Dmitry Soshnikov in ECMAScript | 2010-02-28

ECMA-262-3 in detail. Chapter 6. Closures.

Read this article in: Russian, Chinese, French.

Introduction

In this article we will talk about one of the most discussed topics related to JavaScript — about . The topic, as a matter of fact, is not new and was discussed many times. However we will try to discuss and understand it more from a theoretical viewpoint, and also will look at how closures are implemented in ECMAScript.

Two previous chapters devoted to scope chain and variable object might be worth to read first, since in this chapter we will use material discussed earlier.

General theory

Before the discussion of closures directly in ECMAScript, it is necessary to specify a number of definitions from the general theory of functional programming.

As is known, in functional languages (and ECMAScript supports this paradigm and stylistics), functions are , i.e. they can be , passed as to other functions, from functions etc. Such functions have special names and structure.

Definitions

> A — is an argument which value is a function.

Example:

```
function exampleFunc(funArg) {
  funArg();
}

exampleFunc(function () {
  console.log('funArg');
});
```

The actual parameter related with the “funarg” in this case is the anonymous function to the `exampleFunc` function.

> In turn, a function which another function as the argument is called a .

Another name of a HOF is a or, closer to mathematics — an . In the example above, `exampleFunc` function is a .

As it was noted, a function can be not only passed as an argument, but also as a from another function.

> A function which another function is called (or a function).

```
(function functionValued() {
  return function () {
    console.log('returned function is called');
  };
})()();
```

> Functions which can participate as normal data, i.e. be passed as arguments, receive functional arguments or be returned as functional values, are called .

In ECMAScript functions are .

A function which receives itself as an argument, is called an :

```
(function selfApplicative(funArg) {

  if (funArg && funArg === selfApplicative) {
    console.log('self-applicative');
    return;
  }

  selfApplicative(selfApplicative);

})();
```

A function which returns itself is called an function. Sometimes, the name is used in a literature:

```
(function selfReplicative() {
  return selfReplicative;
})();
```

> One of interesting patterns of *self-replicative* functions is a *declarative form* of working with a single argument of a collection instead of accepting the collection itself:

```
// imperative function
// which accepts collection

function registerModes(modes) {
  modes.forEach(registerMode, modes);
}

// usage
registerModes(['roster', 'accounts', 'groups']);

// declarative form using
// self-replicating function

function modes(mode) {
  registerMode(mode); // register one mode
  return modes; // and return the function itself
}

// usage: we just *declare* modes

modes
```

```
22      ('roster')
23      ('accounts')
24      ('groups')
```

However, in practice working with the collection itself can be more efficient and intuitive.

Local variables which are defined in the passed functional argument are of course accessible at of this function, since the variable object (environment) which stores the data of the context is created every time on entering the context:

```
1  function testFn(funArg) {
2
3    // activation of the funarg, local
4    // variable "localVar" is available
5
6    funArg(10); // 20
7    funArg(20); // 30
8
9  }
10
11 testFn(function (arg) {
12
13   let localVar = 10;
14   console.log(arg + localVar);
15
16 });
```

However, as we know from the chapter 4, functions in ECMAScript can be with and from . With this feature so-called a is related.

Funarg problem

In stack-oriented programming languages local variables of functions are stored on a which is with these variables and function arguments every time when the function is .

On from a function the variables are from the stack. This model is a for using functions as (i.e. returning them from parent functions). Mostly this problem appears when a function uses .

> A is a variable which is used by a function, but is neither a parameter, nor a local variable of the function.

Example:

```
function testFn() {

  let localVar = 10;

  function innerFn(innerParam) {
    console.log(innerParam + localVar);
  }

  return innerFn;
}

let someFn = testFn();
someFn(20); // 30
```

In this example `localVar` variable is for the `innerFn` function.

If this system would use model for storing local variables, it would mean that on return from `testFn` function all its local variables would be from the stack. And this would cause an error at `innerFn` function activation from the .

Moreover, in this particular case, in a stack-oriented implementation, returning of the `innerFn` function would not be possible at all, since `innerFn` is also for `testFn` and therefore is also removed on returning from the `testFn` .

Another problem of functional objects is related with passing a function as an argument in a system with dynamic scope implementation.

Example (pseudo-code):

```
let z = 10;

function foo() {
  console.log(z);
}

foo(); // 10 - with using both static and dynamic scope

(function () {

  let z = 20;
  // NOTE: always 10 in JS!
  foo(); // 10 - with static scope, 20 - with dynamic scope

})();

// the same with passing foo
// as an arguments

(function (funArg) {

  let z = 30;
  funArg(); // 10 - with static scope, 30 - with dynamic scope

})(foo);
```

We see that in systems with dynamic scope, is managed with a . Thus, free variables are searched in the of the — in the place where the function is , but not in the which is saved at .

