Tomer Alon

CS361

Programming Languages Principles and Implementation

Homework 2

**Question 1: History of programming languages**

Put the following programming languages on a chronological timeline. The year must be provided. **In addition,** indicate the name of the designer of the programming language, where it was created (company, national lab, higher education institution etc.), and the country.

* Fortran – 1957, designed by John Backus at IBM, USA.
* Lisp – 1958, designed by John McCarthy at the Massachusetts Institute of Technology (MIT), in Massachusetts, USA.
* COBOL – 1959, designed by Howard Bromberg, Howard Discount, Vernon Reeves, Jean E. Sammet, William Selden, and Gertrude Tierney at the US Department of Defense.
* ISETL – 1969, designed by Jacob T. Schwartz at New York University (NYU) Courant Institute of Mathematical Sciences, in New York, USA.
* PASCAL – 1970, designed by Niklaus Wirth, in Zürich, Switzerland.
* C – 1972, designed by Dennis Ritchie at Bell Labs (then owned by AT&T), in New Jersey, USA.
* Prolog – 1972, designed by Alain Colmerauer, at the University of Montreal, Canada.
* C++ – 1980, designed by Bjarne Stroustrup at AT&T Bell Labs, in New Jersey, USA.
* ADA – 1983, designed by Jean Ichbiah and the team of Cll Honeywell Bull (a French-owned computer company in France) under a contract to the US Department of Defense.
* SML – Design began in 1983; Formal definition established in 1986-89. Designed by Robin Milner at the University of Edinburgh, Scotland, UK.
* EIFFEL – 1985, designed by Bertrand Meyer at Eiffel Software, which was founded by Meyer.
* Perl – 1987, designed by Larry Wall at Unisys, in Pennsylvania, USA.
* Python – 1991, designed by Guido van Rossum at Centrum Wiskunde & Informatica, in Amsterdam, Netherlands.
* Ruby – 1993, designed by Yukihiro Matsumoto in Japan.
* Java – 1995, designed by James Gosling in Sun Microsystems (now owned by Oracle Corporation), in California, USA.
* Kotlin – 2011, designed by JetBrains (founded by Sergey Dmitriev, Valentin Kipiatkov, and Eugene Belyaev), in Prague, Czech Republic.

**Question 2:**

Consider the following code. Each *draw* method has a number.

public class Circle{

public double center\_x, center\_y;

public double radius;

public void draw() {

// **(1)** method to draw circle on the screen

}

public void draw(Color color) {

// **(2)** method to draw circle on the screen with a

// given color

}

}

public class ColoredCircle extends Circle{

public int color;

public void draw() {

// **(3)** method to draw the colored circle

}

}

1. Explain polymorphism on the code above.

The class *Circle* is the parent class, whereas *ColoredCircle* is the child class (subclass), which inherits from *Circle*; therefore, ColoredCircle is polymorphic– it has more than one form. The ColoredCircle IS-A Circle.

1. c is of type Circle and d is of type ColoredCircle. Can we write d = c;? Why?

No, because ColoredCircle is the child class of Circle, and it is not allowed to assign a parent class’s object to a child’s, unless a cast is used. Even then, it is not advised to do so.

1. c is of type Circle and d is of type ColoredCircle. Can we write c = d;? Why? What happens if we execute the code below? What method called *draw* is called? Why?

c = d;

c.draw();

It is legal to assign ColoredCircle d to c, as the child class is assigned to its super class. When the code is executed, the program will compile finely, and the output will invoke the draw method, but of ColoredCirle d, since the assignment has overridden the Circle method of “c.draw();.”

**Question 3:**

Install the following Eclipse Bytecode Outline plugin from: <http://asm.objectweb.org/eclipse/index.html> or from the Eclipse MarketPlace.

1. What Eclipse version are you using?

Version: Oxygen Release (4.7.0)

