Final #1 – Questions and Solutions

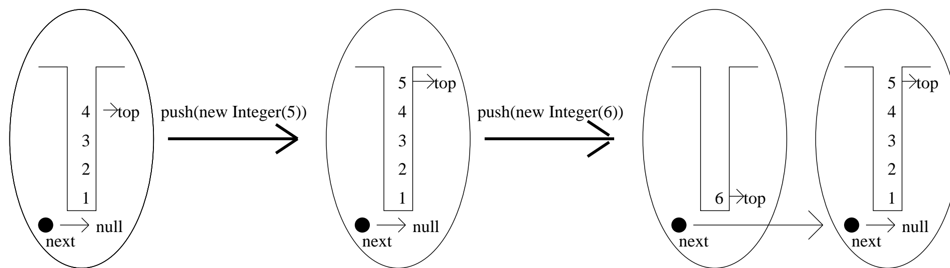
1. In the course we have described two implementations of *StackInterface*: The first – *StackAsArray* – using an array of objects, and the second – *StackAsList* – using a linked list of objects.

   In this question we construct an additional implementation of the interface *StackInterface*, which we call *MultiStack*. The implementation is similar to that of an array-stack in the sense that the elements are held in an array of objects. The difference is that, when the array becomes full, it is possible to add elements by instantiating an additional array-stack, linked to the previous in a structure of a linked list. Here we assume that the arrays in the structure hold five elements each.

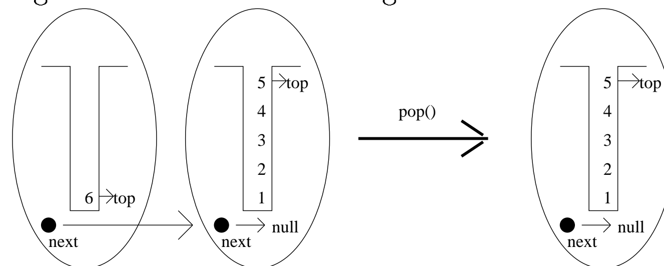
   The implementation of a multi-stack is actually not that complex, as the multi-stack may be described as an object of type list-stack, consisting of a collection of array-stacks, in which the elements of the multi-stack reside. The title of the class is as follows:

   public class MultiStack extends StackAsList
   implements StackInterface

   Pushing an element x into a multi-stack m is performed if possible on the top array-stack in m. If m is empty of stacks, or the top array-stack in m is full, a new array-stack is added to m, and x is added into this array-stack. The following drawing exemplifies two addition operations (of the element 5 and the element 6) to the given multi-stack.



Popping an element from a (non-empty) multi-stack m is performed from the top array-stack in m, but we make sure not to leave empty array-stacks in m. Popping an element from an empty multi-stack returns a *null* value. The following drawing exemplifies the operation of popping the element 6 from the given multi-stack.



In the following you will find an incomplete definition of the class *MultiStack*. In parts (a) and (b) of the question you are required to complete the methods *push*() and *pop*() in the class *MultiStack*.

The stack interface you have to implement is as follows:

```
public interface StackInterface
{
    boolean push (Object a);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}// interface StackInterface
```

The following is a partial implementation, which you have to complete:

```
public class MultiStack extends StackAsList
                        implements StackInterface
{
   public MultiStack()
   {super();}

   public boolean push (Object a)
   {
      /*** FILL IN DEFINITION ***/
   }

   public Object pop()
   {
      /*** FILL IN DEFINITION ***/
   }

   public Object peek()
   {
      /*** ASSUME THIS HAS BEEN IMPLEMENTED ***/
   }

   // public boolean isEmpty(){** is inherited **}
   // public boolean isFull(){** is inherited **}
}
```

When pushing an element, we need to push it into the top array-stack of the multi-stack. Prior to this, we have to make sure the top array-stack is not full; if it is full (or if the multi-stack itself is empty), then we need first to push an array-stack into the multi-stack. This is accomplished by the following code:

```
if (isEmpty() || ((StackAsArray) super.peek()).isFull())
   super.push (new StackAsArray());
((StackAsArray) super.peek()).push(a);
return true;
```

