Java Programs 49- 58

NOTE: These programs help you practice the following areas in Java programming:

- Object-Oriented Programming
- Collections and Generics
- Exception Handling
- File I/O

▪ All the programs are expected to run in command-line mode.
▪ You can use an IDE (Eclipse or NetBeans) to develop the projects.

Program #49: Keyword Analysis

Develop a program that scans a text file and generates a keywords analysis report to a text file. The paths of the input and output text files are given as program's arguments. So the syntax to run this program is as follows:

```
java KeywordTool <input_file> <output_file>
```

For example:

```
java KeywordTool Notes.txt report.txt
```

Then the program scans the file `Notes.txt` in the current directory and writes the keyword analysis report to the file `report.txt`.

The report lists the top 10 keywords in terms of their popularity in the document. That means the keyword which has the most number of occurrences appear first. Here's an example of the report:

```
Keywords Analysis Report for Notes.txt:

Java - 15% (30)

tutorial - 3% (6)

code - 7% (13)

example - 3% (6)

…
```

The number in the parenthesis is the number of occurrences of the keyword in the file. So:

```
Java - 15% (30)
```

meaning that the word `Java` appears 30 times in the document (`15%` of total words).

The program should check the input path before processing:

- If the path doesn't exist, print the message: `The input path does not exist.`
- If the path does exist but is not a file, print the message: `the input path must be a file.`
- If the path is a file but not a text file, print the message: `the input file must be a text file.`

Program #50: Student Management

Develop a program that allows the user to manage a list of students in a binary data file. A student has the following information:

- Student ID: number
- Full name: text
- Birthday: Date
- Email: text
- Address: text
- Faculty: text

By default, the data is stored in a binary file named Studentsdb.dat in the same directory as the program. Upon startup, the program introduces the following menu:

```
Welcome to Student Manager Lite

===============================

1. List all students
2. Find a student
3. Add a student
4. Edit a student
5. Delete a student
6. Quit

Enter your choice (1-6): _
```

Each feature is described in detail as below:

1. List all students:

The program prints all students in the data file in the following format:

```
No. |   ID   |   Full Name  |  Birthday     |   Email   |     Address          |   Faculty
---------------------------------------------------------------------------------------------
1.  | 102 | Johny Tan  |   04/18/1990  | johnytan@gmail.com | Redmon, USA | Software Engineering
2.
3.
```

Note that the list is sorted by student `ID` column in alphanumeric order (ascending). The `No` columns always starts from 1, 2, 3, ... and so on. The Birthday is in the format of `mm/dd/YYYY`.

In case there's no student, the program prints the following message: `There's no student found`.

2. Find a student:

The program asks the user to:

```
Enter keyword: _
```

Then the program searches for the students who have any fields (full name, email, address or faculty) that match the specified keyword. If there are students found, the program prints the result like this:

```
Found <number> students
```

`<number>` is the number of students that match the given keyword. And it is followed by the list in the same format mentioned above.

In case there's no match, the program prints:

```
No student found.
```

3. Add a student:

The program asks the user to enter information for each field of a student sequentially as follows (all fields are required):

```
Enter student ID: _
Enter full name: _
Enter email: _
Enter birthday: _
Enter address: _
Enter faculty: _
```

If the student ID is not a number, the program should ask for it again until it is correct:

```
student ID must be a number. Enter student ID again: _
```

In case there's already a student with the same ID in the database, the program prints the following message:

```
The given student ID already exists. Enter student ID again: _
```

The program should repeatedly ask for the student ID until it is unique in the database.

Note that the user must enter birthday in the following format: `mm/dd/YYYY`. If invalid, the user is asked to enter birthday again until it is correct:

```
birthday is invalid. Enter birthday again: _
```

Once all fields are properly entered, the program inserts the new student into the data file, and prints the following message:

```
New student saved successfully.
```

4. Edit a student:

The program asks for the student ID as follows:

```
Enter student ID: _
```

If the input entered is not a number, the program asks the user to enter again until it is correct.