Build id: 20170620-1800

1. What Java version are you using?

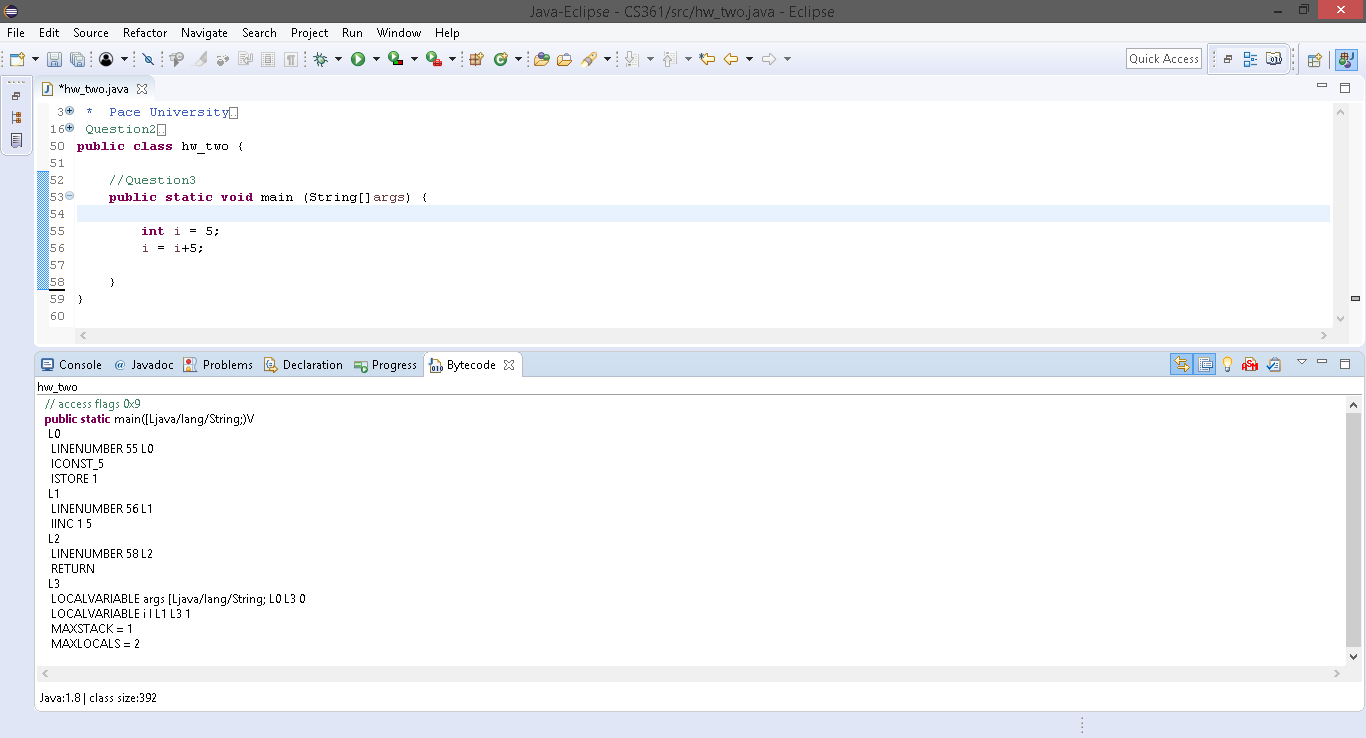
Java Version 8 Update 144 (build 1.8.0\_144-b01)

1. What is the Bytecode generated by the following statements?

int i = 5;

i = i+5;

Explain the syntax of the Bytecode. Provide a screenshot to support your work.



The Java bytecode consists of opcodes specifying the operations to be performed. Each bytecode is composed of one byte (hence the name) which represents an opcode, with zero or more bytes for operands.

For the two statements above, the bytecode reads as follows:

“LINENUMBER 55 L0” refers to the line 55 in the Java code, where the first instruction in the method is performed. On that same line are two opcodes: “ICONST\_5” and “ISTORE 1.” The former is an instruction to *push* the operand constant (in this case an integer constant) 5 into the operand stack. The latter instruction, “ISTORE 1” denotes that the integer in the operand stack is now *popped* and stored in a local variable 1 (in the local variable register).

“LINENUMBER 56 L1” refers to the line of code following the previously mentioned one. This line in the bytecode only has one opcode, which says “IINC 1 5.” This instruction takes two parameters, the first being the local variable, and the second, the number by which the variable will increase. 1 refers to the local variable in the register, and 5 is an integer amount the variable will increment.

Finally, “LINENUMBER 58 L2” has one opcode that says “RETURN.” This is where the method “main,” of return type void ends. In the Java code, this line is the closing curly bracket of the main method ‘}.’

With most instructions in the bytecode, there are prefixes and/or suffixes being used, to refer to the types of operands with which the instructions deal. For example, in the given code, the instructions such as “ICONST\_5” and “ISTORE 1,” as well as “IINC 1 5” have the prefix “I (i),” which refers to the integer operand type.

L0, L1, and L2 are labels that are paired with the line numbers:

(n0, L0)

(n1, L1)

(n2, L2)

Every instruction between L0 and L1 comes from the line n0, and every instruction between L1 and L2 comes from line n1, and so on....

The last lines that follow the “RETURN” instruction and L3 label are “LOCALVARIABLE args,” “LOCALVARIABLE I L1 L3 1,” “MAXSTACK = 1,” and “MAXLOCALS = 2.” The first reads the local variable arguments between labels L0 and L3, then lists the local variable 1, an integer i, between the labels L1 and L3. “MAXSTACK” is the max operand stack depth, and “MAXLOCALS” is the number of local variables.

1. Compare the Bytecode generated by the 2 functions below and write down your conclusions. Provide screenshots to support your work.

