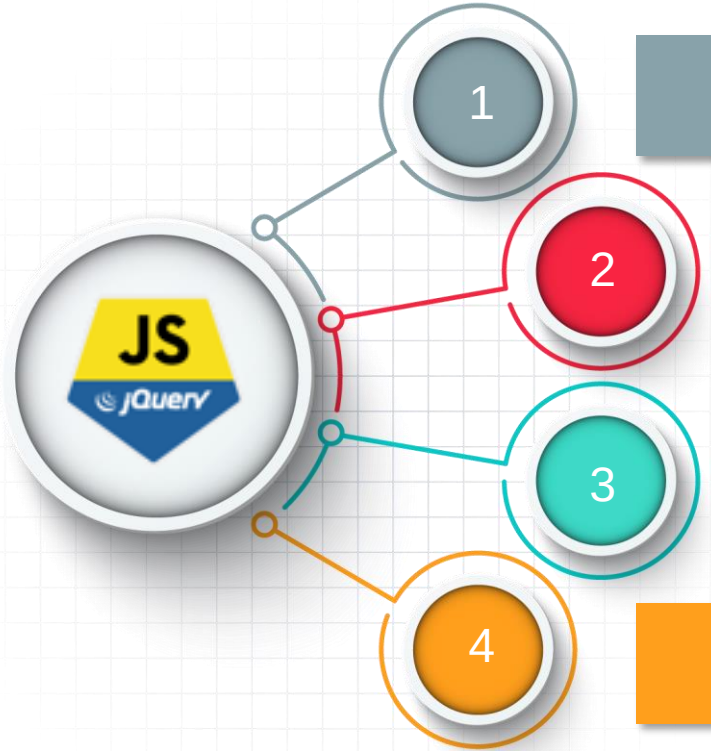
edureka!

JavaScript & JQuery

Course Outline



Introduction to JavaScript

Functions and Internal Memory Management

Event Handling and Objects

Libraries and Frameworks

edureka!

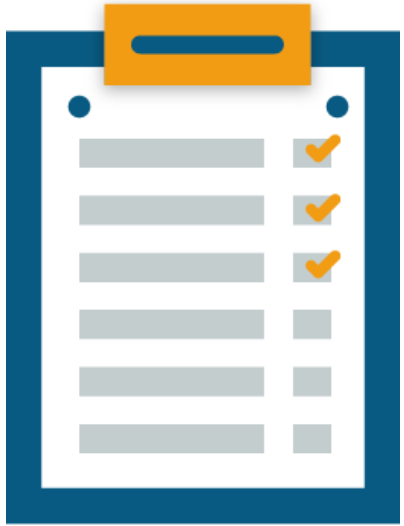
Module 2 - Functions and Internal Memory Management

edureka!

Objectives

After completing this module, you should be able to:

- Minimize your code size by using reusable codes i.e. Functions
- Analyse Internal Memory Management in JavaScript
- Identify the type of declaration that should be applied for a variable
- Explain the concept of Variable Shadowing and Closures
- Understand the role of Garbage Collectors in JavaScript



JavaScript – Functions

Functions are building blocks of JavaScript, which can be reused many times with different arguments to produce different results

Functions

- A set of statements (function body) that performs a task or calculates a value
- Defined somewhere in the scope
- Invoked by a function call

Function Parameters

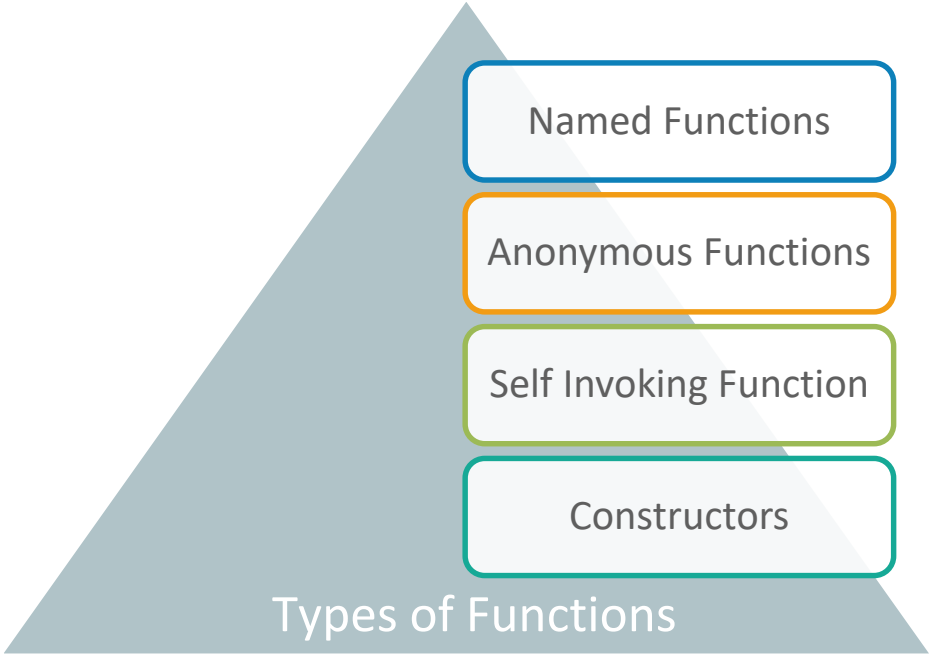
- A function parameter is a value, which is accepted by the function
- Parameters in a function call are arguments
- Arguments are parameters, which are passed to the functions by value

