Homework 2

Yunfei Wang

Seidenberg School of Computer Science

CS 611

**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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LANGUAGE | YEAR | DESIGNER | PLACE/GROUP | COUNTRY |
| Fortran | 1957 | John Backus | IBM | US |
| Lisp | 1958 | John McCarthy | MIT | US |
| Cobol | 1959 | Howard Bromberg Howard Discount Vernon Reeves Jean E. Sammet William Selden Gertrude Tierney | CODASYL  US DoD | US |
| ISETL | 1969 | Jacob T. Schwartz | New York University | US |
| Pascal | 1970 | Niklaus Wirth | / | Switzerland |
| Prolog | 1972 | Alain Colmerauer  Philippe Roussel | Aix-Marseille University | France |
| C | 1972 | Dennis Ritchie | Bell Labs | US |
| Ada | 1980 | Jean Ichbiah (France) | CII Honeywell Bull US DoD | US |
| C++ | 1983 | Bjarne Stroustrup | Bell Labs | US |
| Eiffel | 1986 | Bertrand Meyer (France) | Eiffel Software | US |
| Perl | 1987 | Larry Wall | / | US |
| SML | 1990 | Robin Milner (England) Mads Tofte (Denmark) Robert Harper (US) David MacQueen (US) | The first design meeting was held in Edinburgh, Robin’s living room in 1983 | UK |
| Python | 1991 | Guido van Rossum | Python Software Foundation | Netherlands |
| Java | 1995 | James Gosling | Sun Microsystems | US |
| Ruby | 1995 | Yukihiro Matsumoto | / | Japan |
| Kotlin | 2011 | / | JetBrains | Russia |

**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.

Polymorphism allows a single method to perform in different ways. The code demonstrates polymorphism via inheritance. Circle is the parent class and ColoredCircle is the child. When assigning a child object to a parent reference (Circle c = new ColoredCircle(); ), then calling a *draw* method (c.draw(); ), the third *draw* method in ColoredCircle overrides the first *draw* method in Circle.

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

No, because when assigning a parent object to a child reference must be done with a down-casting: d = (ColoredCircle)c; .

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 *draw* method is called? Why?

c = d;

c.draw();

Yes, because a ColoredCircle is a Circle. The code is assigning a child object to a parent reference, and the third *draw* method is called because of method overriding.

**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?

Oxygen.1a Release (4.7.1a)