If the given student ID not found in the database, the program prints the following message:

```
Could not find a student with the given ID.
```

If exists, the program prints the student's information:

You're about to update Student ID <ID>:

```
Full name: _

Email: _

Birthday: _

Address: _

Faculty: _

(Leave blank to skip)
```

then asks the user to update each field sequentially as follows:

```
Enter full name: _

Enter email: _

Enter birthday: _

Enter address: _

Enter faculty: _
```

Note that the user can leave a field blank (by hitting Enter without typing anything) to skip updating a field. The birthday is validated in the same way as when creating new student.

Once all fields are entered, the program updates the student's information in the database, and prints the following message:

```
The student <ID> has been updated successfully.
```

5. Delete a student:

The program asks for the student ID as follows:

```
Enter student ID: _
```

Again, the input is validated same as mentioned previously. Once the student is found and removed from the database, the program prints:

```
The student <ID> has been deleted.
```

6. Quit:

The program terminates.

NOTES:

- After completing each function, the program re-prints the menu and waits for input from the user.
- If there's any I/O error occurs, the program prints the following message and terminates:

```
Sorry, there has been an error occurred: <exception_message>
```

Program #51: Files Renaming Tool

Develop a program that helps renaming multiple files in a directory quickly. The user can run this program via command line like this:

```
java Renamer <dir_path> <prefix>
```

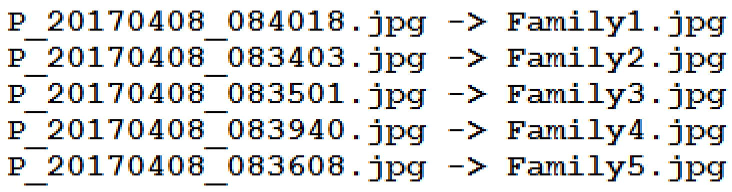
For example:

```
java Renamer photos Family
```

Suppose that the directory photos contain the following files:

```
P_20170408_084018.jpg

P_20170408_083403.jpg

P_20170408_083501.jpg

P_20170408_083940.jpg

P_20170408_083608.jpg
```

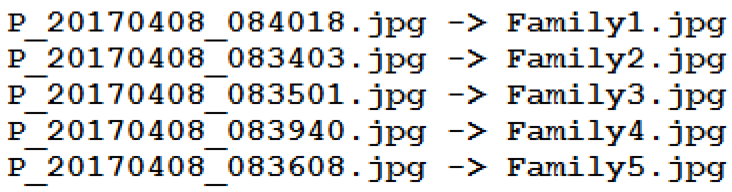
Then the program renames these files using the given prefix like this:



```
P_20170408_084018.jpg -> Family1.jpg
P_20170408_083403.jpg -> Family2.jpg
P_20170408_083501.jpg -> Family3.jpg
P_20170408_083940.jpg -> Family4.jpg
P_20170408_083608.jpg -> Family5.jpg
```

That means the files have same prefix but different count numbers which start from 1. The order for renaming is based on the alphabetical order of names of the original files. Finally the program prints a report like this:

```
Total 5 files were renamed. Details:
```



```
P_20170408_084018.jpg -> Family1.jpg
P_20170408_083403.jpg -> Family2.jpg
P_20170408_083501.jpg -> Family3.jpg
P_20170408_083940.jpg -> Family4.jpg
P_20170408_083608.jpg -> Family5.jpg
```

NOTE: only files in the root of the specified directory are renamed (no recursive renaming).

Program #52: Files Duplication Search

Develop a program for searching duplicate files in a given directory. Two files are considered duplicated if they have same MD5 hash code (Read this tutorial to know how to calculate MD5 hash for a file).