To pop an element, we need (assuming the multi-stack is non-empty) to peek at the array-stack at the top of the multi-stack and perform a *pop*() operation on it. If this operation empties the top array-stack of the multi-stack, we have to pop this empty array-stack. This is accomplished by:

```
Object value = null;
if(!isEmpty())
{
   value = ((StackAsArray)super.peek()).pop();
   if(((StackAsArray)super.peek()).isEmpty())
      super.pop();
}
return value;
```

2. Let *MyStack* be a class implementing the interface *StackInterface*, defined earlier. The details of the implementation are unknown.

   In Java it is possible to extend a regular class by an abstract class. The purpose of the abstract class *FilterStack* is to enable "filtering" the elements of a stack in order to remove those not satisfying some filtering condition. The filtering condition is defined by means of an abstract method *filter*(), whose role is testing an element b and returning a *false* value if (and only if) b is to be removed from the stack. The other elements are to remain in the stack according to the original order. For example, we may employ this method to leave in a stack of numbers only the positive ones.

   (a) Complete the body of the method *filtering*() so that it will go over the stack elements and removes those failing the filtering condition.

   ```
   public abstract class FilterStack extends MyStack
   {
      public FilterStack()
      {super(); }
   ```

```
    public abstract boolean filter (Object b);
    public void filtering()
    {
        /*** FILL IN DEFINITION ***/
    }
}
```

The simplest is to define another stack, say $s1$, of type *MyStack*. Then we pop the elements of the key stack one by one, and push those satisfying the filtering condition into $s1$. At this point $s1$ contains all required elements, but in inverse order. Now pop the elements of $s1$ one by one and push each back into the key stack. Here is the required code:

```
MyStack s1 = new MyStack();
while (!this.isEmpty())
{
    Object d = this.pop();
    if (this.filter(d))
        s1.push(d);
}
while (!s1.isEmpty())
    this.push (s1.pop());
```

(b) Write the class *StringStack*, extending *FilterStack*, so that employing the *filtering*() method in it will leave in the stack only elements of type *String*.

```
public class StringStack extends FilterStack
{
    public StringStack()
    {super();}

    /*** FILL IN ***/
}
```

The filtering condition now is belonging to the class *String*. We need to override the abstract method *filter*() by a method returning the value *true* if and only if the key object is of type *String*. The following code accomplishes the task:

```
public boolean filter (Object b)
{return (b != null && b instanceof String); }
```

3. What will the following program print?

```
public class What
{
   public static void process (int[] a)
   {
      boolean is = true;
      int i = a.length - 1;
      while (is && i >= 1)
      {
         is = false;
         for (int j = 0; j < i; j = j + 1)
            if (a[j] > a[j + 1])
            {
               int temp = a[j];
               a[j] = a[j + 1];
               a[j + 1] = temp;
               is = true;
            }
            i = i - 1;
      }
   }

   public static void main (String[] args)
   {
      int[] arr = {3, 9, 2, 6, 4, 7, 1};
      process (arr);
```

```
    for (int i = 0; i < arr.length; i = i + 1)
    System.out.print (arr[i] + " ");
  }
}// What
```

The method *process*() is a sorting method (known as *bubblesort*). In fact, it first goes over all elements of the array and, whenever it finds two consecutive elements, the smaller of which being preceded by the larger one, switches the two. In particular, after the first pass the maximal element is certainly at its proper place – the last. In the second pass the second largest element will arrive at its proper place, and by the time the outer loop finishes all passes the array will be sorted. (The variable *is* makes sure that, in case the array became sorted before all passes have been completed, the program will terminate after one more pass.) Therefore the output in our case is:
1 2 3 4 6 7 9

In Questions 4–5 we shall relate to the following code:

```
class A
{
  int _x;

  public A (int i)
  {_x = i;}

  public String toString()
  {return "1";}
}

class B extends A
{
  public B (int i)
  {super (i);}
```

```
    public String toString()
    {return "2";}

    public A convertToA()
    {return new A (_x);}
}

class C {}
```

