Final #2 – Questions and Solutions

In questions 5 – 7, mark ALL correct answers for each question.

1. The class *DNode* given below defines a link in a doubly-linked list, in which there is a *data* field of type *Object* as well as fields *previous* and *next* pointing at the preceding and the next link, respectively. The definition of the class *DNode* is given as an extension of the class *Node*, defining a link in a regular list:

```
public class DNode extends Node
{
   public Node previous;

   DNode (Object data, Node previous, Node next)
   {
      super (data, next);
      this.previous = previous;
   }

   public Node get_previous()
   {return previous; }

   public void set_previous (Node a)
   {previous = a; }
}// DNode
```

```
public class Node
{
    public Object data;
    public Node next;

    Node (Object data, Node next)
    {
        this.data = data;
        this.next = next;
    }

    public Object get_data()
    {return data; }

    public Node get_next()
    {return next; }

    public void set_next (Node a)
    {next = a; }
}// Node
```

The class *DList* (given below) defines a structure of a doubly-linked list of links of type *DNode*, in which there are two special links – *left* and *right* – called sentinels. These links contain no data, but provide a convenient access to the leftmost and the rightmost links in the structure, and simplify the definition of operations on the structure.

The task

The class *DList* given below has worked properly until all casting operations have been removed from it. In the code there are five lines in which it is necessary to add casting operations in order that the code will be accepted by the compiler and work correctly. Mark the problematic lines and correct them, so that the code will work correctly.

Do not write more than 5 lines.

```
public class DList
{
    public DNode left, right;

    DList()
    {
        left = new DNode (null, null, null);
        right = new DNode (null, left, null);
        left.set_next (right);
    }

    public boolean isEmpty()
    {return (left.get_next() == right);}

    public void addAtHead (Object data)
    {
        DNode new_node = new DNode (data, left, left.get_next());
        left.get_next().set_previous (new_node);
        left.set_next (new_node);
    }

    public void addAtTail (Object data)
    {
        DNode new_node = new DNode (data, right.get_previous(), right);
        right.get_previous().set_next (new_node);
        right.set_previous (new_node);
    }

    public void forwards_print()
    {
        DNode i = left.get_next();
        while (i != right)
        {
            System.out.print (i.get_data() + "");
            i = i.get_next();
        }
```

```
        System.out.println();
    }

    public void backwards_print()
    {
        Node i = right.get_previous();
        while (i != left)
        {
            System.out.print (i.get_data() + "");
            i = i.get_previous();
        }
        System.out.println();
    }

    public void delete (DNode i) //Node i must be in the list
    {
        i.get_next().set_previous (i.get_previous());
        i.get_previous().set_next (i.get_next());
    }
}// class DList
```

The first missing casting operation occurs at the second instruction of the *addAtHead*() method. In fact, *left.get_next*() should be of type *DNode*, which is fine. However, the compiler does not know that. Since *get_next*() is inherited to *DNode* from *Node*, as far as the compiler can know at this stage *left.get_next*() is going to be of type *Node*, so that the method *set_previous*() cannot be invoked for it. Therefore casting is needed as follows:

```
((DNode) left.get_next()).set_previous (new_node);
```

A similar problem occurs at the first line of the *forwards_print*() method. The compiler expects the right hand side to yield an element of type *Node*, and will not agree to have it referenced by a *DNode* type variable. The line should be changed to:

```
DNode i = (DNode) left.get_next();
```

In the second instruction of the *while* loop of the same method the same error is made, and should be corrected as follows:

$$i \;=\; (DNode)\ i.get\_next();$$

The second instruction of the $backwards\_print()$ method is erroneous as well, but for a slightly different reason. Here i is declared to be of type $Node$, yet we try to invoke the method $get\_previous()$ for it. Even though i indeed references an object of the right type, the compiler does not know it, and the following casting is required:

$$i \;=\; ((DNode)\ i).get\_previous();$$

Finally, in the first line of the $delete()$ method, $i.get\_next()$ is expected at compilation time to be of type $Node$, so that the $set\_previous()$ method cannot be applied to it, and the line should be changed to:

$$((DNode)\ i.get\_next()).set\_previous\ (i.get\_previous());$$

2. In this question we shall use the attached code of the classes $DList$ and $DNode$ to write the class Set, extending $DList$ and representing a set of objects (mutually distinct, namely the same element should not occur more than once in the representation of the set).

