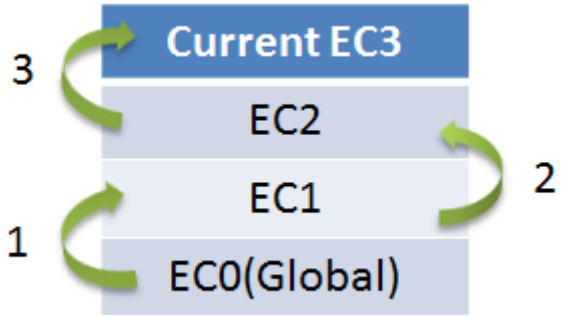
When we write our JavaScript program, usually define some variables and functions. How interpreter found them when executing the program? What happened behind them when we referenced these variables? In this article, let's exploring execution context defined in ECMA-262-3 standard, related mechanism will also be discussed.

*other versions: English-Blog, English-PDF, Chinese-Blog, Chinese-PDF

Definition

When control is transferred to ECMAScript executable code, control is entering an *execution context*. It is possible to enter a new execution context from current execution context, these contexts logically form a stack.
Execution context is created dynamically when program is running. For instance, function creates a new execution context when it is invoked. Repeatedly invoking functions (including recursion) will repeatedly create new execution contexts. Logical stack changing on program running when new function invoked, function returned, unhandled exception thrown, etc. Top of this logical stack always be the current running execution context, bottom is the global context.



EC logical stack in running program

Types of Executable Code

Depending on different types of executable code, execution context may be different. At first, let's understand three types of executable code.

a). Global Code

Global code is any source code text except code parsed as part of function body.

```
var i = 0;                  // global code
function foo() {            // definition part of foo is also global code
```

```
    var j = 1;                  // function body of foo is function code

}

var k = 2;                      // global code
```

Before any program code execution, global execution context is initialized, structure of the logical stack (LS) looks like:

[Pseudo code]

```
LS = {
  globalContext
}
```

b). Function Code

Function code is source code text that parsed as function body, but not includes code in nested inner function bodies.

```
var i = 0;                                  // global code
function foo() {                            // global code
    var j = 0;                              // function code of foo
    function inner() {                      // function code of foo
        var k = 0;                          // function code of inner
    }

    if (i++ == 0) foo();                    // function code of foo
}

foo();
```

[Pseudo code]

```
LS = {// foo invoked at 1st time

    <foo>functionContext,                   // current activated execution context
    globalContext

}
```

```
LS = {// foo invoked at 2nd time

    <foo>functionContextRecursively          // current activated execution context
    <foo>functionContext,                    // waiting <foo>...Recursively
                                             // returned, than activated

    globalContext

}
```

c). Eval Code

When invoking built-in eval function and its parameter is string (not function object), code in this string called “Eval Code”.

```
eval(“function foo2() {/* doSth(); */}”);          // eval code in string double quoted
eval(function foo3() {/* function code */});
```

*Definition in ECMA262-3 doc will be more rigorous, this article may compromise between easily understanding and rigorous definitions.

For eval code, execution context varies depend on calling context. For example, if function eval invoked in global context, the calling context is global context; if invoked in function context, the calling context is this function context.

```
eval(‘var x = 10’);                          // affect globalContext
(function foo() {

  eval(‘var y = 20’);                        // affect <foo>functionContext,
}());

alert(x);                                    // 10
alert(y);                                    // runtime error，y is not defined


[Pseudo code]

LS = {                                       // invoke eval('var x = 10');
    evalCallingContext : globalContext,

  globalContext
```

```
}
```

```
LS = {                                          // eval('var x = 10'); ended
    globalContext                               // evalCallingContext pop up
}
```

```
LS = {                                          // invoke foo, <foo>functionContext
                                                // pushed
    evalCallingContext : <foo>funcitonContext,  // invoke eval('var y = 20');
                                                // evalCallingContext pushed
    globalContext
}
```

Variable Instantiation

Every execution context has its own *variable object* (VO), in the initialization stage, defined variables and functions will be added to variable object as properties. For function code, formal parameters also be added to variable object.

*In ECMA-262-5, terminology "variable object" will be replaced by new "lexical environment" which we'll discuss in next chapter.

