Assigning 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, worth 5 pts., that is similar in nature to the examples below. (The maximum score on the Final Exam will be 40 pts.) For more on call/pass by name, see pp. 455 – 6 of this book, which can be viewed via these links: p. 455* p. 456


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.

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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reference</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W)</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>name</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

SOLUTIONS

Problem 1:

Output for each parameter passing mode:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reference</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W)</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>name</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Example 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.

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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reference</td>
<td>9</td>
<td>-1</td>
<td>-1</td>
<td>9</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>value-result (Algol W)</td>
<td>4</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>name</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

*While Algol W is listed on p. 455 as a language that dropped pass by name, name parameters were in fact supported by Algol W, though the best known Algol W compiler discouraged the use of name parameters by issuing a warning (warning 2031 on p. 80 of this manual) if they were used.
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
   
   ```
   void p(int a, int b)
   {
       a = 4;
       b = 7;
   }
   ```

   where the parameters `a` and `b` are passed by value-result. Suppose `main` calls this function as follows:
   
   ```
   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 pass by value-result doesn't say whether (i) or (ii) occurs first. If (ii) occurs first the final value of `j` will be 4, the final value of `a`; if (i) occurs first the final value of `j` will be 7, the final value of `b`.

   If you were asked to write down the output of `System.out.print(j)` assuming pass by value-result, neither "4" nor "7" would be correct—write "4 or 7". In Ada, pass by value-result is the default parameter passing method for "in out" parameters of a scalar type (like Integer),* but Ada avoids the above-mentioned issue by disallowing calls like `p(j, j)` that pass the same variable as the argument for two such parameters.

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 Algol W, the first language to support pass by value-result. 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 the call was made (i.e., into the location that belonged to the actual argument variable immediately before the called function's body was executed). 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 (i.e., into the location that belongs to the actual argument variable immediately after the called function's body is executed).**

   These two versions of pass by value-result may produce different results if the actual argument is, e.g., an indexed variable `v[expr]` whose index expression `expr` changes in value during execution of the called function's body. As an example, consider a function
   
   ```
   void q(int c)
   {
       c = 55;
       i = 17;
   }
   ```

   where `i` is a global variable and `i`'s parameter `c` is passed by value-result. Suppose this function `q` is called within the function `main` as follows:
   
   ```
   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 immediately before the body of `q` was executed (and so `arr[i]` was `arr[23]`). 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 immediately after the body of `q` is executed (so that `arr[i]` is `arr[17]`).

*This is true of Ada 2012 but not fully consistent with the remarks about Ada on p. 159 of Sethi’s book: Sethi’s remarks apply to an older version of Ada.

**Students interested in learning more may refer to subsections 5.3.2.2 and 7.3.2 in Part II of N. Wirth and C. A. R. Hoare, A contribution to the development of Algol, Communications of the ACM, vol. 9, 1966, 413–32.
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.

Examples of Possible Questions Relating to the Dump (Some explanatory comments are given on p. 8.)

- **Recommendation:** Work on the *green questions* shortly before or soon after doing TJ Assignment 2.

  - **Hint for question 2:** If $x$ is an array variable and $x !=$ null, then $x$ stores a pointer to the location $x[\theta]$, and address of $x[\theta] = n +$ address of $x[\theta]$. If the array is an array of arrays, then each array element $x[k]$ stores a pointer to the location $x[k][\theta]$, and address of $x[k][\theta] = n +$ address of $x[k][\theta].$

  - **Hint for questions 4 and 5:** The FP register—see the “PC=503 ESP=2 FP=…” line of the dump—points to the location at offset 0 in the currently executing method activation’s stackframe.

  - **Hint for questions 10, 11, 12, and 15:** The ESP register—see the “PC=503 ESP=2 FP=…” line of the dump—contains a count of the number of items that are currently on EXPRSTACK. When ESP > 0, EXPRSTACK[ESP-1] is the *top* item on EXPRSTACK and EXPRSTACK[ESP] is the *bottom* item.

- **Recommendation:** Work on the *red questions* soon after the last lecture of the course.

