BILKENT UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CS315

HOMEWORK 1

BERAT BİÇER

21503050

24.11.2017

ANKARA, TURKEY

JavaScript

```
<script>

var a = 0; // Global a

var b = 1; // Global b

var c = 2; // Global c

var str = "";
```

a, b & c are global variables, since they are defined in the script's global scope. Every function in a JS script can access global variables if they are not hidden by another variable. This may not be the case if a variable that is defined later locally uses the same name, which we later illustrate.

```
function f1() {

    str += "f1(): a = " + this.a + ", b = " + b + ", c = " + c + "\n"; // uses global a,b &
c

    f2();

}
```

A call to f1 uses global variables "a", "b" & "c" since there is no declaration for "b" and "c" inside f1. For variable "a", however, situation is different: Normally, when a function call is made, the call is done via an object instance. If the call's not done via an object instance, "***this***" points to the global scope. Thus, "***this.a***" accesses to global variable "a", however, the value of "a" cannot be changed using the same syntax. This is a security precaution that prevents data corruption.

```
function f2() {

       var a = "ebucabbarHüsamettin"; // Local a

       str += "f2(): a = " + a + ", b = " + b + ", c = " + c + "\n"; // uses global b & c,
and local a

       f3();

}
```

A call to f2 is similar to the call to f1, however since f2 includes a locally defined variable a, it will use this local variable instead of the global one. Moreover, since JS uses static scope and NOT dynamic scope, variables are not transferred between function calls automatically. In order to do this, we'd need to give the desired variables as parameters of f2 when we call it. Thus, f2 illustrates both local variable declaration and usage of global variables.

```
function f3() {

       var a = 11; // local a

       var b = 22; // local b


       function f31() {

              var a = window["a"]; // Accesses global "a", still a local variable

              str += "f31(): a = " + a + ", b = " + b + ", c = " + c + "\n"; // uses local
a, b of f3 & global c
```

```
		f32();
	}

	function f32() {
		var a = 23.24; // local a
		var c = 56.87; // local c
		str += "f32(): a = " + a + ", b = " + b + ", c = " + c + "\n"; // uses local a
& c, and b of f3
		f33();
	}

	function f33() {
		var a= "berat"; // local a

		var b = (function () {
			str += "anonymous in f33-1(): a = " + a + ", b = " + b + ", c = " + c
+ "\n"; /* uses a & b of f33 and FAILS at b (since b is not decleared yet), c of global
			still fits the static scoping logic since the first b definition in the
parents is in f33, just wanted to show what would happen*/
			var b = 23; // local b

			str += "anonymous in f33-2(): a = " + a + ", b = " + b + ", c = " + c
+ "\n"; // uses a of f33, local b & c of global
			return 5;
		})();

		str += "f33(): a = " + a + ", b = " + b + ", c = " + c + "\n"; // uses local
a, b & global c
	}

	f31();
}
```

F3 is a complex function, so we'll divide it as follows: In the body of f3, there are 2 variable and 3 function declarations. Variables "a" & "b" are locally declared and if we want to access variable "c" from here, we'd use the global "c".

In f31, we declare a local variable "a" with value "***window["a"]***". This statement allows the locally defined variable "a" to have the value of global "a", however it is important to notice that "a" in f31 is still a LOCAL variable, NOT the global "a". Also, since f31 does not have declarations for "b" & "c", it

will use the “b” of f3 & global “c” because the first declarations encountered in the parents statically are in f3 for “b” & global in “c”

In f32, variables “a” & “c” are declared locally. Thus, we use these local variables and “b” of f3 since they are statically scoped.