**public** **static** **int** sum\_for(**int** n) {

**int** i = 0, sum = 0;

**for** (i = 0; i <= n; i++) {

sum += i;

}

**return** sum;

}

**public** **static** **int** sum\_while(**int** n) {

**int** i = 0, sum = 0;

**while** (i <= n) {

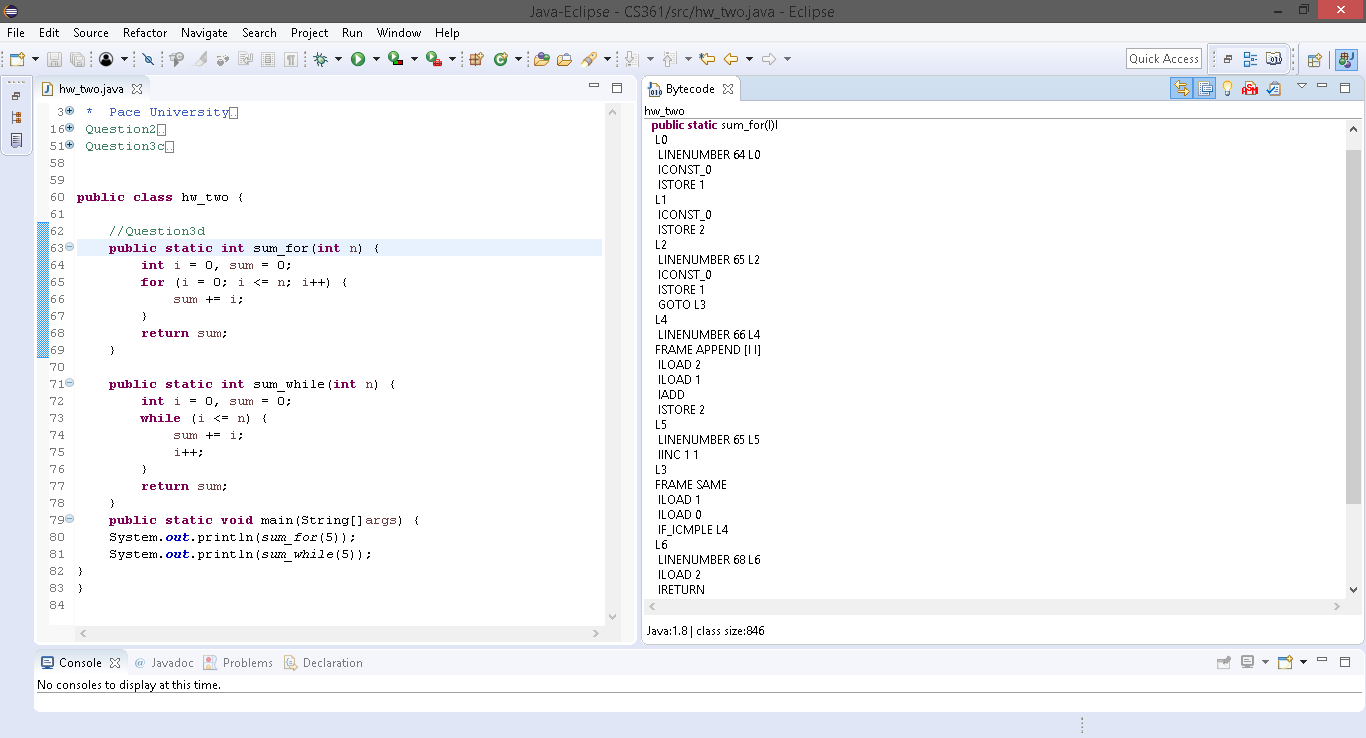
sum += i;