And this can lead to ambiguity. Thus, even if `z` exists (in contrast with the previous example where local variables would be removed from a stack), there is a question: which value of `z` (i.e. `z` from which , from which) should be used in various calls of `foo` function?

> The described cases are of the — depending on whether we deal with the returned from a function , or with the passed to the function .

For solving this problem (and its subtypes) the concept of a was proposed.

Closure

> A is a combination of a and of a context in which this code block is .

Let's see an example in a pseudo-code:

```
let x = 20;

function foo() {
  console.log(x); // free variable "x" == 20
}

// Closure for foo
let fooClosure = {
  code: foo // reference to function
  environment: {x: 20}, // context for searching free variables
};
```

In the example above, `fooClosure` is a pseudo-code since in ECMAScript `foo` function already captures the lexical environment of the context where it is created.

The word "lexical" is often implicitly assumed and omitted — in this case it focuses attention that a closure saves its parent variables in the of the , that is — where the function is . At next activations of the

function, free variables are searched in this saved context, that we can see in examples above where variable `z` always should be resolved as `10` in ECMAScript.

In definition we used a generalized concept — "the code block", however usually the term "function" is used. Though, not in all implementations closures are associated only with functions: for example, in Ruby programming language, a closure can be presented as a procedure object, a lambda-expression or a code block.

Regarding implementations, for storing local variables after the context is destroyed, the implementation (because it contradicts the definition of stack-based structure). Therefore in this case captured environments are stored in the (on the "heap", i.e. implementations), with using a . Such systems are less effective by speed than stack-based systems. However, implementations can always do different optimizations, e.g. not to allocated data on the heap, if this data is not closured.

ECMAScript closures implementation

Having discussed the theory, we at last have reached closures directly in ECMAScript. Here we should notice that ECMAScript uses static (lexical) scope (whereas in some languages, for example in Perl, variables can be declared using both static or dynamic scope).

```
1  let x = 10;
2
3  function foo() {
4    console.log(x);
5  }
6
7  (function (funArg) {
8
9    let x = 20;
10
11   // variable "x" for funArg is saved statically
12   // from the (lexical) context, in which it was created
13   // therefore:
14
15   funArg(); // 10, but not 20
16
17 })(foo);
```

Technically, the parent environment is saved in the internal `[[Scope]]` property of a function. So if we completely understand the `[[Scope]]` and the

topics, which in detail were discussed in the chapter 4, the question on understanding closures in ECMAScript will disappear by itself.

Referencing to algorithm of functions creation, we see that , since of them at creation save scope chain of a parent context. The important moment here is that, regardless — whether a function will be activated later or not — the parent scope in it :

```
let x = 10;

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

// foo is a closure
foo: <FunctionObject> = {
  [[Call]]: <code block of foo>,
  [[Scope]]: [
    global: {
      x: 10
    }
  ],
  ... // other properties
};
```

As we mentioned, for optimization purposes, when a function does not use free variables, implementations may not to save a parent scope chain. However, in ECMAScript specification nothing is said about it; therefore, formally (and by the technical algorithm) — all functions save scope chain in the `[[Scope]]` property at creation moment.

Some implementations allow access to the closured scope directly. For example in Rhino, for the `[[Scope]]` property of a function, corresponds a non-standard property `__parent__` which we discussed in the chapter about variable object:

```
var global = this;
var x = 10;

var foo = (function () {

  var y = 20;

  return function () {
    console.log(y);
  };

})();

foo(); // 20
console.log(foo.__parent__.y); // 20

foo.__parent__.y = 30;
foo(); // 30

```

```
// we can move through the scope chain further to the top
console.log(foo.__parent__.__parent__ === global); // true
console.log(foo.__parent__.__parent__.x); // 10
```

One [[Scope]] value for “them all”

It is necessary to notice that closured [[Scope]] in ECMAScript is the object for several inner functions created in this parent context. It means that modifying the closured variable from one closure, the variable in closure.

> That is, all functions the environment.

```
let firstClosure;
let secondClosure;

function foo() {

  let x = 1;

  firstClosure = function () { return ++x; };
  secondClosure = function () { return --x; };

  x = 2; // affection on AO["x"], which is in [[Scope]] of both closures

  console.log(firstClosure()); // 3, via firstClosure.[[Scope]]
}

foo();

console.log(firstClosure()); // 4
console.log(secondClosure()); // 3
```

There is a widespread mistake related with this feature. Often programmers get unexpected result, when create functions in a , trying to associate with every function the loop’s counter variable, expecting that every function will keep its “own” needed value.

