

**Written exam in
Introductory Programming**

Model solutions

IT-C, June 12, 2002
English version

Question 1

Question 1.1

c: 9 x: 9 y: 3 z: 2
c: 7 x: 9 y: 3 z: 2
c: 5 x: 9 y: 3 z: 2

Question 1.2

c: 6 x: 6 y: 1 z: 2
c: 4 x: 6 y: 1 z: 2
c: 2 x: 6 y: 1 z: 2
c: 5 x: 5 y: 2 z: 3
c: 4 x: 4 y: 3 z: 4

Question 1.3

```
public class ZeroesOnes
{
    public static void classify(int[][] arr) {
        int zeroes = 0;
        int ones = 0;
        int others = 0;
        for (int i = 0; i < arr.length; i++)
            for (int j = 0; j < arr[i].length; j++)
                switch (arr[i][j]) {
                    case 0: zeroes++; break;
                    case 1: ones++; break;
                    default: others++;
                }
        System.out.println("    0: " + zeroes);
        System.out.println("    1: " + ones);
        System.out.println("other: " + others);
    }

    public static void test() {
        int[][] a = {{0, 1, 8, 9}, {4, 4, 1, 0}, {1, 1, 0}};
        classify(a);
    }
}
```

Question 2

Questions 2.1-2.3

```
class Hat {
    protected char printSymbol;

    public Hat(char printSymbol) {
        this.printSymbol = printSymbol;
    }
    // prints a number of characters in a row
    private void draw(char symbol, int times) {
        for (int i = 0; i < times; i++)
            System.out.print(symbol);
    }
    // prints a new line
    private void draw() {
        System.out.println();
    }
    public void print() {
        draw(' ', 3); draw(printSymbol, 5); draw();
        draw(' ', 3); draw(printSymbol, 1); draw(' ', 3); draw(printSymbol, 1); draw();
        draw(' ', 3); draw(printSymbol, 1); draw(' ', 3); draw(printSymbol, 1); draw();
        draw(' ', 3); draw(printSymbol, 1); draw(' ', 3); draw(printSymbol, 1); draw();
        draw(printSymbol, 11); draw();
    }
    public void print(int topWidth, int bottomWidth, int height) {
        int leftMargin = (bottomWidth - topWidth) / 2;
        int hatInteriorWidth = topWidth - 2;
        int hatInteriorHeight = height - 2;
        draw(' ', leftMargin); draw(printSymbol, topWidth); draw();
        for (int i = 0; i < hatInteriorHeight; i++) {
            draw(' ', leftMargin); draw(printSymbol, 1); draw(' ', hatInteriorWidth);
            draw(printSymbol, 1); draw();
        }
        draw(printSymbol, bottomWidth); draw();
    }
    public boolean valid(int topWidth, int bottomWidth, int height) {
        return (topWidth >= 3 && bottomWidth > topWidth &&
            (bottomWidth - topWidth) % 2 == 0 && height >= 3);
    }
    public void printValid(int topWidth, int bottomWidth, int height) throws Exception {
        if (valid(topWidth, bottomWidth, height))
            print(topWidth, bottomWidth, height);
        else
            throw new Exception("Invalid input arguments: topWidth = " +
                topWidth + "; bottomWidth = " +
                bottomWidth + "; height = " + height + ".");
    }
}
```

Question 2.4

The definition of `valid` in `SubHat` overrides the definition of `valid` in `Hat`. Furthermore, `printValid` is inherited by `SubHat` from `Hat`, which calls `valid` by dynamic dispatch. Thus the method invocation

of `valid` during execution of `hat2.printValid` returns true, and the `print` method is consequently called, instead of an exception raised. This in turn, with the given code for `print` here, results in the printing of the following degenerate hat, consisting of 4 symbols on two lines:

```
xx  
xx
```

Note that 'x's are printed not 'o's since `draw` in `Hat` is declared private and is thus *not* overridden.

NB: This is a question testing the student's understanding of the technical details of inheritance, overloading, overriding, dynamic dispatch, access modifiers and subtype polymorphism, and their (complex) interaction in Java. Understanding precisely the semantics of the individual concepts must be considered at the upper edge of 'GP pensum', and their interaction strictly speaking beyond that, but within the reach of independent deductive combination of knowledge acquired during the course.

Question 3

Question 3.1

Freddy
Madonna
George

Question 3.2

The downcast operation is necessary because the return type of the `get`-method is `Object` and the type of name is `String`. Since `Object` is the same as or a subtype of `String` this would result in a compile-time error. The downcast operation makes the type of the right-hand side be `String` and so this compile-time error is avoided.

