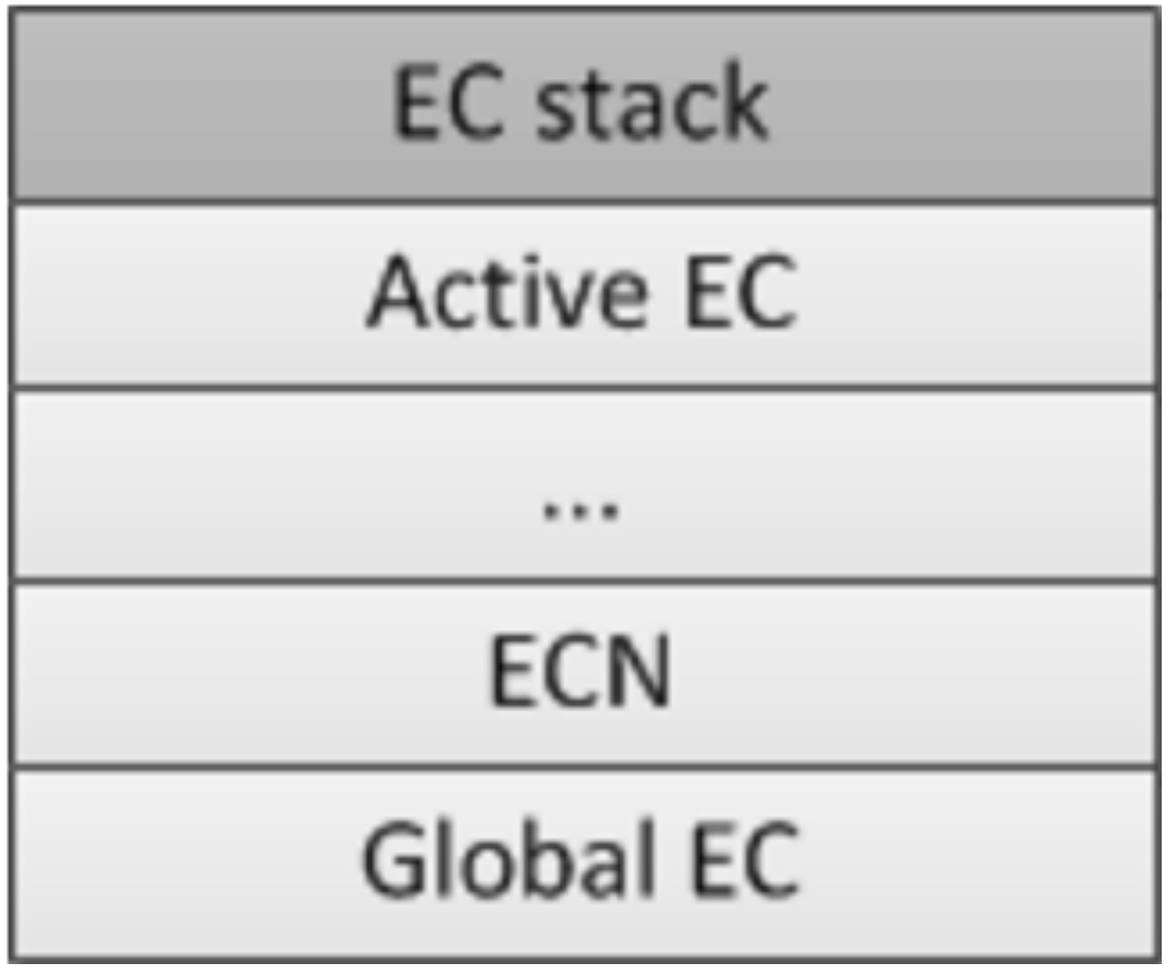
Rodrigo Branas – @rodrigobranas - http://www.agilecode.com.br

Execution Context

O Execution Context, ou EC, é o ambiente onde o código é executado, composto por 3 partes: variable object, scope chain e this.

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

É organizado em uma pilha, ou stack


EC stack
Active EC
...
ECN
Global EC

O código pode ser executado em 3 tipos de contextos: global, function ou eval. Os contextos são empilhados conforme o programa vai sendo executado.

```
1.    var x = 10;
2.    var y = function () {
3.      console.log(x);
4.      var x = 100;
5.      console.log(x);
6.    };
7.    y();
```

Inicialização das variáveis no Variable Object

```
1.    var x = 10;
2.    var y = function () {
3.      if (x === 10) {
4.        var z = 100;
5.      }
6.      console.log(z);
7.    };
8.    y();
```

Inicialização das variáveis no Variable Object

```
1.    function x() {
2.      y();
3.      function y() {
4.        console.log("y");
5.      };
6.    };
7.    x();
```

