Dmitry Soshnikov in ECMAScript | 2010-12-12

ECMA-262-5 in detail. Chapter 3.1. Lexical environments: Common Theory.

Introduction

In this chapter we'll talk in detail about [illegible] — a mechanism used in some languages to manage the [illegible]. In order to understand this concept completely, we'll also discuss briefly the alternative — [illegible] scoping (which though isn't used directly in ECMAScript). We'll see how environments help to manage lexically nested structures of a code and to provide complete support of [illegible].

Lexical environments was introduced in ECMA-262-5 specifications, though this is an independent from ECMAScript theoretical concept which is used in many languages.

Actually, a technical part related with the topic (though, in the alternative terminology) we already discussed in the ES3 series — when were talking about variable and activation objects and also about a scope chain.

Strictly speaking, [illegible] in this case are just more [illegible] and more [illegible] of the previous ES3 concepts. Since ES5 era, in discussions and explanations of ECMAScript, I recommend using exactly these new definitions. Though, more generic concepts, such as an [illegible] (which is an [illegible] in ES3) of a [illegible] (which is an [illegible] in ES), etc of course may also be used, but already in discussions on [illegible].

This chapter is devoted to the [illegible] of environments and also touches several aspects of the [illegible]. We'll consider the topic from several perspectives and implementations in different languages — in order to understand, why lexical environments are needed and how these structures are created. In fact, if we completely understand the common scope theory, the question on understanding environments and scope in exactly ES will disappear by itself, and the topic will become clear.

Common Theory

As we said, all these concepts used in ES (i.e. an activation object, a scope chain, a lexical environment, etc) are related with a generic concept of a [illegible]. Mentioned ES definitions are just [illegible] and a [illegible] of the scope [illegible]. In order to understand these [illegible], let's recall the theoretical concept itself and its types.

Scope

Typically, scope is used to manage the visibility and accessibility of variables from different parts of a program.

Several encapsulating abstractions (such as , , etc) related with a scope, are invented to provide a better modularity of a system and to avoid . In the same respect, there are of and local variables of . Such techniques help to and encapsulate the internal data (not bothering a user of this abstraction with details of the implementation and exact internal variable names).

Concept of a scope helps us to use in one program the same name variables but with possibly different meanings and values. From this perspective:

> is an enclosing context in which a variable is associated with a value.

We may also say, that this is a in which a (or even an) has its . For example, a , a , etc, which generally reflects a logical range of a variable (or).

Block and function concepts lead us to one of the major scope properties — other scopes or . Thus, as we'll see, not all implementations allow nested functions, the same as not all implementations provide block-level scope.

Consider the following C example:

```
// global "x"
int x = 10;

void foo() {

  // local "x" of "foo" function
  int x = 20;

  if (true) {
    // local "x" of if-block
    int x = 30;
    printf("%d", x); // 30
  }

  printf("%d", x); // 20

}

foo();

printf("%d", x); // 10
```

It can be presented with the next figure:

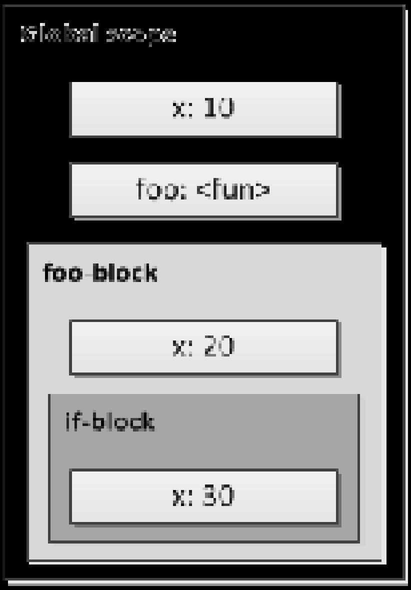


Figure 1. Nested scopes.

ECMAScript before version 6 (aka ES6 or ES2015) didn't support block-level scope:

```
var x = 10;

if (true) {
  var x = 20;
  console.log(x); // 20
}

console.log(x); // 20
```

ES6 standardized `let` keyword which creates block-scoped variables:

```
let x = 10;

if (true) {
  let x = 20;
  console.log(x); // 20
}

console.log(x); // 10
```

Previously the "block-scope" could be implemented as an immediately-invoked function expression (IIFE):

```
var x = 10;

if (true) {
  (function (x) {
    console.log(x); // 20
  })(20);
}

console.log(x); // 10
```

Another major property of a scope — is a method of a variable . Indeed, since very likely that several programmers in one project may use the same variable name (for example, `i` for a loop counter), we should know how to get correct value of an appropriate identifier with the same name. There are two conceptual ways, and as a consequence two types of scope: and . Let's clarify them.

Static (lexical) scope

In static scoping, an identifier refers to its nearest . The word "lexical" in this case relates to a property of a . I.e. where in the source text a variable appears (that is, the exact place in the code) — in it will be later at runtime on referencing this variable. The word "environment" implies again something that lexically the definition.

The word "static" relates to ability to determine the scope of an identifier of a program. That is, if we (by looking at the code) can say the starting the program, in which scope a variable will be resolved — we deal with a scope.

Let's see an example:

```
var x = 10;
var y = 20;

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

foo(); // 10, 20

function bar() {
  var y = 30;
  console.log(x, y); // 10, 30
  foo(); // 10, 20
}

bar();
```

In the example, variable `x` lexically defined in the global scope — that means at runtime it is resolved also in the global scope, i.e. to value `10` .

In case of the `y` name we have two definitions. But as we said, always the lexical scope containing the variable is considered. The own scope has the highest priority and is considered the first. Therefore, in case of the `bar` function, `y` variable is resolved as `30`. The local variable `y` of the `bar` function is said to the same name variable `y` of the global scope.

However, the same name `y` is resolved as `20` in case of `foo` function — it is called the `bar` function, which contains another `y`. I.e. resolution of an identifier is independent from the environment of a (in this case `bar` is a of `foo`, and `foo` is a). And again, this is because at the moment of `foo` function , the lexical context with the `y` name — was the global context.

Today static scope is used in many languages: C, Java, ECMAScript, Python, Ruby, Lua, etc.

We'll mention a bit later the mechanisms of lexical scope (after all it's used in ECMAScript), and also will talk about subtle cases of using it with the . And now let's note briefly an alternative, scope — in order to see the difference and understand, why the dynamic scope cannot be used in supporting . As we'll see ECMAScript also has some features of dynamic scope.

Dynamic scope

In contrast with the static scope, assumes that we determine at , to which value (in which environment) a variable will be resolved. Usually it means, that the variable is resolved in the environment, but rather in the formed . Every met variable declaration just puts the name of the variable onto the stack. After the scope (lifetime) of the variable is finished, the variable is removed (popped) from the stack.

That means, that for a function we may have of the variable name — from which the function is .

