Lab 3. Exercises on Java Basics

Writing Good Programs

The only way to learn programming is program, program and program. Learning programming is like learning cycling, swimming or any other sports. You can't learn by watching or reading books. Start to program immediately. On the other hands, to improve your programming, you need to read many books and study how the masters program.
It is easy to write programs that work. It is much harder to write programs that not only work but also easy to maintain and understood by others – I call these good programs. In the real world, writing program is not meaningful. You have to write good programs, so that others can understand and maintain your programs.
Pay particular attention to:

1. **Coding style**:
   - Read Java code convention: "Google Java Style Guide" or "Java Code Conventions - Oracle".
   - Follow the Java Naming Conventions for variables, methods, and classes STRICTLY. Use CamelCase for names. Variable and method names begin with lowercase, while class names begin with uppercase. Use nouns for variables (e.g., radius) and class names (e.g., Circle). Use verbs for methods (e.g., getArea(), isEmpty()).
   - **Use Meaningful Names**: Do not use names like a, b, c, d, x, x1, x2, and x1688 - they are meaningless. Avoid single-alphabet names like i, j, k. They are easy to type, but usually meaningless. Use single-alphabet names only when their meaning is clear, e.g., x, y, z for co-ordinates and i for array index. Use meaningful names like row and col (instead of x and y, i and j, x1 and x2), numStudents (not n), maxGrade, size (not n), and upperbound (not n again). Differentiate between singular and plural nouns (e.g., use books for an array of books, and book for each item).
   - Use consistent indentation and coding style. Many IDEs (such as Eclipse / NetBeans) can re-format your source codes with a single click.
2. **Program Documentation**: Comment! Comment! and more Comment to explain your code to other people and to yourself three days later.
3. The only way to learn programming is program, program and program on challenging problems. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests (such as International Collegiate Programming Contest (ICPC), International Olympiad in Informatics (IOI)).

1 More Exercises

1.1 Matrices (2D Arrays)

Similar to *Math* class, write a Matrix library that supports matrix operations (such as addition, subtraction, multiplication) via 2D arrays. The operations shall support both *double* and *int*. Also write a test class to exercise all the operations programmed.

Hints

```
public class Matrix {
  // Method signatures
  public static void createRandomMatrix(int[][] matrix);
  public static int[][] createRandomMatrix(int rows, int cols);

  public static void print(int[][] matrix);
  public static void print(double[][] matrix);

  // Used in add(), subtract()
  public static boolean haveSameDimension(int[][] matrix1,
                                          int[][] matrix2);
  public static boolean haveSameDimension(double[][] matrix1,
                                          double[][] matrix2);

  public static int[][] add(int[][] matrix1, int[][] matrix2);
  public static double[][] add(double[][] matrix1,
                               double[][] matrix2);

  public static int[][] subtract(int[][] matrix1, int[][] matrix2);
  public static double[][] subtract(double[][] matrix1,
                                    double[][] matrix2);

  public static int[][] multiply(int[][] matrix1, int[][] matrix2);
  public static double[][] multiply(double[][] matrix1,
                                    double[][] matrix2);
  ......
}
```

1.2 Trigonometric Series

Write a method to compute $\sin(x)$ and $\cos(x)$ using the following series expansion, in a class called **TrigonometricSeries**. The signatures of the methods are:

```
// x in radians, NOT degrees
public static double sin(double x, int numTerms);
public static double cos(double x, int numTerms);
```

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$
$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

Compare the values computed using the series with the JDK methods *Math.sin()*, *Math.cos()* at $x = 0, \pi/6, \pi/4, \pi/3, \pi/2$ using various numbers of terms.

Hints
Do not use int to compute the factorial; as factorial of 13 is outside the int range. Avoid generating large numerator and denominator. Use double to compute the terms as:

$$\frac{x^n}{n!} = \left(\frac{x}{n}\right)\left(\frac{x}{n-1}\right)\cdots\left(\frac{x}{1}\right).$$

1.3 Exponential Series

Write a method to compute the sum of the series in a class called **SpecialSeries**. The signature of the method is:

```java
public static double specialSeries(double x, int numTerms);
```

$$x + \frac{1}{2} \times \frac{x^3}{3} + \frac{1 \times 3}{2 \times 4} \times \frac{x^5}{5} + \frac{1 \times 3 \times 5}{2 \times 4 \times 6} \times \frac{x^7}{7} + \frac{1 \times 3 \times 5 \times 7}{2 \times 4 \times 6 \times 8} \times \frac{x^9}{9} + \cdots ; -1 \le x \le 1.$$

1.4 FactorialInt (Handling Overflow)

Write a program called **FactorialInt** to list all the factorials that can be expressed as an *int* (i.e., 32-bit signed integer in the range of [−2147483648, 2147483647]). Your output shall look like:

Command window

```
The factorial of 1 is 1
The factorial of 2 is 2
...
The factorial of 12 is 479001600
The factorial of 13 is out of range
```

Hints
The maximum and minimum values of a 32-bit int are kept in constants *Integer.MAX_VALUE* and *Integer.MIN_VALUE*, respectively. Try these statements:

```
System.out.println(Integer.MAX_VALUE);
System.out.println(Integer.MIN_VALUE);
System.out.println(Integer.MAX_VALUE + 1);
```

Take note that in the third statement, Java Runtime does not flag out an overflow error, but silently wraps the number around. Hence, you cannot use $F(n) * (n+1) > Integer.MAX\_VALUE$ to check for overflow. Instead, overflow occurs for $F(n+1)$ if ($Integer.MAX\_VALUE$ / Factorial(n)) $< (n+1)$, i.e., no more room for the next number.

Try
Modify your program called **FactorialLong** to list all the factorial that can be expressed as a long (64-bit signed integer). The maximum value for long is kept in a constant called $Long.MAX\_VALUE$.

1.5 FibonacciInt (Handling Overflow)

Write a program called **FibonacciInt** to list all the Fibonacci numbers, which can be expressed as an int (i.e., 32-bit signed integer in the range of [−2147483648, 2147483647]). The output shall look like:

Command window

```
F(0) = 1
F(1) = 1
F(2) = 2
...
F(45) = 1836311903
F(46) is out of the range of int
```

Hints
The maximum and minimum values of a 32-bit int are kept in constants $Integer.MAX\_VALUE$ and $Integer.MIN\_VALUE$, respectively. Try these statements:

```
System.out.println(Integer.MAX_VALUE);
System.out.println(Integer.MIN_VALUE);
System.out.println(Integer.MAX_VALUE + 1);
```

Take note that in the third statement, Java Runtime does not flag out an overflow error, but silently wraps the number around. Hence, you cannot use $F(n) = F(n-1) + F(n-2) > Integer.MAX\_VALUE$ to check for overflow. Instead, overflow occurs for $F(n)$ if $Inte-$

ger.MAX_VALUE $-F(n-1) < F(n-2)$ (i.e., no more room for the next Fibonacci number).

Try
Write a similar program called **TribonacciInt** for Tribonacci numbers.

1.6 Number System Conversion

Write a method call *toRadix()* which converts a positive integer from one radix into another. The method has the following header:

```
1  // The input and output are treated as String.
   public static String toRadix(String in, int inRadix, int outRadix)
```

Write a program called **NumberConversion**, which prompts the user for an input string, an input radix, and an output radix, and display the converted number. The output shall look like:

Command window

```
Enter a number and radix: A1B2
Enter the input radix: 16
Enter the output radix: 2
"A1B2" in radix 16 is "1010000110110010" in radix 2.
```

1.7 NumberGuess

Write a program called **NumberGuess** to play the number guessing game. The program shall generate a random number between 0 and 99. The player inputs his/her guess, and the program shall response with "Try higher", "Try lower" or "You got it in n trials" accordingly. For example:

Command window

```
java NumberGuess
Key in your guess:
50
Try higher
70
Try lower
65
Try lower
61
You got it in 4 trials!
```

Hints

Use *Math.random()* to produce a random number in double between 0.0 (inclusive) and 1.0 (exclusive). To produce an int between 0 and 99, use:

```
final int SECRET_NUMBER = (int)(Math.random()*100);  // truncate to int
```

1.8 WordGuess

Write a program called **WordGuess** to guess a word by trying to guess the individual characters. The word to be guessed shall be provided using the command-line argument. Your program shall look like:

Command window

```
java WordGuess testing
Key in one character or your guess word: t
Trial 1: t__t___
Key in one character or your guess word: g
Trial 2: t__t__g
Key in one character or your guess word: e
Trial 3: te_t__g
Key in one character or your guess word: testing
Congratulation!
You got in 4 trials
```

Hints

1. Set up a *boolean* array (of the length of the word to be guessed) to indicate the positions of the word that have been guessed correctly.
2. Check the length of the input *String* to determine whether the player enters a single character or a guessed word. If the player enters a single character, check it against the word to be guessed, and update the boolean array that keeping the result so far.

Try
Try retrieving the word to be guessed from a text file (or a dictionary) randomly.

1.9 DateUtil

Complete the following methods in a class called **DateUtil**:

- *boolean isLeapYear(int year)*: returns *true* if the given year is a leap year. A year is a leap year if it is divisible by 4 but not by 100, or it is divisible by 400.