For more details, variable object initialized as steps below in order:

a). Parameter Process

For function code, every parameter passed in will be added to variable object as a property whose name is the identifier of the function, value supplied by caller. If parameters are defined but not passed in, new property of variable object still be created, but the value is *undefined*.

```
function  foo(x, y, z) {                    [Pseudo code]

}                                           VO(<foo>functionContext) = {
```

```
foo(1, 2);
```

```
    x : 1,
    y : 2,
    z :  undefined
}
```

b). Function Definition Process

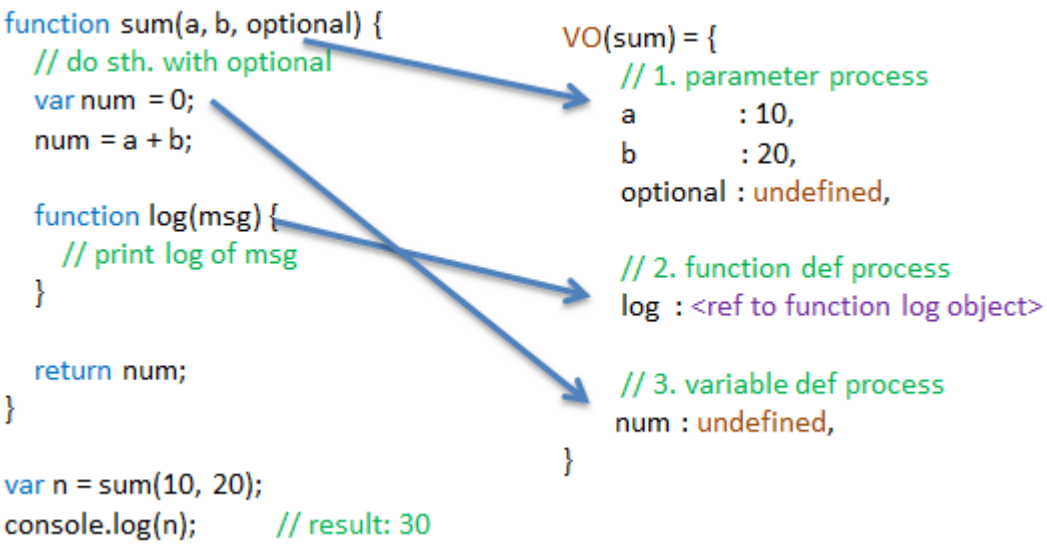
For every function definition (not function expression) creating new property of variable object whose name is identifier, value is this function object.

```
function  foo(x, y, z) {
    function  f() {
    }
}
foo(1, 2);
```

```
[Pseudo code]
VO(<foo>functionContext) = {
    x : 1,
    y : 2,
    z :  undefined,
    f :  <ref to function f object>
}
```

c). Variable Definition Process

For every variable definition, creating new property of variable object whose name is identifier of variable, value is undefined. Caution please, assignment operation is not handled in this stage, for instance, “var i = 0;” is separated into two parts: definition and assignment. Variable object initialization only process variable definition, assignment statement “i = 0” executed in next stage.



d). Resolving Naming conflicts

Conflicts in parameter process:

If the parameter identifier conflicted, its value is decided by the last one.

e.g.: !function(x, y, y){alert(y);}(i1, 2, 3) // alert 3

If second y is not passed in, the value of y supplied by the last y parameter, namely undefined.

e.g.: !function(x, y, y){alert(y);}(1, 2) // alert undefined

*on the left of function, "!" used for making function as expression instead of function definition. Similar to (function(){}()), but shorter.

Conflicts in function definition process:

If function identifier existed in variable object, **replace** it.

e.g.: !function(x) {function x() {};alert(x);}(1) // alert function

Conflicts in variable definition process:

If variable identifier existed in variable object, **ignore** it.

```
e.g.: !function(x) {function x() {};var x; alert(x);}(1)    // still alert function
```

Pay attention again that variable object initialization not handles variable assignment statement. We'll introduce what happened after execution context initialization later.

```
e.g.: !function(x) {function x() {};var x = 1; alert(x);}(1)    // alert 1
```

Scope Chain

Not only variable object but also scope chain and this binding are initialized in execution context initialization stage. Scope chain is a list of objects, used for identifier lookup. When execution context finished initialization, only *with* statement and *try* clause can change scope chain in runtime.

```
[Pseudo code]

activeExecutionContext = {

        VO: { },

        scopeChains: [VO_1, VO_2, …, VO_n ]

        this: thisValue

}
```

For global context, only global object is placed into scope chain. For function context, scope chain looks like a list consist of variable objects in every execution context of logical stack. Scope chain will be used for variable search, in logical stack, from top to bottom.