Return Statement

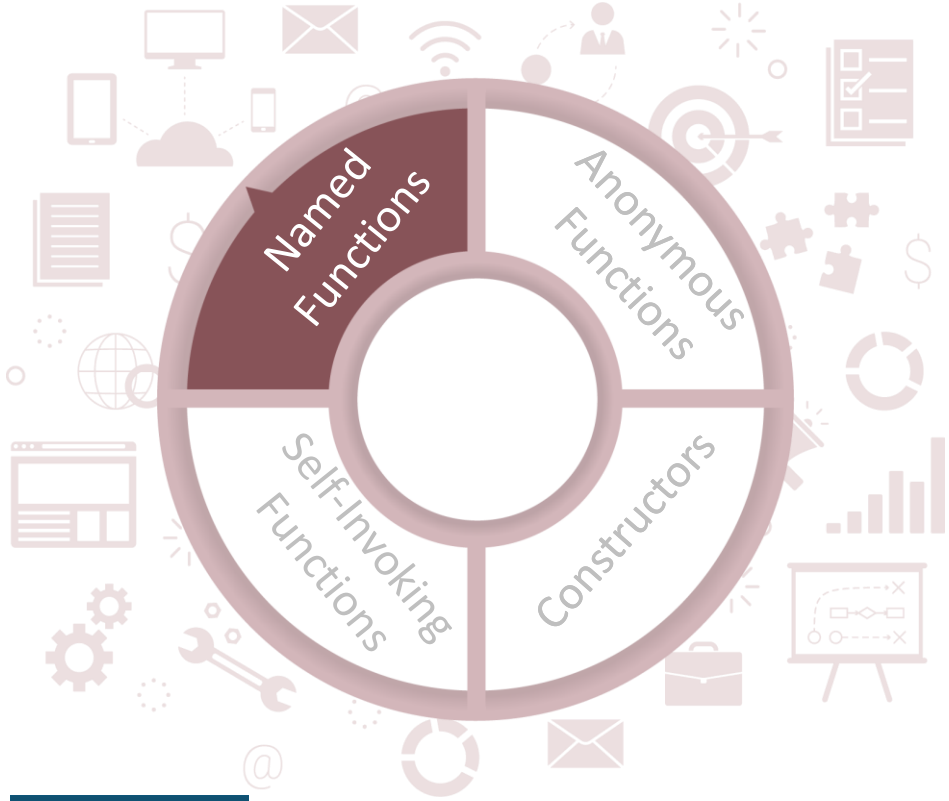
- Every Function should have a return statement in its body, which returns a specific value
- If a function does not have a return statement, a default value is returned
- The default value that is returned is the value **undefined**

Types Functions

On the basis of how it is defined and called, functions are categorized in 4 types :



Named Functions

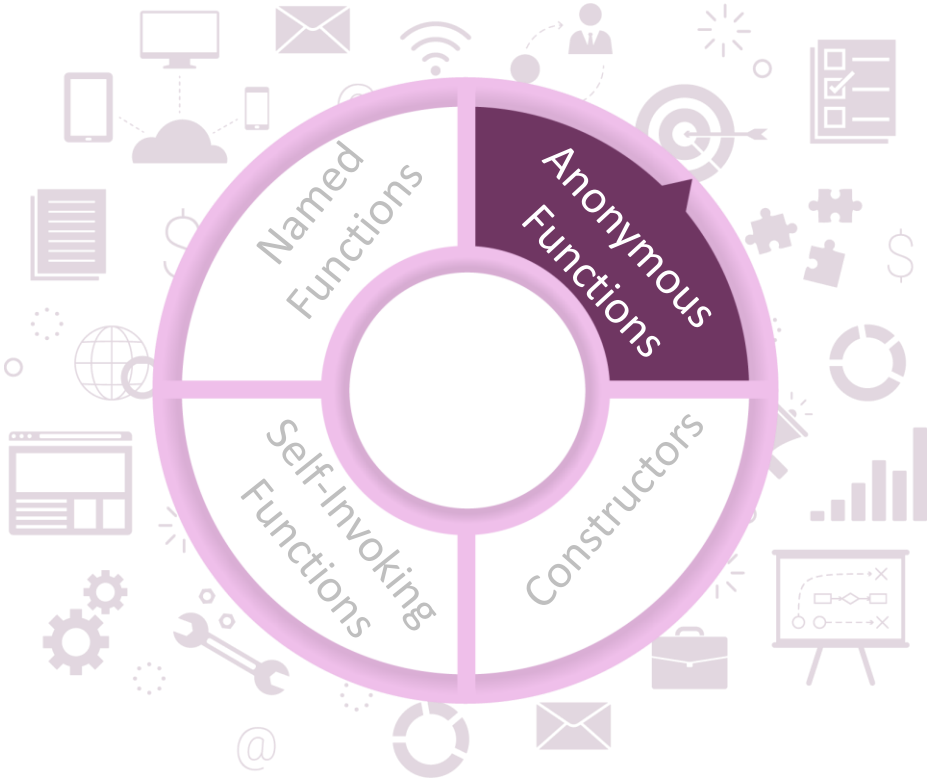


- Has a unique name
- Can be called/used in multiple places
- Example:

```
function addNum(a, b)
{
/*function definition with parameters a and b*/
return  a+b;
return statements
}
var d= addNum(4,3);
/* function call, returns value 7 */
var c= addNum(5,5);
/* function call, returns value 10*/
```

edureka!

Anonymous Functions



- Does not have a name
- Can be used at one place only (when it is called immediately after it is defined, or actual argument to function)
- The function defined is used as an expression
- Can be stored in a variable
- Passed as an actual argument to a function
- Can be returned as a value by a function