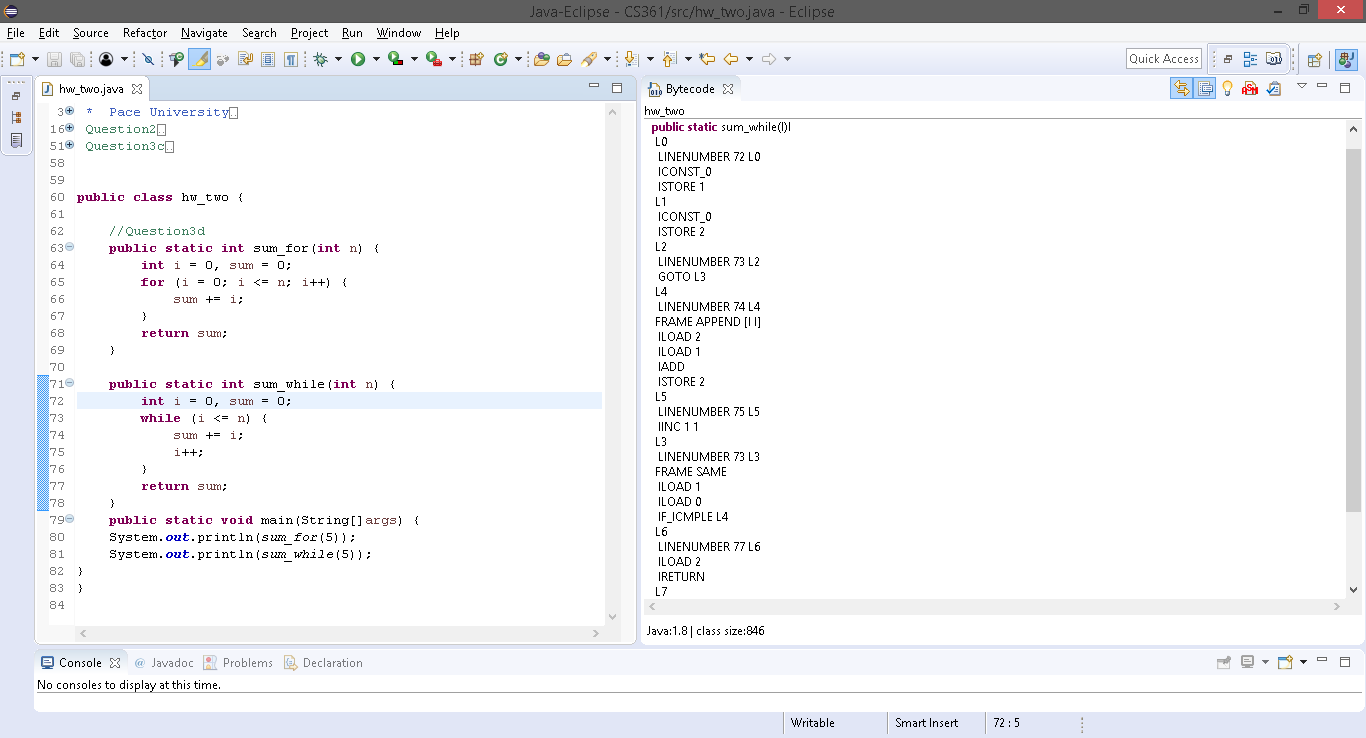
i++;

}

**return** sum;

}





Both functions of calculating the sum of consecutive numbers starting at zero give the same result, but their methods are different, just as some of their bytecode is not the same. One uses a ‘for loop’ and the other a ‘while loop.’

Beginning each method, sum\_for and sum\_while are defined as static methods that return an integer. The instructions that then follow are the same for both: an integer constant zero (0) is stored, or pushed, into the operand stack. The constant is then popped out of the stack and stored in variable 1 in the register (local variable register). The next label, L1, still on the same line, is doing the same two operations as in the label previous to it; an integer constant zero is pushed, then stored as variable 2 in the local variables register. Variable 1 is the declared variable i, and variable 2 is sum.

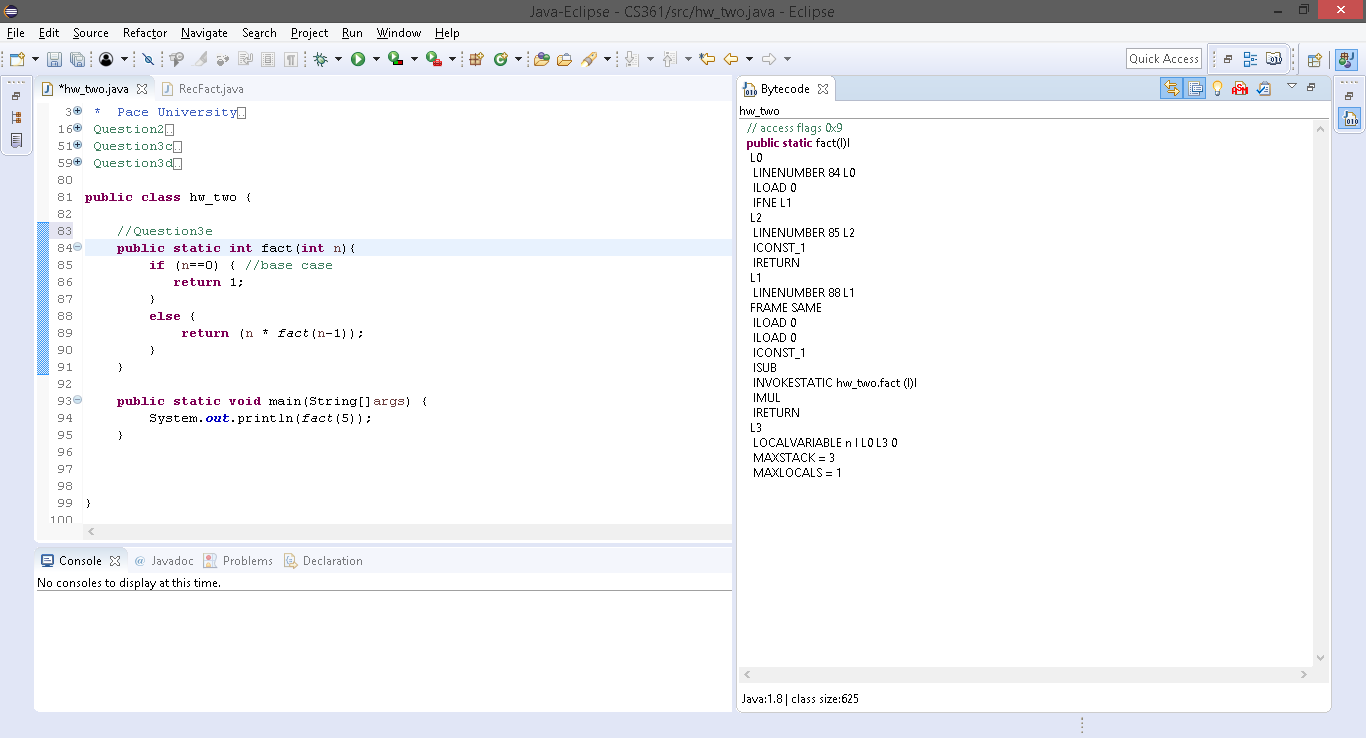
This is where the two methods are no longer similar in their bytecodes. The method sum\_for continues on with the label L2, which stores an integer constant zero in the variable i, which is then popped and stored in a local variable 1, and then instructs to “GOTO L3.” The “GOTO” instruction causes execution to branch to the instruction at the address listed. The sum\_while method simply instructs “GOTO L3,” without initializing “i” and storing it. Looking at L3 label for the sum\_for method, the instructions opt to load the two integer variables, by pushing them back into the operand stack. The two variables inside the ‘for loop’ are then compared– if, according to the statement within the loop, “i” is less than or equal to “n,” the bytecode instructs the program to branch to label L4. “IF\_ICMPLE L#” pops the two integers off the stack and compares them, to see if it can branch to the next address.