1. What Java version are you using?

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

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

int i = 5; ICONST\_5 ISTORE 1

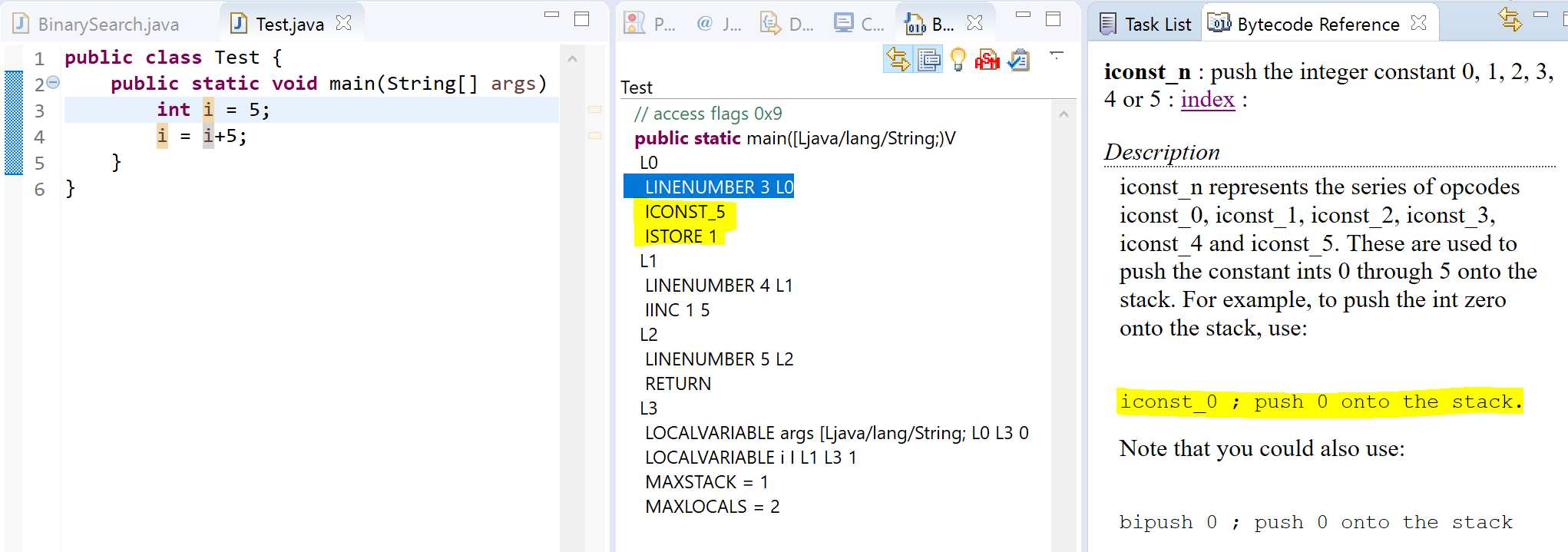
i = i+5; IINC 1 5

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

ICONST\_5: push 5 onto the stack

ISTORE 1: store integer in local variable 1

IINC 1 5: increment local variable 1 by 5



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++;