Anonymous Function: Example

- Stored in a variable

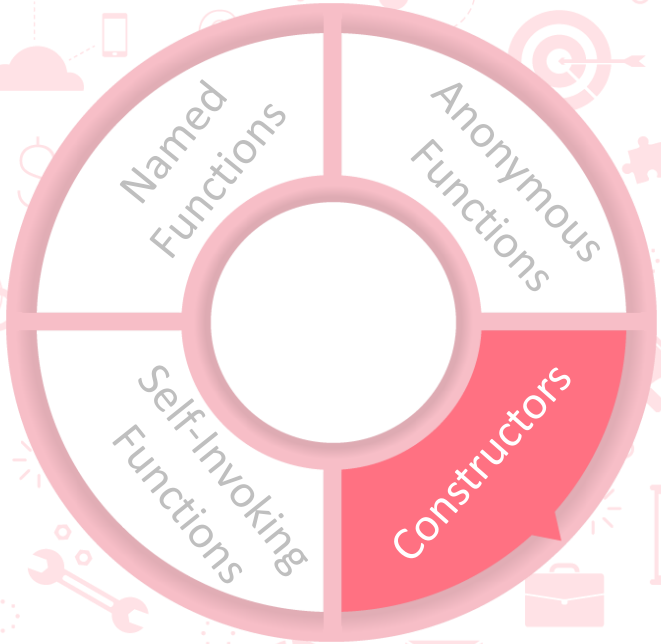
```
var add = function (a, b) { return a+b; };
add(2, 3);
```

- Passed as an argument to another function

```
setTimeout( function()
 {
    alert("this message is displayed after 5 seconds");
}, 5000);
```

No name for the function

Constructors



- A constructor is called when an object is created using the new keyword
- Example –

```
var addFunc = new Function("a", "b",
"return a + b;");
var c = addFunc(2,3);
```

is the same as

```
var addFunc = function(a, b) {
return a + b;
};
var c = addFunc(2,3);
```

edureka!

Self-Invoking Functions

- Self-invoking functions are anonymous functions, which are invoked right after the function has been defined
- You can execute the code once, without declaring any global variables
- No reference is maintained to this function, not even to its return value
- Syntax –

```
(function(){
    //body
}() );
```

```
(function(){
//body
}) ();
```

Self-Invoking Functions - Example

```
(function()
 alert("this is a self invoking function");
}) ();
```

OR

```
 (function()
alert("this is a self invoking function");
}());
```

Demo – Calculating the Square of a Number

edureka!

Calculating the Square of a Number

Step 1: Code your normal HTML page and give reference to your file with extension .js

```
<html>

<body>
    <h2></h2>
    <p id="demo"></p>

</body>

<script src="myJs.js"></script>


</html>
```

Step 2: Write a function, which accepts a number as a parameter

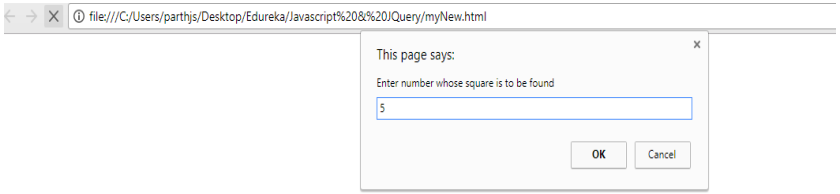
```
function squared(a) {
    return a*a;
}
```

Calculating the Square of a Number

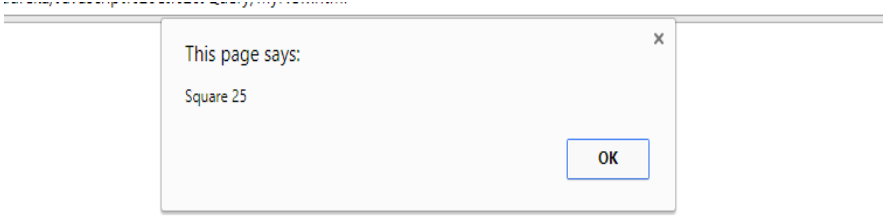
Step 3: To execute the function, we need to call the function

```
3   alert("Square " + squared(temp));
4
```

Step 4: Open your html file with a browser and enter a number in the prompt box. Click OK



Step 5: You will get the output that will be the square of the input number



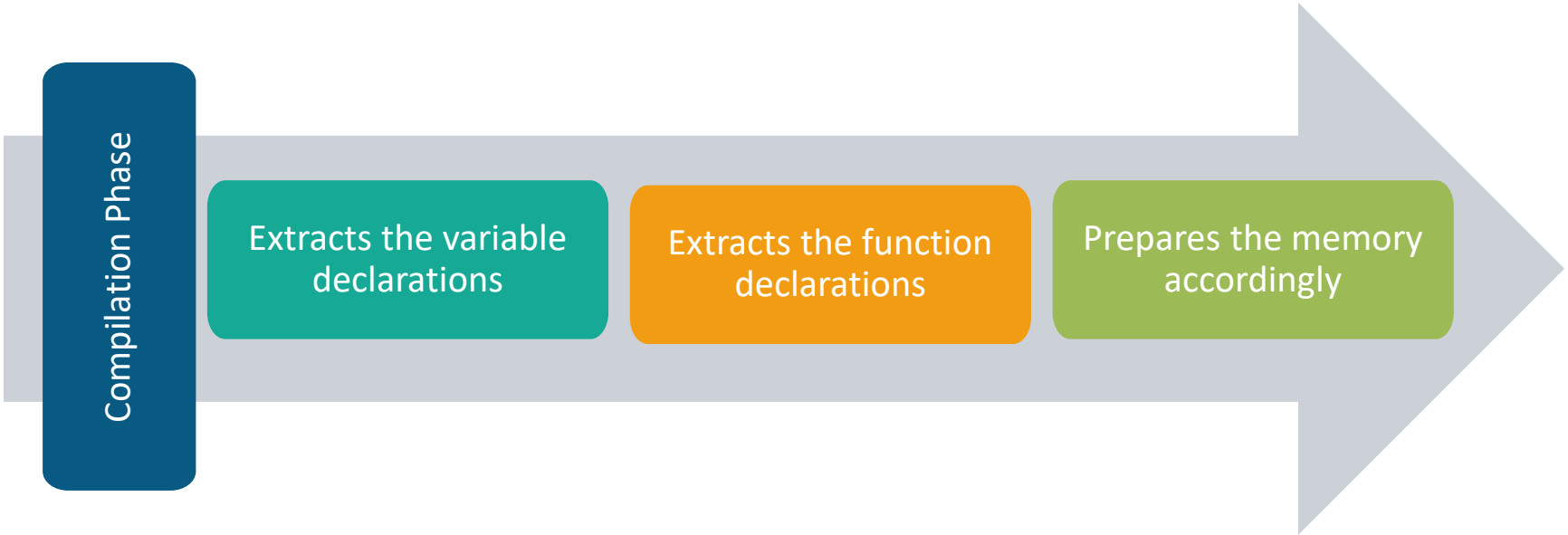
Memory Management

- The way in which memory is allocated, accessed and deallocated is termed as Memory Management
- Allocated memory is in the form of Heap or Stack

