

Java Script

Closure is when a function remembers its lexical environment even when the function is executed outside that lexical scope

Kyle Simpson (<https://www.linkedin.com/in/getify>)

Замыкания - это когда функция "запоминает" свой Lexical Enviroment, даже если ее вызов осуществляется все ее Lexical Scope

Порядок работы функций

Вариант 1 - это не замыкание

```
function test(){ }
```

В программе нет вложенных функций и нет ссылок на этот объект. После того как функция возвратит управление, она будет утилизирована сборщиком мусора.

Это явно не замыкание

```
function nonClosure(){  
  // fExC = {date : undefined, OE: window}  
  var date = new Date();  
  // fExC = {date : object, OE: window}  
  return date.getMilliseconds();  
} // Garbage Collector will delete fExC
```

Переменная `date` не будет доступна после отработки функции, так как объект `fExC` будет уничтожен


Вариант 2 - это не замыкание

```
function outer(){ // LEnv_1
  // fExC_1 = {a: undefined, inner: function(){...}, OE: window };
  var a = 5;
  // fExC_1 = {a:5, inner:function(){...}, OE: window };

  function inner (){
    // fExC_2 = {b:undefined, c:LEnv_1 };
    var b = a * 5;
    // fExC_2 = {b:25, c:LEnv_1 };
    } // delete fExC_2
    inner();
  } // delete fExC_1
```

В программе есть вложенные функции. Каждая из этих функций имеет ссылку на свою цепочку областей видимости, а цепочка будет ссылаться на объекты с локальными переменными. После отработки функции будут утилизированы сборщиком мусора, а вместе с ними будут утилизированы и объекты с локальными переменными, на которые они ссылались.

```
function testClosure() {  
    var x = 5  
  
    function closeX() {  
        return x  
    }  
    return closeX  
}
```



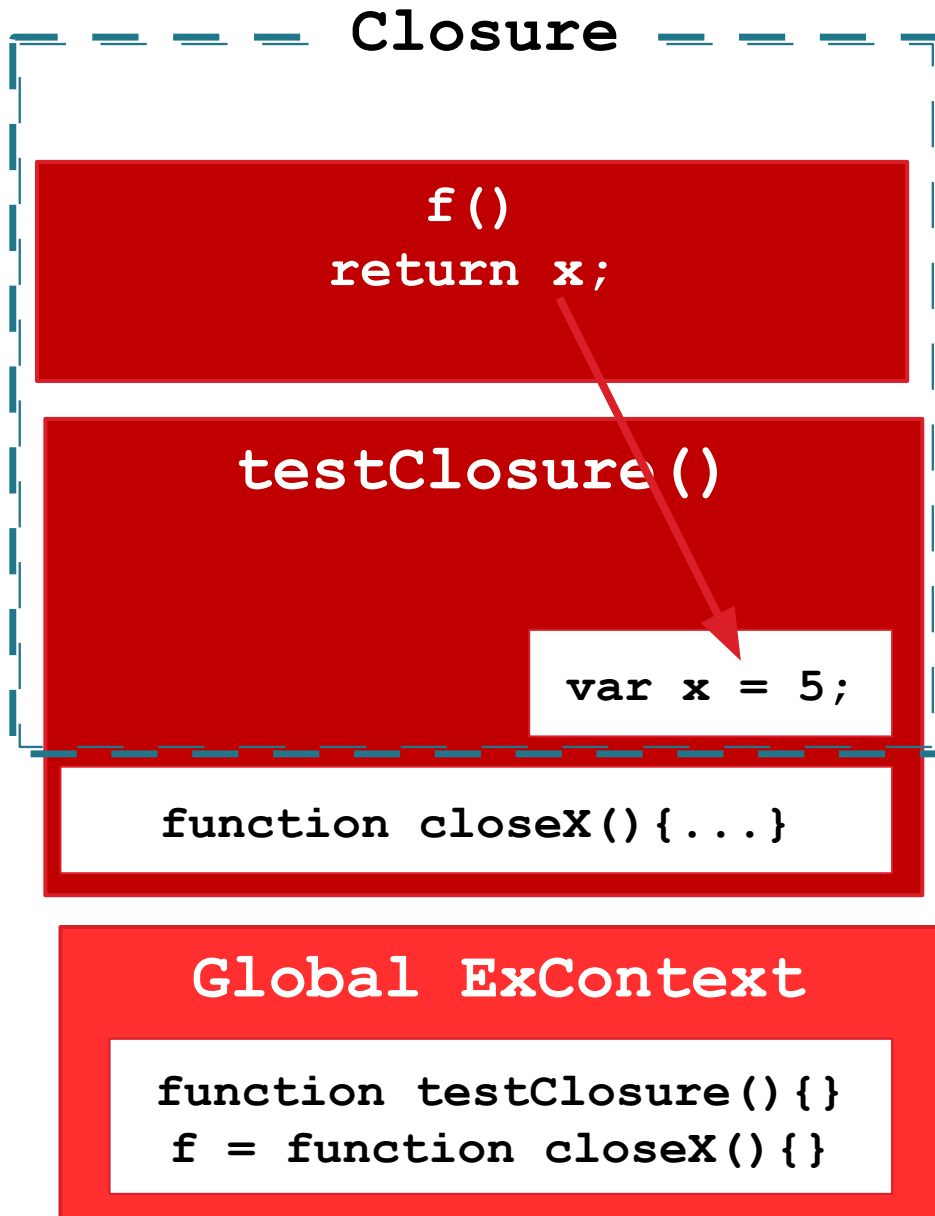
```
var checkLocalX = testClosure();
```

```
checkLocalX();
```