   Remark: Even if you fail to answer some parts of the question, you may use the methods you have been required to define there in your answers to the other parts. You may also use the methods in Question 1, even if you have not corrected the casting problems.

   In parts (A) – (D) you will find instructions for completing the following definitions:

```
public class Set extends DList
{
   public Set()
   {super(); }
```

```
public boolean isMember (Object b)
{
    /*** FILL IN DEFINITION ***/
}

public void add (Object b)
{
    /*** FILL IN DEFINITION ***/
}

public Set sameClass (String type)
{
    /*** FILL IN DEFINITION ***/
}

public Set setsOfClasses()
{
    /*** FILL IN DEFINITION ***/
}
}// class Set
```

(A) Complete the method *isMember* (*Object b*) of the class *Set*, which checks whether the object *b* belongs to the set of elements of the key object. The check is performed by the boolean method *equals* (*Object b*), which is defined for the classes of elements in the set.

We have to go over all elements of the set and check for each whether it is equal to *b*. This is accomplished by:

```
boolean belongs = false;
for (Node t = left.get_next(); t != right; t = t.get_next())
    if (b != null && b.equals (t.data))
        belongs = true;
return belongs;
```

(B) Complete the method *add* (*Object b*) of the class *Set*, which adds an object *b* to the set of elements in the key object.

Adding an element is done using the method *addAtHead*(). The only point to take care of is to check prior to adding an element that it is not already in the set, in which case nothing should be done. The required code is:

```
if (!isMember(b))
   addAtHead(b);
```

(C) Suppose we have added the method *getType*() to the class *Node*, and it returns the name of the class of the *data* field in the key object as a string. For example, if we execute the two instructions

```
Node s = new Node (new Set());
System.out.print (s.getType());
```

then the string "*Set*" will be printed.

Employ the method *getType*() to complete the method
sameClass (*String type*)
of the class *Set*, which deletes from the key object all elements of type *type* and returns them as a new set of elements.

For example, suppose the instructions

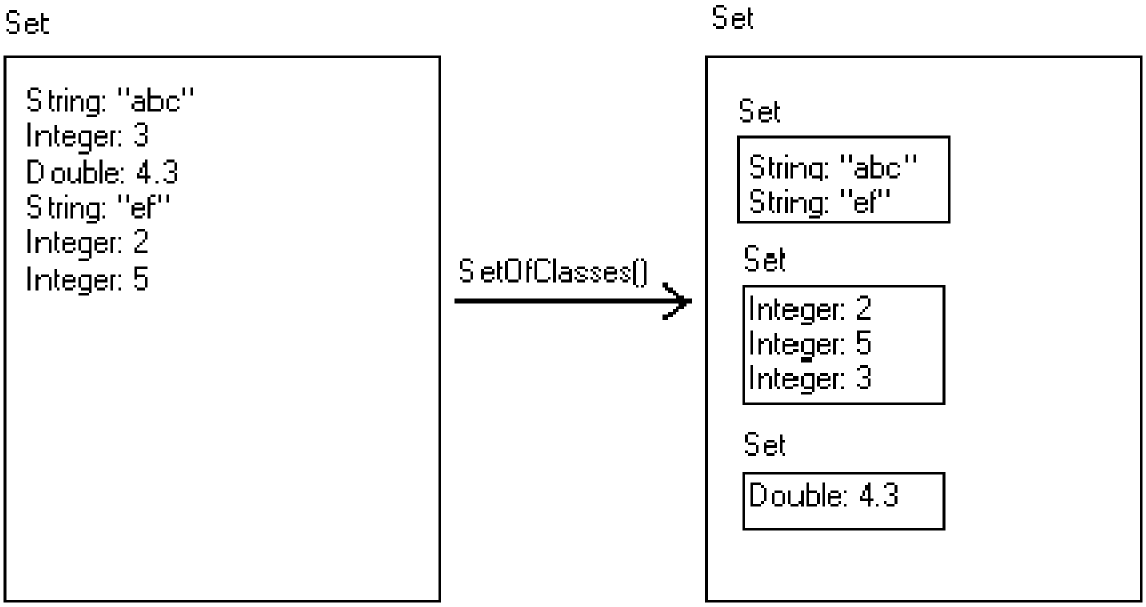
```
Set s1 = new Set();
s1.add ("123");
s1.add (new Double (4.2));
s1.add (new Integer (3));
s1.add (new Double (1.2));
```