In f33, variables “a” & “b” are locally defined and “c” of global is used. However, this function includes an anonymous function call. Thus, to evaluate what “b” is, the program first evaluates the code inside this anonymous scope. In this anonymous scope, there is no declarations for “a”, “b” or “c” at the beginning. Thus, when we try to access these variables we access “a” and “b” of f33 & “c” of global. However, since “b” of f33 is not yet known (because we haven’t finished execution of anonymous function yet), the result is undefined. Thus, first access returns undefined for “b” of f33. Later in the anonymous function we defined variable “b” locally & can access its value without a problem. After this anonymous function call is executed, the value of variable “b” of f33 is known and the program accesses “b” of f33 correctly. This example illustrates how static scope works in JS well; even though value of a variable is not defined, JS still accesses the first variable “b” it finds by statically scoping the function’s parent scopes. Also, we see how anonymous functions can create new static scopes at any point of execution.

```
f1();

str += "main: a = " + a + ", b = " + b + ", c = " + c + "\n"; // uses global a, b & c

document.getElementById("para").innerHTML = str;
```

After f33 completes execution, we return to main program and use variables global “a”, “b” & “c” since all other variables are invisible at this point. Finally, we set the content of “pre” tag with ID value “para” to the string we formed, “str”.

The complete program is in the archive in which this report is located.

This program uses JS and illustrates how static scoping is used. Important things to note is listed below:

- In JS, variables defined in any scope is local to that scope by default.
- To access any variable without its scope, there are 3 methods: 1, we can give that variable to as a parameter to a function. This method uses the VALUE of the variable; it does not directly access the variable itself. This is done via creating a local variable & assigning its value to the parameter. 2, we can use the expression “`window["a"]`” to access the VALUE of global variable “a” if it exists. However, similarly to the previous one, we do not access the variable itself but to its value. 3, we may use “`global`” before a variable to access a global variable with the same name. This method and use of global variables, however, are discouraged due to the following reasons:
    - Global variables are slower to locate, since the program needs to trace all the way to the global scope and find the point of declaration.
    - It is easy to forget that a variable is declared and later this variable may be re-declared. If the first one is not local, the value is overwritten and can cause wrong calculations.
    - Security reasons: every user has access to the global scope. Thus, using global variables allows others to change the value of these variables & violates encapsulation.
    - They reduce scalability and testability of the programs.

The output of the program above is provided below:

```
f1(): a = 0, b = 1, c = 2
f2(): a = ebucabbarHüsamettin, b = 1, c = 2
f31(): a = 0, b = 22, c = 2
f32(): a = 23.24, b = 22, c = 56.87
```

```
anonymous in f33-1(): a = berat, b = undefined, c = 2
anonymous in f33-2(): a = berat, b = 23, c = 2
f33(): a = berat, b = 5, c = 2
main: a = 0, b = 1, c = 2
```

Perl

```
# Demonstrates different scoping mechanisms in Perl

$a = 1; # global a

$b = 2; # global b

$c = 3; # global c

sub function1{

	my $a = 4; # creates lexical scoped variable a

	local $b = 5; # creates dynamically scoped variable b

	local $c = 6; # creates dynamically scoped variable b

	$function1 = "In function1: \$a = $a, \$b = $b, \$c = $c\n";

	function2();

}

sub function2{

	$function2 = "In function2: \$a = $a, \$b = $b, \$c = $c\n";

	function3();

}

sub function3() {

	my $c = 123; # creates a dynamically scoped variable c

	$function3 = "In function3: \$a = $a, \$b = $b, \$c = $c\n";

}

function1(); # call 1

$print = "Execution 1 ::\n $function1 $function2 $function3 In main-1: \$a = $a, \$b = $b,
\$c = $c\n\n";



function2(); # call 2

$print = "$print Execution 2 ::\n $function2 $function3 In main-2: \$a = $a, \$b = $b, \$c =
$c\n\n";



function3(); # call 3

$print = "$print Execution 3 ::\n $function3 In main-3: \$a = $a, \$b = $b, \$c = $c";



print "$print\n";
```

The Perl program above illustrates different scoping rules in Perl. In main function, we define global variables “a”, “b” & “c”. The call 1 to function1 creates 3 new variables: “a” is created with keyword “my”, so its statically scoped. That means, its scope is defined by the user code: it is invisible outside the scope of function1. “b” and “c”, however, are created with keyword “local”. That means, they use dynamic scoping: any function calls made later on can access to the variables “b” and “c” as long as the execution of function1 is not terminated. This is a problem for security and data validity: the result of calculations is not safe because the data might be corrupted. Since function1 has variables “a”, “b” and “c” accessible, it uses these local values.