Inicialização das funções no Variable Object

```
1.  function x() {
2.    y();
3.    var y = function () {
4.      console.log("y");
5.    };
6.  };
7.  x();
```

Inicialização das funções no Variable Object

```
1.    var x = 10;
2.    function y () {
3.      function z() {
4.        console.log(x);
5.      }
6.      z();
7.    }
8.    y();
```

Localização de variáveis livres no Scope Chain

```
1.    var x = {
2.      y: 10,
3.      getY: function () {
4.        console.log(this.y);
5.      }
6.    };
7.    x.getY();
```

Utilização do thisValue

```
1.  var x = {
2.    y: 10,
3.    getY: function () {
4.      (function () {
5.        console.log(this.y);
6.      })();
7.    }
8.  };
9.  x.getY();
```

Utilização do thisValue

```
1.    var x = {
2.      y: 10,
3.      getY: function () {
4.        var that = this;
5.        (function () {
6.          console.log(that.y);
7.        })();
8.      }
9.    };
10.   x.getY();
```

Utilização do thisValue

```
1.  var x = {
2.    y: 10,
3.    getY: function () {
4.      (() => {
5.       console.log(this.y);
6.      })();
7.    }
8.  };
9.  x.getY();
```

Utilização do thisValue

Closures

```
1.      function foo() {
2.        var x = 10;
3.        return function bar() {
4.          console.log(x);
5.        };
6.      }
7.
8.      var returnedFunction = foo();
9.
10.     var x = 20;
11.
12.     returnedFunction();
```

upward funarg problem

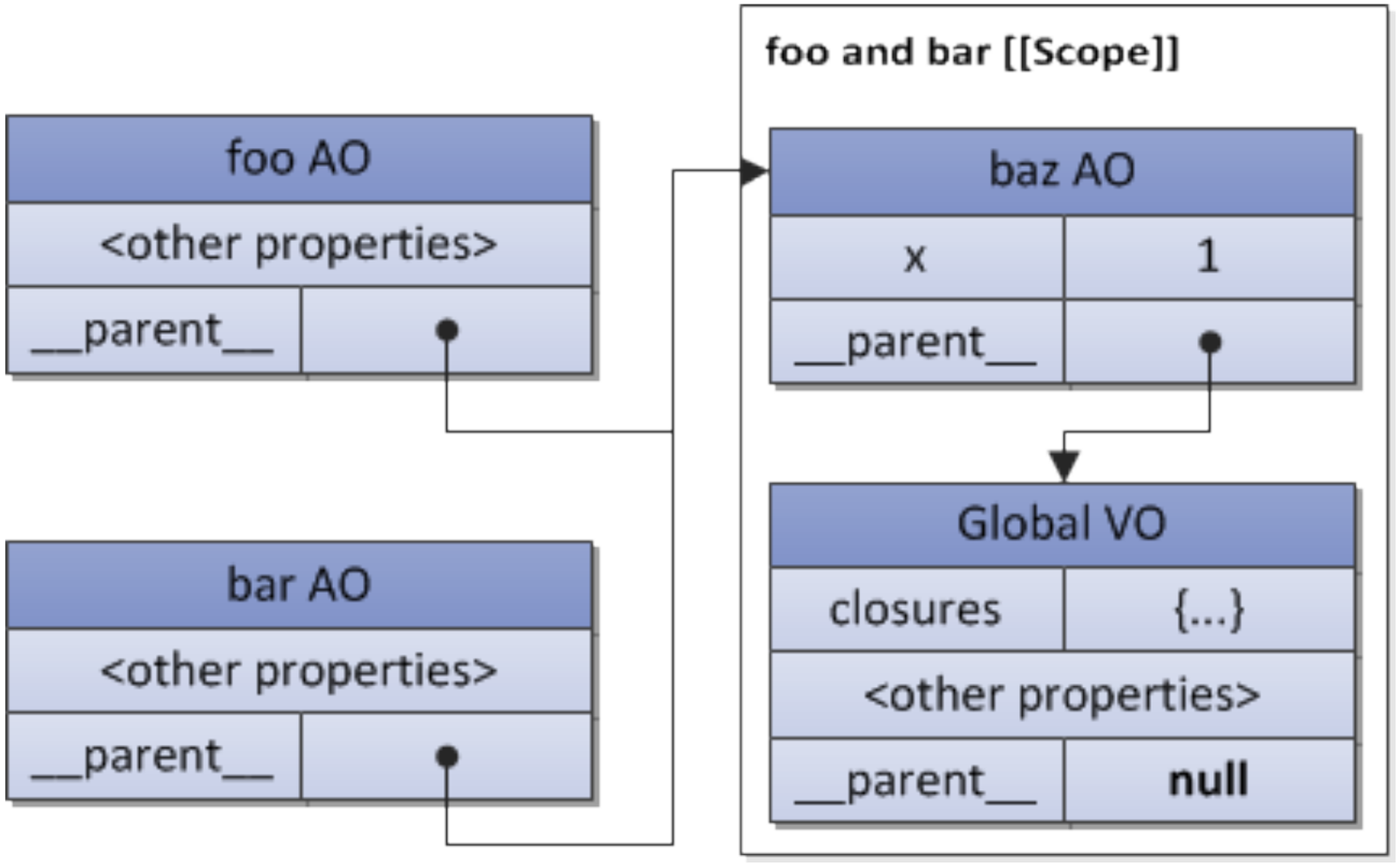
```
1.      var x = 10;
2.
3.      function foo() {
4.        console.log(x);
5.      }

6.      (function (bar) {
7.        var x = 20;
8.        bar();
9.
10.     })(foo);
```