The method sum\_while, after being instructed to “GOTO L3,” does the same operation as sum\_for did with comparing the two integers, after loading them by pushing them back into the operand stack. If “i” is less than or equal to “n,” the method continues with rolling out its instructions.

From here until the termination of both methods, sum\_for and sum\_while carry out the same opcodes. From L4 to L5, both methods load the two variables initially declared, “i” and “sum,” and add the two (“sum += i;”). The addition is then stored into local variable 2, namely, “sum.” Following L5, local variable 1, “i,” is incremented by an integer 1 (“i++;”). Finally, after the sum is stored in local variable 2, the instructions between L6 and L7 load the integer variable by pushing it back into the operand stack, and then return its value. The statement “IRETURN” is similar to the previously-encountered statement “RETURN” in the previous question; however, “IRETURN” uses the i\_ prefix, which means the output is expected to be an integer.

The only thing done differently between the two methods, is that in the ‘while loop,’ the variable “i” is not declared a second time like it is in the ‘for loop.’

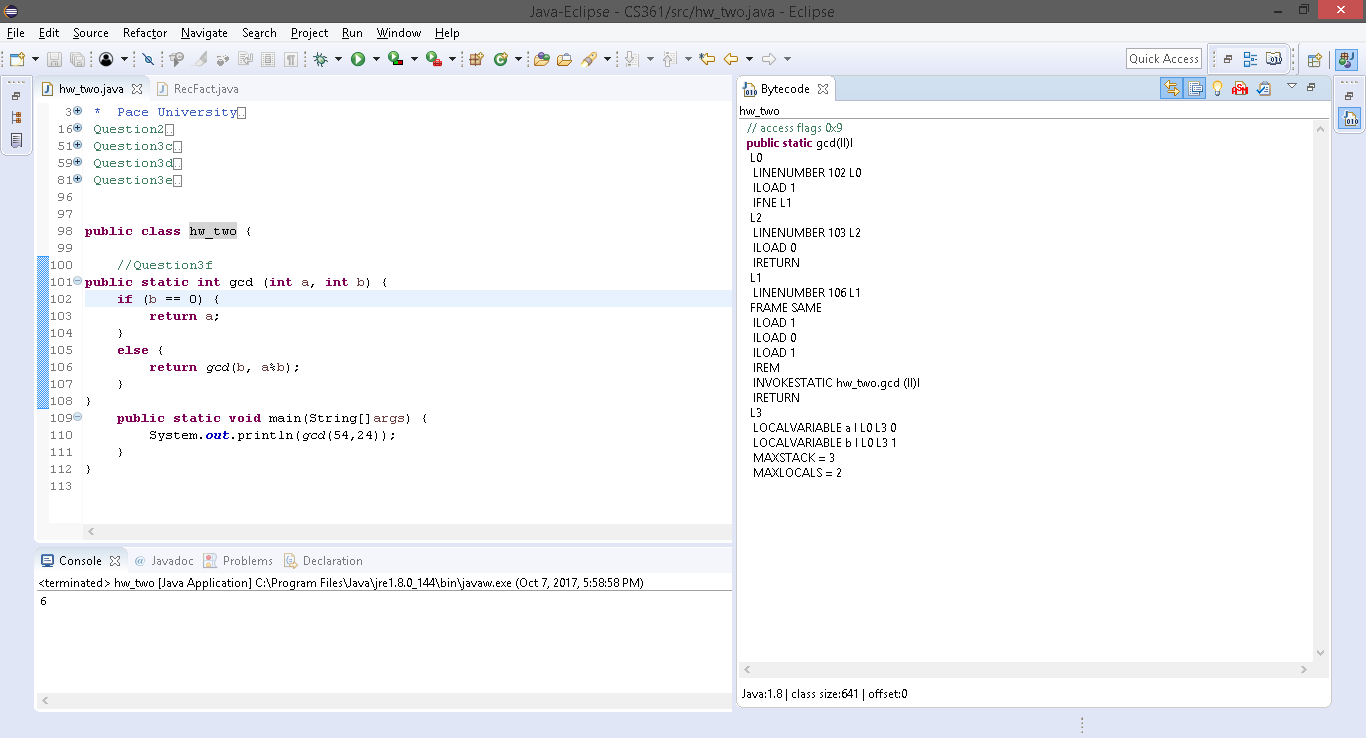
1. Write the factorial function (with the profile: public static fact(int n)) and describe the bytecode generated by this function.