Global Object

Before entering any execution context or any program code execution, a global and unique *global object* is created, built-in properties and

functions are added such as: Math, String, Date, parseInt, etc. Properties defined by host are also added to global context, for instance, HTML document object model, a property "window" will be added. The value is just the global object itself.

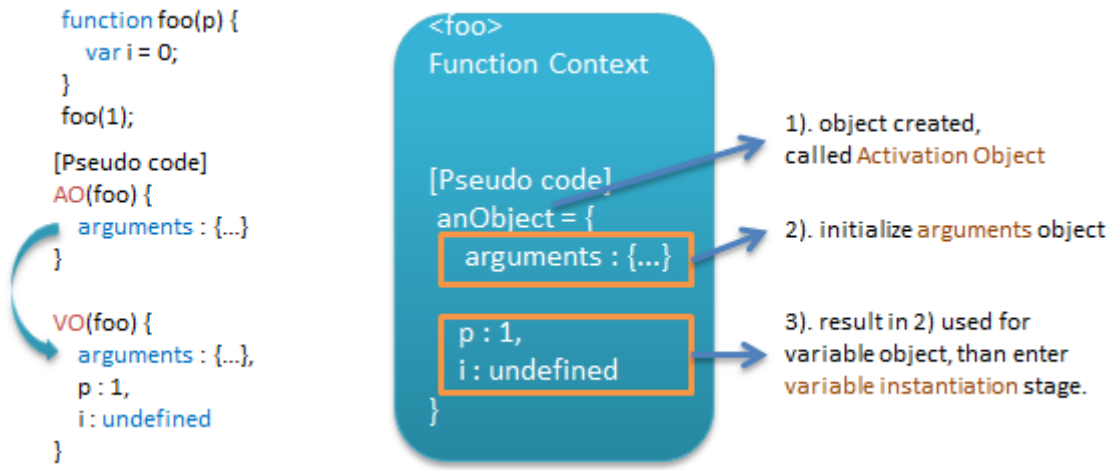
Global object is inner object in mechanism, cannot be directly manipulated by program code, as there's a reference "window" in HTML DOM, we can indirectly interact with global object.

[Pseudo code]

```
globalObject  = {
        Math :  {…},
         String :  {…},
        parseInt : <function>,
        …
         window :   globalObject
}
```

Activation Object

For function context, an *activation object* is created before variable object initialization. This activation object will be initialized with property *arguments*, than this activation object used as variable object and go on entering variable instantiation stage.



Example

First step: entering an execution context, execution context initialized with 1).variable object, 2).scope chain, 3).this binding.
Second step: executing code and interacting with execution context.

```
function  test(a, b) {
    var  c = 10;
    function  d() {}
    var  e =  function _e() {};
    (function x() {});
    b = 20;
}
test(10);
```

```
[Pseudo code] - <test>scopeChain
ScopeChain(<test>functionContext) = {
    VO(<test>functionContext),
    globalObject
}
[Pseudo code] - VO initialize
VO(<test>functionContext) = {
    a : 10,
    b :  undefined,
    d :  <ref to function d object>,
    c :  undefined,
    e :  undefined
```

```
}

[Pseudo code] - VO changing

VO(<test>functionContext) = {

    a : 10,

    b :  20,

    d :  <ref to function d object>,

    c :  10,

    e :  <ref to function _e object>

}
```

Conclusion

Execution context defined in ECMA-262-3 varies on three types of executable code:

1. For global context, initialize global object, than this object is used as variable object for variable instantiation.

2. For function context, initialize activation object, than this object is used as variable object for variable instantiation.

3. For eval context, depending on the calling context:

    a) If eval is invoked in global context, eval code interacting with global context in initialization and execution.

    b) If eval is invoked in function context, eval code interacting with this function context in initialization and execution.

In next chapter<<JSinDeep2: Exploring Execution Context in ECMA-262-5>>, we'll discuss lexical environment. Compared with ES3, there are

differences on performance, object model, ways processing with statement and try clause, etc.

About

Author：Bosn Ma
Intro： I am Peking based web developer, interested in JavaScript, J2EE and MySQL. Currently I am working at Taobao as Senior Web R&D Engineering
E-mail: bosn@outlook.com
GitHub：https://github.com/bosnma/JsInDeep

Reference

ECMA262-3 Doc
http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf

<<ECMA-262-3 in detail. Chapter 1. Execution Contexts>>

http://dmitrysoshnikov.com/ecmascript/chapter-1-execution-contexts/

<<ECMA-262-3 in detail. Chapter 2. Variable object>>

http://dmitrysoshnikov.com/ecmascript/chapter-2-variable-object/