Assigned Reading for the Final Exam: Read Sec. 5.2 on pp. 155 – 60 and the discussion of call by name on p. 165 in Sethi. Also, read the comments on call / pass by value-result on the next page. [You should already understand call / pass by value and call / pass by reference, as those parameter passing modes are used in C++. So most of the material on pp. 155 – 8 should be familiar to you.] There will be a problem on the Final Exam that is similar in nature to the examples below.


Complete the table below to show the output that is produced when the following program is executed. When completing each row of the table, assume that parameters are passed by the indicated mode.

```java
class Example {
    static int e;
    static int a[] = new int[3];

    static void test (int x) {
        a[1] = 6;
        e = 2;
        x += 3;
    }

    public static void main(String[] args) {
        a[1] = 1; a[2] = 2; e = 1;
        test(a[e]);
        System.out.println(a[1] + " " + a[2] + " " + e);
    }
}
```

Output for each parameter passing mode:

- value:
  - a[1]: 6
  - a[2]: 2
  - e: 2
- reference:
  - a[1]: 9
  - a[2]: 2
  - e: 2
- value-result:
  - a[1]: 4
  - a[2]: 2
  - e: 2
- value-result (Algol W):
  - a[1]: 6
  - a[2]: 4
  - e: 2
- name:
  - a[1]: 6
  - a[2]: 5
  - e: 2

Solutions

Problem 1:

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Problem 2 (based on an old exam question):

Complete the table below to show the output that is produced when the following program is executed. When completing each row of the table, assume that parameters are passed by the indicated mode.

```java
class FinalExam {
    static int e = 1;
    static int a[] = {0,1,2};

    public static void main(String[] args) {
        test(a[e], a[e-1]);
        System.out.println(a[0] + " " + a[1] + " " + a[2] + " " + e);
    }

    static void test (int x, int y) {
        a[1] = 6;
        e = 2;
        x += 3;
        y--;
        System.out.print(x + " " + y + " ");
    }
}
```

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>-1</td>
<td>-1</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Comments on Call / Pass by Value-Result

There are two subtleties relating to call / pass by value-result:

1. If the same variable is passed as two different arguments, then the final value of that argument variable may depend on the order in which formal parameter values are copied back into the actual argument variables' locations. As an example, consider a function of the form

   ```java
   void p(int a, int b)
   {
       a = 4;
       b = 7;
   }
   ```

   where the parameters a and b are passed by value-result. Suppose this function p() is called within the function main() as follows:

   ```java
   p(j,j);
   System.out.print(j)
   ```

   Then, when control returns to main() from the call p(j, j), the following must happen:
   (i) The final value of formal parameter a (i.e., 4) is copied into argument variable j.
   (ii) The final value of formal parameter b (i.e., 7) is copied into argument variable j.

   The definition of call/pass by value-result does not say which of (i) and (ii) occurs first. If (i) occurs after (ii), then the final value of j will be 4, the final value of formal parameter a. But if (ii) occurs after (i), then the final value of j will be 7, the final value of parameter b. [If you were given the above code and asked to write down the output of System.out.print(j) assuming pass by value-result, then neither "4" nor "7" would be correct—you would be expected to write "4 or 7".]

2. The discussion of call/pass by value-result on pp. 159–60 of Sethi applies to "standard" pass by value-result. A slightly different version of pass by value-result was used in the language Algol W. In standard pass by value-result, when control returns to the caller the final value of each formal parameter is copied into the location that belonged to the corresponding actual argument variable at the time of the call. But, in Algol-W style pass by value-result, when control returns to the caller the final value of each formal parameter is copied into the location that belongs to the corresponding actual argument variable at that time. These two versions of pass by value-result are usually indistinguishable, but they may produce different results if the actual argument is, e.g., an indexed variable whose index is changed by the function that is called. As an example, consider a function

   ```java
   void q(int c)
   {
       c = 55;
       i = 17;
   }
   ```

   where i is a global variable and parameter c is passed by value-result. Suppose this function q() is called within the function main() as follows:

   ```java
   i = 23;
   q(arr[i]);
   ```

   In standard pass by value-result, when control returns to main() from the call q(arr[i]) the final value of q's parameter c (i.e., 55) will be copied into arr[23] because i's value was 23 when q(arr[i]) was called. However, in Algol-W style pass by value-result the final value of parameter c will instead be copied into arr[17] because i's value is 17 when control returns to main() from the call q(arr[i]).
The dump on pp. 5 – 7 below was produced when TJasn.TJ compiled the TinyJ program on p. 2 and then executed the generated code with a debugging stop after execution of exactly 23,172 instructions with the following sequence of input values: 4, 5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0. The code that was generated is shown on pp. 3 – 5. Note that the INITSTKFRM instructions in this code are at code memory addresses 4, 322, 391, and 490.