- *boolean isValidDate(int year, int month, int day)*: returns true if the given year, month and day constitute a given date. Assume that year is between 1 and 9999, month is between 1 (Jan) to 12 (Dec) and day shall be between 1 and 28|29|30|31 depending on the month and whether it is a leap year.
- *int getDayOfWeek(int year, int month, int day)*: returns the day of the week, where 0 for SUN, 1 for MON, ..., 6 for SAT, for the given date. Assume that the date is valid.
- *String toString(int year, int month, int day)*: prints the given date in the format "xxxday d mmm yyyy", e.g., "Tuesday 14 Feb 2012". Assume that the given date is valid.

Hints

To find the day of the week (Reference: Wiki "Determination of the day of the week"):

1. Based on the first two digit of the year, get the number from the following "century" table.

1700-	1800-	1900-	2000-	2100-	2200-	2300-	2400-
4	2	0	6	4	2	0	6

   Take note that the entries $4, 2, 0, 6$ repeat.

2. Add to the last two digit of the year.
3. Add to "the last two digit of the year divide by 4, truncate the fractional part".
4. Add to the number obtained from the following month table:

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Non-Leap Year	0	3	3	6	1	4	6	2	5	0	3	5
Leap Year	6	2	3	6	1	4	6	2	5	0	3	5

5. Add to the day.
6. The sum modulus 7 gives the day of the week, where 0 for SUN, 1 for MON, ..., 6 for SAT.

For example: 2012, Feb, 17

EX

```
(6 + 12 + 12/4 + 2 + 17) % 7 = 5 (Fri)
```

The skeleton of the program is as follows:

```java
/**
 * Utilities for Date Manipulation
 */
public class DateUtil {

   // Month's name - for printing
   public static String[] strMonths
     = {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

   // Number of days in each month (for non-leap years)
   public static int[] daysInMonths
     = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

   /*
    * Returns true if the given year is a leap year
    */
   public static boolean isLeapYear(int year) { ..... }

   /*
    * Return true if the given year, month, day is a valid date
    * year: 1-9999
    * month: 1(Jan)-12(Dec)
    * day: 1-28|29|30|31. The last day depends on year and month
    */
   public static boolean isValidDate(int year, int month, int day) {
      ↪ ..... }

   /*
    * Return the day of the week, 0:Sun, 1:Mon, ..., 6:Sat
    */
   public static int getDayOfWeek(int year, int month, int day) { .....
      ↪ }

   /*
    * Return String "xxxday d mmm yyyy" (e.g., Wednesday 29 Feb 2012)
    */
   public static String printDate(int year, int month, int day) { .....
      ↪ }

   // Test Driver
   public static void main(String[] args) {
     System.out.println(isLeapYear(1900));  // false
     System.out.println(isLeapYear(2000));  // true
     System.out.println(isLeapYear(2011));  // false
     System.out.println(isLeapYear(2012));  // true

     System.out.println(isValidDate(2012, 2, 29));  // true
     System.out.println(isValidDate(2011, 2, 29));  // false
     System.out.println(isValidDate(2099, 12, 31)); // true
     System.out.println(isValidDate(2099, 12, 32)); // false
```

```
49
        System.out.println(getDayOfWeek(1982, 4, 24));  // 6:Sat
51      System.out.println(getDayOfWeek(2000, 1, 1));   // 6:Sat
        System.out.println(getDayOfWeek(2054, 6, 19));  // 5:Fri
53      System.out.println(getDayOfWeek(2012, 2, 17));  // 5:Fri

55      System.out.println(toString(2012, 2, 14)); // Tuesday 14 Feb 2012
    }
57  }
```

Notes

You can compare the day obtained with the Java's Calendar class as follows:

```
1   // Construct a Calendar instance with the given year, month and day
    Calendar cal = new GregorianCalendar(year, month - 1, day);  // month
       ↪ is 0-based
3   // Get the day of the week number: 1 (Sunday) to 7 (Saturday)
    int dayNumber = cal.get(Calendar.DAY_OF_WEEK);
5   String[] calendarDays = { "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday" };
7   // Print result
    System.out.println("It is " + calendarDays[dayNumber - 1]);
```

The calendar we used today is known as Gregorian calendar, which came into effect in October 15, 1582 in some countries and later in other countries. It replaces the Julian calendar. 10 days were removed from the calendar, i.e., October 4, 1582 (Julian) was followed by October 15, 1582 (Gregorian). The only difference between the Gregorian and the Julian calendar is the "leap-year rule". In Julian calendar, every four years is a leap year. In Gregorian calendar, a leap year is a year that is divisible by 4 but not divisible by 100, or it is divisible by 400, i.e., the Gregorian calendar omits century years which are not divisible by 400. Furthermore, Julian calendar considers the first day of the year as march 25th, instead of January 1st.

This above algorithm work for Gregorian dates only. It is difficult to modify the above algorithm to handle pre-Gregorian dates. A better algorithm is to find the number of days from a known date.