In function2, since “b” and “c” of function1 are dynamically scoped and function1 is still executing, they are accessible inside function2. However, “a” of function1 uses static scoping and therefore it is not accessible to function2. Instead, function2 uses global “a” because function2 is inside the scope of global “a”.

In function3, a local variable “c” is created statically. Thus, even though “b” and “c” of function1 are accessible to function3, it only uses “b” of function1. Similar to function2, function3 uses “a” of global because “a” of function1 is invisible to function3.

In call 2, function2 accesses global “a”, “b” & “c” since it has no declaration for these variables and execution of function1 is over. Thus, dynamically scoped variables “b” & “c” of function1 are invisible from now on. Similar to function2, function3 also uses global “a” and “b”, however it has its own declaration for “c”, which is a local statically scoped variable; so it uses the local “c”.

In call 3, function3 acts similar to the previous one, and accesses global “a” and “b”, however it has its own declaration for “c”, which is a local statically scoped variable; so it uses the local “c”.

In Perl, variables may use either dynamic scoping or static scoping. Dynamic scoping allows variables to be accessible in the upcoming function calls but introduces security problems related to the risk of data corruption. Static scoping allows variables to be more encapsulated and enables the reader to trace which variable is used where, however it reduces flexibility.

In the archive this report is located, there are 2 Perl programs: one writes output to an HTML file whereas the other writes output to the console.

The output of the program above is as follows:

```
Execution 1 ::
 In function1: $a = 4, $b = 5, $c = 6
 In function2: $a = 1, $b = 5, $c = 6
 In function3: $a = 1, $b = 5, $c = 123
 In main-1: $a = 1, $b = 2, $c = 3

 Execution 2 ::
 In function2: $a = 1, $b = 2, $c = 3
 In function3: $a = 1, $b = 2, $c = 123
 In main-2: $a = 1, $b = 2, $c = 3

 Execution 3 ::
 In function3: $a = 1, $b = 2, $c = 123
 In main-3: $a = 1, $b = 2, $c = 3
```

PHP

In PHP, new scopes can be created with closures. However, it does not allow variables to be visible to its child scopes even though PHP is a statically scoped PL. By using such strict rules, PHP allows programs to be more scalable and secure at the cost of flexible and writability.

```
echo "\n              <pre>\n";

$a = 0; # global a

$b = 1; # global b

$c = 2; # global c
```

The statements above create new global variables named “a”, “b” & “c”. They are normally accessible only in the global scope, outside any function definitions.

```
function f1($c) { # Expands c of global's scope into f1's scope

        $a = 10; # local a

        global $b; # use global b

        echo "In f1: \$a = $a, \$b = $b & \$c = $c\n";
```

The code above is a part of f1 that declares its own local variable “a”, uses global “b” with “*global*” keyword and uses the value of global “c” without accessing it. Accessing global variables however discourage as usual.

```
function f11($b, $a) { # Accepts $b as a parameter

                $c = $GLOBALS["c"]; # use value of global c to create a local variable c

                echo "In f11: \$a = $a, \$b = $b & \$c = $c\n";

                echo "In f11-2: \$a = $a, \$b = $b & \$c = “. $GLOBALS["c”]. “\n";
```