Some Examples of Possible Questions Relating to the Dump

1. (a) For each method, say how many locations are allocated to local variables in its stackframe.
   ANSWER: main: 7; readRow: 2
   transpose: 5; writeOut: 3

   (b) Write down the size of a stackframe of readRow(), transpose(), and writeOut().
   ANSWER: readRow: 7; transpose: 10; writeOut: 8

Questions 2 – 9 are about the state of the TinyJ virtual machine at the time of the debugging stop after execution of 23,172 instructions:

2. Consider the static variables mat, count, and tm.
   What values are stored in the following locations?
   (a) count
   ANSWER: 1
   (b) mat[2][4]
   ANSWER: 5
   (c) tm[0][3][2]
   ANSWER: 8

3. Which method is being executed?
   ANSWER: writeOut

4. Which data memory locations constitute the stackframe of the executing method?
   ANSWER: addresses 200 through 207

5. What values are stored in the stackframe locations of the local variables and formal parameters of the executing method?
   ANSWER: i = 742, j = 0, mm = null,
   rows = -2, cols = 0,
   matr[ ][ ] = PTR to 10085

6. Which method called the executing method?
   ANSWER: writeOut

7. Which method called the caller?
   ANSWER: readRow

8. Which method called the caller’s caller? And which method called that method?
   ANSWER: transpose; main

9. What are the addresses of main’s local variables h and j?
   ANSWER: h's addr is 171; j's is 172

Now suppose the debugging stop had not occurred.

10. What would be the code memory addresses of the next 10 instructions to be executed (i.e., the 23,173rd through 23,182nd instructions to be executed)?
    ANSWER: 503 through 511, then 500.

11. What output, if any, would be produced by execution of those instructions? ANSWER: None

12. Which data memory locations, if any, would be changed in value by execution of the 10 instructions? Name the variable(s) whose values are stored there.
    ANSWER: address 205; i

13. Write down what the PC, FP, ASP and ESP registers would contain after execution of the first 3 of the above 10 instructions, and also write down the value or values of EXPRSTACK[j] for 0 ≤ j < ESP.
    ANSWER: PC=506, FP=PTR TO 204,
    ASP=PTR TO 208, ESP=1
    EXPRSTACK[0]=PTR TO 205

14. When the currently executing method activation RETURNS to its caller, what will PC, FP, and ASP be set to?
    ANSWER: PC=568, FP=PTR TO 196
    ASP=PTR TO 200

15. Answer questions 10 – 13 (regarding the 23,173rd through 23,182nd instructions to be executed) again under the (very unlikely!) assumption that, immediately before executing the 23,173rd instruction, soft errors in one of the computer’s memory chips change the LT instruction at code memory address 503 to a GE instruction.
    ANSWER: 503 and 504, then 512 – 519.
    Output: 1 then newline
    No data memory location's value is changed.
    After execution of the first 3 or these 10 instructions,
    PC=513, FP=PTR TO 204,
    ASP=PTR TO 208, ESP=1
    EXPRSTACK[0]=PTR TO 1
import java.util.Scanner;

class DumpEx {
    static int mat[ ][ ][], count;
    static int tm[ ][][] = new int[5][ ][ ];

    public static void main (String args[ ]) {
        int r[ ] = new int[5], c[ ] = new int[5], n = 1;
        int layer = -1;
        while (n == 1) {
            if (layer < 4)
                layer = layer + 1;
            else
                layer = 0;
            Scanner input = new Scanner(System.in);
            System.out.print("Enter number of rows: ");
            r[layer] = input.nextInt();
            System.out.print("Enter number of columns: ");
            c[layer] = input.nextInt();
            mat = new int [r[layer]][ ];
            tm[layer] = mat;
            int i = 0;
            while (i < r[layer]) {
                mat[i] = new int[c[layer]];
                readRow(i + 1, mat[i], c[layer]);
                i = i + 1;
            }
            int h = 0;
            while (h <= layer) {
                System.out.println("Given matrix: ");
                writeOut(r[h], c[h], tm[h]);
                System.out.println("Transposed matrix: ");
                writeOut(c[h], r[h], transpose(tm[h], r[h], c[h]));
                h = h + 1;
            }
            System.out.println("Doubled matrices: ");
            h = 0;
            while (h <= layer) {
                System.out.println("\nType 1 to continue, 0 to quit: ");
                n = input.nextInt();
            }
        }
    }