The function fact(n) has generated a bytecode similar to what has been shown so far. The first label, L0 lists the instructions “ILOAD 0” and “IFNE L1.” The first opcode says that the integer is retrieved from the local variable register, in this case “n.” Then comes the “IF“ statement, in the form of “IFNE L1,” which means the function jumps if the integer is a nonzero number. In this case, the statement “if (n==0)” is the base case. “IFNE L1” pops the variable “n” to check if it is zero or not. In the case that it is a nonzero integer, the execution branches to label L1, as noted by the opcode, otherwise, the execution continues to label L2, which states “ICONST\_1” and “IRETURN.” The integer value 1 is stored in the operand stack, and then returned. Again, this only happens if “n” equals to zero.

Label L1 carries out the main recursive function, which is invoked within the “ELSE” statement. The only two new opcodes that are present here are “ISUB,” “IMUL,” and “INVOKESTATIC hw\_two.fact(I)I.” The first two opcodes’ purposes can be easily guessed– one is subtraction of two integers, and the second is multiplication of two integers. The variable “n” is loaded twice, since it is written twice in the function within the “ELSE” statement, and is then pushed into the operand stack. The integer 1 is then subtracted from “n” within the then-invoked static function fact(int n). The called-upon function is then multiplied by “n,” and finally the result is returned in integer form (“IRETURN”).

1. Choose a tail recursive function and describe the bytecode generated by this function. Compare with the code generated for a recursive function obtained in c.



For a tail-recursive function, I chose the Greatest Common Division function. The bytecode begins with loading the variable 1 (“b”) by pushing it to the operand stack. As this line is the base case for the recursive function, “IFNE L1” checks if “b” is a nonzero integer so the execution could branch to label L1. If “b” does equal zero, the variable “a” is pushed into the operand stack and is then returned.

When “b” does not equal zero, the two variables are pushed into the operand stack (“b” is loaded twice), then the opcode “IREM” is performed. This opcode pops off the two operands from the stack and calculates their remainder. That integer value is then pushed back into the operand stack. The method is then invoked again with the use of “INVOKESTATIC hw\_two.gcd(II)I,” and eventually an integer result is returned.

Compared with recursive function obtained in c, the gcd function generates a couple more bytecodes than does the former function. In c, the function simply declares a variable, stores an integer 5 in it, and then stores that variable in the local variable register. It then increments that local variable by 5 and terminates. The gcd function uses an “IF-ELSE” statement, in which the base case is established. Additionally, the gcd function’s bytecode specifically mentions the invocation of the static method, as part of its recursive trait.

**References**

* The Java Virtual Machine Specification <https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf> (Java 8 SE)
* Java Bytecode Basics <http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html> (1996)
* <http://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/> (2015)

**Question 4:**

1. Write a PROLOG program that describes the British family until nowadays. Kate, William and their children should be cited in the facts. Your program will start with the facts available in the slides (slide 31) and ends with Kate, William and their children.

(See attached file)

parent(victoria, edward\_vii).

parent(edward\_vii, george\_v).

parent(alexandra, george\_v).

parent(george\_v, george\_vi).

parent(george\_vi, elizabeth\_ii).

parent(elizabeth\_ii, charles).

parent(charles, william).

parent(william, george).

parent(kate, george).

parent(william, charlotte).

parent(kate, charlotte).

male(edward\_vii).

male(george\_v).

male(george\_vi).

male(charles).

male(william).

male(george).

