Midterm #2

In questions 1–7, mark ALL correct answers for each question.

1. Based on the class *Point* of Exercise 3, designed to model points in the plane, a class *Line* is written, intended to represent straight lines in the plane. A *Line* object has the following instance variables:
   Point p;
   double slope;
   boolean vertical;
   The variable p is some point on the represented line. The variable *slope* is the slope of the line (in case the line is not vertical to the x-axis, and an arbitrary value if it is vertical), and the variable *vertical* indicates whether it is vertical to the x-axis: It assumes the value *true* if the line is vertical and the value *false* otherwise.

   The following method has been written to test whether a line equals another:

   boolean equals (*Line L*)
   {*return p.equals* (*L.p*) & *slope* == *L.slope* & *vertical* == *L.vertical*; }

   (a) The compiler will reject the code since the boolean condition has to be parenthesized.

   (b) The method works well for vertical lines, but not for all lines in the plane.

   (c) If the method returns a *true* value, then the lines are equal, but the converse is not necessarily true.

   (d) The method works well for all lines.

(e) The following method tests correctly whether the line in question is parallel to another:

$$\begin{array}{l} boolean\ parallel\ (Line\ L) \\ \{ \\ \quad if\ (!vertical\ \&\ !L.vertical) \\ \qquad return\ slope\ ==\ L.slope; \\ \quad else\ return\ vertical\ ==\ L.vertical; \\ \} \end{array}$$

(f) None of the above.

A line has many possible representations, as p may be any point on the line. Thus, if the instance variables of two lines assume the same values, the lines coincide, but the converse is false.

The method in part (e) returns the value *true* if and only if the lines have the same direction, namely iff they are either equal or parallel.

Thus, only (c) is correct. (As many students marked (e) because they considered equal lines to be parallel, it has been decided to ignore this part.)

2. Two methods have been added to the class *Point* of Exercise 3: The method
Point midpoint $(Point\ Q)$
which returns the midpoint of the interval connecting the given point with another point Q, and the method
void invertColor $(Point\ Q)$
which inverts the color of all points lying on that interval. (Here assume that each point on the screen may be colored either white or black.) Now a third method is added as follows:

$$
\begin{array}{l}
void\ invertColorPartly(Point\ Q,\ int\ n)\\
\{\\
\quad if\ (n\ >\ 0)\\
\quad \{\\
\quad\quad invertColor(Q);\\
\quad\quad Point\ P1\ =\ midpoint\ (Q);\\
\quad\quad Point\ P2\ =\ midpoint\ (P1);\\
\quad\quad Point\ P3\ =\ P1.midpoint\ (Q);\\
\quad\quad P2.invertColorPartly\ (P3,\ n-1);\\
\quad \}\\
\}
\end{array}
$$

Suppose that somewhere in the program we have two points P and Q of distance 1 apart, and the command
$P.invertColorPartly\ (Q,\ 10)$
is given at a situation where all the screen is white. After this command is executed

(a) the screen is still all white.

(b) there are two black line segments of total length 1/2 on the screen.

(c) there are 10 black line segments of total length 341/512 on the screen.

(d) there are 10 black line segments of total length 1/2 on the screen.

(e) there are 1023 black line segments of total length 1023/1024 on the screen.

(f) there are 20 black line segments on the screen.

The method inverts the color of all points between the given point and Q, and then continues to apply itself (with the int parameter decreased by 1) on the interval comprising the "middle half" of the original interval. When we start with a white screen, the first stage will color the interval in question black, the second stage will return the middle half to white, the third stage will again turn to black the middle half of that middle half, and so forth. That is, in the kth stage an interval of length $1/2^{k-1}$ changes its color. For odd k the change is from white to black, while for even k it is from black to white, and thus in any case the number of black intervals increases by 1. Hence the number of black line segments by the end of the execution of the method is 10, and the total length of these intervals is

$$1 - \frac{1}{2} + \frac{1}{4} - \ldots - \frac{1}{2^9} = \frac{341}{512}\,.$$

Thus, only (c) is correct.

The next two questions (3 – 4) relate to the following classes:

```
class A
{
   public A() {;}
   public int get() {return this.getY();}
   public int getY() {return -1;}
}
```

```
class B extends A
{
   private int _y;
   public B (int t)
   {
      super();
      _y = t;
   }
   public int getY() {return _y;}
   public void setY (int t) {this._y = t;}
}
```

3. What will the output of the following code section be? (We have written the output from left to right even though it is supposed to be from top to bottom.)

```
public class Bohan2
{
   public static void main (String[] a)
   {
      A s1 = new A();
      A s2 = new B(20);
      B s3 = new B(40);
      System.out.println (s1.get());
      System.out.println (s2.get());
      System.out.println (s3.get());
      System.out.println (((A)s3).get());
      System.out.println (((B)s1).get());
   } // main
}
```

(a) -1, -1, 40, -1, runtime error.