have been given. If at this stage *s1.left.next* points to a link whose *data* field is of type *Double*, and we exectute the additional instruction
Set s2 = s1.sameClass (s1.left.next.getType());
then the set *s1* will contain the string "123" and the *Integer* 3, while *s2* will contain the *Double* objects 1.2 and 4.2.

We have to go over all elements of the set. Each of them, which is of type *type*, is removed from the key object and added to the set which is the return value of the method. This is accomplished by:

$Set\ s1\ =\ new\ Set();$
$for\ (Node\ t\ =\ left.get\_next();\ t\ !=\ right;\ t\ =\ t.get\_next())$
$\{$
$\quad if\ (type.equals(t.getType()))$
$\quad \{$
$\qquad s1.add\ (t.data);$
$\qquad delete\ ((DNode)\ t);$
$\quad \}$
$\}$
$return\ s1;$

(D) Complete the method $setOfClasses()$ of the class Set, which divides the set of elements in the key object into sets of elements according to their types. For example, if we let the method act on the set drawn on the left, then the returned value will be the set of sets drawn on the right:



(There is no restriction as to what should happen to the original set; it may change during the process, or may stay the same.)

The simplest (although perhaps not the most efficient) solution may be as follows: Take any type which appears in the set, for example the type of the first element, and generate from all elements of this type in the set a new set, which will be the first element of the returned set, while omitting all these elements from the original set. Next take another type which appears in the set, for example the type of the first of the remaining elements, and generate from all elements of this type in the set a new set, which will be the second element of the returned set, again omitting all those elements from the original set. Continue in the same way until the original set becomes empty. This is accomplished by:

```
Set ans = new Set();
while (!isEmpty())
   ans.addAtHead (sameClass (left.get_next().getType()));
return ans;
```

3. Complete the method $subsetsOfSize\ (int\ n,\ int\ k,\ String\ s)$, so that the method $subsetsOfSize\ (int\ n,\ int\ k)$, which accepts two integers n and k (with $0 \leq k \leq n$ and $n \geq 1$), will print all subsets of size k of the set $\{1, 2, \ldots, n\}$ (where printing a subset means printing all its elements as a string). For example, the given method $main$ invokes $subsetsOfSize(n, k)$ with $n = 5$ and $k = 3$, and its output should consist of the strings

   123 124 125 134 135 145 234 235 245 345

   (one per line, in any order).

```
public class subsets
{
   static void subsetsOfSize (int n, int k)
   {sub (n, k, ""); }

   static void subsetsOfSize (int n, int k, String s)
   {
      /*** FILL IN DEFINITION ***/
   }

   public static void main (String[] args)
   {subsetsOfSize (5, 3); }
}
```