Heap Memory	Stack Memory
• Heap memory is dynamic • It will be destroyed when there is no way to reach it	• Stack memory is static • Very few memory is allocated in stack • Stack includes : – global variable declarations – function return references

Program Execution – Compilation Phase

- The first phase is referred as the Compilation phase



edureka!

Program Execution – Execution Phase

- The second phase, referred to as the Execution phase
- In the Execution Phase:
  - The Interpreter begins to execute lines of code
  - The code can reference Variables and Functions that were placed in memory space during the Compilation phase

Consider a Block of Code to Understand the Memory Management

Note : The  symbol denotes the compilation phase

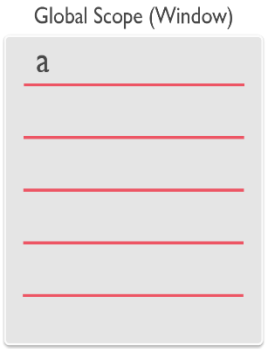
The  symbol denotes the execution phase

edureka!

Step 1

- Declaration for variable “a” in stack memory

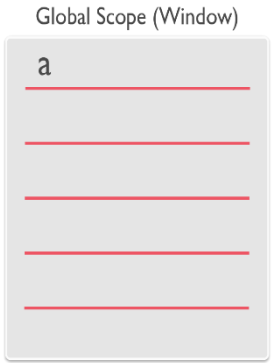
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 2

- Variable “b” is not declared. Hence, memory cannot be allocated

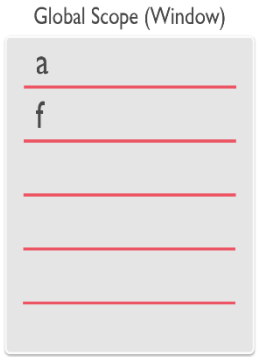
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 3

- Function f() is declared and memory for f() is allocated

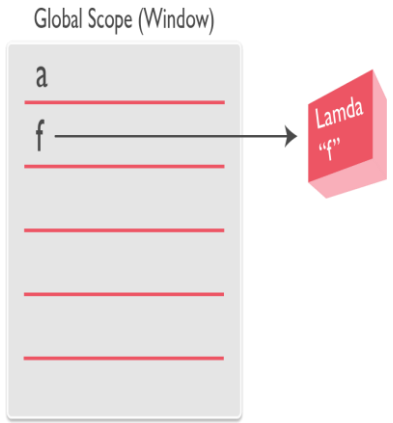
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 4

- After the function f() is declared in the stack, they will have a reference to the string block, lets say Lamda “f”
- Lamda “f” includes all the content of function f() body
- Skips and reaches the end of function f() block

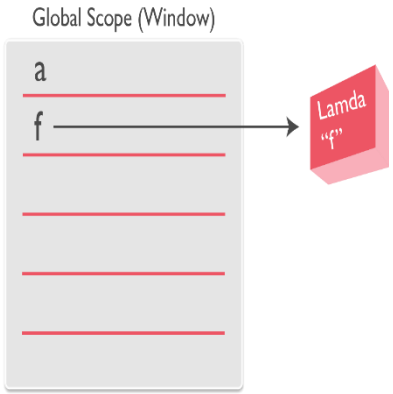
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 5

- f(1) is a function call, so it wont be added in the stack memory
- Compilation of the main block is finished

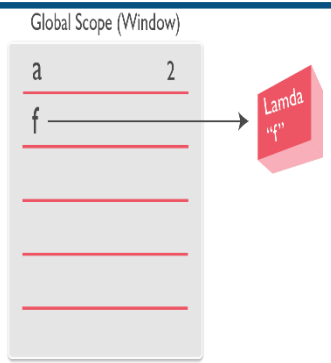
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 6

- The execution phase starts
- “a” is assigned the value 2

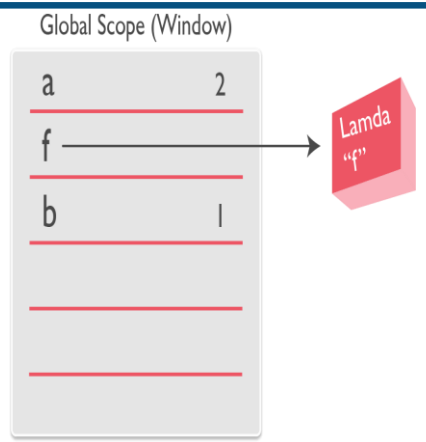
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 7

- Variable “b” does not have a declaration statement
- JavaScript forgives the mistake and allocates “b” with its value in global scope

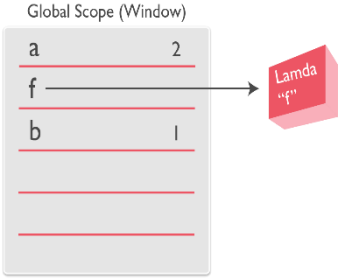
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```

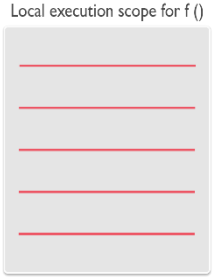


Step 8

- When execution reaches the f(1) function call, it creates a heap memory with local scope

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



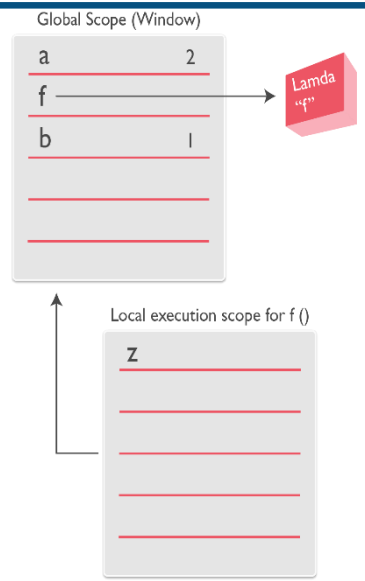


edureka!