The code above uses the values of “b” and “a” from f1, in other words, it accesses the values of variables “a” and “b” used in f1. In this case, they are local “a” of f1 and global “b”. It also creates its own local variable “c” using the value of global variable “c”. Using “***$GLOBALS["c"]***” allows access to the global variable “c” and any changes made using either “*global*” keyword or “***$GLOBALS["c"]***” statement will affect the global version of the variable. For example, “***$GLOBALS["c"] = 4;***” will set the global “c” to 4 whereas “***$c = $GLOBALS["c"]; $c = 4;***” will only effect the local “c”. So, in the first case, we create our own local variable “c” using the value of global “c”. In the second echo, we directly use global “c”.

```
function f111() {

        # echo "In f1: \$a = $a, \$b = $b & \$c = $c" --> Would give an error since we're
trying to access nondefined variables for f111's scope

        $a = function () { # closure, a local variable

                global $a; # global a

                $c = 111; # local c

                echo "In anonymous-1 of f111: \$a = $a, \$b = " . $GLOBALS["b"] . " & \$c =
$c\n"; # use global b

                return $c;

        };
```

The code above tries to access variables that do not yet defined inside the scope, so the first echo that’s commented out would print an error. Later, we create 3 closures to define local variables “a”, “b” & “c”. The first one is for “a”, and it creates 3 variables. The first one accesses global “a” directly,

the second one accesses directly to global “b” and the last one creates a local variable “c”. After execution, the value of the a call to “$a()” returns 111.

```
$b = function () use ($a) { # closure, use statement expands the scope of a and allows it to be used in anonymous-2

	global $b; # global b

	$c = 676; # local c

	echo "In anonymous-2 of f111: \$a = ". $a() . " \$b = $b & \$c = $c\n";

	return $c;

};
```

The code above is similar to the previous one, except it accesses the function inside “a” of f111 whose scope is expanded by using “***use***” keyword. Thus, “a” of f111 is accessible inside this closure. It also accesses global “b” and creates a local “c”.

```
$c = function () use ($b) { # closure

	global $a; # global a

	$c = 45; # local c

	echo "In anonymous-3 of f111: \$a = $a, \$b = " . $b() . " & \$c = $c\n";

	return $c;

};
```

The code above is similar to the previous one, except it accesses the function inside “b” of f111 whose scope is expanded by using “***use***” keyword. Thus, “b” of f111 is accessible inside this closure. It also accesses global “a” and creates a local “c”.

```
		echo "In f111: \$a = ". $a() . " \$b = " . $b() . " & \$c = " . $c() . "\n";

	}


	f111();

}

f11($b, $a);

f2($c);

}
```

The “***echo***” above belongs to the f111, so it accesses the functions inside “a”, “b” & “c” of f111 and uses the returned values to print.

```
function f2($c) { # $c is c from f1, which is global c

	# echo "In f2-1: \$a = $a, \$b = $b & \$c = $c\n"; --> Error since a & b are not defined atm

	$a = 9174; # local a

	echo "In f2-2: \$a = $a, \$b = " . $GLOBALS["b"] . " & \$c = $c\n";
```

```
}
```

The code above is called from inside f1, however, it cannot access any variables of f11 directly. It creates a local variable “c” using the value of the parameter, creates a local variable “a” and accesses the global “b”.

```
f1($c);

echo "In main: \$a = $a, \$b = $b & \$c = $c\n            </pre>\n";
```

The code above accesses global “a”, “b” & “c” since its inside the global scope.

The complete version of the program above is located in the directory this report is placed.

The output of the program above is as follows:

```
In f1: $a = 10, $b = 1 & $c = 2

In f11: $a = 10, $b = 1 & $c = 2

In f11-2: $a = 10, $b = 1 & $c = 2

In anonymous-1 of f111: $a = 0, $b = 1 & $c = 111

In anonymous-1 of f111: $a = 0, $b = 1 & $c = 111

In anonymous-2 of f111: $a = 111 $b = 1 & $c = 676

In anonymous-1 of f111: $a = 0, $b = 1 & $c = 111

In anonymous-2 of f111: $a = 111 $b = 1 & $c = 676

In anonymous-3 of f111: $a = 0, $b = 676 & $c = 45

In f111: $a = 111 $b = 676 & $c = 45

In f2-2: $a = 9174, $b = 1 & $c = 2

In main: $a = 0, $b = 1 & $c = 2
```