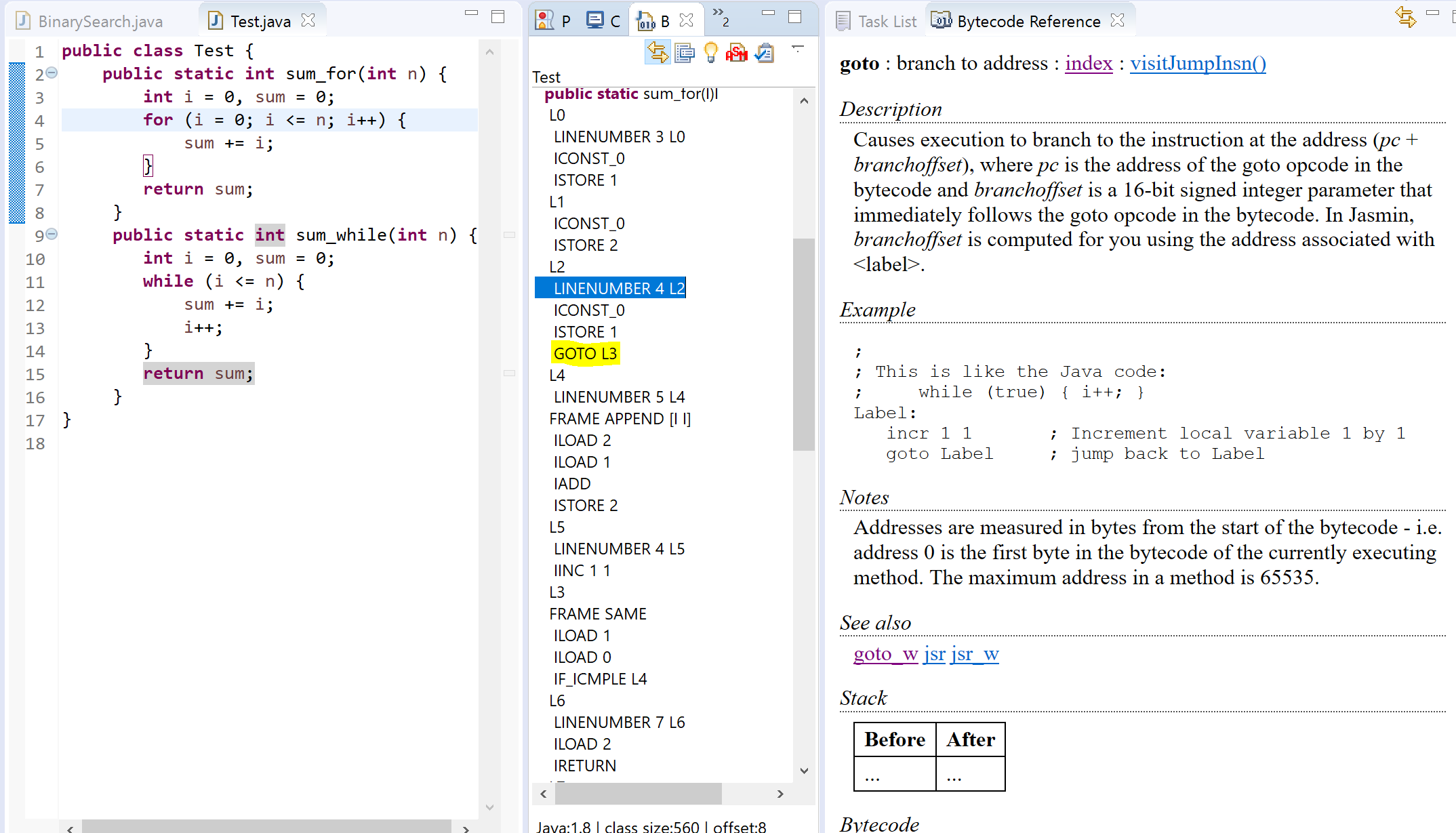
}

return sum;

}

|  |  |  |
| --- | --- | --- |
| step | sum\_for | sum\_while |
| 1 | ICONST\_0 ISTORE 1 | ICONST\_0 ISTORE 1 |
| 2 | ICONST\_0 ISTORE 2 | ICONST\_0 ISTORE 2 |
| 3 | ICONST\_0 ISTORE 1 GOTO L3 | GOTO L3 |
| 4 | ILOAD 2 ILOAD 1 IADD ISTORE 2 | ILOAD 2 ILOAD 1 IADD ISTORE 2 |
| 5 | IINC 1 1 | IINC 1 1 |
| 6 | ILOAD 1 ILOAD 0 IF\_ICMPLE L4 | ILOAD 1 ILOAD 0 IF\_ICMPLE L4 |
| 7 | ILOAD 2 IRETURN | ILOAD 2 IRETURN |

Above is a step-by-step comparison of the Bytecode of *sum\_for* and *sum\_while*; they are basically the same. The only difference is at step 3: *sum\_for* has ICONST\_0, ISTORE 1, and GOTO L3, whereas *sum\_while* only has GOTO L3. GOTO means “branch to address”, and “jump back to label L3” in this example. *sum\_for* has two extra steps because it has i = 0; which requires a push and a store action. Since int i = 0; is already assigned in global scope, it is somewhat redundant 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.

|  |  |  |
| --- | --- | --- |
| **public static int** fact(**int** n){ |  |  |
| **if** (n==0) | ILOAD 0  **IFNE** L1 | jump if nonzero |
| **return** 1; | ICONST\_1  IRETURN |  |
| **else** **if** (n>0 && n<=12) | ILOAD 0  **IFLE** L3  ILOAD 0  BIPUSH 12  **IF\_ICMPGT** L3 | jump if less than or equal to zero;  jump if one integer is greater than another |
| **return** n\*fact(n-1); | ILOAD 0  ILOAD 0  ICONST\_1  **ISUB**  **INVOKESTATIC Test.fact (I)I**  **IMUL**  IRETURN | subtract two integers;  call *fact()*;  multiply two integers |
| **else** |  |  |
| **throw new**  IndexOutOfBoundsException  (n+" is out of bounds!"); | **NEW** ...IndexOutOfBoundsException  **DUP**  NEW java/lang/StringBuilder  DUP  ILOAD 0  INVOKESTATIC …String  INVOKESPECIAL …V  **LDC** " is out of bounds!"  INVOKEVIRTUAL …StringBuilder;  INVOKEVIRTUAL …String;  INVOKESPECIAL …V  **ATHROW** | create an object;  duplicate top single-word item on the stack;  push single-word constant onto stack;  throw an exception |
| } |  |  |

1. Choose a tail recursive function and describe the bytecode generated by this function. Compare the code that is generated with the code of a recursive function. What do you observe?