For example, consider a similar case as from above, but with using the dynamic scope. We use pascal-like pseudo-code syntax:

```
// *pseudo* code - with dynamic scope

y = 20;

procedure foo()
  print(y)
end


// on the stack of the "y" name
// currently only one value 20
// {y: [20]}

foo() // 20, OK

procedure bar()

  // and now on the stack there
  // are two "y" values: {y: [20, 30]};
  // the first found (from the top) is taken

  y = 30

  // therefore:
  foo() // 30!, not 20

end

bar()
```

We see that the environment of a affects on the variables resolution. Since a function (a) may be called from different locations and with different states, it's hard to determine the exact environment statically, at parsing stage. That's why this type of scope is called as .

That is, a variable is resolved in the environment of , rather than the environment of as we have in the static scope.

One of the dynamic scope benefits is ability to apply the same code for different (mutable over the time) states of a system. However, such a benefit requires to keep in mind all possible cases of the function executions.

Obviously, with the dynamic scope it's to create a for a variable.

Today, most of the modern languages dynamic scope. However, in some languages, and notably in Perl (or some variations of Lisp), a programmer may choose how to define a variable — with static or dynamic scope.

Take a look on the following Perl example. Keyword `my` there captures a variable , meanwhile keyword `local` makes the variable dynamically-scoped:

```
# Perl example of static and dynamic scopes

```

```
$a = 0;

sub foo {
  return $a;
}

sub staticScope {
  my $a = 1; # lexical (static)
  return foo();
}

print staticScope(); # 0 (from the saved global frame)

$b = 0;

sub bar {
  return $b;
}

sub dynamicScope {
  local $b = 1; # dynamic
  return bar();
}

print dynamicScope(); # 1 (from the caller's frame)
```

`With` and `eval` as dynamic scope features in ECMAScript

As we said, ECMAScript use dynamic scope (in the view we described it above) also. However, a pair of ES instructions sometimes are considered as to the static scope. Therefore, modifications made by such instructions can also be related to . But notice again — not in respect of the global variables stack as in the standard dynamic scope definition, but in respect of to determine at in which environment a variable will be resolved. These instructions are `with` and `eval` . The effect they apply to the ECMAScript static scope can be called a "Runtime scope augmentation".

Consider the following example:

```
var x = 10;

var o = {x: 30};
var storage = {};

(function foo(flag) {

  if (flag == 2) {
    eval("var x = 20;");
  }

  if (flag == 3) {
    storage = o;
  }

  with (storage) {

    // "x" may be resolved either
    // in the global scope - 10, or
    // in the local scope of a function - 20
    // (created via "eval" function), or even
    // in the "storage" object - 30

    alert(x); // ? - scope of "x" is undetermined at compile time

  }

  // organize recursion on 3 calls

  if (flag < 3) {
    foo(++flag);
  }

})(1);
```

As we'll see shortly, static lexical scoping increase efficiency, and the `with` and `eval` in contrast may decrease performance of lexical environments storage and variables lookup at implementation level. Therefore, statement `with` was completely removed from ES5 strict mode. And the `eval` function in the strict mode may not create variables in the calling context. That is, strict mode provides a lexical scope in ES.

Further in this chapter we'll discuss only lexical (static) scope and details of its implementation. But before it we should consider briefly concept of a , since it's actively used with concept of environments.

Name binding

Having highly-abstracted languages we usually operate not with low-level addresses to refer some data in memory, but rather with convenient variable names (identifiers), which reflect that data.

> A is the association of an with an .

An identifier can be or . If an identifier is bound to an object, it this object. The following use of the identifier results the object it's bound to.

With bindings concepts two major operations are related (which often cause confusion in discussing or strategies of passing arguments and assignment). These operations are and .

Rebinding

A relates to an . This operation the identifier (if it was previously bound) from an old object and it to another one (to another block of memory). Often (and in ECMAScript in particular) rebinding is implemented via a simple operation of .

For example:

```
// bind "foo" to {x: 10} object
var foo = {x: 10};

console.log(foo.x); // 10

// bind "bar" to the *same* object
// as "foo" identifier is bound

var bar = foo;

console.log(foo === bar); // true
console.log(bar.x); // OK, also 10

// and now rebind "foo"
// to the new object

foo = {x: 20};

console.log(foo.x); // 20

// and "bar" still points
// to the old object

console.log(bar.x); // 10
console.log(foo === bar); // false
```

Often rebinding is with assignment . One could thought that after assigning the new object to the `foo` variable in the example above, `bar` variable would also point to the new object. However, as we see, `bar` still refers to the old object, meanwhile `foo` was to the new memory block. The next figure shows these two actions:

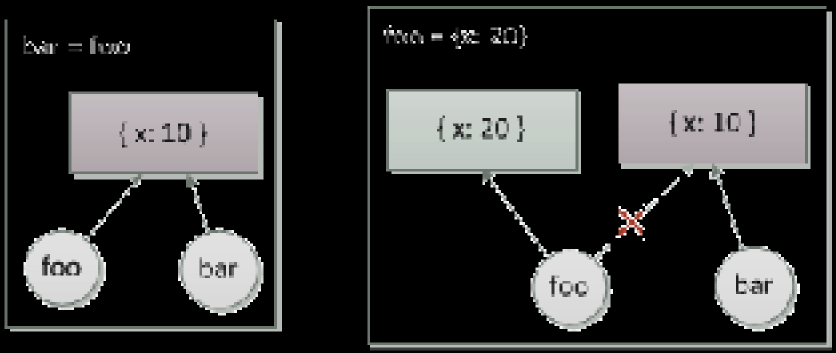


Figure 2. Rebinding.

Think about bindings not as by-reference, but (from C viewpoint) as (or sometimes —) operation. Often it's also called as a special case of where value is an address. Assignment just changes (rebinds) the (the address) from one memory block to another. And when we assign one variable to another we just the address of the same object to the second variable. Now two identifiers are said to the one object. From here the name — .

Mutation

In contrast with rebinding, the operation of already affects the of the object.

Consider the following example:

```
// bind an array to the "foo" identifier
var foo = [1, 2, 3];

// and here is a *mutation* of
// the array object contents
foo.push(4);

console.log(foo); // 1,2,3,4

// also mutations
foo[4] = 5;
foo[0] = 0;

console.log(foo); // 0,2,3,4,5
```

This code is presented in the following figure:

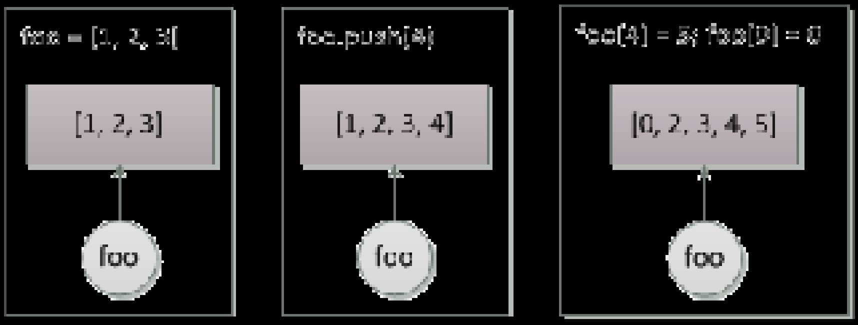


Figure 3. Mutation.