4. What will the following program print?

```
public class Exam4
{
    public static void main (String[] a)
    {
        A s1 = new A (10);
        A s2 = new B (20);
        B s3 = new B (30);
        System.out.print (s1);
        System.out.print (s2);
        System.out.print (s3);
        System.out.print ((A) s3);
        System.out.print (s3.convertToA());
    }
}
```

The string representation of an A object is "1", while that of a B object is "2". Hence the output of the first print instruction is "1" and that of the third is "2". The second print command refers to a B object (although pointed to by a variable of type A), and therefore it produces a "2" as well. Similarly, even though the expression $(A)\ s3$ is of type A, the object referred to is a B object, whence the fourth print command

also produces a "2". However, since the method $convertToA()$ of class B returns an A object, $s3.convertToA()$ belongs to A, whence the last print command yields a "1".

Thus the output is (in five separate lines): 1 2 2 2 1.

5. Consider the following program:

```
public class Exam5
{
   public static void main (String[] a)
   {
      Object s5 = new A (5);
      C s4 = (C) s5;
   }
}
```

   (a) The compiler will accept the code, but there will be a runtime error due to casting.
   (b) The casting will cause a compilation error, as it is disallowed to cast from type A to type C.
   (c) If $s5$ was a variable of type A (instead of *Object*), the compiler would not accept the code, as then it would be clear that it is impossible to perform the casting already at the stage of type checks of the compiler.
   (d) The compiler will accept the code, and the program will run without errors, since the class C is a special case of the class A.
   (e) The compiler will reject the code since it is disallowed to define a variable of an abstract type, and *Object* is an abstract class.
   (f) None of the above.

At the time of compilation, it is impossible to know precisely the type of an object to which a certain variable refers. It may be of the type

of the variable or any type extending it. Since $s5$ is declared of type $Object$, the compiler will "believe" that the casting is possible. During runtime, however, there will be an error since an A object cannot be cast to class C. If $s5$ was declared to be of type A, the casting would be obviously wrong at compilation time.

A variable may well be of a type of an abstract class, although any object it refers to during runtime must be of a type of an extending non-abstract class. $Object$ is not an abstract class.

Thus, only (a) and (c) are correct.

6. We are given two arrays of integers $arr1$ and $arr2$. Both arrays are long, $arr2$ being much longer than $arr1$. For example, you may suppose that the length of $arr2$ is the square of the length of $arr1$. A common operation, which we will be required to perform many times (henceforward "the operation"), is the following: Given an integer d, find an element x in $arr1$ and an element y in $arr2$ such that $y - x = d$. Then:

   (a) If both arrays are sorted, then the operation may be performed by a simple binary search on the differences.

   (b) It is possible to perform this operation by means of a double loop, which examines all pairs of elements (one from each array). Using this method, it makes no difference if the arrays are sorted before performing the search.

   (c) If we sort both arrays beforehand, then we can perform the operation by means of a loop, which examines all elements of one of the arrays, and for each of them performs a binary search for an appropriate value (according to d) in the other array.

   (d) It is possible to improve the implementation described in (c) by sorting only one of the two arrays, and it makes no difference if this is the array all of whose elements are searched or the other array.

   (e) It is possible to improve the implementation described in (c) by sorting $arr1$ in increasing order and $arr2$ in decreasing order.

(f) It is possible to improve the implementation described in (c) by sorting only one of the two arrays. It is better to sort only $arr2$, even though it is much longer and requires more operations for its sorting than does $arr1$.

If one of the arrays, say $arr1$, is sorted, then one needs to go over all the elements of the other array, and for each such element $arr2[i]$ search for an occurence of the number $arr2[i] - d$ in $arr1$. The search may be binary since $arr1$ is sorted. With this algorithm it would be of no consequence if $arr2$ was sorted as well. (It is possible, though, to take advantage, albeit very small, of $arr2$ being sorted, but this is true for another algorithm.)