    static void readRow(int rowNum, int m[ ], int c)
    {
        if (rowNum >= 0) {
            System.out.print("Row ");
            System.out.println(rowNum);
        }
        int i = 0;
        while (i < c) {
            if (rowNum == -1) {
                int mm[ ][ ] = new int[1][ ];
                writeOut(-1, 10, mm);
                i = i + 1;
            } else {
                Scanner input = new Scanner(System.in);
                System.out.print("Enter value in column ");
                System.out.print(i+1);
                System.out.print(" ");
                m[i] = input.nextInt();
                i = i + 1;
            }
        }
    }

    static int[ ][ ] transpose(int m[ ][ ], int r, int c)
    {
        int k, i, m1[ ][ ] = new int[c][ ];
        k = 0;
        while (k < c) {
            m1[k] = new int[r];
            System.out.println("\n");
            j = 0;
            while (j < r) {
                System.out.print(m1[j][i]);
                j = j + 1;
            }
        i = i + 1;
        }
        return m1;
    }

    static void writeOut (int rows, int cols, int matr[ ][ ][ ])
    {
        System.out.println("\n\nType 1 to continue, 0 to quit: ");
        n = input.nextInt();
    }
}

static void readRow(int rowNum, int m[ ], int c[ ]){
    if (rowNum >= 0) {
        System.out.print("Row ");
        System.out.println(rowNum);
    }
    int i = 0;
    while (i < c) {
        if (rowNum == -1) {
            int mm[ ][ ] = new int[1][ ];
            writeOut(-1, 10, mm);
            i = i + 1;
        } else {
            Scanner input = new Scanner(System.in);
            System.out.print("Enter value in column ");
            System.out.print(i+1);
            System.out.print(" ");
            m[i] = input.nextInt();
            i = i + 1;
        }
    }
}

static int[ ][ ] transpose(int m[ ][ ], int r, int c){
    int k, i, m1[ ][ ] = new int[c][ ];
    k = 0;
    while (k < c) {
        m1[k] = new int[r];
        System.out.println("\n");
        j = 0;
        while (j < r) {
            System.out.print(m1[j][i]);
            j = j + 1;
        }
        i = i + 1;
    }
    return m1;
}