   The simplest solutions to this problem are recursive. One possibility is to decompose the set of all subsets of size k of $\{1, 2, \ldots, n\}$ into two parts, one consisting of all those subsets containing the element n and the other consisting of all those subsets not containing n. Listing all subsets belonging to the second part is a problem similar to the original, but with n decreased by 1. Listing all subsets belonging to the first part is basically the same as listing all subsets of size $k-1$ of $\{1, 2, \ldots, n-1\}$, each followed by n. Developing this idea further, we are led to writing a method $subsetsOfSize\ (int\ n,\ int\ k,\ String\ s)$, more general than the method $subsetsOfSize\ (int\ n,\ int\ k)$. The 3-parameter method should print all subsets of size k of $\{1, 2, \ldots, n\}$, each followed by the string s. In view of the above discussion, a possible code for that is:

```
if (k == 0)
   System.out.println (s);
else for (int i = 1; i <= n; i++)
   subsetsOfSize (i - 1, k - 1, i + " " + s);
```

Another possibility is to split the set into a disjoint union of n (indeed, $n-k+1$) sets, depending on the largest element belonging to the set (which may be any one of the numbers $k,\ k+1, \ldots,\ n$). Developing the idea as in the preceding solution, we arrive at the following possible code:

```
if (n >= 0 && k == 0)
   System.out.println (s);
else
   if (n >= 0 && k > 0)
   {
      subsetsOfSize (n - 1, k, s);
      subsetsOfSize (n - 1, k - 1, n + " " + s);
   }
```

It is interesting to note that the two proposed solutions are analogous to two recursive formulas for binomial coefficients, namely

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

and

$$\binom{n}{k} = \sum_{i=k-1}^{n-1} \binom{i}{k-1},$$

(where in both formulas we omitted the cases where the coefficient is defined directly).

4. (A) The program defined in the class *what*:

   (a) is rejected by the compiler.

(b) is accepted by the compiler, but generates a runtime error.

(c) runs well, and generates the output ... .

(B) Repeat part (A), assuming that the two lines denoted with ***1*** and ***2*** are interchanged.

Note that the class *Node* has been defined in Question 1 and the class *List* is brought below for convenience.

```
class what
{
   public static void what (List x)
   {
      if (x.first != null && x.first.get_next() != null)
      {
         Node n = x.first;
         x.first = x.first.get_next(); // ***1*** //
         n.set_next (null); // ***2*** //
         what (x);
         Node node = x.first;
         while (node.get_next() != null)
            node = node.get_next();
         node.set_next (n);
      }
   }

   public static void main (String[] args)
   {
      List x = newList();
      x.addAtHead (new Integer (5));
      x.addAtHead (new Integer (4));
      x.addAtHead (new Integer (3));
      x.addAtHead (new Integer (2));
      x.addAtHead (new Integer (1));
      what (x);
      x.print();
   }
}
```

Following is the implementation of a linked list:

```
public class List
{
   public Node first;

   List()
   {first = null;}

   public void addAtHead (Object data)
   {first = new Node (data, first);}

   public void deleteHead()
   {
      // we assume that this method is called only
      // when first is not equal to null
      first = first.get_next();
   }

   public void print()
   {
      for (Node i = first; i != null; i = i.get_next())
         System.out.print (i.get_data() + " ");
   }
}//classList
```

The *what* method creates a reference n to the first node of the given list x. (Assume x consists of at least two elements.) Then it moves the $first$ field of x to refer to the second element of x, effectively deleting the first element of x. After a recursive call to *what* for the modified x, the original first element of x, referred to by n, is linked to x at the tail of x, its *next* field being now *null*. Thus, ignoring the recursive call, we just moved the first element of x to the last place. The recursion acts on the shortened x in the same way, namely the first element is moved to the last place, etc. Altogether, *what* reverses the order of the elements of x. (Formally, this should be done by induction on the length of x.) In our case, the *main* method initializes x to be the list consisting of the (object) numbers 5 4 3 2 1 in this order, so that *what* changes x to be the list 1 2 3 4 5, and this numbers will form the output of the program.

The above is completely correct for the program as is. Performing the indicated line interchange, we introduce an error, as follows. The first element of x, pointed at by n, is made to point to *null* before the $first$ field of x is changed to point to the second element of x. Consequently, all the elements of x but the first are cut from the list. Trying to change $x.first$ to $x.first.get\_next()$ we get an error, as $x.first$ has no next element.