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = function () {
    console.log(k);
  };
}

data[0](); // 3, but not 0
data[1](); // 3, but not 1
data[2](); // 3, but not 2
```

The previous example explains this behavior — a scope of a context which creates functions is the for functions. Every function refers it through the [[Scope]] property, and the variable k in this parent scope can be easily changed.

Schematically:

```
activeContext.Scope = [
  ... // higher variable objects
  {data: [...], k: 3} // activation object
];

data[0].[[Scope]] === Scope;
data[1].[[Scope]] === Scope;
data[2].[[Scope]] === Scope;
```

Accordingly, at the moment of function activations, last assigned value of `k` variable, i.e. `3` is used.

> This relates to the fact that all variables are created before the code execution, i.e. on entering the context. This behavior is also known as "hosting".

Creation of additional enclosing context may help to solve the issue:

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = (function _helper(x) {
    return function () {
      console.log(x);
    };
  })(k); // pass "k" value
}

// now it is correct
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

Let's see what has happened in this case.

First, the function `_helper` is created and with the argument `k`.

Then, returned value of the `_helper` function is , and exactly is saved to the corresponding element of the `data` array.

This technique provides the following effect: being activated, the `_helper` every time creates a which has argument `x`, and the of this argument is the value of `k` variable.

Thus, the `[[Scope]]` of returned functions is the following:

```
data[0].[[Scope]] === [
  ... // higher variable objects
  AO of the parent context: {data: [...], k: 3},
  AO of the _helper context: {x: 0}
];

data[1].[[Scope]] === [
  ... // higher variable objects
  AO of the parent context: {data: [...], k: 3},
  AO of the _helper context: {x: 1}
];

data[2].[[Scope]] === [
  ... // higher variable objects
  AO of the parent context: {data: [...], k: 3},
  AO of the _helper context: {x: 2}
];
```

We see that now the `[[Scope]]` property of functions have the reference to the needed value — via the `x` variable which is captured by the created scope.

Notice, that from the returned functions we still may of course reference `k` variable — with the same correct for all functions value `3`.

Often JavaScript closures reduced only to the showed above pattern — with creation of the additional function to capture the needed value. From the practical perspective, this pattern really is known, however, from the theoretical perspective as we noted, in ECMAScript are closures.

The described pattern is not a unique though. To get the needed value of `k` variable is also possible, for example, using the following approach:

```
var data = [];

for (var k = 0; k < 3; k++) {
  (data[k] = function () {
    console.log(arguments.callee.x);
  }).x = k; // save "k" as a property of the function
}

// also everything is correct
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

> Note: ES6 standardized *block scope*, which is achieved using `let`, or `const` keywords in variable declarations. The example above can be rewritten simply as:

```
let data = [];

for (let k = 0; k < 3; k++) {
```

```
    data[k] = function () {
      console.log(k);
    };
  }

  // Also correct output.
  data[0](); // 0
  data[1](); // 1
  data[2](); // 2
```

Funarg and return

Another feature is from closures. In ECMAScript, a `return` statement from a closure returns the control flow to a (a caller). In other languages, for example in Ruby, various forms of closures, which process `return` statement differently, are possible: it may be return to a caller, or in others cases — a full exit from an active context.

ECMAScript standard `return` behavior:

```
function getElement() {

  [1, 2, 3].forEach(element => {

    if (element % 2 == 0) {
      // return to "forEach" function,
      // but not return from the getElement
      console.log('found: ' + element); // found: 2
      return element;
    }

  });

  return null;
}

console.log(getElement()); // null, but not 2
```

Though, in ECMAScript in such case throwing and catching of some special "break"-exception may help:

```
const $break = {};

function getElement() {

  try {

    [1, 2, 3].forEach(element => {

      if (element % 2 == 0) {
        // "return" from the getElement
        console.log('found: ' + element); // found: 2
        $break.data = element;
        throw $break;
      }

    });
```

```
17
18    } catch (e) {
19      if (e == $break) {
20        return $break.data;
21      }
22    }
23
24    return null;
25  }
26
27  console.log(getElement()); // 2
```

Theory versions

As we noted, often developers understand as closures functions returned from parent context.

> Let's make a note again, that , independently from their type: , , function expression or function declaration, because of the scope chain mechanism, .

An exception to this rule can be functions created via Function constructor which `[[Scope]]` contains .

And to clarify this question, let's provide two correct versions of closures regarding ECMAScript:

:

- from the viewpoint: , since all they save variables of a parent context. Even , referencing a global variable refers a and therefore, the general scope chain mechanism is used;

- from the viewpoint: those functions are interesting which:
  - continue to exist after their parent context is finished, e.g. inner functions returned from a parent function;
  - use .