static void writeOut (int rows, int cols, int matr[ ][ ][ ]){
    System.out.println("\n\nType 1 to continue, 0 to quit: ");
    n = input.nextInt();
}
472: CALLSTATMETHOD 322 518: LOADFROMADDR 564: PUSHLOCADDR 3
473: PUSHLOCADDR 4 519: PUSHNUM 1 565: LOADFROMADDR
474: PUSHLOCADDR 4 520: ADD 566: PASSPARAM
475: LOADFROMADDR 521: SAVETOADDR 567: CALLSTATMETHOD 490
476: PUSHNUM 1 522: PUSHLOCADDR 1 568: PUSHLOCADDR 2
477: ADD 523: LOADFROMADDR 569: PUSHLOCADDR 2
478: SAVETOADDR 524: PUSHLOCADDR 570: LOADFROMADDR
479: JUMP 434 525: LOADFROMADDR 571: PUSHNUM 1
480: PUSHLOCADDR 2 526: LT 572: ADD
481: PUSHLOCADDR 2 527: PUSHLOCADDR 573: SAVETOADDR
482: LOADFROMADDR 528: LOADFROMADDR 574: JUMP 593
483: PUSHNUM 1 529: PUSHNUM 1 575: PUSHLOCADDR 2
484: ADD 530: CHANGESIGN 576: LOADFROMADDR
485: SAVETOADDR 531: EQ 577: PUSHLOCADDR 1
486: JUMP 425 532: PUSHLOCADDR 1 578: LOADFROMADDR
487: PUSHLOCADDR 3 533: LOADFROMADDR 579: ADDTOPTR
488: LOADFROMADDR 534: PUSHLOCADDR 580: LOADFROMADDR
489: RETURN 3 535: LOADFROMADDR 581: PUSHLOCADDR 2
490: INITSTKFRM 3 536: LT 582: LOADFROMADDR
491: PUSHLOCADDR 1 537: AND 583: ADDTOPTR
492: PUSHNUM 0 538: OR 584: LOADFROMADDR
493: SAVETOADDR 539: JUMPNFALSE 607 585: WRITEINT
494: PUSHLOCADDR -4 540: PUSHLOCADDR 2 586: WRITESTRING 164 164
495: LOADFROMADDR 541: PUSHNUM 0 587: PUSHLOCADDR 2
496: PUSHNUM 2 542: SAVETOADDR 588: PUSHLOCADDR 2
497: CHANGESIGN 543: PUSHLOCADDR 2 589: LOADFROMADDR
498: EQ 544: LOADFROMADDR 590: PUSHNUM 1
499: JUMPNFALSE 522 545: PUSHLOCADDR -3 591: ADD
500: PUSHLOCADDR 1 546: LOADFROMADDR 592: SAVETOADDR
501: LOADFROMADDR 547: LT 593: JUMP 543
502: PUSHNUM 1000 548: JUMPNFALSE 594 594: PUSHLOCADDR -4
503: LT 549: PUSHLOCADDR -4 595: LOADFROMADDR
504: JUMPNFALSE 512 550: LOADFROMADDR 596: PUSHNUM 0
505: PUSHLOCADDR 1 551: PUSHNUM 1 597: GE
506: PUSHLOCADDR 1 552: CHANGESIGN 598: JUMPNFALSE 600
507: LOADFROMADDR 553: EQ 599: WRITELNOP
508: PUSHNUM 1 554: JUMPNFALSE 575 600: PUSHLOCADDR 1
509: ADD 555: PUSHLOCADDR 3 601: PUSHLOCADDR 1
510: SAVETOADDR 556: PUSHNUM 1 602: LOADFROMADDR
511: JUMP 500 557: HEAPALLOC 603: PUSHNUM 1
512: PUSHSTATADDR 1 558: SAVETOADDR 604: ADD
513: LOADFROMADDR 559: PUSHNUM 2 605: SAVETOADDR
514: WRITELNOP 560: CHANGESIGN 606: JUMP 522
515: WRITELNOP 561: PASSPARAM 607: RETURN 3
516: PUSHSTATADDR 1 562: PUSHNUM 0
517: PUSHSTATADDR 1 563: PASSPARAM

***** Debugging Stop *****

Data memory dump

Data memory--addresses 0 to top of stack, and allocated heap locations:
  0: 2147428131 = POINTER TO 10019
  1: Ctrl-A
  2: 2147428113 = POINTER TO 10001
  3: 'E'
  4: 'n'
  5: 't'
  6: 'e'
  7: 'r'
  8: ' '
  9: 'n'
 10: 'u'
 11: 'm'
 12: 'b'
 13: 'e'
 14: 'r'
 15: ' '
 16: 'O'
 17: 'T'
 18: 'F'
 19: 'I'
 20: 'O'
 21: 'L'
 22: 'S'
 23: ':'
 24: ' '
 25: 'E'
 26: 'n'
 27: 't'
 28: 'e'
 29: 'r'
 30: ' '
 31: 'n'
 32: 'u'
 33: 'm'
 34: 'b'
 35: 'e'
 36: 'r'
 37: ' '
 38: 'O'
 39: 'O'

