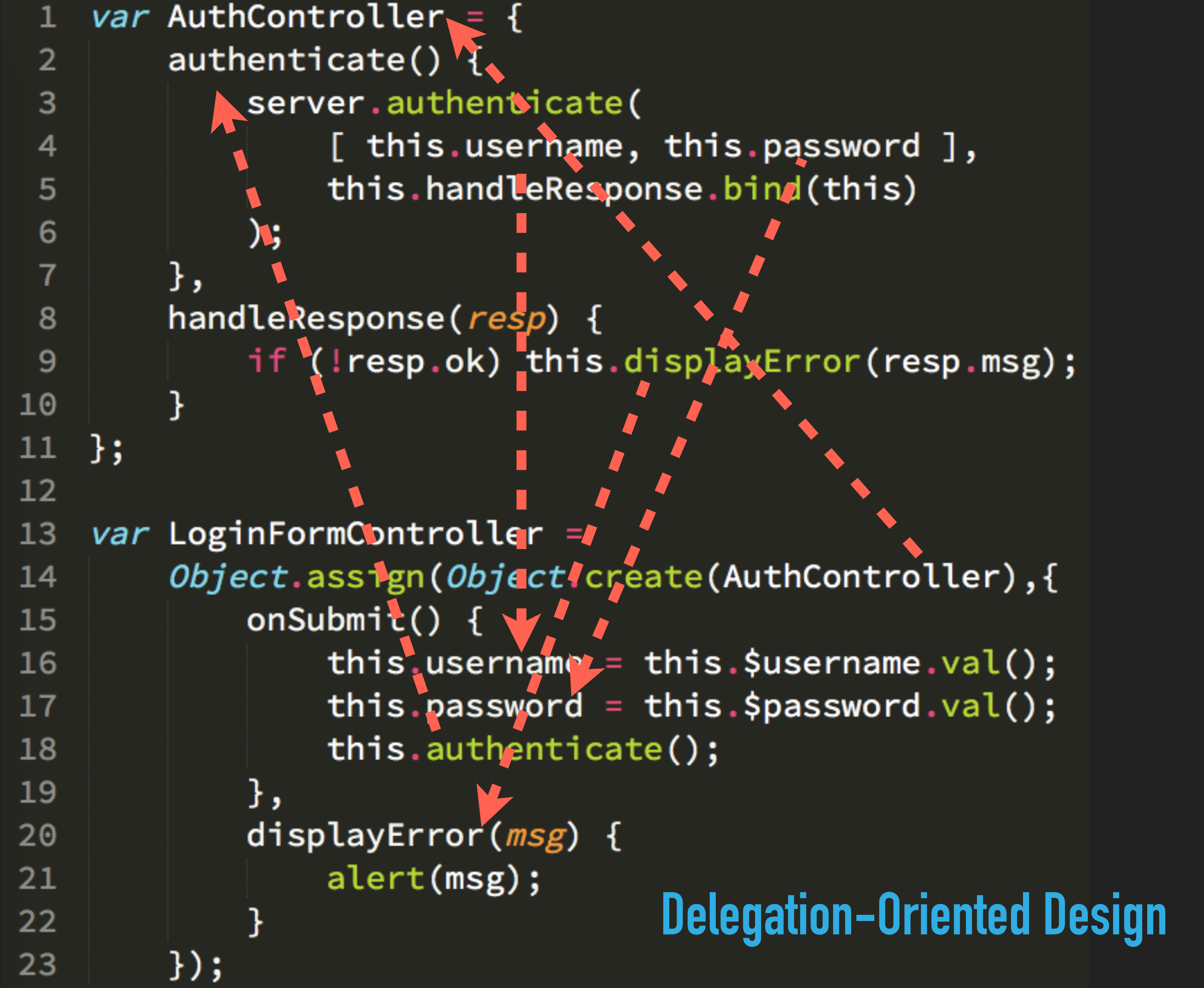


OLOO: new hotness

Delegation: Design Pattern

~~Parent–Child~~

Peer–Peer



```
var AuthController = {
    authenticate() {
        server.authenticate(
            [ this.username, this.password ],
            this.handleResponse.bind(this)
        );
    },
    handleResponse(resp) {
        if (!resp.ok) this.displayError(resp.msg);
    }
};

var LoginFormController =
    Object.assign(Object.create(AuthController),{
        onSubmit() {
            this.username = this.$username.val();
            this.password = this.$password.val();
            this.authenticate();
        },
        displayError(msg) {
            alert(msg);
        }
});
```

Delegation-Oriented Design

Quiz

1. How is JavaScript's [[Prototype]] chain not like traditional/classical inheritance?
2. What does [[Prototype]] "delegation" mean and how does it describe object linking in JS?
3. What are the benefits of the "behavior delegation" design pattern? What are the tradeoffs of using [[Prototype]]?

(exercise #8: 10min)

THANKS!!!!

KYLE SIMPSON GETIFY@GMAIL.COM

DEEP JS FOUNDATIONS