5. Consider the classes $MyInt$ and $Exam$:

```
class MyInt
{
   private int _n;

   MyInt (int i)
   {_n = i;}

   int intValue()
   {return _n;}

   static void inc (MyInt n)
   {
      if (n != null)
         n = new MyInt (n.intValue() + 1);
   }
}

public class Exam
{
   public static void main (String[] a)
   {
```

```
        MyInt i2 = new MyInt (2);
        i2.inc (i2);
        System.out.println (i2.intValue());
    }
}
```

(a) The compiler will accept the code, the program will run and will print 2.

(b) The compiler will accept the code, the program will run and will print 3.

(c) The compiler will accept the code, but there will be a runtime error.

(d) The compiler will reject the code since the static method *inc* activates methods of *MyInt*.

(e) The compiler will reject the code since, in the method *main*, the static method *inc* is not operated on the name of the class *MyInt*, but rather on an object of type *MyInt*.

(f) If we change the title of the method *inc* to be
private void inc (*MyInt n*),
then the compiler will accept the code.

In the method *main*, we first instantiate an object *i2* with an *_n* field of 2. Next we invoke the *inc* method of this object (which could indeed be invoked by the command *MyInt.inc*(*i2*) instead of the command *i2.inc*(*i2*)). The call creates a new variables table, containing a variable *n* which points to the same memory location as *i2*. Since *n* is not *null*, another object of type *MyInt* is instantiated, this one with an *_n* field of 3, and *n* is changed to reference this newly generated object. Observe that *n* references a new location, but nothing relevant to *i2* has occurred. Upon terminating the operation of *inc*, the former variables table takes over, the variable *n* ceases to exist, and the old *i2* returns to existence with no change in either the memory location its refers to or the contents of this memory location.

If the change suggested in (f) is performed, then the method *inc* is unknown outside *MyInt* class, so that the code is rejected by the compiler.

Thus, only (a) is correct.

6. To the class *Point* from homework assignment #3 we have added a method *Point midpoint*(*Point Q*), which returns the midpoint of the interval connecting the key point with the given point Q.

   The class *Triangle* represents triangles. It contains a constructor *Triangle* (*Point P*1, *Point P*2, *Point P*3), which accepts the vertices of the triangle, a method *double area*() which returns the area of the triangle, and a method *Point*[] *getVertices*() which returns an array containing the three vertices of the triangle. The following method belongs also to the class *Triangle*:

```
double strangeFunction()
{
   double result = 0;
   if (area() >= 0.001)
   {
      Point Q = getVertices()[0].midpoint(getVertices()[1]);
      Point R = getVertices()[1].midpoint(getVertices()[2]);
      Point S = getVertices()[2].midpoint(getVertices()[0]);
      Triangle T = new Triangle (Q, R, S);
      result = area() + 2 * T.strangeFunction();
   }
   return result;
}
```

   Suppose that, in some program, T denotes the triangle with vertices at the points $(1, 0)$, $(-1, 0)$ and $(0, 1)$, and the command *double* x = *T.strangeFunction*() is given. The value of x after this command is executed is:

(a) 0.

(b) $\frac{31}{16}$.

(c) $\frac{1023}{512}$.

(d) $\frac{5}{4}$.

(e) $\frac{3}{2}$.

(f) None of the above.

The method $strangeFunction()$ is recursive. Calculating it for a given triangle (of area at least 0.001) involves calculating it for the triangle whose vertices are the midpoints of the edges of the original triangle. Now the area of the small triangle is a fourth of that of the big one. Hence the value returned by $strangeFunction()$ depends only on the area of the initial triangle. Denoting by $f(S)$ the returned value if this area is S, we obtain:

$$f(S) = \begin{cases} 0, & S < 0.001, \\ S + 2f(S/4), & S \geq 0.001, \end{cases}$$