Page 5 of 8
198: 1 = Ctrl-A
199: 2147428197 = POINTER TO 10085
200: -2
201: 0 = Ctrl-@
202: 2147428197 = POINTER TO 10085
203: 568
204: 214748308 = POINTER TO 196
205: 742
206: 0 = Ctrl-@
207: 0 = Ctrl-@
10000: 2147428118 = POINTER TO 10006
10001: 2147428131 = POINTER TO 10019
10002: 0 = Ctrl-@
10003: 0 = Ctrl-@
10004: 0 = Ctrl-@
10005: 0 = Ctrl-@
10006: 214748124 = POINTER TO 10012
10007: 4 = Ctrl-D
10008: 0 = Ctrl-@
10009: 0 = Ctrl-@
10010: 0 = Ctrl-@
10011: 0 = Ctrl-@
10012: 2147428130 = POINTER TO 10018
10013: 5 = Ctrl-E
10014: 0 = Ctrl-@
10015: 0 = Ctrl-@
10016: 0 = Ctrl-@
10017: 0 = Ctrl-@
10018: 2147428135 = POINTER TO 10023
10019: 2147428136 = POINTER TO 10024
10020: 2147428142 = POINTER TO 10030
10021: 2147428148 = POINTER TO 10036
10022: 2147428154 = POINTER TO 10042
10023: 2147428141 = SOURCE TO 10029
10024: 1 = Ctrl-A
10025: 2 = Ctrl-B
10026: 3 = Ctrl-C
10027: 4 = Ctrl-D
10028: 5 = Ctrl-E
10029: 2147428147 = POINTER TO 10035
10030: 6 = Ctrl-F
10031: 7 = Ctrl-G
10032: 8 = Ctrl-H
10033: 9 = Ctrl-I
10034: 0 = Ctrl-@
10035: 2147428153 = SOURCE TO 10041
10036: 1 = Ctrl-A
10037: 2 = Ctrl-B
10038: 3 = Ctrl-C
10039: 4 = Ctrl-D
10040: 5 = Ctrl-E
10041: 2147428159 = SOURCE TO 10047
10042: 6 = Ctrl-F
10043: 7 = Ctrl-G

Total number of instructions executed: 23172
Last instruction to be executed: 502: PUSHNUM
Expression evaluation stack:

PC=503  ESP=2  FP= POINTER TO 204  ASP= POINTER TO 208
HP= POINTER TO 10086  HMAX=POINTER TO 15000

10044: 8 = Ctrl-H
Comments on the Answers

1(a) The answers are deduced from the operands of the methods' INITSTKFRM instructions at code memory addresses 4, 322, 391, and 490. [It is also possible to work out the answers from the local variable declarations in each method. In main(), for example, the local variables r, c, n, and layer are given the stackframe offsets 1, 2, 3, and 4; i is given offset 5; h is given offset 6; and j is given offset 7. Note that the scopes of local variable declarations need to be taken into account. Thus if we add a declaration of a local variable hh inside the block of the while (h <= layer) { ... } loop that follows the declaration of h, then both hh and j will be given the offset 7 because the scopes of the declarations of hh and j will not overlap.]

(b) For any method other than main():

stackframe size = no. of parameters + 2 + no. of locations allocated to local vars.

The 2 extra locations are for the dynamic link (at offset 0) and the return address (at offset -1).

For main():

stackframe size = 1 + no. of locations allocated to local vars.

In TinyJ, main() is not called by another method and its stackframe has no return address. The INITSTKFRM instruction always allocates a location (offset 0) for a dynamic link, but in the case of main() that location serves no purpose and always points to the illegal data memory address 20000. (The highest legal data memory address is 19999; moreover, data memory addresses 10000 - 19999 are reserved for use as heap memory.)

2. mat's address is 0, count's address is 1, and tm's address is 2. (b) and (c) are intended to test your understanding of arrays. (c) is solved as follows: tm's address is 2. That location points to tm[0], so tm[0]'s addr is 10022. That location points to tm[0][3][0], so tm[0][3][0]'s addr is 10042, and hence tm[0][3][2]'s addr is 10044. That location contains the answer, 8.

3. From the addresses of the INITSTKFRM instructions, we see that main's code is at 4 - 321, readRow's code is at 322 - 390, transpose's code is at 391 - 489, writeOut's code is at 490 - 607. The last instruction to be executed was at 502 (as stated on the 5th-last line of the dump). This is within writeOut's code.

4. We see from FP that offset 0 of the stackframe is at 204. The beginning and end of the stackframe can be deduced from this and the answers to 1(a) and (b) for writeOut.

5. The answers are deduced from the stackframe offsets of the parameters and variables, and the fact that offset 0 is at 204. [In fact the variables j and mm are not in scope in the "while (i < 1000)" loop that is being executed at this time. So the values stored in the locations of j and mm are just "garbage" values!]

