



Examiner: Carsten Butz, IT-University of Copenhagen
Censor: Fritz Henglein, DIKU, Copenhagen University
Exam Form: Written Exam
Date: January 3, 2003
Time: 9:00 – 13:00

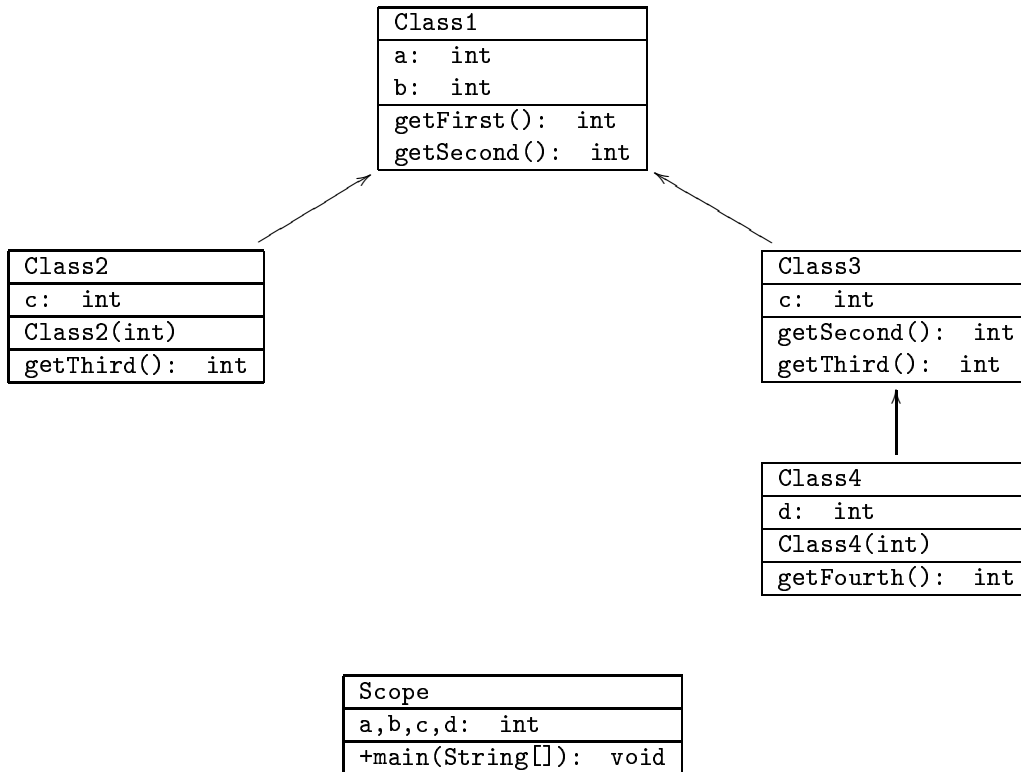
SOLUTION TO THE WRITTEN EXAM, JANUARY 3, 2003

Please read the following instructions carefully before starting answering the questions.

- The exam is a written exam. You have four (4) hours to answer the questions.
- The exam is graded on the Danish 13 scale.
- The question set is available in English only. You are allowed to answer the questions in either *English* or *Danish*.
- The exam is an open book exam, which for this exam means that the following aids may be used:
 - The text book *Java by Dissection*.
 - Additional course notes, like copies of slides, the notes by Peter Sestoft on searching and sorting, and on software testing.
 - Handwritten notes.
 - A language dictionary.
- No electronic devices are permitted.
- The exam consists of four main questions. The weight of each question is given (in percentage points) next to the question. All four questions should be answered.
- When using classes or methods from the book or from the notes by Peter Sestoft then it is not necessary to copy them. However, you have to give the precise location of those (for example, source, edition, section number, page number). It is your responsibility that we can identify that location.

(Weight 25%) QUESTION 1

SOLUTION 1.1



SOLUTION 1.2

The scope refers "to range of statements [in the program] that can access the variable"¹, cf. the textbook sections 4.4 and 6.11.

1. The variable is *local*, and can thus be accessed only after its declaration in line 20 until the end of the method (line 22).
2. The variable is *local*, and can thus be accessed only after it is declared, and inside the block where it occurs. Thus the scope is lines 37 and 38 only.
3. The variable is a *class variable* and its scope is thus the entire class `Scope` (regardless of where the actual declaration is placed within the class). Note that the variable can also be accessed in the method `main()`, even within the loop! Since inside the loop (lines 37, 38) the static variable `c` is hidden by the local variable `c` declared in line 37 the static variable can only be accessed as `Scope.c` in lines 37 and 38.