(b) -1, 20, 40, -1, -1.

(c) -1, 20, 40, 40, -1.

(d) -1, 20, 40, 40, runtime error.

(e) The compiler will reject the code.

The variable $s1$ points to an object of type A, while $s2$ and $s3$ point to type B objects. Hence for $s1$ the method $get()$ returns -1, while for the other variables it returns $\_y$, which is 20 for $s2$ and 40 for $s3$. It follows that the output due to the first 4 $println()$ commands is -1, 20, 40, 40. Upon getting to the last $println()$ command, the compiler accepts the code since it is not possible to know at this stage if the expression $(B)s1$ makes sense. It will depend if at runtime the object $s1$ refers to has the capacity to be casted to type (B). In this example when running the program we find that $s1$ points to an object of type A which cannot be casted to an object of type B. Hence this command will cause a runtime error.

Thus, only (d) is correct.

4. What will the output of the following code section be?

```
public class Bohan3
{
   public static void main (String[] a)
   {
      A[] arr = new A[4];
      arr[0] = new A();
      arr[1] = new B(10);
      arr[2] = new B(20);
      arr[3] = arr[2];
      arr[1] = arr[0];

      ((B) arr[3]).setY(5);

      int sum = 0;
      for (int i = 0; i < 4; i = i + 1)
      sum = sum + arr[i].get();
      System.out.println (sum);
   }// main
}
```

(a) 38.

(b) 28.

(c) 8.

(d) The compiler will reject the code.

(e) The execution of the program will result in a runtime error.

The statements assigning objects to the entries of the array arr make $arr[0]$ and $arr[1]$ point to some (and the same) object of type A, while $arr[2]$ and $arr[3]$ point to some (and the same) object of type B. The command $((B)\ arr[3]).setY(5)$ sets the $\_y$ field of the object that $arr[2]$ and $arr[3]$ point to to be 5.

In the for-loop statement for $i = 0, 1$ the method $get()$ returns -1; and for $i = 2, 3$ it returns 5. Summing the 4 numbers, we obtain 8.

Thus, only (c) is correct.

5. Consider the code section

```
int[][] M = {{1, 2}, {3, 4}};
int p = product (M[1]) + product (M[0]) + product (M[1]);
System.out.print (p);
```

where the function $product()$ is defined by:

```
static int product (int[] M)
{
   for (int i = M.length - 2; i >= 0; i = i - 1)
      M[i] = M[i] * M[i + 1];
   return M[0];
}
```

(a) The compiler will reject the code since the argument for the function $product()$ is a 2-dimensional array, whereas the function is actually invoked for a 1-dimensional array.

(b) The compiler will reject the code, but due to another reason.

(c) The output due to this code section is 62.

(d) The output due to this code section is 26.

(e) It would save the computer work, and make no difference for the program, if the second line in the code was replaced by:
$int\ p = 2 * product\ (M[1]) + product\ (M[0]);$

(f) The change suggested in (e) would yield the same output, but there would be no significant runtime saving, assuming that the arithmetic operations are efficient anyway.

(g) None of the above.

Initially, $M[0] = \{1,\ 2\}$ and $M[1] = \{3,\ 4\}$. It is easy to see that the first invocation *product* $(M[1])$ returns $3 \cdot 4 = 12$. We observe that in the process of this calculation $M[1]$ itself is changed to become $\{12,\ 4\}$. The invocation *product* $(M[0])$ returns $1 \cdot 2 = 2$. The invocation *product* $(M[1])$ now operates on the array $\{12,\ 4\}$, and therefore returns $12 \cdot 4 = 48$. Altogether $p = 12 + 2 + 48 = 62$, which is the result to be printed.

The change suggested in (e) would lead to a result of 26 as *product* $(M[1])$ would not be calculated a second time with a new value of $M[1]$.

Thus, only (c) is correct.

6. A, B, C and $UseABC$ are public classes in some directory. The classes B and C extend A (but do not extend each other). Each of the three classes A, B and C has a single constructor without parameters. The body of the constructor of A consists of the single line
$System.out.println$ ("A");
The body of the constructor of B consists of two lines:
$super()$;
$System.out.println$ ("BB");
and that of C consists of:
$super()$;
$System.out.println$ ("CCC");
The class $UseABC$ is the main class and the body of the function $main()$ is:

```
A a;
for (int i = 1; i <= 100; i++)
   if (Math.random() < 0.5)
      a = new B();
```

$else$
    $a = new\ C();$

(a) The compiler will reject the code since the variable a which is declared as a type A variable cannot serve both for new objects of type B and for new objects of type C.