⇒ 5

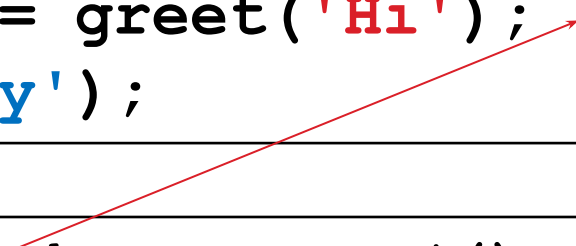
```
function testClosure(){
  var x = 5;

  function closeX(){
    return x;
  }
  return closeX;
}
var f = testClosure();
f();
```



```
function greet(say){  
    return function(name){  
        print(say + " " + name);  
    }  
}
```

```
var sayHi = greet('Hi');  
sayHi('Tomy');
```



В этом месте программы функция `greet()` уже отработала, и ее `Execution Context` должен быть уже уничтожен, включая переданный ей при вызове аргумент `say` – строку `'Hi'`.

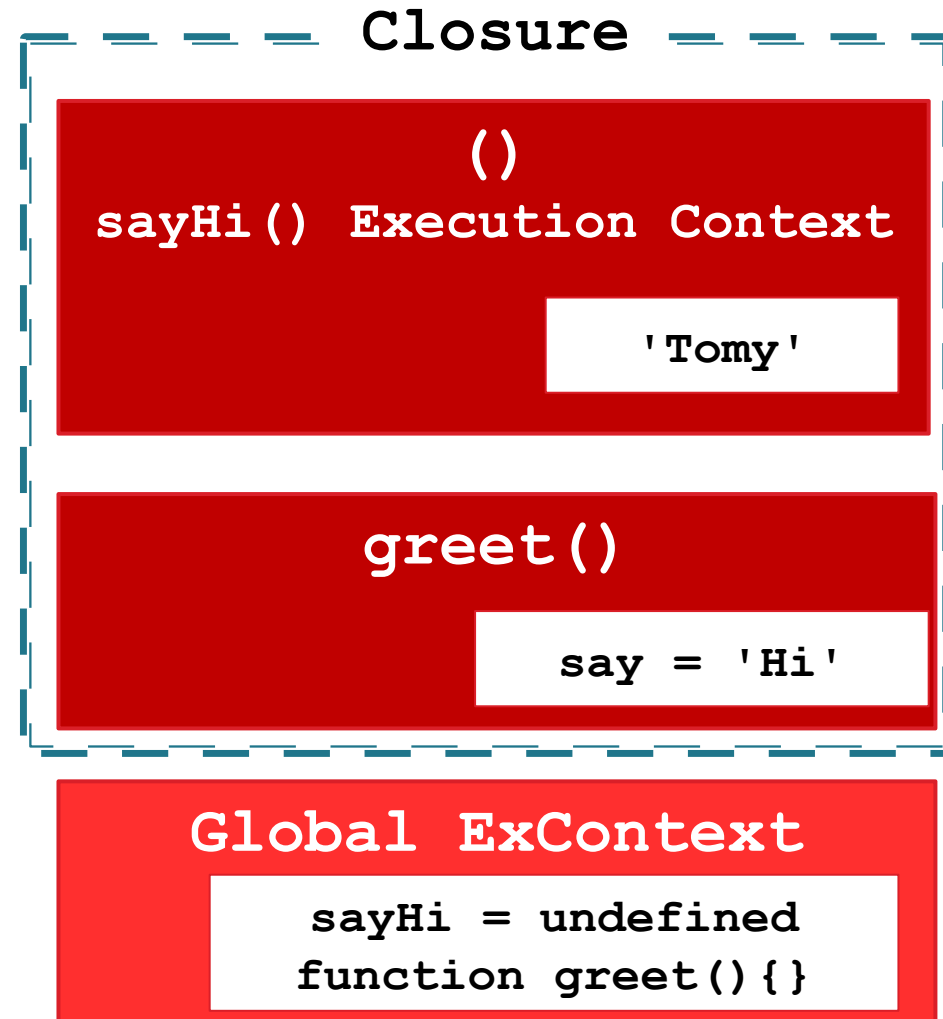
И в точке вызова функции `sayHi()` этой переменной быть не должно, но тем не менее она не была уничтожена и доступна в функции `sayHi()`.