Step 9

- Compilation phase for f(z) begins where “z” is allocated in local scope of function f()
- A parameter passed is a local variable of the function
- Local scope of f() has a reference to the global scope, as it can access all the global variables

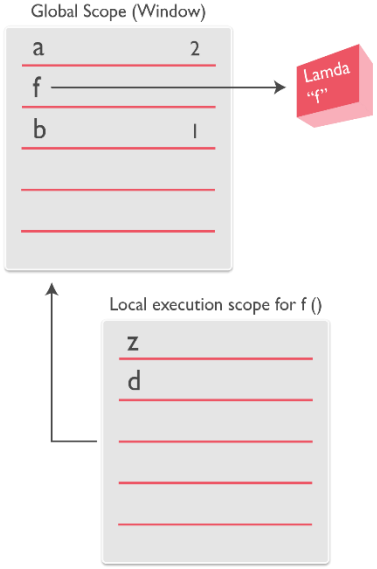
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 10

- Skips variables “b” and “c” as they are not declaration statements
- Allocates memory for “d”

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```

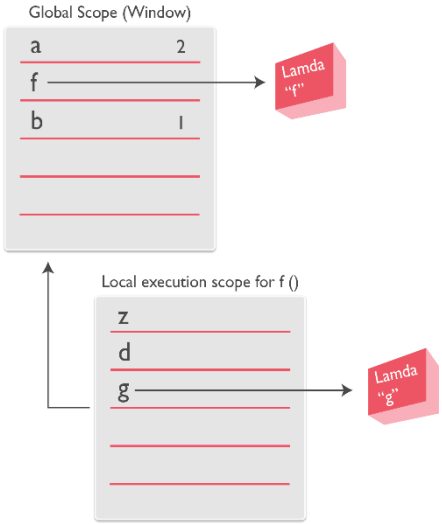


edureka!

Step 11

- Similar to f() function, g() is compiled
- A string block with the data of function g() is referenced from the local memory of function f(), lets name it Lamda “g”

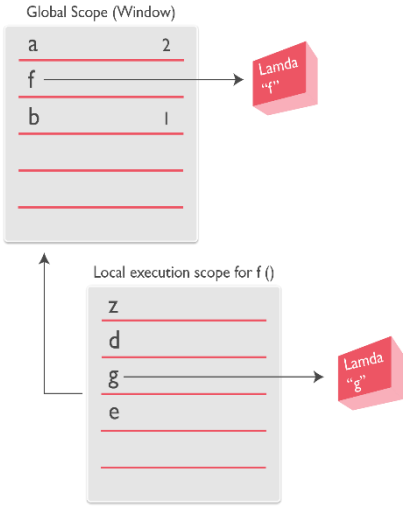
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 12

- Memory for variable “e” is allocated
- We find that the variable “e” is used before it has been declared
- This is possible due to the property of variable hoisting, wherein all the declarative statements are pulled above the rest of the statements

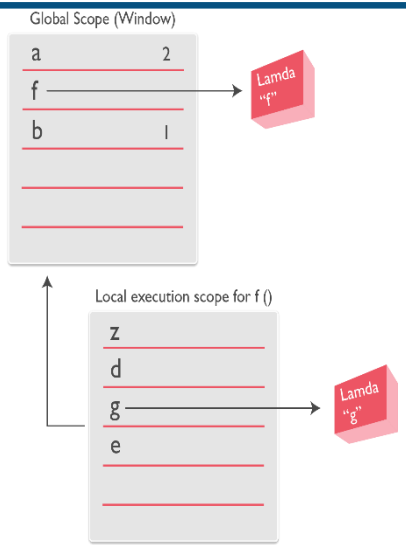
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 13

- Execution of function f(z) begins
- Variable “z” is passed with value 1

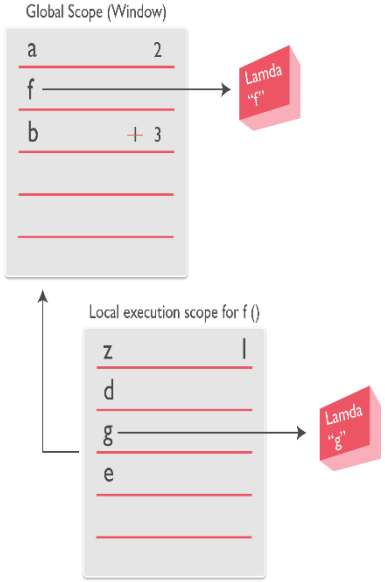
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 14

- Local of f() has a reference to the global scope
- It will check for variable “b” in the global
- Once “b” is found, the initial value of “b” is changed to 3

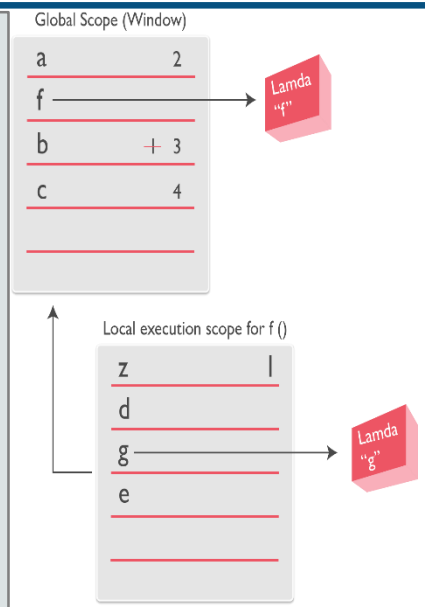
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 15

