Michael Camara
Honor Code Pledge: This work is mine unless otherwise cited
CMPSC 220
Professor Roos
September 10, 2015

Lab 2: Scope, Frames, and Other Issues

3) “More about scope in JavaScript”

In the JavaScript code provided, the first console output will be “x = 10” and the second console output will be “x = hello”.

a) *Explain how JavaScript’s “function scope” rule is interpreted.*
A function in JavaScript shares some of the same scope rules as Java. Any variables that are declared locally (using the “var” declaration) in a function are not visible to any outer scopes once control returns to where the function was originally called, unless they are specified in a return statement. Further, if such a local declaration is made using the same variable name as a global declaration, then this will create “hole” in the outer scope. Essentially, any future calls to that variable from within the function will always refer to the local version. However, the function can still access other variables that have been declared globally, and can create global variables that persist upon subroutine closure. The latter can be accomplished by declaring any variable without including “var,” such as “y = 10,” where “y” is a novel variable name that has not appeared in an outer scope.

b) *State whether or not JavaScript requires “declare before use” for variables.*
The output from the provided JavaScript code shows that it does **not** require “declare before use” for variables. Even though the assignment “x = 10” is located lexically before the declaration “var x,” the scope of a declaration in JavaScript is the entire block in which it appears. Therefore, the “var x” declaration hides the global declaration of “x,” even though it appears after the “x = 10” assignment at the beginning of the function. This means that the global “x” is not affected by the behavior of the function “f,” such that the final console output shows that the global value of “hello” has not been altered.

5) “Look at stack structure in Java”

Syntax = “Frame Location : Name of Variable at Frame Location”

1 : i
2 : j
3 : a
4 : a
5 : b
6 : b

7 : p
8 : q
9 : sum
10 : prod
11 : prod
12 : max

6) “A stack machine computation”

Syntax = “Frame Location : Name of Variable at Frame Location”

1 : x
2 : y
3 : I

(NOTE: For the following visualization, the stack grows downwards)

Bytecode	**Stack Contents**
iload_1	10 (x)
iload_2	10 (x) 20 (y)
iadd	30
iload_1	30 10 (x)
iload_2	30 10 (x) 20 (y)
iadd	30 30
imul	900
iload_1	900 10 (x)
iload_2	900 10 (x) 20 (y)

iadd	900
	30
iload_1	900
	30
	10 (x)
iload_2	900
	30
	10 (x)
	20 (y)
iadd	900
	30
	30
imul	900
	900
iadd	1800
istore_3	___ (1800 is stored in i, stack is now empty)
getstatic java/lang/System/out Ljava/io/PrintStream;	___
iload_3	1800
invokevirtual java/io/PrintStream/println(I)V	___
return	___

<u>7)	"One more look at optimization"</u>

The source code for Stack2.java is written with many redundancies for the assignment: i = ((x+y)*(x+y))+((x+y)*(x+y)). This is reflected in the bytecode, which repeats many instructions to generate values that have already been calculated earlier in the stack. In order to optimize this bytecode and reduce the number of instructions needed, we can use the "dup" instruction. This will essentially duplicate the current top element in the stack and push it onto the stack. The following bytecode shows one way this instruction can be used, reducing the number of instructions in Stack2.class from 20 to 12.

(NOTE: For this visualization, the stack grows downwards)

Bytecode	Stack Contents
iload_1	10 (x)
iload_2	10 (x)
	20 (y)
iadd	30
dup	30 // This duplicates 30 and pushes it onto the stack
	30
imul	900
dup	900 // This duplicates 900 and pushes it onto the stack
	900
iadd	1800
istore_3	___ (1800 is stored in i, stack is now empty)
getstatic java/lang/System/out Ljava/io/PrintStream;	___
iload_3	1800
invokevirtual java/io/PrintStream/println(I)V	___
return	___