SOLUTION 1.3

1. Invalid, since there is no variable with that name declared in either the class `Class1` or the method `getSecond()`.
2. Valid, and it refers to the variable of class `Class3` declared in line 16.
3. Valid, and it refers to the variable declared in the method `getSecond()` in line 20. Note that that variable declaration *hides* the class variable with the same name. That one can only be accessed through `this.c`.
4. Valid, and it refers to the variable `c` declared in class `Class3` (line 16), which the class `Class4` inherits.
5. Valid, and it refers to the static variable `c` declared in line 24 of class `Scope`.

¹JbD, p. 101.

6. Valid, and it refers to the static variable `c` declared in line 24 of class `Scope`. Note that it does not refer to the local variable declared inside the `for` loop since the scope of that variable is lines 37 and 38.

SOLUTION 1.4

1. There is only one constructor call, so one object of class `Class1` is created, and two pointers (`c1` and `c2`) referencing it.
2. There are two constructor calls, thus two objects of class `Class2` are created (those are also objects of class `Class1`).
3. There is one constructor call, so one object of class `Class3` (which is also an object of its super class `Class1`). There are two references to this objects, one the pointer `c`, the other the pointer `d`. This is legitimate since an object of class `Class3` is in particular an object of class `Class1`.
4. One object is created, which is an array of references.
5. Two objects are created since there are two constructor calls.

SOLUTION 1.5

One could insert a (static) counter variable, for example,

```
static int counter = 0;
```

in the class definition of `Class1`. Each time a constructor is called (creating an object) we should increment the counter by one. Since the only constructor for this class is the default constructor we should re-implement it as

```
Class1(){counter++;}
```

In the `main()` method we can always check how many objects of class `Class1` were created by printing the value of variable `counter`, for example, by inserting at the end of the `main()` method the line

```
System.out.println(Class1.counter);
```

Since attributes shouldn't be declared public it would be even better to declare `counter` a private variable, and use a method in class `Class1`, for example,

```
static int howMany(){return counter;}
```

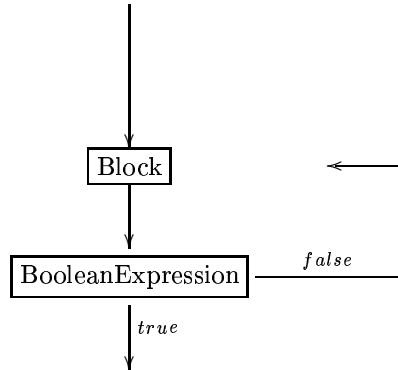
(Weight 20%) QUESTION 2

SOLUTION 2.1

The language construct has the form

```
do Block until( BooleanExpression )
```

A flow diagram is shown below.



Since the block is executed at least once, the output will be

`i = -4`

SOLUTION 2.2

```
String prettyPrint(String s, int l){  
  
    if(l <= 0)  
        return "";  
  
    if(l <= s.length())  
        return s.substring(0,l);  
    else{  
        String r= "";  
        int i = l - s.length();  
        do{  
            r = r + " ";  
            i--;  
        }until(i==0);  
        return r+s;  
    }  
}
```

The code above closely follows the given specification. A nicer way of implementing the algorithm (using only one return statement) is the following:

```
String prettyPrint(String s, int l){  
    int r = "";  
  
    if(l > 0){  
        if(l <= s.length())  
            r = s.substring(0,l);  
        else{  
            int i = l - s.length();  
            do{  
                r = r + " ";  
                i--;  
            }until(i==0);  
            r = r + s;  
        }  
    }  
}
```

```

        }until(i==0);
    }
}
return r;
}

```

Yet another solution is the following:

```

String prettyPrint(String s, int l){
    int r = "";

    if(l > 0){
        if(l <= s.length())
            r = s.substring(0,l);
        else{
            r = s;
            do{
                r = " " + r;
            }until(r.length() == l);
        }
    }
    return r;
}

```

If you do not remember the `substring()` method then you can replace it by repeated calls of `charAt()`, see Section 6.1.2 of the textbook for a list of `String` methods.

SOLUTION 2.3

```

class TestPrettyPrint{

    public static void main(String[] args){

        System.out.println(prettyPrint(args[0], Integer.parseInt(args[1].trim())));
    }

    static String prettyPrint(String s, int l){
        ...
    }
}

```

Note that the method `prettyPrint()` has to be declared static.