- Variable “c” is not found in global scope
- Memory for variable “c” will be allocated and value 4 will be assigned to it

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```

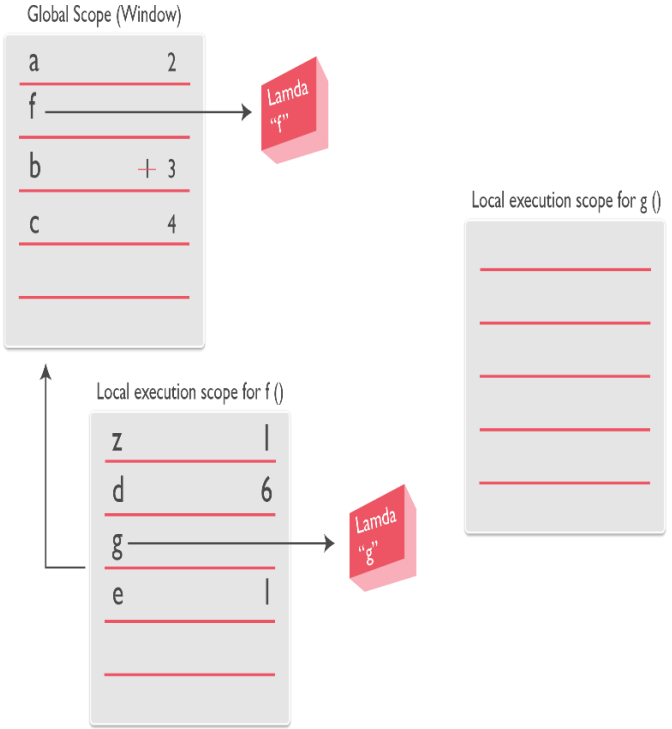


edureka!

Step 16

- Function g() is called through the return statement
- Local scope heap memory for g() is allocated

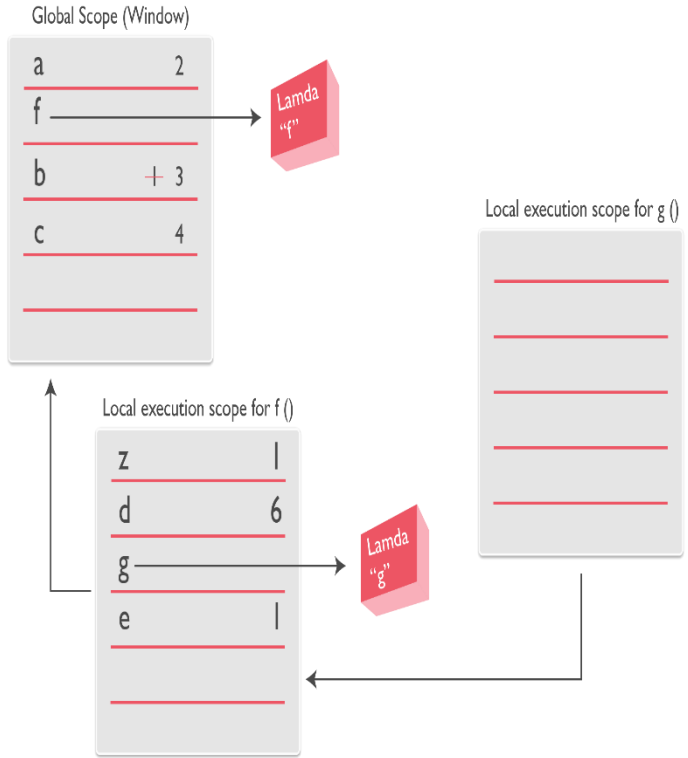
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 17

- Now similar to f(), g() starts compiling

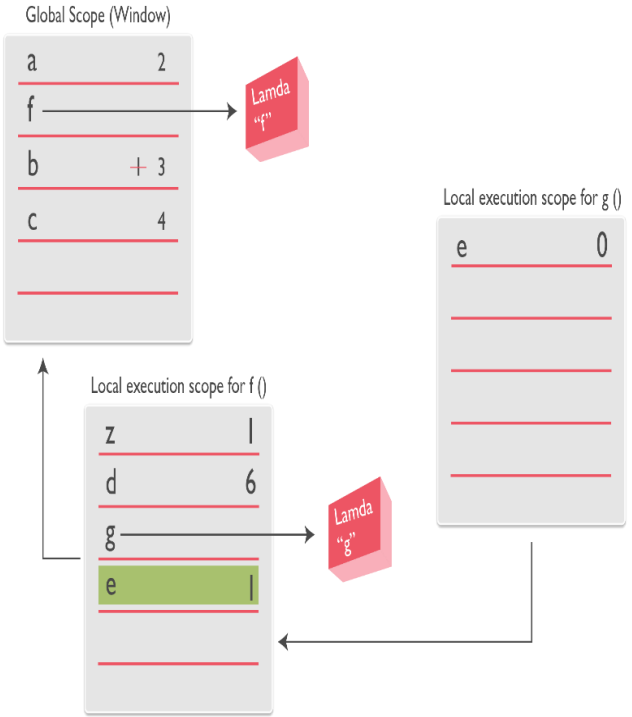
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 18

- Variable “e” of g() will be considered, as it is declared in local scope of g(), even though it has reference to local scope of f()
- This is called Variable Shadowing

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Variable Shadowing

- A variable only exists within the function/method/class in which it is present
- This variable will override any variables which belong to a wider scope
- This is also known as **Variable Scope**
- Example:

```
 var currencySymbol = "$";
function showMoney(amount) {
  var currencySymbol = "€";
  document.write(currencySymbol + amount);  }
showMoney("100");
```