Это произошло благодаря замыканию (`closure`).

Когда выполняется строка `print(say + " " + name)` интерпретатор начинает искать переменную `say` - по цепочке `LexicalEnvironment`, то есть в родительской области. И благодаря *замыканию* переменная `say` не была уничтожена.

```
function greet(say){  
  return function(name){  
    print(say + " " + name);  
  }  
}
```

```
var sayHi = greet('Hi');  
sayHi('Tomy');
```




```

function buildFunc() {
  var arr = [];
  var i;
  for(i=0; i < 3; i++) {
    arr.push(function () {
      print(i);
    });
  }
  return arr;
}
var fs = buildFunc();

```

```

fs[0](); // 3
fs[1](); // 3
fs[2](); // 3

```

Execution Context

fs[0]()

buildFunc()

i

3

arr

[f0, f1, f2]

Global ExecutionContext

fs = undefined
function buildFunc() {}

Вариант 3 – это замыкание

```
var myVar; // window {myVar:undefined}
function outer(){ // LE_1
  // fExC_1 = {a: undefined, inner: function(){...}, OE: window };
  var a = 5;
  // fExC_1 = {a:5, inner:function(){...}, OE: window };

  function inner(){
    // fExC_2 = { OE: LE_1 };
    return a * 5;
  }
  myVar = inner; // window {myVar:function inner(){...}}
}
outer();
alert( myVar() ); // выводится 25
```

В программе есть вложенные функции. Внешняя функция определяет вложенную функцию и сохраняет ее в переменной **myVar**.

Образуется внешняя ссылка на вложенную функцию. Объект **fExC_1** функции **outer()** не будет утилизирован сборщиком мусора, так как функция **inner** хранится в переменной **myVar**, и ее **fExC_2** имеет ссылку **[[Scope]]** на внешнюю область видимости (на **LE_1**). Таким образом интерпретатор не удаляет объекты переменных **fExC_1** и **fExC_2**.

Вариант 4 - это замыкание

```
function outer(){ // LEnv_1
    // fExC_1 = {a: undefined, OE: window };
    var a = 5;
    // fExC_1 = {a: 5, OE: window };
    return function (){
        // fExC_2 = {OE: LEnv_1 };
        return a * 5;
    }
}

var f = outer(); // что сейчас находится в переменной f ?
f();
```

В программе есть вложенные функции.

Ситуация аналогична варианту 3 – объекты переменных функций не будут утилизированы сборщиком мусора.

В замыкания включаются

- параметры самой функции;
- все переменные внешней области видимости, даже те, которые были декларированы позже;
- все переменные, **которые определены** в области видимости самой функции.

```
var outerValue = "outerVal";
```

closure.html

```
var later;
```

```
function outerFunction(){
```

```
    var innerValue = "innerVal";
```

```
    function innerFunction(param) {
```

```
        var num = 25;
```

```
        assert(outerValue, "I can see outerValue");
```

```
        assert(innerValue, "I can see innerValue");
```

```
        assert(param, "I can see param");
```

```
    }
```

```
    later = innerFunction;
```

```
}
```

```
function makeUser(name) { // LEnv_1
  fExc_1 = {name: Thomas, newName: undefined, OE: window}
  var newName = name.toUpperCase();
  fExc_1 = {name: Thomas, newName: THOMAS, OE: window}

  return function() {      // fExc_2 = {OE: LEnv_1 }
    newName += "!";
    assert(true, newName);
  }
}

var user = makeUser("Thomas ");
user(); // Thomas !
user(); // Thomas !!
user(); // Thomas !!!
```

Вывод - в объекте *fExc_2* находятся не значения внешних переменных, а ссылки на них, поэтому значения этих внешних переменных могут изменяться в процессе выполнения скрипта независимо от замыкания.

```
var hello = "Hello ";

function sayHi(name){
    return function(){
        alert(hello + "My name is " + name);
    }
}

var f = sayHi("Bill");

hello = "Hello there !!! ";

f(); // Hello there !!! My name is Bill
```

```
function greet(lang) {  
  return function(name) {  
    if(lang == "en") {  
      return "Hello" + name;  
    }  
  
    if(lang == es) {  
      return "Hola" + name;  
    }  
  }  
}
```

```
var greetEn = greet("en");  
var greetSp = greet("es");  
  
greetEn("John");  
greetSp("John");
```

greetSp()

name = John



greet()

lang = es

greetEn()

name = John



greet()

lang = en

Global ExecutionContext

greet, greetEn, greetSp

```
function makeCounter() {  
    var count = 0;  
    return function() {  
        return ++count;  
    }  
}
```

```
var c1 = makeCounter();  
var c2 = makeCounter();  
  
c1();  
c1();  
alert(c1()); // выводится 2  
  
c2();  
c2();  
c2();  
alert(c2()); // выводится 4
```