**private** **static** **int** tailFact(**int** n, **int** accm) {

**if** (n == 0)

**return** accm;

**else if** (n>0 && n<=12)

**return** *tailFact*(n-1, n\*accm);

**else**

**throw new** IndexOutOfBoundsException

(n + " is out of bounds!");

}

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

**return** *tailFact*(n, 1);

}

In the recursive case (see code in question e), first you break down the steps from n to 0, then the result is built backwards from 0 to n. In the tail recursive case, the result is saved in the accumulator as the second parameter of the method, and it only takes half number of the steps of the recursive case.

**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.

H(Albert, Victoria). % H(X,Y): X is Y’s husband.

P(Victoria, Edward VII).

P(Albert, Edward VII).

H(Edward VII, Alexandra).

P(Edward VII, George V).

P(Alexandra, George V).

H(George V, Mary).

P(George V, George VI).

P(George V, Edward VIII).

P(Mary, George VI).

P(Mary, Edward VIII).

H(George VI, Elizabeth).

P(George VI, Elizabeth II).

H(Philip, Elizabeth II).

P(Elizabeth II, Charles).

P(Philip, Charles).

H(Charles, Diana).

P(Charles, William).

P(Charles, Harry).

P(Diana, William).

P(Diana, Harry).

H(William, Kate).

P(William, George).

P(William, Charlotte).

P(Kate, George).

P(Kate, Charlotte).

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

Father(X,Y):-

H(X,Z), P(Z,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 || n>=31)

**throw new** IllegalArgumentException

("Illegal Power Argument");

**if**(n==0) **return** 1;

**int** half = recPow(n/2);

**if**(n%2==1){

**return** 2\*half\*half;

} **else** {

**return** half\*half;

}

}

Testing:

n=0, output: 1;

n=5, output: 32;

n=10, output: 1024;

n=31, output: Illegal Power Argument.

**Question 6:**

Write a **recursive** function that implements merge sort in Java. The function will have the following profile:

public static int[] mergeSort(int[])

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.

**static** **int**[] mergeSort(**int** []arr) {

**if** (arr==**null** || arr.length<=1) **return** arr;

**int** []arr1 = **null**;

**int** []arr2 = **new** **int**[arr.length/2];

**if** (arr.length % 2 == 0) {

arr1 = **new** **int**[arr.length/2];

} **else** {

arr1 = **new** **int**[arr.length/2 + 1];

}

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

**if** (i%2 == 0) {

arr1[i/2] = arr[i];

} **else** {

arr2[i/2] = arr[i];

}

}

arr1 = *mergeSort*(arr1);

arr2 = *mergeSort*(arr2);

**return** *merge*(arr1, arr2);

}

**static** **int**[] merge(**int**[] arr1, **int** []arr2) {

**int** []result = **new** **int**[arr1.length + arr2.length];

**int** i1 = 0, i2 = 0, cur = 0;

**while** (i1<arr1.length && i2<arr2.length) {

**if**(arr1[i1] < arr2[i2]) {

result[cur++] = arr1[i1++];

} **else** {

result[cur++] = arr2[i2++];

}

}

**while** (i1 < arr1.length) {

result[cur++] = arr1[i1++];

}

**while** (i2 < arr2.length) {

result[cur++] = arr2[i2++];

}

**return** result;

}

Testing:

Input: [7, 3, 2, 5, 8, 1, 9], output: [1, 2, 3, 5, 7, 8, 9].

Resource: www.mycstutorials.com/articles/sorting/mergesort