Midterm #1

Answer all the following questions. For the multiple-choice questions, mark all correct answers to each question.

1. Consider the following code section:

```
int d = 1;
while (number > 0)
{
   number = number - d;
   d = d + 1;
}
System.out.println(number);
```

   Assume that *number* is a variable of type *int*, which has been initialized earlier. Then:

   (a) If the value of *number* prior to the execution of this block is 5051, then the output to be printed on the screen is -100.

   (b) There are more than two initial values of *number* for which the output is -20.

   (c) There are exactly two initial values of *number* for which the output is -20.

   (d) The output cannot be 0.

   (e) The output cannot be -1.

   (f) None of the above.

At the first execution of the loop, *number* is decreased by 1, at the second execution by 2, and so forth. After 100 executions we have

$$number = 5051 - 1 - 2 - \ldots - 100 = 5051 - \frac{100(100+1)}{2} = 1\,.$$

At the next execution we shall have $number = 1 - 101 = -100$, and the loop will not be executed any more.

There are many possibilities for the output to be -20. This is the case:

- if the initial value of *number* is -20;
- or if the initial value of *number* is:

  $$-20 + 21 + 20 + 19 + \ldots + 1 = 211\,;$$

- or if the initial value of *number* is:

  $$-20 + 22 + 21 + 20 + \ldots + 1 = 233\,;$$

- or other possibilities obtained similarly.

If *number* is initially $0,\ 1,\ 3,\ 6, \ldots,$ then the output is 0, while if it is initially $-1,\ 2,\ 5,\ 9, \ldots,$ then the output is -1.

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

2. A code is written to reverse integers, namely, given a non-negative integer, to construct the integer obtained by reversing its digits. In the following code, *number* is a positive integer of type *int*, and after the execution of the code, *reverse* is supposed to hold the reverse number. For example, if *number* is 58040, then *reverse* should be 4085.

```
int reverse = 0, digit = 0;
while (number > 0)
{
   digit = number % 10;
   number = number / 10;
   reverse = 10 * reverse + digit;
}
```

(a) The code works correctly for each initial value of *number*. Moreover, if we represent *number* and *reverse* in base 12, we will still obtain a string of digits and its reverse.

(b) The code works correctly for each initial value of *number*, except for values which are smaller than *Integer.MAX_VALUE* whose reverse is larger than it. For such numbers the result will be incorrect. (The largest value of type *int* is 2147483647.)

(c) To make the code work correctly it suffices to replace the first line in the body of the loop by:

$digit \ = \ number \ - 10 \ * \ (number \ / \ 10);$

(d) The code works well for palindromes (i.e., numbers such as 38683, which are the same when read from right to left as when read from left to right), but not for other numbers.

(e) None of the above.

In principle, the code works well for any number. In fact, the integer division of *number* by 10 at each stage means truncating its current least significant digit. Multiplying the current value of *reverse* by 10 and adding *digit* to it then “glues” the truncated digit to *reverse* as its most significant digit. Thus at each stage *number* holds a block consisting of several leading digits of the initial representation of *number*, while *reverse* consists of those digits which were erased already, in the reverse order. (A formal proof would proceed by induction.) At the end of the process, *number* becomes 0 and reverse holds all the initial digits in reverse order. Note that, if *number* originally ends with a block of 0's, then these 0's are “lost”.

However, the above fails if at some point the calculations produce an incorrect value due to the limitations on some variables. In our case, this may happen as *reverse* becomes larger and larger, since its value “tries” to exceed the maximal possible type *int* value. Consider, for example, the case where $number = 1111111119$. In the last step we encounter the calculation $10 \cdot 911111111 + 1$, which theoretically should produce 9111111111, but fails to do this as it is above *Integer.MAX_VALUE*. Note that for palindromes, since initially *number* was bounded above by *Integer.MAX_VALUE*, so will be *reverse*, and therefore the problem will not arise.

The code suggested in (c) is equivalent to the code it is supposed to replace, and therefore the change is immaterial.

There is no reason why the process should work on representations in bases other than 10. For example, if *number* = 12 then the code assigns *reverse* the value 21. In base 12, these values are represented as 10 and 19, respectively.

The phrasing in (d) turned out to be vague, in the sense that the last part of the sentence may be interpreted as claiming that the program does not work well for all non-palindromes although it may work well for some such numbers (which is correct), or as claiming that the program works well for no non-palindrome (which is incorrect). It was therefore decided to disregard this part.