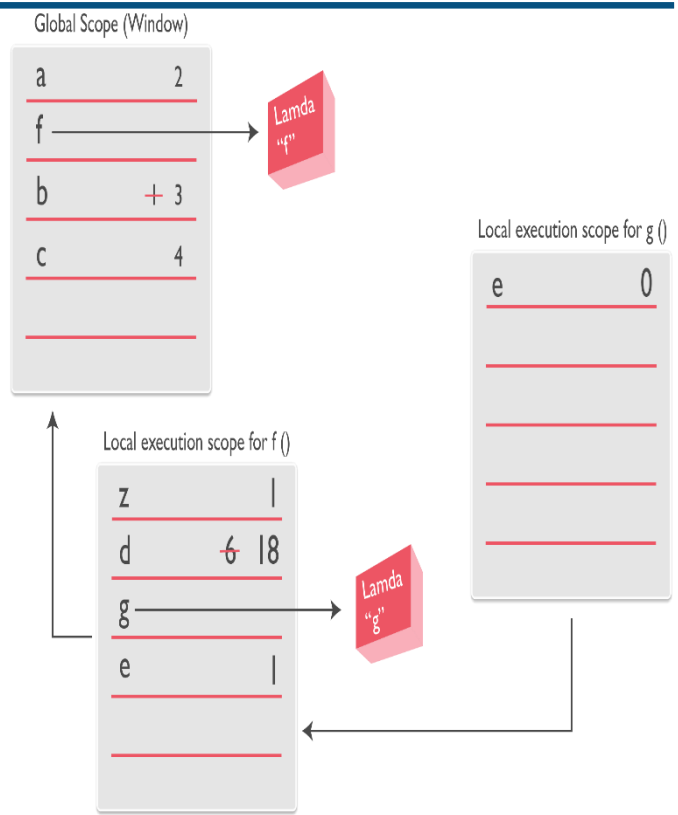
A euro sign will be shown, and not a dollar. (Because the currencySymbol containing the dollar is at a wider (global) scope than the currencySymbol containing the euro sign)

edureka!

Step 19

- Variable “d” is present in local scope of f(), and g() has a reference to local of f()
- Variable “d” is assigned the value 18 in local scope of f()

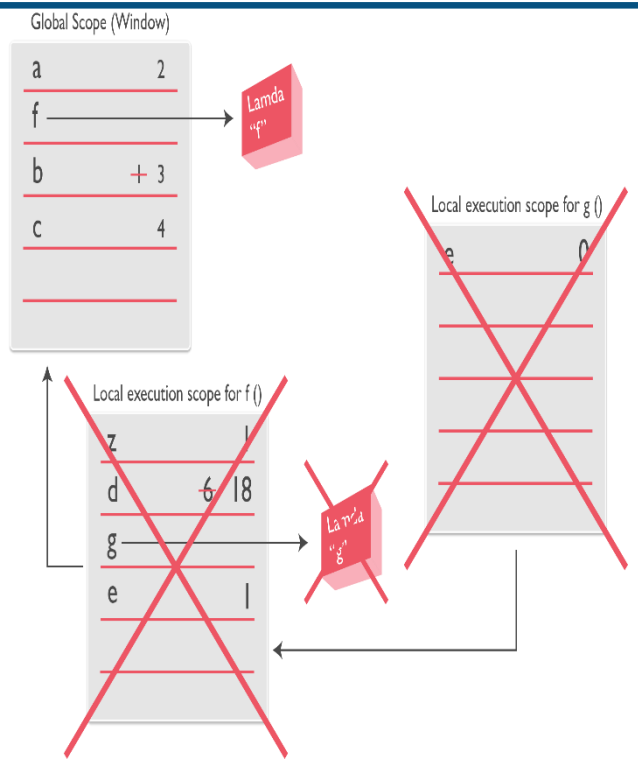
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Step 20

- Execution is completed
- As there is no way to access local of g() and local f(), they are garbage collected

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
f(1);
```



Garbage Collection

- Memory life cycle in all programming languages:
  - Allocate the memory you need
  - Use the allocated memory (read, write)
  - Release the allocated memory when it is not needed anymore
- In JavaScript a memory is released, once there is no reachability to the code
- Release of an allocated memory is called **Garbage Collection**

Let’s Modify our Code to Understand Closures

edureka!

Step 1

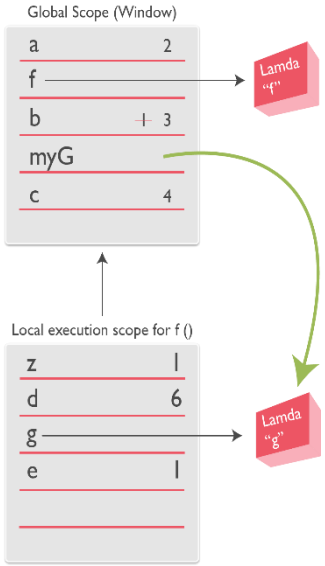
- The code will run similar to the previously discussed code, until the function f() is executed

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
var myG=f(1);
myG(); //28
```

Step 2

- myG() has a pointer pointing to Lamda "g", as f() returns function g()