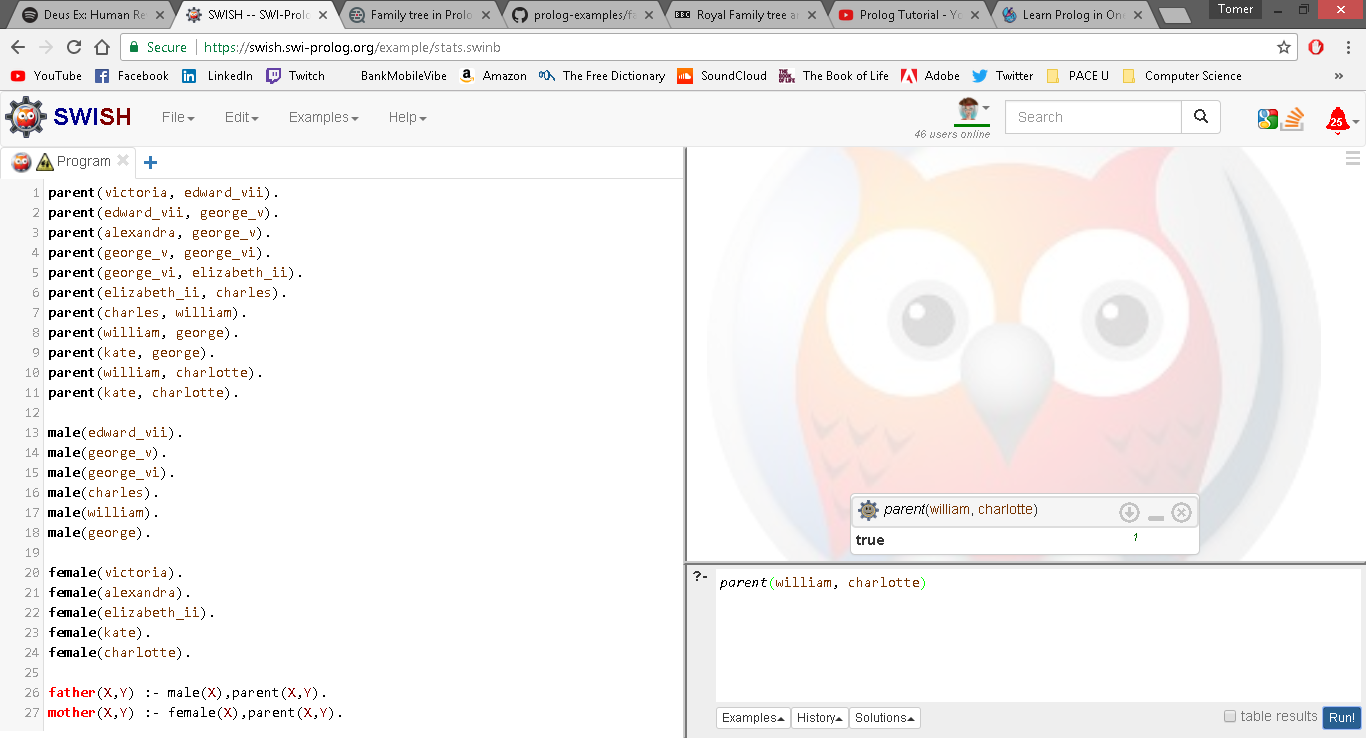
female(victoria).

female(alexandra).

female(elizabeth\_ii).

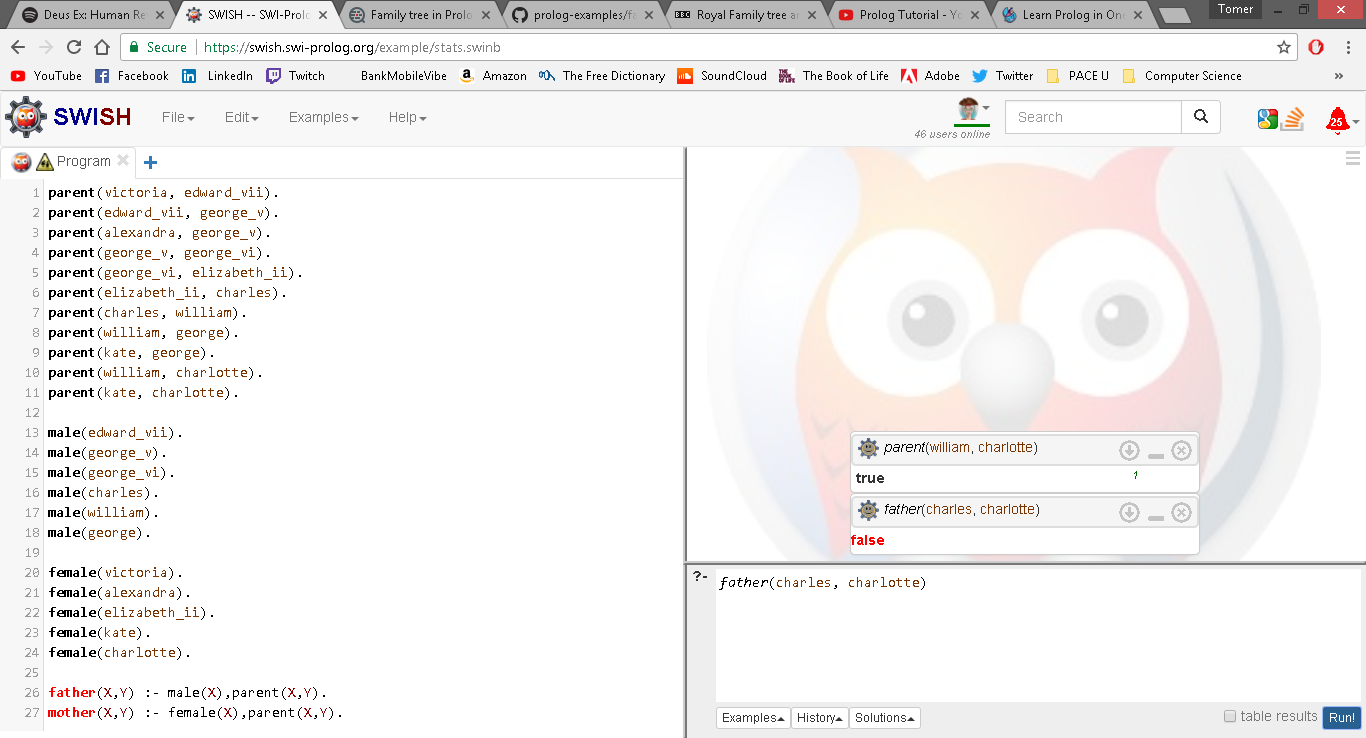
female(kate).