Practical usage of closures

In practice closures may create elegant designs, allowing customization of various calculations defined by a "funarg". An example the `sort` method of arrays which

accepts as an argument the sort-condition function:

```
[1, 2, 3].sort((a, b) => {
  ... // sort conditions
});
```

Or, for example, so-called, as the `map` method of arrays which a new array by the condition of the functional argument:

```
[1, 2, 3].map(element => {
  return element * 2;
}); // [2, 4, 6]
```

Often it is convenient to implement search functions with using functional arguments defining almost unlimited conditions for search:

```
someCollection.find(element => {
  return element.someProperty == 'searchCondition';
});
```

Also, we may note as, for example, a `forEach` method which a function to an array of elements:

```
[1, 2, 3].forEach(element => {
  if (element % 2 != 0) {
    console.log(element);
  }
}); // 1, 3
```

By the way, methods of function objects `apply` and `call`, also originate in of functional programming. We already discussed these methods in a note about this value; here, we see them in a role of — a function is to arguments (to a list of arguments — in `apply`, and to positioned arguments — in `call`):

```
(function (...args) {
  console.log(args);
}).apply(this, [1, 2, 3]);
```

Another important application of closures are :

```
let a = 10;
setTimeout(() => {
  console.log(a); // 10, after one second
}, 1000);
```

And also :

```
...
let x = 10;
// only for example
xmlHttpRequestObject.onreadystatechange = function () {
  // callback, which will be called deferral ,
  // when data will be ready;
  // variable "x" here is available,
  // regardless that context in which,
  // it was created already finished
  console.log(x); // 10
};
..
```

Or e.g. creation of an encapsulated module scope in order to hide implementation details:

```
// initialization
const M = (function () {

  // Private data.
  let x = 10;

  // API.
  return {
    getX() {
      return x;
    },
  };
})();

console.log(M.getX()); // get closured "x" - 10
```

Conclusion

In this article we tried to discuss closures more from a general theory perspective, i.e. the “Funarg problem”, which I hope made understanding closures in ECMAScript simpler. If you have any questions, as usually I’ll be glad to answer them with in comments.

Additional literature

- Javascript Closures (by Richard Cornford)
- Funarg problem
- Closures

Translated by: Dmitry Soshnikov.
Published on: 2010-02-28

Originally written by: Dmitry Soshnikov [ru, read »]
Originally published on: 2009-07-20 [ru]

Dmitry Soshnikov

Software engineer interested in learning and education. Sometimes blog on topics of programming languages theory, compilers, and ECMAScript.

Published
2010-02-28

Write a Comment

56 COMMENTS

← Older Comments

Dmitry Soshnikov
2017-11-07

@Joiner, the exists only for object environment records (used in `with` statement, and for global environment). Declarative environment records (used for function variables) do not have the binding object.

```
var x = 10;

// from the binding object of the global environment
console.log(this.x);

with ({x: 20}) {
  // From the binding object of the {x: 20}.
  console.log(x);
}
```

Joiner
2017-11-08

I get it, thank you very much

Aron
2018-01-09

Hi Dmitry,
Thanks for taking time to lay out the under the hood concepts of Javascript. I have few queries of my own like.

1.) What exactly does block scope mean. Is it just that the variables declared within "{}"("{}" meaning code within if,for,switch,while etc) is just restricted to that specific block, what is its behavior with respect to Execution Context ?

2.) For one of the cases where you have used "let" of ES6 instead of IIFEs what exactly is the "let" doing there. Could you please elaborate on that?

ogostos
2018-01-12

Hi Dmitry,

Thanks for your great articles. Could you, please, give more insights on how does this code work:

```
var data = [];

for (var k = 0; k < 3; k++) {
  (data[k] = function () {
    console.log(arguments.callee.x);
  }).x = k; // save &amp;quot;k&amp;quot; as a property of the function
}

// also everything is correct
```

```
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

I mean, to the expression enclosed in parenthesis' property named x we assigned value of k. Is it a function expression?

Dmitry Soshnikov
2018-01-14

@Aron

Prior ES6 variables were only :

```
if (false) {
  var x;
}

// Variable exists!
console.log(x); // undefined
```

Starting since ES6, `let`, and `const` create block-scoped variables (`for`, `if`, simple block, etc):

```
if (false) {
  let x;
}

// "x", is not defined!
console.log(x); // ReferenceError
```

Block creates a , which is set to current execution context (so inside the block the variables are accessible, but once it's finished, the environment is destroyed — unless is closured!).

```
let fn;

// Simple block:
{
  let x = 10;

  // Closure, capture `x`;
  fn = function() { console.log(x); };
}

fn(); // 10

// But `x` doesn't exists after block:

console.log(x); // ReferenceError
```

Dmitry Soshnikov
2018-01-14

@ogostos, the value of an array entry is a function. Functions are objects, so can store any other properties. And we store the of `k` in the `x` property in each created function. When the function is called (and is accessible via the `arguments.callee`), we can access that of `k` — via the `x` property of the function.

← Older Comments

Write a Comment

RELATED CONTENT BY TAG CLOSURE ECMA-262-3 ECMASCRIPT FIRST-CLASS OBJECTS FUNARG FUNCTIONAL PROGRAMMING