You may find an additional information about the binding and evaluation strategies (by-reference, by-value, by-sharing) in the appropriate Chapter 8. Evaluation strategy of the ES3 series.

And now we are ready to discuss in detail general concept of and to see how they are made from within.

Environment

In this section we'll mention the of the lexical scope implementation. Also, since we operate with highly-abstracted entities and talk about lexical scoping, in the further explanation we'll mainly use the concept of an , rather than , since exactly this terminology is used in ES5. E.g. the same — a , a of a function, etc.

As we mentioned, an environment specifies the of an (of a) in the expression. Indeed, it is meaningless to speak of the value of an expression such as e.g. `x + 1` without specifying any information about the that would provide a meaning for the symbol `x` (or even for the symbol `+`, if to treat it as a syntactic sugar for a simple function of addition — `add(x, 1)`, though in the last case we should provide the meaning of the `add` name as well).

ECMAScript manages executions of functions with using the model of a , which is called here the execution context stack. Let's consider some generic models for the variables (bindings). The things interesting for us — systems with and without them.

Activation record model

Until we have no functions (i.e. such functions which may participate as — we'll talk about them a bit later) or simply , the easiest way to store local variables is to use the itself.

A special data structure of the which is called an activation record is used as a storage of the environment bindings. Sometimes it's also called a .

Every time a function is activated, its activation record (with formal parameters and local variables) is pushed onto the call-stack. Thus, if the function calls another function (or itself — recursively) another stack-frame is pushed onto the stack. After the context is finished, the activation record is (popped) from the stack (which means — all local variables are destroyed). This model is used e.g. in C programming language.

For example:

```
void foo(int x) {
  int y = 20;
  bar(30);
}

void bar(x) {
  int z = 40;
}

foo(10);
```

Then the call-stack has the following modifications:

```
callStack = [];

// "foo" function activation
// record is pushed onto the stack

callStack.push({
  x: 10,
  y: 20
});

// "bar" function activation
// record is pushed onto the stack

callStack.push({
  x: 30,
  z: 40
});

// callStack at the moment of
// the "bar" activation

console.log(callStack); // [{x: 10, y: 20}, {x: 30, z: 40}]

// "bar" function ends
callStack.pop();

// "foo" function ends
callStack.pop();
```

On the next figure we see two activation records pushed onto the stack — it's the moment of `bar` function activation:

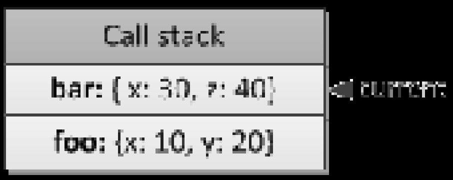


Figure 4. A call stack with activation records.

And logical approach of functions execution is used in ECMAScript. However, with some .

First, as we know and said above, for the call-stack concept here stands the and for the activation record stands (in ES3) the .

However, the terminology difference is not so essential in this case. The main difference, is that in contrast with C, ECMAScript does remove the activation object from the memory if there is a . And the most important case is when this closure is some inner function which uses variables of the parent function in which it's created, and this inner function is returned upwards to the outside.

That means that the activation object should be stored in the itself, but rather in the (a ; sometimes such languages are called languages — in contrast with languages). And it is stored there until there are references from closures which use (free) variables from this activation object. Moreover, not only one activation object is saved, but if needed (in case of several nested levels) — parent activation objects.

```
var bar = (function foo() {
  var x = 10;
  var y = 20;
  return function bar() {
    return x + y;
  };
})();

bar(); // 30
```

In the next figure abstract representation of the heap-based activation records is presented. We see that if `foo` function creates a closure, then after `foo` is finished, its frame isn't removed from memory because there is still reference to it from the closure:

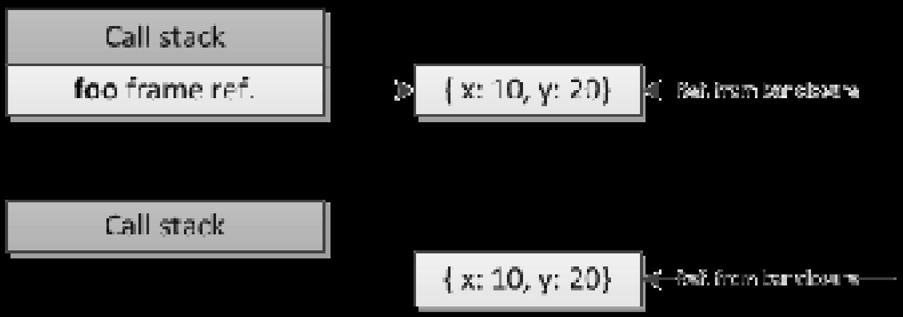


Figure 5. Heap-based call-frames.

One of the used terminologies in the theory for these activation objects is (analogy with). We use this terminology to underline the difference of the implementation — that environment frames continue to exist if there are references to them from closures. Also we use this terminology to underline the higher-abstracted concepts — i.e. without concentrating on lower level stack and addresses structures, we just say that these are , and how they are implemented — is already a derived question.

Environment frames model

As is said, in ECMAScript, in contrast with C, we inner functions and closures. Moreover, all functions in ES are the . Let's recall the definition of such functions and also other definitions of the . We'll see that these concepts are closely related with lexical environments model.

We'll also clarify that the problem of (or — the "funarg problem" as will be mentioned below) is actually the problem of exactly . That's why in this section we'll mostly speak about basic concepts of functional languages.

First-class functions

> A is one that may participate as a , i.e. be , be , or from another function.

A simple example:

```
// create a function expression
// dynamically at runtime and
// bind it to "foo" identifier

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

// pass it to another function,
// which in turn is also created
// at runtime and called immediately
// right after the creation; result
// of this function is again bound
// to the "foo" identifier


```

```
foo = (function (funArg) {

  // activate the "foo" function
  funArg(); // "foo"

  // and return it back as a value
  return funArg;

})(foo);
```

First-class functions may be stratified into more exact sub-definitions.

Funargs and higher-order functions

When a function is passed as an argument, it's called a — an abbreviation of the concept.

In turn, a function which accepts the "funarg" is called a or, closely to mathematics, an .

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

With these concepts, as we'll see below, a so-called is related. And as also we'll see shortly, the solution of this problem are exactly and .

In the example above, `foo` function is a "funarg" which is passed to the (it accepts `foo` "funarg" by the formal parameter name `funArg`). This anonymous function in turn returns the — and again the `foo` function itself. And all these functions are grouped by the definition.

Free variable

Another important concept which is related with first-class functions and which we should recall — is the concept of a .

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

In other words, a free variable is one that is placed not in the environment, but probably in some environments. Notice, that a free variable may the same be as (i.e. found in some parent environment) or . The last case will cause a `ReferenceError` in ECMAScript.