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

2. Consider the static variables mat, count, and tm.

   - What values are stored in the following locations?
     - (a) count
     - (b) mat[2][4]
     - (c) tm[0][3][2]
   
   **ANSWER:** count: 1; mat[2][4]: 5; tm[0][3][2]: 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

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

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

15. Answer questions 10 – 12 and 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 of the 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) {
                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;
                }
                h = h + 1;
                System.out.println("\n");
            }
            System.out.print("\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) {
                System.out.print(-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];
            k = k + 1;
        }
        i = 0;
        while (i < r) {
            int j = 0;
            while (j < c) {
                m1[j][i] = m[i][j];
                int mm[ ] = new int [1];
                readRow(-1, mm, 10);
                j = j + 1;
            }
            i = i + 1;
        }
        return m1;
    }

    static void writeOut (int rows, int cols, int matr[ ][ ]) {
        int i = 0;
        if (rows == -2) {
            while (i < 1000) i = i + 1;
            System.out.println(count);
            count = count+1;
        }
        while (i < rows | rows == -1 & i < cols) {
            int j = 0;
            while (j < cols) {
                if (rows == -1) {
                    int mm[ ][ ] = new int[1][ ];
                    writeOut(-1, 0, mm);
                    j = j + 1;
                } else {
                    System.out.print(matr[i][j]);
                    System.out.print(" ");
                    j = j + 1;
                }
            }
            if (rows >= 0) System.out.println();
            i = i + 1;
        }
    }
}
Instructions Generated:

0: PUSHSTATADDR 2
1: PUSHNUM 5
2: HEAPALLOC
3: SAVEADDR
4: INITSTXFRM 7
5: PUSHLOCADDR 1
6: PUSHNUM 5
7: HEAPALLOC
8: SAVEADDR
9: PUSHLOCADDR 2
10: PUSHNUM 5
11: HEAPALLOC
12: SAVEADDR
13: PUSHLOCADDR 3
14: PUSHNUM 1
15: SAVEADDR
16: PUSHLOCADDR 4
17: PUSHNUM 1
18: CHANGESIGN
19: SAVEADDR
20: PUSHLOCADDR 3
21: LOADFROMADDR
22: PUSHNUM 1
23: EQ
24: JUMPONFALSE 321
25: PUSHLOCADDR 4
26: ADDTOPTADDR
27: PUSHNUM 1
28: ADD
29: JUMPONFALSE 127
30: PUSHLOCADDR 4
31: LOADFROMADDR
32: PUSHLOCADDR 1
33: ADDTOPTADDR
34: ADD
35: PUSHLOCADDR
36: JUMP 40
37: PUSHLOCADDR 4
38: PUSHNUM 0
39: SAVEADDR
40: WRITESTRING 3 24
41: PUSHLOCADDR 1
42: LOADFROMADDR
43: PUSHLOCADDR 4
44: LOADFROMADDR
45: ADDTOPTADDR
46: READINT
47: SAVEADDR
48: WRITESTRING 25 49
49: PUSHLOCADDR 2
50: LOADFROMADDR
51: PUSHLOCADDR 4
52: LOADFROMADDR
53: ADDTOPTADDR
54: READINT
55: SAVEADDR
56: PUSHSTATADDR
57: PUSHLOCADDR 1
58: LOADFROMADDR
59: PUSHLOCADDR 4
60: LOADFROMADDR
61: ADDTOPTADDR
62: LOADFROMADDR
63: PUSHLOCADDR 2
64: LOADFROMADDR
65: PUSHLOCADDR 4
66: LOADFROMADDR
67: PUSHLOCADDR 4
68: LOADFROMADDR
69: ADDTOPTADDR
70: PUSHSTATADDR
71: LOADFROMADDR
72: SAVEADDR
448: ADDTOPYTR 502: PUSHNUM 1000 556: PUSHNUM 1
449: PUSHLOCADDR -4 503: LT 557: HEAPALLOC
450: LOADFROMADDR 504: JUMPPFALSE 512 558: SAVETOADDR
451: PUSHLOCADDR 2 505: PUSHLOCADDR 1 559: PUSHNUM 2
452: LOADFROMADDR 506: PUSHLOCADDR 1 560: CHANGESIGN
453: ADDTOPYTR 507: LOADFROMADDR 561: PASSPARAM
454: LOADFROMADDR 508: PUSHNUM 1 562: PUSHNUM 0
455: PUSHLOCADDR 4 509: ADD 563: PASSPARAM
456: LOADFROMADDR 510: SAVETOADDR 564: PUSHLOCADDR 3
457: ADDTOPYTR 511: JUMP 500 565: LOADFROMADDR
458: LOADFROMADDR 512: PUSHSTATADDR 1 566: PASSPARAM
459: SAVETOADDR 513: LOADFROMADDR 567: CALLSTATMETHOD 490
460: PUSHLOCADDR 5 514: WRITETYPE 568: PUSHLOCADDR 2
461: PUSHNUM 1 515: WRITENOP 569: PUSHLOCADDR 2
462: HEAPALLOC 516: PUSHSTATADDR 1 570: LOADFROMADDR
463: SAVETOADDR 517: PUSHSTATADDR 1 571: PUSHNUM 1
464: PUSHNUM 1 518: LOADOFRADDR 572: ADD
465: CHANGESIGN 519: PUSHNUM 1 573: SAVETOADDR
466: PASSPARAM 520: ADD 574: JUMP 593
467: PUSHLOCADDR 5 521: SAVETOADDR 575: PUSHLOCADDR -2
468: LOADFROMADDR 522: PUSHLOCADDR 1 576: LOADFROMADDR
469: PASSPARAM 523: LOADFROMADDR 577: PUSHLOCADDR 1
470: PUSHNUM 10 524: PUSHLOCADDR -4 578: LOADFROMADDR
471: PASSPARAM 525: LOADFROMADDR 579: ADDTOPYTR
472: CALLSTATMETHOD 322 526: LT 580: LOADFROMADDR
473: PUSHLOCADDR 4 527: PUSHLOCADDR -4 581: PUSHLOCADDR 2
474: PUSHLOCADDR 4 528: PUSHLOCADDR 582: LOADFROMADDR
475: LOADFROMADDR 529: PUSHNUM 1 583: ADDTOPYTR
476: PUSHNUM 1 530: CHANGESIGN 584: LOADFROMADDR
477: ADD 531: EQ 585: WRITETYPE
478: SAVETOADDR 532: PUSHLOCADDR 1 586: WRITESTRING 164 164
479: JUMP 434 533: LOADFROMADDR 587: PUSHLOCADDR 2
480: PUSHLOCADDR 2 534: PUSHLOCADDR -3 588: PUSHLOCADDR 2
481: PUSHLOCADDR 535: LOADFROMADDR 589: LOADFROMADDR
482: LOADFROMADDR 536: LT 590: PUSHNUM 1
483: PUSHNUM 1 537: AND 591: ADD
484: ADD 538: OR 592: SAVETOADDR
485: SAVETOADDR 539: JUMPPFALSE 607 593: JUMP 543
486: JUMP 425 540: PUSHLOCADDR 2 594: PUSHLOCADDR -4
487: PUSHLOCADDR 3 541: PUSHNUM 0 595: LOADFROMADDR
488: LOADFROMADDR 542: SAVETOADDR 596: PUSHNUM 0
489: RETURN 3 543: PUSHLOCADDR 2 597: GE
490: INITSTKFRM 3 544: LOADFROMADDR 598: JUMPPFALSE 600
491: PUSHLOCADDR 1 545: PUSHLOCADDR -3 599: WRITETYPE
492: PUSHNUM 0 546: LOADFROMADDR 600: PUSHLOCADDR 1
493: SAVETOADDR 547: LT 601: PUSHLOCADDR 1
494: PUSHLOCADDR -4 548: JUMPPFALSE 594 602: LOADFROMADDR
495: LOADFROMADDR 549: PUSHLOCADDR -4 603: PUSHNUM 1
496: PUSHNUM 2 550: LOADFROMADDR 604: ADD
497: CHANGESIGN 551: PUSHNUM 1 605: SAVETOADDR
498: EQ 552: CHANGESIGN 606: JUMP 522
499: JUMPPFALSE 553 553: EQ 607: RETURN 3
500: PUSHLOCADDR 1 554: JUMPPFALSE 575
501: LOADFROMADDR 555: PUSHLOCADDR 3