В данном примере функция, возвращаемая **makeCounter**, использует переменную **count**, которая сохраняет нужное значение между ее вызовами.

Важно !!!

Счетчики – независимы друг от друга, то есть для каждого из них при запуске функции создается свое замыкание – то есть своя область памяти в которой хранятся значения переменных.

Задание :

1. Используя код счетчика написать функцию, которая бы возвращала объект для работы с счетчиком, и имела бы методы

<code>set(val)</code>	Установка значения счетчика в значение <code>val</code>
<code>reset()</code>	Сброс значения счетчика в 0
<code>getNext()</code>	Увеличение значения счетчика на 1

Использование:

```
var c = makeCount();  
c.getNext();  
c.reset();  
c.set(5);
```

2. Изменить код так, чтобы убрать метод `getNext()`, и использование было таким:

```
var counter = makeCounter();  
counter(); // увеличивает значение счетчика на 1  
counter.reset(); // сбрасывает значение счетчика в 0  
counter.set(5); // Установка значения счетчика в значение 5
```

Примеры замыканий

```
1 function foo() {  
2     var bar = "bar"  
3  
4     function baz(){  
5         console.log(bar)  
6     }  
7  
8     bam(baz)  
9 }  
10 function bam(baz){  
11     baz()  
12 }  
13  
14 foo()
```

```
1 function foo() {  
2     var bar = "bar"  
3  
4     setTimeout(function(){  
5         console.log(bar)  
6     }, 1000)  
7 }  
8  
9 foo()
```

```
1 function foo() {  
2     var bar = "bar"  
3  
4     $('#btn').click(function(e){  
5         console.log(bar)  
6     })  
7 }  
8  
9 foo()
```

Shared scope

```
1 function foo() {  
2     var bar = 0  
3  
4     setTimeout(function(){  
5         console.log(bar++)  
6     }, 100)  
7     setTimeout(function(){  
8         console.log(bar++)  
9     }, 200)  
10 }  
11  
12 foo()
```

Вопрос - это замыкание или нет ?

```
1 var foo = (function(){  
2     var o = {bar: "bar"}  
3  
4     return {obj: o}  
5  
6 }())();  
7  
8 console.log(foo.obj.bar) // ???
```

Выводы

1. Для того, чтобы осуществлять несколько действий одновременно с несколькими объектами (анимация, AJAX, события) нам не подходят глобальные переменные.

Например в нашем коде все 3 переменные – `timer`, `ссылка на счетчик тактов tick`, и `ссылка на элемент DOM el` должны сохраняться на протяжении анимации и ни в коем случае не должны быть в глобальной области видимости.

2. Переменные, находящиеся в замыкании могут обновляться.

То есть замыкание это не просто одномоментный снимок состояния области определения функции, а активная сущность, которая может изменяться в течении времени существования замыкания.

Выводы

1. По умолчанию при использовании системы привязки событий функцией `addEventListener()` к обработчику события привязывается сам объект, который генерирует событие.
2. Используя функции `call()` и `apply()` мы можем привязывать к обработчику событий нужный нам контекст.
Но при этом привязку надо осуществлять через анонимную функцию, которая создаст замыкание и запомнит этот контекст в замыкании.

Используя такой синтаксис можно создавать временные области видимости функций – ведь области видимости переменных определяются границами функций.

Например

```
(function () {  
    var count = 0;  
    document.addEventListener("click", function () {  
        alert(count++);  
    }, false);  
})();
```

Так как вызов функции происходит немедленно, то событие `click` сразу же привязывается к `document`.

Создается замыкание для обработчика события `click`, в которое входит переменная `count`.

Таким образом переменная `count` хранится самостоятельно с обработчиком события и соотносится только с данным обработчиком и ничем более.

Еще один пример

```
document.addEventListener("click",  
    ( function(){  
        var count = 0;  
        return function(){  
            alert( ++ count);  
        };  
    }) (), false);
```

Опять мы создаем функцию, вызов которой происходит немедленно, но в этот раз она возвращает внутреннюю функцию, которая передается в обработчик события `click`.
Здесь внутренняя функция имеет доступ к переменной `count` через замыкание.

Thank you for attention