Take a look on the example:

```
// Global environment (GE)

var x = 10;

function foo(y) {

  // environment of "foo" function (E1)

  var z = 30;

  function bar(q) {
    // environment of "bar" function (E2)
    return x + y + z + q;
  }

  // return "bar" to the outside
  return bar;

}

var bar = foo(20);

bar(40); // 100
```

In this example we have three environments: `GE`, `E1` and `E2`, which correspond respectively to the global object, `foo` function and `bar` function.

Thus, for the `bar` function, `x`, `y` and `z` variables are — they are neither formal parameters, nor local variables of `bar`.

Notice, that `foo` function use free variables. However, since `x` variable is used inside the `bar` function, and because `bar` function is created during of the `foo` function, the later one should nevertheless save the bindings of the parent environment — in order to pass the information about the `x` binding further to the deeper nested functions (to the `bar` in our case).

Correct and expected result `100` after the `bar` function activation means, that `bar` function somehow the environment of the `foo` function activation (where internal `bar` function is), the context of the `foo` is finished. Repeat again, this is the difference from the stack-based activation record model used in C.

Obviously, if we allow nested inner functions and want to have the static (lexical) scope, and at the same time — to have all these functions as , we should at the moment of the function's .

Environment definition

The most straightforward and the easiest way to implement such an algorithm, is to save the parent environment in which we were created. Later, at our activation (in this case, at activation of the `bar` function), we'll create our environment, fill it with local variables and parameters, and set as our environment the saved one — in order to find free variables there.

It is possible to use the term either for a bindings object, or for the list of all binding objects corresponding to the deepness of the nested level. In the later case we may call the binding objects as of the environment. From this viewpoint:

> An is a sequence of . Each frame is a (possibly empty) of , which associate variable names with their corresponding values.

Notice, since this is a generic definition, we use the abstract concept of a without specifying exact implementation structure — it may be a hash-table placed in the heap, or a stack memory, or even registers of the virtual machine, etc.

For example, environment `E2` from the example above has three frames: own — `bar`, `foo` and `global`. Environment `E1` contains two frames: `foo` (own) and the `global` frame. Global environment `GE` in turn consists only from one, `global`, frame.

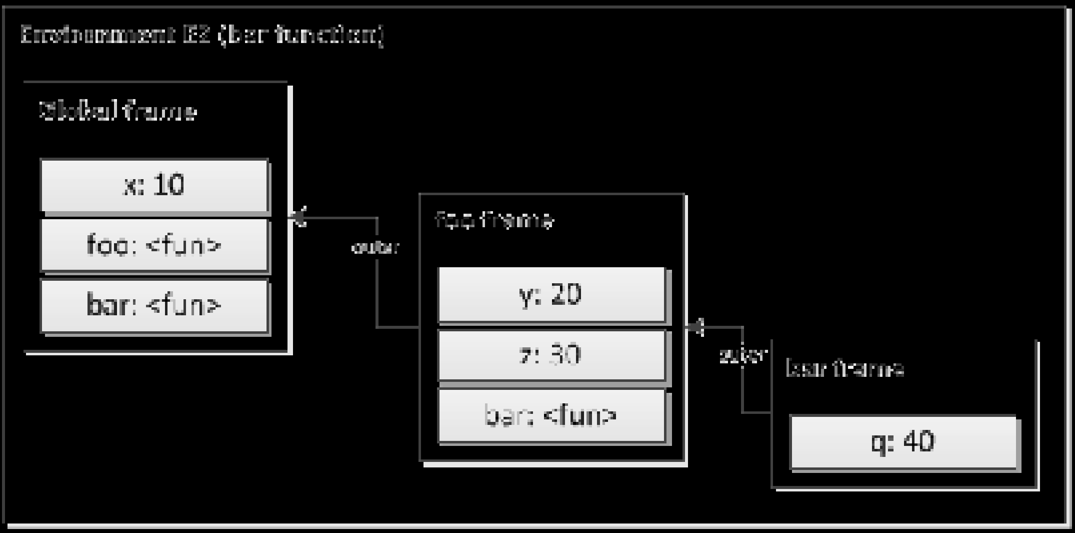


Figure 6. An environment with frames.

A single frame may contain binding for any variable. Each frame also has a pointer to its (or an) environment. The outer reference of the frame is `null` . The value of a variable with respect to an environment is the value given by the binding of the variable in the frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be in the environment (the case of a `ReferenceError`).

```
var x = 10;

(function foo(y) {

  // use of free-bound "x" variable
  console.log(x);

  // own-bound "y" variable
  console.log(y); // 20

  // and free-unbound variable "z"
  console.log(z); // ReferenceError: "z" is not defined

})(20);
```

I.e. backing again to the concept of , this sequence of environment frames (or in a different view — a of environments) forms something that we may call as a . Not surprisingly, ES3 had exactly this terminology for that — a .

Notice, a environment may serve as an environment for inner environments:

```
// Global environment (GE)

var x = 10;

function foo() {

  // "foo" environment (E1)

  var x = 20;
  var y = 30;

  console.log(x + y);

}

function bar() {

  // "bar" environment (E2)

  var z = 40;

  console.log(x + z);
}
```

Thus in pseudo-code:

```
// global
GE = {
  x: 10,
  outer: null
};

// foo
E1 = {
  x: 20,
  y: 30,
  outer: GE
};

// bar
E2 = {
  z: 40,
  outer: GE
};
```

The next figure shows these relations:

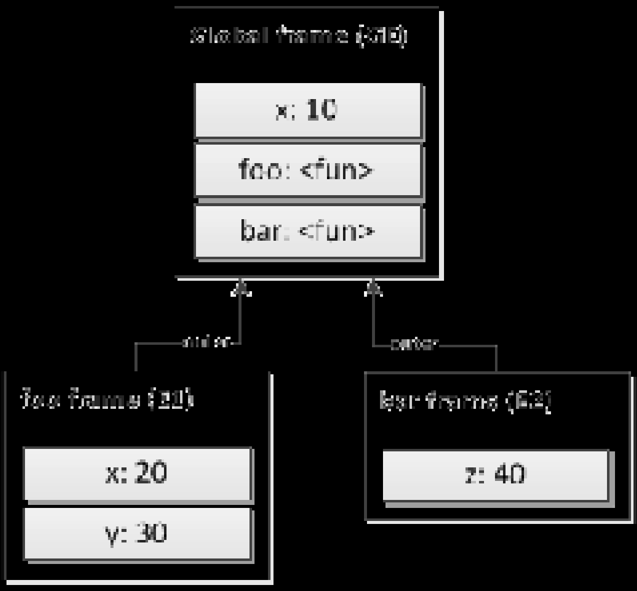


Figure 7. Common parent environment frame.

That is, binding `x` in respect of the environment `E1` shadows the same name binding in the global frame.

Rules of function creation and application

From all this we have the generic rules for creating and applying (calling) functions:

> A function is relatively to a given environment. The resulting function object is a consisting of the (function body) and a in which the function was .