It is better to have $arr2$ sorted than to have $arr1$ sorted. Indeed, since binary search is very fast, it makes more sense to go over all elements of $arr1$ and have the big saving on the longer array $arr2$, rather than doing it vice versa. (One can quantize these intuitive argument as follows. If $arr1$ is sorted, then the loop over the elements of $arr2$ is of length n^2, and the length of each cycle is $\log n$, so that the time required for the whole search is of the order of magnitude of $n^2 \log n$. However, if $arr2$ is sorted, then the outer loop extends over n elements, and the length of each cycle is $\log n^2 = 2 \log n$. Now for large n the first expression is much larger than the second.)

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

7. The following sorting algorithm for integer arrays has been proposed: Given an array arr of length n of integers, perform the following operations:
i. Interchange $arr[0]$ and $arr[k]$, where k is the number of elements among $arr[1],\ arr[2], \ldots,\ arr[n-1]$ which are smaller than $arr[0]$.
ii. Interchange $arr[1]$ and $arr[k+1]$, where k is the number of elements among $arr[2],\ arr[3], \ldots,\ arr[n-1]$ which are smaller than $arr[1]$.
iii. Continue for $i = 2,\ 3, \ldots,\ n-2$ in the same way. Namely, interchange $arr[i]$ and $arr[k+i]$, where k is the number of elements among $arr[i+1], \ldots,\ arr[n-1]$ which are smaller than $arr[i]$.

(a) The proposed algorithm does not always work correctly.

(b) The proposed algorithm works correctly for arrays consisting of distinct integers.

(c) The proposed algorithm works correctly and performs the same element interchanges as $insertionSort()$.

(d) The proposed algorithm works correctly and performs the same element interchanges as $selectionSort()$.

(e) The proposed algorithm is correct, but distinct from the algorithms we have encountered in class.

(f) None of the above.

The initial stage of the algorithm performs a reasonable operation – it places $arr[0]$ at the place it should occupy, which is the k-th place if there are exactly k elements in the array below $arr[0]$. However, in all subsequent stages, the element moved to place 0 at the first stage remains untouched. For example, if arr is the array $\{1, 2, 0\}$, then the first stage changes it to $\{2, 1, 0\}$, and the second (and last) stage changes it to $\{2, 0, 1\}$.

Thus, only (a) is correct.

8. A, B and $UseAB$ are three given classes in some directory. The class B extends A. The (only) constructors of A and B are as follows:

```
A (int n)
{
   for (int i = 0; i < n; i++)
      new A (i);
   System.out.println ("Computer Science");
}
```

```
B (int n)
{
```

```
    super (n);
    for (int i = 0; i < n; i++)
        new B (i);
}
```

The *main* method of the class *UseAB* contains the line: $B\ b = new\ B\ (4);$
The number of lines printed due to this instruction is:

(a) 16.
(b) 31.
(c) 32.
(d) 48.
(e) 64.
(f) None of the above.

For $n = 0$, the loop in the constructor of A is not executed, so that the output due to instantiating an element of A with parameter 0 consists of a single line. For $n = 1$, we instantiate an A element with parameter 0, and then print another line, and thus 2 lines are printed. For $n = 2$ we similarly find that $1 + 2 + 1 = 4$ lines are printed, for $n = 3$ we obtain $1 + 2 + 4 + 1 = 8$ lines, and for general n we find (formally by induction) that 2^n lines are printed upon instantiating an element of A with parameter n.

Instantiating an element of B with parameter 0 is equivalent to instantiating an element of A with the same parameter, so that a single line is printed. In general, the number of printed lines is the same as that generated by instantiating an element of A with the same parameter and in addition by instantiating elements of B with all parameters between 0 and $n - 1$. Thus, for $n = 1$ we obtain $2 + 1 = 3$ lines, for $n = 2$ we obtain $4 + 1 + 3 = 8$ lines, and in the general case (again, formally by induction) the number of printed lines is $(n + 2)2^{n-1}$. For $n = 4$, this gives 48 lines.

Thus, only (d) is correct.