6. Return addr (at offset -1, addr 203) is 568. This is within writeOut's code.

7,8. The dynamic link in the stackframe of the currently executing method points to addr 196. That location points to 189. That location points to 179. That location points to 165. Thus 196, 189, 179, and 165 are the addresses of the offset 0 locations in the stackframes of the caller, the caller's caller, the caller's caller's caller, and the caller's caller's caller's caller, respectively.

Note: Another way to tell that the caller's caller's caller's caller is main is to observe that offset 0 in its stackframe (addr 165) points to the illegal data memory address 20000--see the above comment on question 1(b).

9. Offset 0 in main's frame is at addr 165 (see comments on questions 7,8). h's stackframe offset is 6 and j's is 7.

10. PC contains 503, so 503: LT is the first of the 10 instructions. We see from the last few lines of the dump (on p. 7) that at this time ESP = 2, EXPRSTACK[0] = 742, and EXPRSTACK[1] = 1000. Thus 1000 is on top of EXPRSTACK and 742 is the second item from the top. Since 742 < 1000, execution of LT replaces these two integers with the value 1 (which represents true), so the JUMPONFALSE at 504 does not jump after popping off this value.

11. Only WRITEINT, WRITESTRING, and WRITELNOP produce output.

12. Data memory is changed only by SAVETOADDR, PASSPARAM, CALLSTATMETHOD, INITSTKFRM, and HEAPALLOC. The only one of these that is executed here is SAVETOADDR (at 510). When this is executed, the pointer that is second from top on EXPRSTACK was put there by 505: PUSHLOCADDR 1. This refers to offset 1 in the currently executing method's stackframe, which is the location of i and has address 205 (since offset 0 has address 204).

[Note: HEAPALLOC changes data memory only because it sets the location that immediately precedes the block of heap memory it allocates to point to the location that immediately follows the block. This allows allocated blocks of heap memory that have become inaccessible to be deallocated by the garbage collector, and makes it possible to check at runtime that every array index is less than the length of the array.]

13,14. Questions like these are intended to test your understanding of what specific machine instructions do to the TinyJ virtual machine. Here the instructions you are being tested on are LT, JUMPONFALSE, PUSHLOCADDR, and RETURN.
The dump below was produced when TJasn.TJ compiled the TinyJ program on p. 2 and executed the generated code with a debugging stop after execution of 1,209,788 instructions. The sequence of input values was 4, 3, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2. The INITSTKFRM instructions in the generated code are: 4: INITSTKFRM 7 339: INITSTKFRM 4 408: INITSTKFRM 6 509: INITSTKFRM 5

The instructions at addresses 351 – 407 in the generated code are shown on page 3.

Some Examples of Possible Questions Relating to the Dump

1. (a) For each method, say how many locations are allocated to local variables in its stackframes.
   
   ANSWER: main: 7; readRow: 4
   transpose: 6; writeOut: 5

   (b) Write down the size of a stackframe of readRow(), transpose(), and writeOut().
   
   ANSWER: readRow: 10; transpose: 13; writeOut: 10

Questions 2 – 8 are about the state of the TinyJ virtual machine at the time of the debugging stop after execution of 1,209,788 instructions:

2. Consider the static variables mat, count, and tm. What values are stored in the following locations?
   
   (a) count
   ANSWER: 100

   (b) mat[3][2]
   ANSWER: 2

   (c) tm[0][1][2]
   ANSWER: 6

3. Which method is being executed? ANSWER: readRow

4. Which data memory locations constitute the stackframe of the executing method?
   ANSWER: addresses 170 through 179

5. What values are stored in the stackframe locations of the formal parameters and first two local variables of the executing method?
   ANSWER: rowNum = -1
   m = PTR TO 10059
   c = 10  d = 6
   i = 0  mm = PTR TO 10061

6. Which method called the executing method?
   ANSWER: transpose

7. Which method called the caller? ANSWER: main

8. What are the addresses of main’s local variables layer and hhhhh?
   ANSWER: layer’s addr is 151; hhhhh’s addr is 154

Next, suppose the debugging stop had not occurred.

9. What would be the code memory addresses of the next 10 instructions to be executed (i.e., the 1,209,789th through 1,209,798th instructions to be executed)?
   ANSWER: 383–5, then 406, then 353–8

10. What output, if any, would be produced by execution of these 10 instructions? ANSWER: None

11. Which data memory locations, if any, would be changed in value by execution of the 10 instructions? Name the variable(s) stored there and say what its/their value(s) is/are after execution of the 10 instructions.
   