The code:

```
// global "x"
var x = 10;

// function "foo" is created relatively
// to the global environment

function foo(y) {
  var z = 30;
  console.log(x + y + z);
}
```

Corresponds in the pseudo-code to:

```
// create "foo" function

foo = functionObject {
  code: "console.log(x + y + z);"
  environment: {x: 10, outer: null}
};
```

This function object is shown on the following figure:

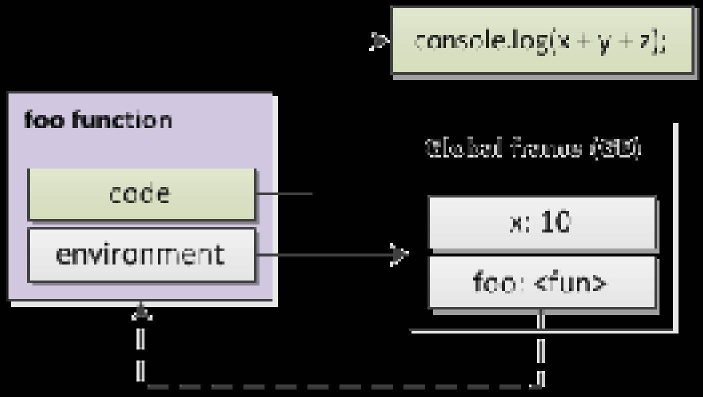


Figure 8. A function.

Note, the function refers to its environment, and one of the environment bindings — the function — to the function object.

> A function is with (or to) a set of arguments by constructing a , binding the formal parameters of the function to the arguments of the call, creating bindings for local variables in this frame, and then executing the body of

> the function in the context of the constructed. The has as its the environment part of the function object being applied.

And the application:

```
// function "foo" is applied
// to the argument 20

foo(20);
```

Corresponds to the following pseudo-code:

```
// create a new frame with formal
// parameters and local variables

fooFrame = {
  y: 20,
  z: 30,
  outer: foo.environment
};

// and evaluate the code
// of the "foo" function

execute(foo.code, fooFrame); // 60
```

The next figure shows the function application using environment:

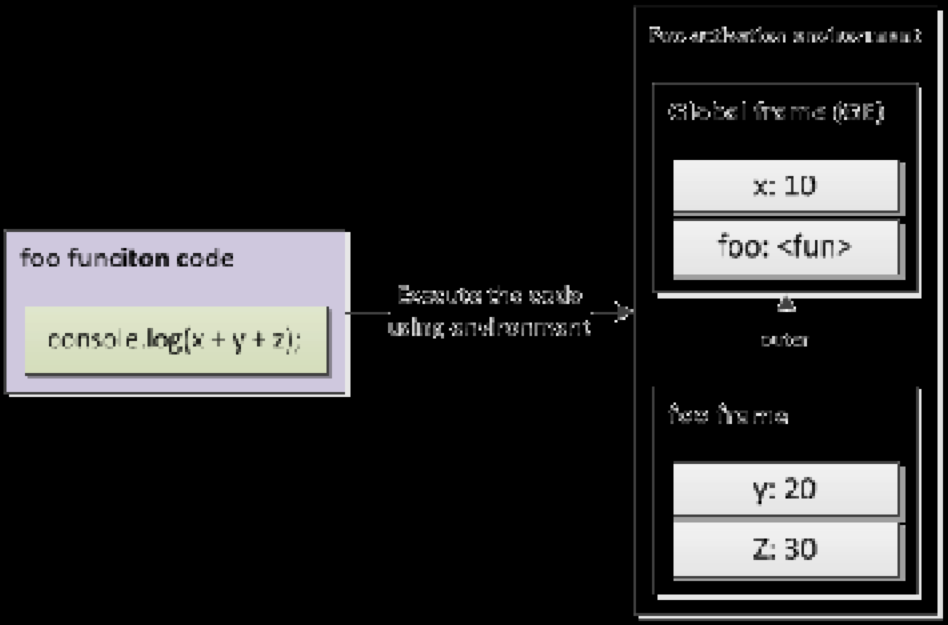


Figure 9. Function application.

The first point from this conclusion directly gives us a definition of a .

Closure

> A is a consisting of the function and the in which the function is .

As we mentioned above, closures are invented as a solution for the "funarg problem". Let's recall it in order to have the complete understanding.

Funarg problem

The funarg problem is divided into two sub-problems which are directly related with concepts of , and .

> corresponds to the an inner function to the (upward) — i.e. how can we implement the returning of the function if this function uses free variables of the parent environment in which it's created?

```
(function (x) {
  return function (y) {
    return x + y;
  };
})(10)(20); // 30
```

As we already know, the with saving enclosing frame on the heap — the key and the answer. And the strategy of storing bindings on the (used in C) anymore. Let's repeat again, this saved code block and the environment — a .

> corresponds to the of the variable name when we pass a function which uses free variables as an argument to another function. In which scope these free variables should be resolved — in the scope of the function or in the scope of the function ?

```
var x = 10;

(function (funArg) {

  var x = 20;
  funArg(); // 10, not 20

})(function () { // create and pass a funarg
  console.log(x);
});
```

I.e. relates to the choice of the and scopes discussed in the beginning of the chapter. As we already know and said above — again, the — the answer. We should save exactly the variables to avoid such ambiguities. And again — this saved lexical variables and the code of our function — is what is called a .

So what do we have at the end? Concepts of functions, and are . And the lexical environments is exactly the which is used in implementation of and the .

At this step (though, running forward) mention that ECMAScript uses with . However, concrete ES terminology we'll discuss in the appropriate section.

Detailed explanation of closures may be found in the Chapter 6. Closures of the ES3 series.

For the completeness let's also clarify an alternative environments implementation used in some other languages.

Combined environment frame model

Always remember, that in order to understand some technology (e.g. ECMAScript) , we should always first understand the mechanisms of the and also to see how languages implement the technology. Then we'll see that these are become apparent in many similar languages. Though, the different languages may treat the implementation also differently. This section is devoted to environments in such languages as Python, Ruby and Lua.

An alternative way to save all free variables is to create environment frame which contains , but free variables collected from enclosing environments.

Obviously, if some variables are not needed for inner functions, there is no need to save them. Consider the following example:

```
// global environment

```

```
var x = 10;
var y = 20;

function foo(z) {

  // environment of "foo" function
  var q = 40;

  function bar() {
    // environment of "bar" function
    return x + z;
  }

  return bar;

}

// creation of "bar"
var bar = foo(30);

// applying of "bar"
bar();
```

We see that none of the functions use global `y` variable. Therefore, we don't save it neither in `foo` 's closure, nor in the `bar` 's.

Global variable `x` is not used by the `foo` function, however as we mentioned before, we should save it in the `foo` 's closure since the deeper inner `bar` function use it and should take information about `x` from the environment in which it is created (i.e. from the environment of the `foo` function).

With the `q` variable of the `foo` function the same situation as with the global `y` — no one uses it on deeper levels, so — we do not save it in `bar` 's closure. Variable `z` of course is saved in the `bar` .