(Weight 30%) **QUESTION 3**

SOLUTION 3.1

```
class DoubleTuple {  
  
    private Tuple v;  
  
}
```

SOLUTION 3.2

Note for the following methods that we have to use the wrapper class `Double`, since the primitive data type `double` is not derived from class `Object`.

```
DoubleTuple(){v = new Tuple();}  
  
void add(int index, double d){  
    v.add(index,new Double(d));  
    return;  
}  
  
void clear(){v.clear();}  
  
double elementAt(int index){  
    return ((Double)v.elementAt(index)).doubleValue();  
}  
  
int indexOf(double d){  
    return v.indexOf(new Double(d));  
}  
  
int size(){return v.size();}
```

SOLUTION 3.3

```
boolean contains(double d){  
    found = false;  
    int i = 0;  
  
    while(!found && i < v.size()){  
        if(d == Double.getValue(v.elementAt(i)))  
            found = true;  
    }  
  
    return found;  
}
```

Alternatively you can use the method `indexOf()` of class `Tuple` to determine whether or not the `double` value is contained in the vector or not, compare the next question.

SOLUTION 3.4

```
boolean contains(double d){  
    Double newd = new Double(d);  
    int index = v.indexOf(newd);  
  
    if(index != -1){  
        v.remove(index);  
        v.add(0,newd);  
    }  
}
```

```

    }

    return index != -1;
}

```

SOLUTION 3.5

```

boolean contains(double d){
    Double newd = new Double(d);
    int index = v.indexOf(newd);

    if(index != -1 && index != 0){
        v.remove(index);
        v.add(index-1,newd);
    }

    return index != -1;
}

```

SOLUTION 3.6

The two improvements suggested in the exam questions are moving a requested element to the top of the tuple, or move it just one cell nearer to the head of the tuple. Another method could be to move the requested element half way to the head of the tuple. Thus, if the current requested item is found in cell k , then swap the contents of cells k and $k/2$ (if k is even) or $(k - 1)/2$ if k is odd.

SOLUTION 3.7

The methods require to re-arrange cells, which can be expensive (with respect to time). If the `Tuple` class is implemented as a linked list, then re-arranging cells is automatic (but accessing a cell with a specified cell index can take time). If the `Tuple` class is implemented as an array (see the next question), then accessing a particular cell is cheap, but re-indexing takes a lot of time. For example, if currently 200 cells are occupied, and we remove the contents of cell 49, then the contents of cell 50 has to be copied into the array cell with index 49, the contents of cell 51 has to be copied into the cell with index 50, etc.

(Weight 25%) QUESTION 4

SOLUTION 4.1

```
public Tuple(){
    dataTuple = new String[10];
    numooc = 0;
}

public int size(){return numooc;}

public int clear(){numooc = 0;}

public String elementAt(int index){
    return dataTuple[index];
}
```

SOLUTION 4.2

```
public int indexOf(String s){
    int i = numooc - 1;
    boolean found = false;

    while(!found && i >=0){
        if(dataTuple[i].equals(s))
            found = true;
        i--;
    }

    if(found)
        return i+1;
    else
        return -1;
}
```

SOLUTION 4.3

```
public String remove(int index){

    /* Retrieve the contents of cell index.
     * We do not worry if the index is out of bound, in which
     * case the method will throw an exceptions, which is fine.
     */
    String s = dataTuple[index];

    /* Next we have to shift the contents of the cells.
     * The first cell into which we want to write something
     * is the cell index (explaining the int i = index).
     * The last cell from which is want to read is the last
     * occupied cell, which has index numooc-1. Thus the last
     * cell into which we read is cell numooc-2, explaining
     * i < numooc - 1.
     */
    for(int i = index; i < numooc - 1; i++)
        dataTuple[i] = dataTuple[i+1];

    /* If we did delete a cell, then now one cell less
     * is occupied.
     */
}
```

```

    if(numooc >= 0 )
        numooc--;

    /* And we return the found string. */
    return s;
}

```

SOLUTION 4.4

```

void add(int index, String s){
    /* Check first whether there is enough space in dataTuple.
     * If not, double the size of dataTuple.
     */
    if(numooc == dataTuple.length){
        String[] tmp = dataTuple;
        dataTuple = new String[2*numooc];
        for(int i = 0; i<numooc; i++)
            dataTuple[i] = tmp[i];
    }
    /* Add the element: */
    /* If index is larger than the number of occupied cells,
     * add at the end.
     */
    if(index > numooc)
        index = numooc;

    /* Move data upward, so that cell index becomes available.
     */
    for(int i = numooc; i <= index + 1)
        dataTuple[i] = dataTuple[i-1];

    dataTuple[index] = s;
    numooc++;
    return;
}

```