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

Data memory dump

Data memory--addresses 0 to top of stack, and allocated heap locations:
0: 2147428131 = PTR TO 10019
1: Ctrl-A
2: 2147428133 = PTR TO 10001
3: 69 = 'E'
4: 110 = 'n'
5: 116 = 't'
6: 101 = 'e'
7: 114 = 'r'
8: 32 = ' ' 25: 69 = 'E'
9: 110 = 'n'
10: 117 = 'u'
11: 109 = 'm'
12: 98 = 'b'
13: 101 = 'e'
14: 114 = 'r'
15: 32 = ' ' 2147428113 = PTR TO 10019
16: 111 = 'o'
17: 102 = 'f'
18: 32 = ' ' 2147428114 = PTR TO 10001
20: 111 = 'o'
21: 119 = 'u'
22: 115 = 'a'
23: 58 = 'i'
24: 32 = ' ' 25: 69 = 'E'
26: 110 = 'n'
27: 116 = 't'
28: 101 = 'e'
29: 114 = 'r'
180: 5 = Ctrl-E
181: 0 = Ctrl-0
182: 2147428160 = PTR TO 10048
183: 0 = Ctrl-0
184: 2147428191 = PTR TO 10079
185: -1
186: 2147428191 = PTR TO 10079
187: 10 - Ctrl-J
188: 473
189: 2147428191 = PTR TO 10079
190: 0 = Ctrl-0
191: 2147428193 = PTR TO 10081
192: -1
193: 10 = Ctrl-J
194: 2147428193 = PTR TO 10081
195: 361
196: 2147428193 = PTR TO 10081
197: 0 = Ctrl-0
198: 1 = Ctrl-A
199: 2147428197 = PTR TO 10085
200: -2
201: 0 = Ctrl-0
202: 2147428197 = PTR TO 10085
203: 568
204: 2147428130 = PTR TO 196
205: 742
206: 0 = Ctrl-0
207: 0 = Ctrl-0
10000: 214742818 = PTR TO 1006
10001: 2147428131 = PTR TO 10019
10002: 0 = Ctrl-0
10003: 0 = Ctrl-0
10004: 0 = Ctrl-0
10005: 0 = Ctrl-0
10006: 2147428124 = PTR TO 10012
10007: 4 = Ctrl-D
10008: 0 = Ctrl-0
10009: 0 = Ctrl-0
10010: 0 = Ctrl-0
10011: 0 = Ctrl-0
10012: 2147428130 = PTR TO 10018
10013: 5 = Ctrl-E
10014: 0 = Ctrl-0
10015: 0 = Ctrl-0
10016: 0 = Ctrl-0
10017: 0 = Ctrl-0
10018: 2147428135 = PTR TO 10023
10019: 2147428136 = PTR TO 10024
10020: 2147428142 = PTR TO 10030
10021: 2147428148 = PTR TO 10036
10022: 2147428154 = PTR TO 10042
10023: 2147428141 = PTR TO 10029
10024: 1 = Ctrl-A
10025: 2 = Ctrl-B
10026: 3 = Ctrl-C
10027: 4 = Ctrl-D
10028: 5 = Ctrl-E
10029: 2147428147 = PTR TO 10035
10030: 6 = Ctrl-F
10031: 7 = Ctrl-G
10032: 8 = Ctrl-H
10033: 9 = Ctrl-I
10034: 0 = Ctrl-0
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 10001. That location points to tm[0][0], so tm[0][0]'s addr is 10019, and hence tm[0][0][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, and 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. The return addresses stored in the first three of these stackframes (at addresses 195, 188, and 178) are 361, 473, and 199, which are instructions in the code of readRow, transpose, and main, 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 205) 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 JUMPFALSE 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 VM. Here the instructions you are being tested on are LT, JUMPFALSE, PUSHLOCADDR, and RETURN.

Regarding question 14: RETURN must set FP to point to offset 0 in the caller's stackframe; the dynamic link is a pointer to that location. RETURN must put the return address into PC. RETURN must set ASP to point to the first location (i.e., the location that has the most negative offset) in the currently executing method activation's stackframe; that will deallocate the stackframe. The dynamic link and the return address are respectively stored at offset 0 and offset -1 in the currently executing method activation's stackframe.
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

- **Recommendation**: Work on the green questions shortly before or soon after doing TJ Assignment 2.
  
  **Hint for question 2**: If \( x \) is an array variable and \( x \neq \text{null} \), then \( x \) stores a pointer to the location \( x[0] \), and address of \( x[n] = n + \text{address of } x[0] \). If the array is an array of arrays, then each array element \( x[k] \) stores a pointer to the location \( x[k][0] \), and address of \( x[k][n] = n + \text{address of } x[k][0] \).

- **Recommendation**: Work on the red questions soon after the last lecture of the course.

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
import java.util.Scanner;

class DumpEx2 {
    static int tm[][][] = new int[5][5][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]][c[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) {
                System.out.println("\n");
                i = 0;
                while (i < r[h]) {
                    int j = 0;
                    while (j < c[h]) {
                        tm[h][i][j] = tm[h][i][j] * 2;
                        System.out.println(tm[h][i][j]);
                        j = j + 1;
                    }
                    System.out.println();
                    i = i + 1;
                }
                h = h + 1;
                System.out.println("\n");
            }

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

    static void readRow (int rowNum, int m[], int c, int d) {
        if (rowNum >= 0) {
            System.out.print("Row ");
            System.out.println(rowNum);
        }
        int i = 0;
        while (i < c) {
            int mm[] = new int[c];
            mm[i] = input.nextInt();
            i = i + 1;
        }
    }