Thus, only (b) is correct (and (d) is disregarded).

3. Consider the following code section:

```
int x = 13487, y = x;
do
{
   x = x − 2;
   y = x − 2;
}
while (x > y);
```

   (a) The body of the loop will be executed only once, since after the first execution the values of x and of y will coincide. (Both will be $x - 2$.)
   (b) The body of the loop will be executed infinitely many times (at least theoretically; at some point the computer is doomed to fail).
   (c) If we omit the part of the command $y = x$ from the first code line, the compiler will complain about the variable y not being initialized.
   (d) The body of the loop will be executed finitely many times only, since arithmetic operations yielding values beyond the minimal (or maximal) possible *int* value move cyclically to the other "end".

(e) The number 13487 is too simple to be represented on 32 bits, and thus the compiler will demand that we use type *short* instead of *int*.

(f) None of the above.

Each time the loop is executed, x is decreased by 2, and y is assigned a value smaller by 2 than this new value. Hence in principle the loop should be infinite. In practice, though, at the point when the value of x will become $Integer.MIN\_VALUE + 1$ (notice that x varies over odd numbers only), the value assigned to y will be $Integer.MIN\_VALUE - 1$, which, due to the limitations on values of *int* type, will be translated to $Integer.MAX\_VALUE$.

If y is not declared, the compiler will reject the code. Since the complaint of the compiler would be that y was not defined, and not that it was not initialized, it was decided to cancel (c).

Thus, only (d) is correct (and (c) is disregarded).

4. In this question you are required to write a code section. The function *count* (*int*[] *A*, *int k*) receives an array A of numbers and a number k. It returns the number of occurrences of k within A. Write only the body of the function, namely the inner block. The block should contain no more than 4 lines. Write a rough draft first, and check it carefully.

```
static int count (int[] A, int k)
{
  <YOUR ANSWER>
}
```

We need to keep a counter, initialized to 0, go over all array entries and increment the counter each time we find a k. This can be accomplished by:

```
int howMany = 0;
for (int i = 0; i < A.length; i = i + 1)
  if (A[i] == k)
    howMany = howMany + 1;
return howMany;
```

5. Consider the following code section:

```
int m = (int) Math.random();
System.out.println (m);
```

   (a) The code is rejected by the compiler.
   (b) Certain executions of this code will produce a runtime error, while others will not.
   (c) The program definitely produces a runtime error.
   (d) The output is always 1.
   (e) The output is sometimes 1 and sometimes different from 1.
   (f) The output is always 0.
   (g) The output is sometimes 0 and sometimes different from 0.
   (h) None of the above.

   Reminder: The function *Math.random*() returns a value between 0 (inc.) and 1 (exc.).

   Since *Math.random*() returns a value in the interval $[0, 1)$, and since the casting operation on (non-negative) *double* values simply truncates the fractional part, the value of m must be 0.

   Thus, only (f) is correct.

6. Consider the following code section:

```
int j = Console.readInt ("enter an integer");
int[] A = {1,2,3,4};
if ((−1 < j & j < 4) & A[j] % 2 == 0)
   System.out.println ("yes");
System.out.println ("no");
```

   (a) The code is rejected by the compiler.
   (b) There exist integer j values which will produce a runtime error.

(c) For every j the code will produce a runtime error.

(d) There exist integer j values for which the program will not terminate its execution.

(e) The program always prints "yes".

(f) The program sometimes prints "yes" and sometimes prints "no".

(g) None of the above.

If the value of j is 0, 1, 2 or 3, there will be no problem in running the code. For $j = 0,\ 2$, the output will be only "no", whereas for $j = 1,\ 3$ both "yes" and "no" will be printed. For all other values of j there will be a runtime error due to $A[j]$ being undefined.

Thus, only (b) is correct.

7. Consider the following code section:

```
int j = Console.readInt ("enter an integer");
int[] A = {1, 2, 3, 4};
if ((-1 < j & j < 4) && A[j] % 2 == 0)
   System.out.println ("yes");
else
   System.out.println ("no");
```

(a) The code is rejected by the compiler.

(b) There exist integer j values which will produce a runtime error.

(c) For every j the code will produce a runtime error.

(d) There exist integer j values for which the program will not terminate its execution.

(e) The program always prints "yes".

(f) The program sometimes prints "yes" and sometimes prints "no".

(g) The program always prints "no".

(h) None of the above.