Question 3.3

```
public class FlexibleVector extends SimpleVector {
    public FlexibleVector(int size) {
        super(size);
    }
    public void add(Object obj) {
        arr[length++] = obj;
    }
}
```

Question 3.4

```
public class UnboundedFlexibleVector extends SimpleVector {
    public UnboundedFlexibleVector(int size) {
        super(size);
    }

    // double length of array reference by arr
    private void doubleArray() {
        Object[] newarr = new Object[2 * arr.length];
        for (int i = 0; i < arr.length; i++)
            newarr[i] = arr[i]; // copy old array to (first half of) new array
        arr = newarr; // update instance variable to point to new array
    }
    public void set(int index, Object obj) {
        try {
            super.set(index, obj);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            doubleArray(); // make arr twice as big
            set(index, obj); // try set operation again
        }
    }
    public void add(Object obj) {
        if (length >= arr.length)
            doubleArray();
        arr[length++] = obj;
    }
}
```

Question 4

Question 4.1

Question 4.1.1

```
Employee2 henrik = new Employee2();
henrik.name = "Henrik"; // set name
henrik.id = Employee2.nextId++; // set id and increment nextId
System.out.println("Employee: " + henrik.name + ", " + henrik.id);
```

Question 4.1.2

Class `Employee` does not allow public access to instance variables `name`, `id` and to class variable `nextId`, only read access to `name` and `id` through accessor methods. This guarantees that neither `name` nor `id` can be changed by other code. Furthermore, the constructor guarantees that every instance variable is assigned a unique, sequentially numbered `id`. Class `Employee2` allows write access to `name`, `id` and `nextId`, which means that names and ids of employee instances can be changed arbitrarily and unpredictably by client code; e.g., two or more employees may have the same `id`.

Question 4.2

Question 4.2.1

```
public class ManagedEmployee extends Employee
{
    ManagedEmployee manager;
    public ManagedEmployee(String name) {
        super(name);
    }
    public void setManager(ManagedEmployee manager) {
        this.manager = manager;
    }
    public ManagedEmployee getManager() {
        return manager;
    }
}
```

Question 4.2.2

```
ManagedEmployee fritz = new ManagedEmployee("Fritz");
ManagedEmployee henrik = new ManagedEmployee("Henrik");
ManagedEmployee mads = new ManagedEmployee("Mads");
henrik.setManager(mads);
frtiz.setManager(henrik);
```

Question 4.3

Question 4.3.1

```
// getDirector (recursive)
public ManagedEmployee getDirector() {
    if (manager == null)
        return this;
    else
        return manager.getDirector();
}
```

```

// getDirector (iterative)
public ManagedEmployee getDirector2() {
    ManagedEmployee candidate = this;
    ManagedEmployee candidatesManager = manager;
    while (candidatesManager != null) {
        candidate = candidatesManager;
        candidatesManager = candidatesManager.getManager();
    }
    return candidate;
}

```

Question 4.3.2

The 2 calls to `setManager` introduce a cycle into the management relations. Since Per is Hans's manager and Hans is Per's manager, the instance variable `manager` in both is nonnull. For the recursive definition of `getDirector`, the recursive calls to `getDirector` thus never terminate. (At some point a stack overflow occurs, though.) For the iterative definition, the same argument applies: Since the value of `candidatesManager` never becomes nonnull, the while-loop never terminates.

Question 4.4

Question 4.4.1

```

public class EmployeeInf {
    public static void printNumSubordinates(ManagedEmployee[] emps) {
        int[] subords = new int[10001]; // new array whose elements are set to 0
        for (int i = 0; i < emps.length; i++) {
            ManagedEmployee mgr = emps[i].getManager(); // find manager of employee i
            if (mgr != null)
                subords[mgr.getId()]++; // add 1 to subordinates
        }
        for (int j = 1; j < subords.length; j++) {
            System.out.println("Employee " + j + " has " + subords[j] + " subordinates.");
        }
    }
}

```

Question 4.4.2

To produce the employees in sorted order we allocate an array `mgrs` of type `int[]` and size 10000 and initialize it with the numbers 1 to 10000 in ascending order. These represent the employees 1 to 10000. We then sort the array using the following comparison function:

```

private boolean lessThan(int i, int j) {
    return subords[i] < subords[j];
}

```

where `subords` contains the number of subordinates for each managed employee, as in the answer to 4.4.1.

We cannot use insertion sort, selection sort or bubble sort since they require roughly $10000 * 10000$ comparisons, which is too slow even on a fast computer. So we use quicksort, mergesort or heapsort, which requires only about $10000 * 13$ comparisons.