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

    static void writeOut (int rows, int cols, int matrix[][][]) {
        int i = 0, tmp, tmp1;
        if (rows == -2) {
            while (i < 1000) i = i + 1;
        }
        while (i < rows) {
            int j = 0;
            while (j < cols) {
                if (rows == -1 & j < cols) {
                    int j = 0;
                    while (j < cols) {
                        if (rows == -1) {
                            int mm[] = new int[1];
                            mm[j] = input.nextInt();
                            j = j + 1;
                        }
                        System.out.println(matrix[i][j]);
                    }
                    System.out.println("\n");
                }
                if (rows == 0) System.out.println();
                j = j + 1;
            }
        }
    }
}
351: PUSHNUM 0 10: 117 = 'u'
352: SAVETOADDR 11: 109 = 'm'
353: PUSHLOCADDR 12: 98 = 'b'
354: LOADFROMADDR 13: 101 = 'e'
355: PUSHLOCADDR 14: 114 = 'r'
356: LOADFROMADDR 15: 32 = '
357: LT 16: 111 = 'o'
358: JUMPNFALSE 17: 102 = 'f'
359: PUSHLOCADDR 18: 32 = '
360: LOADFROMADDR 19: 114 = 'r'
361: PUSHNUM 20: 111 = 'o'
362: CHANGESIGN 21: 119 = 'w'
363: EQ 22: 115 = 's'
364: JUMPNFALSE 23: 58 = ':'
365: PUSHLOCADDR 24: 32 = '
366: PUSHNUM 25: 69 = 'E'
367: HEAPALLOC 26: 110 = 'n'
368: SAVETOADDR 27: 116 = 't'
369: PUSHNUM 28: 101 = 'e'
370: CHANGESIGN 29: 114 = 'r'
371: PASSPARAM 30: 32 = '
372: PUSHNUM 31: 110 = 'n'
373: PASSPARAM 32: 117 = 'u'
374: PUSHLOCADDR 33: 109 = 'm'
375: LOADFROMADDR 34: 98 = 'b'
376: PASSPARAM 35: 101 = 'e'
377: CALLSTATMETHOD 36: 114 = 'r'
378: NOP 37: 32 = '
379: PUSHLOCADDR 38: 111 = 'o'
380: PUSHLOCADDR 39: 102 = 'f'
381: LOADFROMADDR 40: 32 = '
382: PUSHNUM 41: 99 = 'c'
383: ADD 42: 111 = 'o'
384: SAVETOADDR 43: 108 = 'l'
385: JUMP 44: 117 = 'u'
386: WRITESTRING 45: 109 = 'm'
387: PUSHLOCADDR 46: 110 = 'n'
388: LOADFROMADDR 47: 115 = 's'
389: PUSHNUM 48: 58 = ':'
390: ADD 49: 32 = '
391: WRITEINT 50: 71 = 'G'
392: WRITESTRING 51: 105 = 'i'
393: PUSHLOCADDR -4 52: 118 = 'v'
394: LOADFROMADDR 53: 101 = 'e'
395: PUSHLOCADDR 54: 110 = 'n'
396: LOADFROMADDR 55: 32 = '
397: ADDTOPTR 56: 109 = 'm'
398: READINT 57: 97 = 'a'
399: SAVETOADDR 58: 116 = 't'
400: PUSHLOCADDR 59: 114 = 'r'
401: PUSHLOCADDR 60: 105 = 'i'
402: LOADFROMADDR 61: 120 = 'x'
403: PUSHNUM 62: 58 = ':'
404: ADD 63: 32 = '
405: SAVETOADDR 64: 84 = 'T'
406: JUMP 353 65: 114 = 'r'
407: RETURN 4 66: 97 = 'a'

Data memory dump
Data memory—addresses 0 to top of stack, and allocated heap locations:
0: PTR TO 10001 72: 101 = 'e'
1: PTR TO 10019 73: 100 = 'd'
2: 100 = 'd' 74: 32 = '
3: 109 = 'e' 75: 109 = 'm'
4: 110 = 'n' 76: 97 = 'a'
5: 116 = 't' 77: 116 = 't'
6: 101 = 'e' 78: 114 = 'r'
7: 114 = 'r' 79: 105 = 'i'
8: 32 = ' ' 80: 120 = 'x'
9: 110 = 'n' 81: 58 = 'i'
10: 117 = 'u' 82: 32 = ' '