As opposed to the preceding question, here $A[j]$ will be tested only for values of j satisfying the condition tested earlier, namely $j = 0,\ 1,\ 2,\ 3$. Hence no runtime error may be produced.

Thus, only (f) is correct.

8. Consider the following code section:

```
int sum = 0, number;
for (number = 1; number <= 10; number = number + 1)
   number = number - 1;
System.out.println (sum);
```

   (a) The code is rejected by the compiler.
   (b) The program prints 0.
   (c) The program prints 55.
   (d) The program prints 45.
   (e) None of the above.

   The program enters an infinite loop. Indeed, each time we execute the body of the loop *number* is decreased by 1, only to be increased by 1 at the updating stage of the loop. Hence each time the loop condition is tested we have *number* $= 1$, and in particular the condition will always be satisfied.

   Thus, only (e) is correct.

9. Consider the following code section:

```
int k = 10, sum = 0;
for (int j = 0; j < k; j =k - 1)
   sum = sum + 1;
```

   (a) By the end of the execution of this code we have $k = 10$.
   (b) The code is rejected by the compiler.

(c) By the end of the execution of this code we have $j = 9$.

(d) The program enters an infinite loop.

(e) None of the above.

The first time the loop condition is tested we have $j = 0$ and $k = 10$, so that the condition is satisfied. Since the update will always produce $j = 9$, the condition will be satisfied on the consecutive tests as well, and consequently the program enters an infinite loop.

Thus, only (d) is correct.

10. Consider the following code section:

```
for (int j = 0; j < 10; j = j + 1)
   for (j = 10; j > 0; j = j − 1)
      System.out.print (" ∗ ");
```

(a) The code will print 100 times the star *.

(b) The code is rejected by the compiler.

(c) By the end of the execution of this code we have $k = 1$.

(d) The program enters an infinite loop.

(e) None of the above.

The first time the program gets to this section, it sets $j = 0$, the condition in the outer loop is satisfied, and then the initialization of the inner loop changes j to 10, so that the condition of the inner loop is satisfied as well. Now the inner loop terminates with $j = 0$, the updating step of the outer loop changes it to 1, and therefore the outer loop is executed a second time. The entry and exit values on consecutive runs will be the same, which means the program enters an infinite loop.

Thus, only (d) is correct.

11. Consider the following code section:

```
for (int i = 0; i < 10; i = i + 1)
   for (i = 0; i < 9; i = i + 1)
      System.out.print ("*");
```

(a) The code will print 90 times the star *.

(b) The code is rejected by the compiler.

(c) The code will print 10 times the star *.

(d) The program enters an infinite loop.

(e) None of the above.

The first time the program gets to this section, it sets $i = 0$, and then the inner loop prints 9 times a star *. Upon exiting from the inner loop we have $i = 9$, and then the updating stage of the outer loop increases it to 10. Hence this time the condition of the outer loop is false, and the code is finished.

Thus, only (e) is correct.

12. A palindrome is a string which is read the same in both directions (from right to left as from left to right). You are required to complete the missing parts in the program below. In the code there are three missing segments (denoted by ?? ??), which you have to complete in such a way that the program will determine whether or not the string transferred to it is a palindrome. For extra clarification, here are three examples of how the program should work:

    i. If *str* = "*abcdcba*", the program should print:
       abcdcba is a palindrome

    ii. If *str* = "*abccba*", the program should print:
       abccba is a palindrome

    iii. If *str* = "*adbccba*", the program should print:
       adbccba is not a palindrome

```
public static void palindrome (String str){
   boolean pal = true;
```

```
    String is = ??1??;

    for (int i = 0; i < str.length(); i = i + 1)
       if ??2??{
          pal = false;
          is = ??3??;
       }
    System.out.print (str + is + "a palindrome");
}
```

To test a string for being a palindrome one needs to check if its last character coincides with the first, its second last coincides with the second, and so forth. If the answers to all these queries are affirmative, then the string is a palindrome, while if at least one is wrong the string is not a palindrome. The conditional statement should result in deciding that *str* is not a palindrome if for some i the ith character is different than the last ith character. Hence we may take

$$\textbf{??2??} = str.charAt(i)\ != str.charAt(str.length() - i - 1)\,.$$

Since we always want to print the word " is ", and add "not " if the string is a palindrome, we should take:

$$\textbf{??1??} = \text{"}\ is\ \text{"}, \qquad \textbf{??3??} = \text{"}\ is\ not\ \text{"}\,.$$