(b) Suppose in some execution of the code of $UseABC$ the function $Math.random()$ returns exactly 50 values below 0.5. In such an execution the number of printed characters is 250 (ignoring space and new line characters).

(c) On the average, the number of printed characters is 350. If we change $UseABC$ by interchanging B and C, we will still obtain the same number on the average.

(d) If C is declared as extending B instead of extending A, then the compiler will reject the code since a is of type A only.

(e) If B is declared as extending C instead of extending A, then the number of characters to be printed must be even (independently of the results of the random function).

(f) None of the above.

An instantiation of a B object brings about the printing of the string "ABB", while that of a C object brings about the printing of the string "$ACCC$". Since we should expect $Math.random()$ to return approximately 50 numbers below 0.5 and 50 above it out the 100 times it is invoked, the number of printed characters should be approximately $50 \cdot 3 + 50 \cdot 4 = 350$. The same is valid if we reverse the condition determining each time whether a B object or a C object is instantiated.

If B extends C instead of A, then an instantiation of a B object causes the string "$ACCCBB$" to be printed, while that of a C object produces the same result as before. Hence, in any case an even number of characters is printed, and the same holds for the total for the whole loop.

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

7. A, B, C and $UseABC$ reside in one directory. The class C extends B, which itself extends A. Each of the three classes has a public $toString()$ method, returning "A" (for A), "BB" (for B) and "CC" (for C). Each of the three classes contains also a $print()$ method. In the class A this method consists of the single line
$System.out.print(this);$
while for B it consists of the same line written twice, and for C the same written three times. The main method of $UseABC$ contains the following code:

```
A a = new A();
B b = new B();
C c = new C();

A[] R = {a, b, c};
for (int i = 0; i < 3; i = i + 1)
   R[i].print();

((A) b).print();
((A) c).print();
((B) c).print();
```

(a) The code section is legal and brings about the printing of 36 characters.

(b) If the last three lines of the code are replaced by

```
((B) a).print();
((C) a).print();
((C) b).print();
```

then there will be a compilation error.

(c) The change described in the preceding part will leave the code valid. However, the number of characters printed by it is only 20.

(d) (This part has been cancelled.)

(e) The line $System.out.print(this)$ is problematic, and will be rejected by the compiler.

(f) None of the above.

When the $print()$ method is invoked for an A object – it prints "A", for a B object it prints "$BBBB$", and for a C object it prints "$CCCCCC$". Hence the for loop prints altogether $1+4+6=11$ characters, and the 3 lines following it print 4, 6, 6 characters, respectively. (Note that the castings are immaterial.) Therefore, the total number of characters printed is 27.

The change suggested in (b) will cause a runtime error, as the objects in those lines "pretend" to be of types $B,\ C,\ C$, but are really of types $A,\ A,\ B$, respectively.

Thus, only (f) is correct.

8. Let s be a string. A *permutation* of s is a string composed of the characters of s in some order (equal or not to the original order). The requirement is that in a permutation all characters of s will appear the same number of times as in s. For simplicity, we will assume that there are no repetitions of characters in s.

   You are required to complete the definition of the recursive static function $perm(String\ s1,\ String\ s2)$, by means of which we will obtain the printing of all the permutations of a given string s. For your convenience, we recall the following two methods of the class $String$:

   1) $String\ subString\ (startIndex,\ lastIndex)$
   // returns the substring of the current object between $startIndex$ and
   // $lastIndex - 1$.

   2) $char\ charAt(i)$
   //returns the character at index i of the current object.

```
public class X
{
   public static void main (String[] args)
   {
      System.out.println ("all permutations of " + args[0]);
      perm (args[0]);
   }

   static void perm (String s)
   {perm (s,""); }

   static String delete (String s, int i)
   {return s.substring(0, i) + s.substring (i + 1, s.length()); }

   // prints all permutations of String s1 followed by s2
   static void perm (String s1, String s2)
   {
      /* FILL IN DEFINITION (AT MOST 6 LINES */
   }
}
```

Printing all permutations of $s1$ followed by $s2$ may be accomplished as follows: If $s1$ is the empty string, we simply have to print $s2$. Otherwise, for each of the characters of $s1$ we print all permutations of $s1$ with this character omitted, followed by the string consisting of the character concatenated with $s2$. This gives rise to the following code:

```
if (s1.length() == 0)
   System.out.println (s2);
else
   for (int i = 0; i < s1.length(); i++)
      perm (delete (s1, i), s1.charAt (i) + s2);
```