female(charlotte).



1. Write a **rule** that describes the father predicate. *Father(X,Y)* means that *X* is the father of *Y*.

father(X,Y) :- male(X),parent(X,Y).



**Question 5:**

Write a **recursive** function *recPow* that computes 2n for n >= 0 in Java. The function will have the following profile:

public static int recPow(int n)

The function must consider all cases and be tested exhaustively. Show your testing!

**public** **static** **int** recPow(**int** n) {

**if** (n < 0) {

**throw** **new** RuntimeException("n has to be an integer greater than zero.");

}

**else** **if** (n>=31) {

**throw** **new** RuntimeException("Integer Overflow");

}

**else** **if** (n == 0) {

**return** 1;

}

**else** **if** (n == 1) {

**return** 2;

}

**else** {

**return** 2 \* *recPow*((n-1));

}

}

**public** **static** **void** main (String[]args) {

**for**(**int** i = 0; i <= 31; i++) {

System.***out***.println("2^"+i + " = " + *recPow*(i));

}

}

**Question 6:**

Write a **recursive** function merge that merges 2 arrays in Java. The function will have the following profile:

public static int[] mergeSort(int[] a, int[] b)

You will use the split function of slide 18 (odd and even positions).

The function must be tested exhaustively. Show your testing!

If you use code online, you will need to cite your sources.

**public** **static** **int**[] sort(**int**[] a){

// Sort the contents of array a in ascending numerical order

**if**(a.length > 1){

**int** i,mid = a.length/2;

**int**[] half1 = **new** **int**[mid];

**int**[] half2 = **new** **int**[a.length - mid];

**for**(i = 0; i < mid; i++)

half1[i] = a[i];

**for**(; i < a.length; i++)

half2[i - mid] = a[i];

*sort*(half1);

*sort*(half2);

**int** ja = 0, ka = 0;

**for**(i = 0; ja < half1.length && ka < half2.length; i++)

**if**(half1[ja] < half2[ka]){

a[i] = half1[ja];

ja++; }

**else**{

a[i] = half2[ka];

ka++;

}

**for**(; ja < half1.length; i++, ja++)

a[i] = half1[ja];

**for**(; ka < half2.length; i++, ka++)

a[i] = half2[ka];

}

**return** a;

}

**public** **static** **int**[] mergeSort(**int**[] a, **int**[] b) {

//function to merge two arrays

**int**[] answer = **new** **int**[a.length + b.length];

**int** i = 0, j = 0, k = 0;

**while** (i < a.length && j < b.length){

**if** (a[i] < b[j]){

answer[k] = a[i];

i++;

}

**else**{

answer[k] = b[j];

j++;

}

k++;

}

**while** (i < a.length){

answer[k] = a[i];

i++;

k++;

}

**while** (j < b.length){

answer[k] = b[j];

j++;

k++;

}

**return** *sort*(answer);

}

**public** **static** **void** printArray ( **int**[] anArray ){ //function to print array

**for** ( **int** i = 0; i < anArray.length; i++ )

System.***out***.print ( anArray[i] + " " );

System.***out***.println ();

}

**public** **static** **void** main (String[]args) {

**int** []a = {5,4,6,8,10,7,3};

**int** []b = {1,3,2,6,11,15,0,9};

**int** []answera = (*sort*(a));

**int** []answerb = (*sort*(b));

**int** []ans = *mergeSort*(a,b);

*printArray*(answera);

*printArray*(answerb);

*printArray*(ans);}}