Thus, we have a environment frame of the function containing all needed free variables:

```
bar = closure {
  code: <...>,
  environment: {
    x: 10,
    z: 30,
  }
}
```

A similar model is used for example in the programming language. There functions have one saved environment frame which is called simply and directly as `__closure__` (reflecting the of the lexical environments). Global variables are not included in this frame, since they may always be found in the global frame. Not used variables are also not in the `__closure__` . Take a look on the example:

```python
# Python environments example

# global "x"
x = 10

# global "foo" function
def foo(y):

    # local "z"
    z = 40

    # local "bar" function
    def bar():
        return x + y

    return bar

# create "bar"
bar = foo(20)

# execute "bar"
bar() # 30

# Saved environment of the "bar" function;
# it's stored in the __closure__ property;
#
# It contains only {"y": 20};
# "x" is not in the __closure__ since it's
# always can be found in the global scope;
# "z" is not saved either since it's not used

barEnvironment = bar.__closure__
print(barEnvironment) # tuple of closure cells

internalY = barEnvironment[0].cell_contents
print(internalY) # 20, "y"
```

Notice, that Python doesn't save non-used variables even in case of using `eval`, i.e. when it is not known in advance will a variable be used in the context or not. In the following example internal `baz` function captures free variable `x`, and the `bar` function does not:

```python
def foo(x):

    def bar(y):
        print(eval(k))

    def baz(y):
        z = x
        print(eval(k))

    return [bar, baz]

# create "bar" and "baz" functions
[bar, baz] = foo(10)

# "bar" does not closure anything
print(bar.__closure__) # None

# "baz" closures "x" variable
print(baz.__closure__) # closure cells {'x': 10}

k = "y"

baz(20) # OK, 20
bar(20) # OK, 20

k = "x"
```

```

baz(20) # OK, 10 - "x"
bar(20), # error, "x" is not defined
```

And again, ECMAScript in contrast, having chained frames of the environment, manages this case normally:

```
function foo(x) {

  function bar(y) {
    console.log(eval(k));
  }

  return bar;

}

// create "bar"
var bar = foo(10);

var k = "y";

// execute bar
bar(20); // OK, 20 - "y"

k = "x";

bar(20); // OK, 10 - "x"
```

For the brief but detailed explanation of closures in Python see this Python code-article.

I.e. the main difference is that model of (used in ECMAScript) optimizes the moment of function , however at the the scope chain, considering environment frames (until the needed binding will be found or the `ReferenceError` will be thrown), should be traversed.

Meanwhile the model of the optimizes the (all identifiers are resolved in the nearest single frame without long scope chain lookup), however requires more complex algorithm of the function with parsing all inner function and determining which variables should be saved and which are not.

Notice though, that this conclusion is only according to the ECMA-262-5 specification. In practice, ES engines may easily optimize the ECMAScript implementation and save variables. But about ECMAScript implementation we'll talk in the following chapter 3.2.

Also notice, that strictly speaking a combined frame may not be the single. This means, that the combined frame is optimized to contain bindings from several parent frames, however the environment may include some additional frames. The same in Python — at execution it has an of the activation, the `__closure__` , and the .

programming language is one that also uses the single frame, with capturing , but only at the moment of the closure variables. In the following example on Ruby variable `x` is captured by the second closure, but not by the first one:

```
# Ruby lambda closures example

# closure "foo", which has
# free variable "x"

foo = lambda {
  print x
}

# define the "x"
x = 10

# second closure "bar" with
# the same body - it also
# refers free variable "x"

bar = lambda {
  print x
}

bar.call # OK, 10
foo.call # error, "x" is not defined
```

However, as mentioned, Ruby saves all existing variables, and the described above case with `eval` resolves (the same as ES and in contrast with Python) an unused variable:

```
k = "y"

foo = lambda { |x|
  lambda { |y|
    eval(k)
  }
}

# create "bar"
bar = foo.call(10)

print(bar.call(20)) # OK, 20 - "y"

k = "x"

print(bar.call(20)) # OK, 10 - "x"
```

Some languages, e.g. (which also has a single environment frame) allow to set the needed environment of a function dynamically at runtime. Consider the following Lua example:

```
-- Lua environments example:

-- global "x"
x = 10

-- global function "foo"
function foo(y)

  -- local variable "q"
  local q = 40

  -- get environment of the "foo"
  fooEnvironment = getfenv(foo)

  -- {x = 10, globals...}
  print(fooEnvironment) -- table

  -- "x" and "y" are accessible from here,
  -- since "x" is in the environment, and
  -- "y" is a local variable (argument)
  print(x + y) -- 30

  -- and now change the environment of the "foo"
  setfenv(foo, {
    -- use reference to "print" function,
    -- but give it another name
    printValue = print,

    -- reuse "x"
    x = x,

    -- and define a new "z" binding
    -- with value of the "y"
    z = y

  })

  -- use new bindings

  printValue(x) -- OK, 10
  printValue(x + z) -- OK, 30

  -- local variables are still accessible
  printValue(y, q) -- 20, 40

  -- but not other names
  printValue(print) -- nil
  print("test") -- error, "print" name is nil, can't call

end

foo(20)
```

Conclusion

At this step we're completing the common theory consideration. The next sub-chapter 3.2 will be devoted to exactly ECMAScript implementation. We'll consider such structures as (which correspond to the of environments in the theory we've discussed here), and talk about their different types: and , we'll see which structure has an in the ES5 and how its different parts are related with different types of functions — known for us and .

The summary which we have for this chapter:

- Concept of an is related with a concept of a .
- In the theory there are two types of scope: and .
- ECMAScript uses (lexical) scope.
- However, `with` and `eval` instructions may be considered as to the static scope.
- Concepts of scope, environment, activation object, activation record, call-stack frame, environment frame, environment record and even execution context — are all and may be used in discussions. Thus, technically in ECMAScript some of them are parts of another — e.g. an environment record is a part of the lexical environment which in turn is a part of the execution context. However, logically in abstract definitions they all may be used nearly equally. It's normal to say: "a global scope", "a global environment", "a global context", etc.
- ECMAScript uses model of the . In ES3 it was called a . In ES5 as we'll see an is called an .
- An environment may enclose inner environments.
- Lexical environment are used to implement and to solve the .
- functions in ECMAScript are and .

If you have questions, additions or corrections, feel free to discuss them in comments.

Additional literature

Structure and Interpretation of Computer Programs (SICP):

- 3.2 The Environment Model of Evaluation

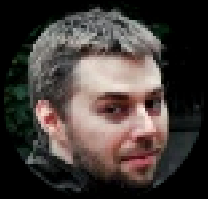
ECMA-262-5 in detail:

- Chapter 3.2. Lexical environments: ECMAScript implementation.

Other useful literature on common theory:

- Scope
- Name binding
- Call stack
- Activation record
- Funarg problem
- Free variable

Written by: Dmitry A. Soshnikov.
Published on: 2010-12-12.



Dmitry Soshnikov

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

Published

2010-12-12

Write a Comment

29 COMMENTS

John Merge
2010-12-15

Amazing stuff, really!
The best articles I have ever seen – this one too!

Keep going, man, you rules!

John Merge
2010-12-15

Typos:

resoled
association of of an identifier
The following use of the identifier results the object it's bound too. (to)

> *The word "environment" implies again something that lexically surrounds the definition.*

What exactly did you mean?