In our case the given triangle is of area 1, and therefore the value returned by the method will be:

$$f(1) \quad \begin{aligned} &= 1 + 2f(1/4) \\ &= 1 + 2 \cdot 1/4 + 4f(1/4^2) \\ &= 1 + 2/4 + 4 \cdot 1/4^2 + 8f(1/4^3) \\ &= 1 + 2/4 + 4/4^2 + 8 \cdot 1/4^3 + 16f(1/4^4) \\ &= 1 + 2/4 + 4/4^2 + 8/4^3 + 16/4^4 + 32f(1/4^5) \\ &= 1 + 2/4 + 4/4^2 + 8/4^3 + 16/4^4 = 31/16. \end{aligned}$$

Thus, (b) is the correct answer.

7. $MyIntegers$ is a class with an instance variable n of type int and a constructor

public $MyIntegers$ (*int* m)
$\{n \ = \ m;\}$

It also contains a method $int\ randomPrimeDivisor()$ which returns a random prime divisor of n for $n \geq 2$ and returns 1 for $n = 1$. (We shall not invoke this method with $n \leq 0$.) For example, if the value of the n field of an object x of type $MyIntegers$ is 17, then $x.randomPrimeDivisor()$ will return 17, while if $n = 60$ then one of the values 2, 3 and 5 will be returned (each with equal probability).

Now we add to $MyIntegers$ two methods, $int\ random1()$ as well as $int\ random2()$. Both methods return 1 for $n = 1$. For $n \geq 2$, the body of $int\ random1()$ consists of

$MyIntegers\ y\ =\ new\ MyIntegers\ (n\ /\ randomPrimeDivisor());$
$return\ randomPrimeDivisor()\ *\ y.random1();$

and that of $random2()$ is

$int\ l\ =\ randomPrimeDivisor();$
$MyIntegers\ y\ =\ new\ MyIntegers\ (n\ /\ l);$
$return\ l\ *\ y.random2();$

Now consider the following code section, where m is of type int:

$MyIntegers\ x\ =\ new\ MyIntegers\ (m);$
$int\ a\ =\ x.random1\ (),\ b\ =\ x.random2\ ();$

After the execution of this code:

(a) The variables a and b are necessarily equal.

(b) There exist (theoretically, ignoring the fact that int is restricted to a finite range of numbers) infinitely many values of m for which we must have $a = b$ and there exist infinitely many for which we may have $a \neq b$.

(c) For $m = 30$ there are 8 possible values for a and only one for b.

(d) For $m = 42$ there are 10 possible values for a and only one for b.

(e) For $m = 109$ there are 5 possible values for a.

(f) None of the above.

If m is a prime, then $randomPrimeDivisor()$ necessarily returns m, and therefore both $random1()$ and $random2()$ return m, so that $a = b = m$. Similarly, if m is a prime power, then $a = b = m$. (Formaly, this is done by induction.) When m is not a prime power, however, the situation is different. In $random2()$ we extract each time a prime divisor of the value of the n field, employ the method for the "rest" of the number, and then multiply the result by the first prime divisor. Therefore, $random2()$ always returns simply the n field, so that $b = m$. In $random1()$ the prime divisor extracted from the n field is not necessarily the same as the divisor we use after finishing the recursion. Consequently, if the prime factorization (into not necessarily distinct primes) of the n field is $p_1p_2\ldots p_k$, then $random1()$ may return any product of k of the p_i's (where some of the indices may repeat several times and some may not appear at all). For example, if $k = 2$ we may obtain one of the three numbers p_1^2, p_1p_2 or p_2^2 (which are distinct if $p_1 \neq p_2$). If $k = 3$ we may obtain either some p_i^3 (three possibilities) or some $p_i^2p_j$ with $i \neq j$ (six possibilities) or $p_1p_2p_3$ (one possibility), which yields ten possible outcomes. In particular, for $m = 109$ there is a single possibility for b (since 109 is prime), while for $m = 30$ and $m = 42$ there are ten possible values for b (since both 30 and 42 are products of three primes).

Thus, only (b) and (d) are correct.