   ANSWER: address 176, i, 1

12. Write down what the PC, FP, ASP and ESP registers would contain after execution of the first 3 of the above 10 instructions.
   ANSWER: PC=406, FP=PTR TO 175, ASP=PTR TO 180, ESP=0

13. When the currently executing method activation RETURNs to its caller, what will PC, FP, and ASP be set to?
   
   ANSWER: PC=492, FP=PTR TO 163, ASP=PTR TO 170

14. What is the code memory address of the next instruction to be executed after the execution of the 10 instructions listed in your answer to question 9?
   
   ANSWER: 359

15. What will be on top of EXPRSTACK after execution of the instruction in your answer to question 14?
   
   ANSWER: PTR TO 170
```java
import java.util.Scanner;

class DumpEx2 {
    static int tm[][][ ] = new int[5][ ][ ];
    static int mat[][], count;
    static Scanner input = new Scanner(System.in);

    public static void main (String args[]) {
        int r[] = new int[5], c[] = new int[5], n = 1;
        int layer = -1;
        while (n == 1) {
            if (layer < 4)
                layer = layer + 1;
            else
                layer = 0;

            System.out.print("Enter number of rows: ");
            r[layer] = input.nextInt();
            System.out.print("Enter number of columns: ");
            c[layer] = input.nextInt();

            mat = new int[r[layer]][ ];
            tm[layer] = mat;
            int i = 0;
            while (i < r[layer]) {
                int iiiii = i;
                mat[i] = new int[c[layer]];
                readRow(i + 1, mat[i], c[layer], iiiii);
                i = i + 1;
            }

            int h = 0;
            while (h <= layer) {
                int hhhhh = h*2;
                System.out.println("Given matrix: ");
                writeOut(r[h], c[h], tm[h]);
                System.out.println("Transposed matrix: ");
                writeOut(c[h], r[h], transpose(tm[h], r[h], c[h], hhhhh, hhhhh));
                h = h + 1;
            }
            h = 0;
            while (h <= layer) {
                i = 0;
                while (i < r[h]) {
                    int j = 0;
                    while (j < c[h]) {
                        tm[h][i][j] = tm[h][i][j] * 2;
                        System.out.print(" ");
                        j = j + 1;
                    }
                    System.out.println();
                    i = i + 1;
                }
                System.out.println("\n");
            }

            int jjjjj;
            System.out.print("*\nType 1 to continue, 0 to quit: ");
            n = input.nextInt();
        }
    }

    static int[][] transpose(int m[][], int r, int c, int p, int q) {
        int temp, k, ml[][], r1, c1;
        while (k < c) {
            ml[k] = new int[r1];
            k = k + 1;
        }
        i = 0;
        while (i < r) {
            int j = 0;
            while (j < c) {
                ml[j][i] = m[i][j];
                System.out.print(" ");
                j = j + 1;
            }
            System.out.println();
            i = i + 1;
        }
        return ml;
    }

    static void readRow (int rowNum, int m[ ], int c[ ], int r, int c, int p, int q) {
        int mm[][] = new int[1][1];
        mm[0][0] = input.nextInt();
        System.out.println("Enter value in column ");
        System.out.print(i+1);
        System.out.print(" ");
        m[i] = input.nextInt();
        i = i + 1;
    }

    static void readRow (int rowNum, int m[][ ], int c[ ], int r, int c, int n) {
        int temp, k, ml[][], r1, c1;
        while (k < c) {
            ml[k] = new int[r1];
            k = k + 1;
        }
        i = 0;
        while (i < r) {
            int j = 0;
            while (j < c) {
                ml[j][i] = m[i][j];
                System.out.print(" ");
                j = j + 1;
            }
            System.out.println();
            i = i + 1;
        }
        return ml;
    }

    static void writeOut (int rows, int cols, int matrix[][ ][]) {
        int i = 0, tmp, tmp1;
        if (rows == -2) {
            while (i < 1000) i = i + 1;
            System.out.println(matrix[0][1] + 1);
            count = count+1;
        }
        while (i < rows | rows == -1) {
            int j = 0;
            while (j < cols) {
                matrix[i][j] = tm[i][j];
                System.out.println();
                j = j + 1;
            }
            System.out.println("\n");
            i = i + 1;
        }
    }
}
```