Dmitry A. Soshnikov
2010-12-16

@John Merge

Thanks, glad to see more people interesting in deep JS.

> *What exactly did you mean?*

I meant an analogy with the real meaning of the "environment" word. E.g. ecological environment (surrounding) — something that surrounds us.

The same is here:

```
var x = 10;

function foo() {

  var y = 20;

  function bar() {
    var z = 30;
  }
}
```

The definition of the `foo` function is by the global . The definition of the `bar` function is surrounded by the `foo` 's environment. And "lexically" means — the exact nearest place in the source code position.

P.S.: thanks for fixing typos (I let myself to combine them in one message). Please inform me if there are more.

Dmitry.

John Merge
2010-12-16

Typos:

"Notice, a one environment…"
"Rules of function creation and application"

> *And the most important case, when this closure — is some inner function which uses variables of the parent function in which it's created, and this inner function is returned upwards to the outside.*

Did you mean:

"And the most important case is when this closure is some inner function which uses variables of the parent function in which it's created, and this inner function is returned upwards to the outside."?

Also, this part: might be improved.

joseanpg
2010-12-16

I will be a laconic commenter, Dmitry this is **superb!!** 😉

Dmitry A. Soshnikov
2010-12-17

@John Merge

Yep, corrected, thanks. Feel free to propose other improvements.

@joseanpg

Thanks, Jose 😉 Also thanks for proposals sent via mail.

Robert Polovsky
2010-12-19

It's just awesome stuff, Dmitry! Thank you for writing so scientific articles, they are really the best I have ever read on JavaScript.

Also special thanks for consideration closures in Python, I'm also interested in this language.

Dmitry A. Soshnikov
2010-12-20

@Robert Polovsky, yep, thanks, glad it's useful.

> *Also special thanks for consideration closures in Python*

Yes, Python and ECMAScript have many similar design features. But as we saw, having the similar core features they nevertheless differ in some implementation aspects. And exactly this difference helps us to analyze the theoretical topic in-depth.

Dmitry.

monolithed
2011-01-09

Дмитрий, походу прочтнения вашей статьи у меня возникло несколько вопросов:

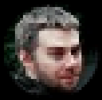
1. как правильно переводится эти выражение: Lexical environments и Lexical scope?. (для себя перевел как Лексическое окружение/контекст и лексическая область видимости)

2. правильно ли я понял, что Dynamic scope как таковой в ES отсутствует?

3. откуда у вас сведения, что инструкция let появится в ES6 (интересно же почитать о том что может нас порабовать в будущем)

4. и если вас не затруднит могли бы вы хоть в кратце прояснить ситуацию о Variable Environment (т.к. она здесь не обсуждаеься) и ее отличиями с Lexical environments

Заранее благодарен за ответы!



Dmitry A. Soshnikov
2011-01-10

@monolithed

> *1. как правильно переводится эти выражение: Lexical environments и Lexical scope?. (для себя перевел как Лексическое окружение/контекст и лексическая область видимости)*

Да, все верно. “Scope” — область видимости. По поводу “environment” — чаще используется “лексическое окружение” (т.е. та , которая в коде лексически объявление функции), но и “лексический контекст” тоже подойдет.

> *2. правильно ли я понял, что Dynamic scope как таковой в ES отсутствует?*

Да. Но, как отмечено, `with` и `eval` могут быть рассмотрены как конструкции, в лексический скоп. Но не в классическом смысле динамической ОВ (области видимости), а в плане, что невозможно на этапе парсига определить, в какой ОВ переменная будет разрешена позже при обращении.

ES5 strict (и, соответственно, ES6 Harmony) отменили `with` , а `eval` запускается в своей “песочнице” и не может создать переменную в вызывающем контексте (кроме косвенного вызова eval'а). Т.е. из ES5-strict удалены фичи динамического скопа.

> *3. откуда у вас сведения, что инструкция let появится в ES6 (интересно же почитать о том что может нас порабовать в будущем)*

Основное обсуждение дизайна ES ведется в листе es-discuss. Периодически результаты обсуждений и предложений описываются на официальном сайте http://ecmascript.org. По поводу `let`, можно почитать здесь, здесь и здесь.

> *4. и если вас не затруднит могли бы вы хоть в кратце прояснить ситуацию о Variable Environment (т.к. она здесь не обсуждаеься) и ее отличиями с Lexical environments*

Данная статья — это общая теория безотносительно конкретных языков программирования (однако, описанная на примерах различных языков). Предполагается, что после прочтения этой статьи, конкретика уже должна восприниматься именно как конкретика, которая использует внутри именно механизмы этой общей теории.

Но, поскольку данный цикл статей посвящен именно ECMAScript, мы конечно будем подробно и в деталях разбирать всю конкретику ES в следующей статье "Chapter 3.2. Lexical Environments. ECMAScript implementations", т.к. там тоже много своих нюансов (но, повторю, "ядром" этих всех нюансов является именно эта общая теория, описанная здесь в 3.1).

Здесь же кратко отмечу. Оба компонента: и `VariableEnvironment` (VE), и `LexicalEnvironment` (LE) являются свойствами контекста исполнения. При этом первый служит для при входе в контекст (т.е. это тот самый variable object, VO и ES3), а второй — для уже в рантайме.

Изначально, LE равна VE. Однако в процессе исполнения кода, LE может меняться (например, заменяться средой, созданной `with`), в то время как VE никогда не меняется на протяжении кода контекста.

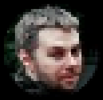
Основное отличие — при работе с разными видами функций. FD запоминает в качестве `[[Scope]]` VE, а FE, соответственно, LE, т.к. в отличие от FD может быть создана в рантайме и, например, внутри `with`, когда LE контекста будет подменена.

Подробней будем разбирать в 3.2.

NekR
2011-03-14

Да, спасибо за хороший цикл статей. Особенно интересна будет следующая, хотелось бы по подробнее познакомиться с современными движками и обработкой им областей видимости.
В 5ой редакции по идее движки должны сохранять только нужные (используемые) переменные в области видимости, а вот в 3ей редакции не понятно как поступают движки, ведь если судить по тому, что через eval можно получить любую переменную и области видимости, то при "встрече" в коде сохраняются все переменные "родительских" областей.

П.С. "Гармоничный" это в том смысле, что ES4 (ActionScript) и ES3/ES5 (JavaScript) смогут перейти на него бесконфликтно?



Dmitry A. Soshnikov
2011-03-15

@NekR

> *Особенно интересна будет следующая, хотелось бы по подробнее познакомиться с современными движками и обработкой им областей видимости.*

Да, ES5 вводит некоторую оптимизацию для обработки лексических окружений. Выделяют и, соответственно, .
Фактически первая, для хранения биндингов (переменных), может использовать не объекты на куче, а прямо регистры виртуальной машины. Вторая используется для `with` и глобального объекта.

Про конкретные реализации в конкретных движках сказать не могу, т.к. я не имплементер.