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
var myG=f(1);
myG(); //28
```

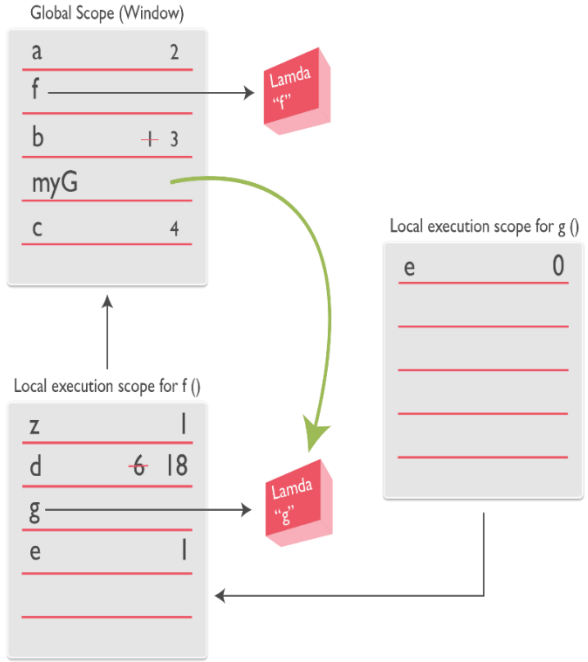


edureka!

Step 3

- g() gets executed

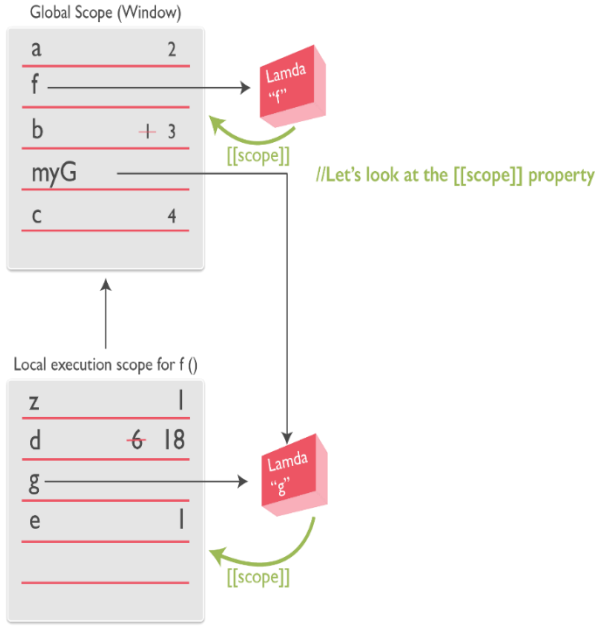
```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
var myG=f(1);
myG(); //28
```



Step 4

- There is one reference pointing to Lamda “g”, as g() is defined only in the context of f()
- This is the reason the memory of f() is not garbage collected
- This is called Closure

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
var myG=f(1);
myG(); //28
```

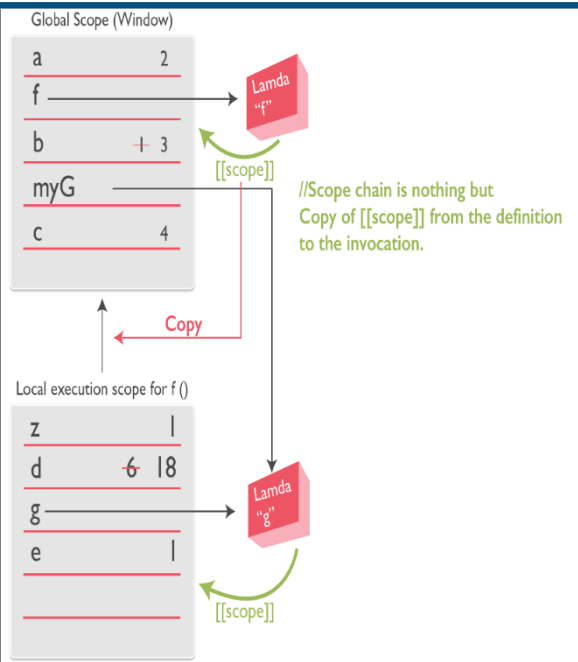


edureka!

Step 5

- The scope chain of Lamda "f" is copied to the local scope of f()

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
var myG=f(1);
myG(); //28
```

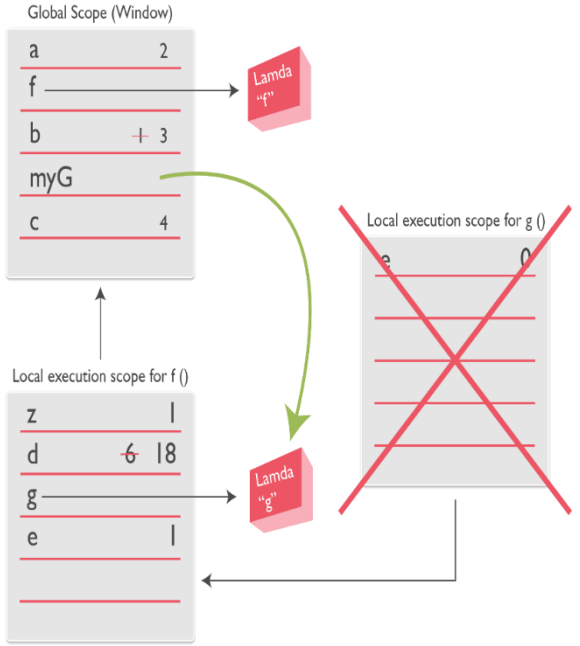


edureka!

Step 6

- The local scope of g() is garbage collected once it is executed

```
var a=2;
b= 1;
function f(z){
        b= 3;
        c=4;
        var d=6;
        e= 1;
        function g(){
                var e=0;
                d=3*d;
                return d;
        }
        return g();
        var e;
}
var myG=f(1);
myG(); //28
```



edureka!

Closures

- It is an implicit permanent link between the function and its scope chain
- A function definition's (Lamda) hidden [[scope]] ("[[]]" denotes internal property) reference:
  - Holds the Scope Chain (preventing garbage Collection)
  - It is used and copied as the "outer environment reference" anytime the function is run
- We saw an example of closure in the previous slide, where local memory of f() is not garbage collected even when there is no way to access f()

Summary

In this module, you should have learnt:

- How to define and call functions
- The memory representation of JavaScript
- Benefits of the variable hoisting
- The concept of variable shadowing or variable scope
- The concept of Garbage Collection and Closures

edureka!

edureka!

Thank You

For more information please visit our website
www.edureka.co