downward funarg problem

```
1.      function baz() {
2.        var x = 1;
3.        return {
4.          foo: function foo() { return ++x; },
5.          bar: function bar() { return --x; }
6.        };
7.      }
8.
9.      var closures = baz();
10.
11.     console.log(
12.       closures.foo(),
13.       closures.bar()
14.     );
```

Compartilhando o mesmo Scope


foo and bar [[Scope]]
foo AO
<other properties>
__parent__
baz AO
x
1
__parent__
bar AO
<other properties>
__parent__
Global VO
closures
{...}
<other properties>
__parent__
null

```
1.      var frutas = ["morango", "banana", "laranja", "abacaxi",
        "manga"];
2.
3.      var saladaDeFrutas = {};
4.
5.      for(var i = 0; i < (frutas.length - 1); i++) {
6.          saladaDeFrutas[frutas[i]] = function () {
7.              console.log("Meu nome é " + frutas[i]);
8.          };
9.      }
10.
11.     saladaDeFrutas.morango();
12.     saladaDeFrutas.banana();
13.     saladaDeFrutas.laranja();
14.     saladaDeFrutas.abacaxi();
```

Criando funções dinamicamente

```
1.      var frutas = ["morango", "banana", "laranja", "abacaxi",
        "manga"];
2.
3.      var saladaDeFrutas = {};
4.
5.      for(var i = 0; i < (frutas.length - 1); i++) {
6.          saladaDeFrutas[frutas[i]] = function (fruta) {
7.              return function () {
8.                  console.log("Meu nome é " + fruta);
9.              };
10.         }(frutas[i]);
11.     }
12.
13.     saladaDeFrutas.morango();
14.     saladaDeFrutas.banana();
15.     saladaDeFrutas.laranja();
16.     saladaDeFrutas.abacaxi();
```

Criando funções dinamicamente, dentro de um novo execution context

bind

Toda função possui a operação bind() que serve para definir o escopo interno de uma função.

função.bind(escopo)

Tail call optimization

Quando a última coisa que uma função execute é apenas o retorno de uma chamada para uma outra função, temos uma tail call. Se a chamada for para a mesma função, o que opcional, temos uma tail-recursive call.

```
1.    let a = function (value) {
2.      return b(value);
3.    };
4.
5.    let b = function (value) {
6.      return c(value);
7.    }
8.
9.    let c = function(value) {
10.     console.log(value);
11.   }
12.   a("JavaScript");
```

Como ficaria a pilha?

```
1.    let a = function (value) {
2.      console.trace();
3.      return b(value);
4.    };
5.
6.    let b = function (value) {
7.      console.trace();
8.      return c(value);
9.    }
10.
11.   let c = function(value) {
12.     console.trace();
13.     console.log(value);
14.   }
15.   a("JavaScript");
```

Utilizando o console.trace para monitorar a pilha

Requisitos para a otimização

- "use strict"
- O último statement precisa ser a chamada de uma função, sem qualquer interação após o retorno
- Habilitar o interpretador com --harmony

```
1.    "use strict";

3.    function fib (n) {
4.      if (n <= 1) return n;
5.      console.trace();
6.      return fib(n - 1) + fib(n - 2);
7.    }
8.    fib(10);
```

Sem tail call optimization

```
1.    "use strict"

3.    function fib (n) {
4.      return fibI(n, 1, 0);
5.    }

7.    function fibI (n, a, b) {
8.      if (n === 0) return b;
9.      console.trace();
10.     return fibI(n - 1, a + b, a);
11.   }

13.   fib(10);
```

Com tail call optimization