In PHP, variables are only accessible in the scopes they are defined. Unlike usual static scoping rules, variables are not visible in their child scopes. To access global variables, we can either use “`global`” or “`$GLOBALS["b"]`”. To access a non-global variable inside an anonymous function or a closure, we use “`function () use ($b)`” syntax to increase the visible range of a variable so that it becomes accessible. To access the value of any variable inside a function, we may give them as parameters to functions under rules of encapsulation.

Python

In python, variables are static locals by default if they are defined without a keyword. Other static scoping rules are still valid, that a variable is visible inside and children of its scope.

```
a = 0

b = 0

c = 0
```

These are the global variables “a”, “b” & “c”. They are accessible anywhere in the program by default.

```
def function1():
```

```
        a = 123 # A local variable defined statically

        b = 456 # A local variable defined statically

        c = 789 # A local variable defined statically

        print("In function1, a = {} & b = {} & c = {}".format(a,b,c)) # Uses local a, b & c
of function1 which are local variables
```

The code above declares 3 variables and by default they are static locals. So, the print statement uses those variables.

```
def function2():

                global a # Global keyword means this a is global a

                nonlocal b # Nonlocal b means this is the first b definition encountered in
function2's parents, in this case its b of function1

                c = 6776 # A local variable defined statically

                print("In function2-1, a = {} & b = {} & c = {}".format(a,b,c)) # Uses global
a, b of function1 & c of function2 which is local
```

The "***global***" keyword allows access to the global variable "a" instead of creating a new, static local variable "a". "***nonlocal***" keyword looks upward in the parent scopes to find a declaration for a variable "b", and makes it accessible inside function2. In this case, it is "b" of function1. Lastly, "c" is a static local variable.

```
def function3():

                        global b # This is global b

                        nonlocal c # This is c of function2 since the first definition of c as
we look larger scopes upwards is found in function2

                        a = 9449 # A local variable defined statically

                        print("In function3, a = {} & b = {} & c = {}".format(a,b,c)) # Uses a
of function3 which is local, b of global & c of function2

                        return;
```

The "***global***" keyword allows access to global "b", "***nonlocal***" keyword allows access to "c" in function2 and "a" is a static local variable.

```
function3()

                print("In function2-2, a = {} & b = {} & c = {}".format(a,b,c)) # Uses global
a, b of function1 & c of function2 which is local

                return;

        function2()

        print("In function1-2, a = {} & b = {} & c = {}".format(a,b,c)) # Uses a, b & c of
function1 which are local variables

        return;

print("In main-1, a = {} & b = {} & c = {}".format(a,b,c)); # Uses global a,b & c

function1()

print("In main-2, a = {} & b = {} & c = {}".format(a,b,c)); # Uses global a,b & c
```

First print belongs to the function2, second one belongs to the function1, third and fourth ones belong to the main; so they use suitable variables.

In Python, variables are static local variables by default, and accessible in their child scopes. To access a variable that's defined outside the current scope, we can either create a variable with "***nonLocal***" keyword, or use "***global***" keyword to access global versions, or simply not define a variable to access the parent's variables. However, these are not safe ways to access out-of-scope variables. Instead of directly accessing variables of outer scopes, we should pass them as function parameters in order not to violate encapsulation & prevent data corruption.

The complete code of the function above is located in the directory this report is placed.

The output of the program above is as follows:

In main-1, a = 0 & b = 0 & c = 0

In function1, a = 123 & b = 456 & c = 789

In function2-1, a = 0 & b = 456 & c = 6776

In function3, a = 9449 & b = 0 & c = 6776

In function2-2, a = 0 & b = 456 & c = 6776

In function1-2, a = 123 & b = 456 & c = 789

In main-2, a = 0 & b = 0 & c = 0