> *В 5ой редакции по идее движки должны сохранять только нужные (используемые) переменные в области видимости, а вот в 3ей редакции не понятно как поступают движки, ведь если судить по тому, что через eval можно получить любую переменную и области видимости, то при "встрече" в коде сохраняются все переменные "родительских" областей.*

Вообще, и в 5-ой редакции ничего не сказано про то, что не все переменные должны замыкаться. Как и в 3-ей редакции родительское окружение сохраняется полностью (точнее, ссылка на него). Соответственно, если функция содержит другие внутренние функции или `eval` (или еще ~~хуже~~ интересней – `eval` по неизвестным заранее родительским переменным внутри вложенных функций), то должен быть способ получить эти переменные. Здесь уже оптимизация с регистрами виртуальной машины не прокатит. Однако, как мы видели, Python, например, забивает на `eval` и не сохраняет все подряд, а только нужное.

> *П.С. "Гармоничный" это в том смысле, что ES4 (ActionScript) и ES3/ES5 (JavaScript) смогут перейти на него бесконфликтно?*

Не уверен насчет ES4 😉 он мало коррелирует с Harmony. "Гармоничный" скорей всего, что все-таки договорились в свое время, в какую сторону двигаться.

NekR
2011-03-15

Хорошо, что JS/ES продолжает развиваться, это радует и всё таки открывает новые возможности для программистов.
В последнее время как раз стал задумываться над тем, как правильнее … выгоднее … пользоваться замыканиями. То есть стоит ли оставлять ссылку на родительскую область видимости, если мне из неё нужна будет только одна переменная, но там естественно будут ещё вспомогательные.
Даже не знаю как объяснить по простому все ситуации, попробую на примере кода, но всё равно это не совсем то.
Пример ака код из jQuery …

```
var objectType = (function(typeByClass){
    // ES5 forEach method
    'Object Array Function RegExp Number String'.split(' ').forEach(funct
        typeByClass['[object '+key+']'] = key.toLowerCase();
    });
    return function(obj){
        return typeByClass[Object.prototype.toString.call(obj)];
    }
}({}));
```

В этом примере замыкание, да, вполне оправданно, но в других случаях вспомогательных функций/переменных может быть гораздо больше и которые в конечной функции не будут замыкаться. Можно конечно сделать и по другому, но всё таки замыкания как раз хороши для таких примеров, по этому мне видится важным сохранения только нужных переменных.

П.С. Это скорее не вопрос, а рассуждения с знающим человеком 😉

Dmitry A. Soshnikov
2011-03-15

@NekR

> *мне видится важным сохранения только нужных переменных.*

Ну вот Python'у тоже так видится, и поэтому, повторю, он даже забивает на `eval` во внутренних функциях.

JS (по спецификации, и практически) — оставляет такую возможность. Однако, я думаю, движки вполне делают какие-то оптимизации, если нет `eval` 'а во вложенных функциях.

Поэтому, еще раз — Python оптимизирует момент разрешения идентификаторов и сохраняет только то, что нужно (однако процесс создания функции сложный — с парсингом внутренних функции и т.д). А JS оптимизирует момент создания функции и просто сохраняет ссылку на родительское окружение, оставляя его полностью в памяти.

NekR
2011-03-15

> *Ну вот Python'у тоже так видится, и поэтому, повторю*

Да, я читал статью и понял про Python, но Питон-Питоном – не переходить же из-за этого на него 😉
Ну в общем ясно, раз скорее всего ссылка всегда сохраняется то можно спокойно пользоваться.
Ждём следующую статью 🙂

Patrick Mahoney
2012-02-03

Closure

A closure is a pair of the function code and the saved at creation lexical environment.

Did you intend “bindings” or “environment” after saved here? Just curious, great article! *subscribes*

Dmitry Soshnikov
2012-02-04

@Patrick Mahoney

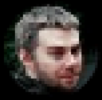
Yeah, it turned out a little bit confusing statement. I edited it. So a closure is a pair consisting of the function code the environment in which the function is created.

AMS
2012-04-10

Hi Dmitry great series of articles really learned a lot from it. What books and on papers do you recommend for some one to really get the background about how functional and non functional paradigms are implemented.

LR
2012-05-18

Is it possible to get an exact reference to the scope chain/environmental record frame? I am trying to create a function at runtime from a string without using eval, but the Function constructor is native code and will only form a closure over the global object’s variables. If there were some way I could pass the entire chain from the activation where the call to create this function takes place, then I could do it, but I'm having trouble finding information about where these properties are stored and how they're ordered. The JS inspector tools for browsers seem to break it down well.



Dmitry Soshnikov
2012-05-25

@LR

No, unfortunately (or fortunately — it depends), the spec doesn't allow to get direct access to the lexical environment object. However, in some implementations, such an access was — e.g. __parent__ property of some versions of Rhino.

Joshua Ramirez
2012-11-16

You're a genius. Thank you for writing straightforward explanations to such abstract concepts. It's a sign of mastery. Much appreciated.

shiva
2013-03-29

One of the best article I have seen on the subject – detailed, clear and couldn't be any simpler.

manohar
2015-08-14

Hats off dude, loved it !

Eric
2017-04-21

Awesome !

Utku
2017-07-12

"If an identifier is bound to an object, it references this object. The following use of the identifier results the object it's bound to."

I would have rewritten it as:

"If an identifier is bound to an object, it references this object. This situation (that is, the binding) causes the subsequent uses of the identifier to yield the object it's bound to."

What is your opinion on this?

Utku
2017-07-12

And it might be more correct to show the execution of "foo(20)" in two steps, since there are two statements in foo, as follows:

```
// create a new frame with formal
// parameters and local variables

fooFrame = {
  y: 20,
  z: undefined,
  outer: foo.environment
};

// and evaluate the code
// of the "foo" function

execute(foo.code, fooFrame); // 60
```

and remove the "z:30" from foo's frame in the first step and add "z = 30;" to the code part of foo.

Dmitry Soshnikov
2017-07-23

@**Utku**, thanks for the suggestions, all make sense. The detailed explanation of ECMAScript specifics (entering the context, and execution stages) are in the appropriate chapters, so in this case of the generic theory it's fine (to make it shorter), to put `z: 30` right away.

Marco
2018-03-15

Thanks Dmitry also for this article, it's very interesting.

I would have a question related to the "Activation record model" section, where you write "for the call-stack concept here stands the execution contexts stack and for the activation record stands (in ES3) the activation object". So, in Es3 we have the activation object, but in Es5? What's the equivalent of the "activation object" in Es5?

Thank you

Dmitry Soshnikov
2018-03-15

@**Marco**, in ES5 the are combined in the single "environments" model.

Write a Comment

READERS WHO SHARED THIS EVERYTHING ABOUT JAVA · THANK YOU!

RELATED CONTENT BY TAG ACTIVATION RECORD CALL-STACK CLOSURE DYNAMIC SCOPE ECMA-262-5 ENVIRONMENT FRAME FUNARG LEXICAL ENVIRONMENT LEXICAL SCOPE NAME BINDING STATIC SCOPE