The user can run this program from command line as follows:

```
java DuplicationFinder <dir_path>
```

For example:

```
java DuplicationFinder D:\JavaCode
```

This tells the program scans all files in the directory D:\JavaCode recursively (including sub files and sub directories) to find duplicate files and print a report like this:

```
Directory scanned: D:\JavaCode

Total duplicate files: 5

File #1: JavaIOTest.java

      Location #1: D:\JavaCode\FileIO\JavaIOTest.java

      Location #2: D:\JavaCode\Test\JavaIOTest.java

File #2: ReadTextFileExample.java

      Location #1: D:\JavaCode\FileIO\Examples\ReadTextFileExample.java

      Location #2: D:\JavaCode\FileIO\Test\ReadTextFileExample.java

File #3:…

File #4:…

File #5:…
```

In case the program did not find any duplication in the specified directory, it prints the following message:

```
        No duplicate files were found.
```

Program #53: Simple Text File Comparer

Develop a program that compares two text files line by line. The user can run this program via command line as follows:

```
        java FileComparer <file_path_1> <file_path_2>
```

First, the program must validate the arguments in the following sequence:

- If there’s no argument (`<file_path_1>` and `<file_path_2>` are not specified), the program prints the following message then terminates:

```
        Please provide the paths of two files. Syntax:

        FileComparer <file_path_1> <file_path_2>
```

- If either `<file_path_1>` or `<file_path_2>` does not exist, print the following message:

```
        <file_path> does not exist.
```

- If either `<file_path_1>` or `<file_path_2>` exist but not a file, print the following message:

```
        <file_path> is not a file.
```

- If either `<file_path_1>` or `<file_path_2>` is not a text file, print the following message:

```
        <file_path> must be a text file.
```

Once the paths of two files are valid, the program compares two text files line by line, and produces the following report to the standard output:

- If two files are identical, print the following message:

```
        Files are identical.
```

- If two files are different, produce the following report:

```
        Differences in <file_path_1>:

                <line_number>: line content

                <line_number>: line content

                <line_number>: line content
                …

        Differences in <file_path_2>:

                <line_number>: line content

                <line_number>: line content

                <line_number>: line content

                …
```

For example, suppose there are two text files with the following content:

```
File1.txt

When I'm old and mankey,
I'll never use a hanky.
I'll wee on plants
and soil my pants
and sometimes get quite cranky.
```

```
File2.txt

When I'm old and mankey,
I'll never use some money.
I'll wee on plants
and soil my chances
and sometimes get quite cranky.
```

Then the program should produce the following report:

```
Differences in File1.txt:

        2: I'll never use a hanky

        4: and soil my pants

Differences in <File2.txt>:

        2: I'll never use some money

        3: and soil my chances

        <line_number>: line content
```

NOTE: the program compares only text files having less than 50 lines. If either file exceeds this number, the program prints the following message:

```
The file <file_path> is too long.
```

Program #54: Files Backup Tool

Develop a Files Backup Tool that copies a set of specified files to a given directory. The program must read settings from `settings.ini` file for the following information:

- `InputFile = ListFilesToBackup.txt`
- `OutputDir = F:\Backup`
- `DirectoryPrefix = Backup`
- `DateFormat = MM-dd-yyyy`

Following is the description of these settings:

- `InputFile`: specifies the path of a text file that contains a list of files which need to be backed up.
- `OutputDir`: specifies the path of the destination directory to which the files will be copied.
- `DirectoryPrefix`: all files will be copied to a directory whose name starts with this prefix. The program has to create this directory in the destination directory.
- `DateFormat`: specifies the format of the date which is concatenated to the directory prefix. The format follows the rules of date time format in Java which is described here.

For example, given the above settings, the program must copy files specified in the file `ListFilesToBackup.txt` to the following directory (suppose that current date is March 13th 2017):

```
F:\Backup\Backup_03_13_2017
```

The input file contains a list of absolute paths of files need to be backed up. Each file is on one line. For example, the file `ListFilesToBackup.txt` contains the following files:

```
D:\Java\Code\IOExamples.java

D:\Java\Tutorials\JavaIO.html

D:\Java\Videos\JavaIO.mp4

D:\Java\Projects\JavaIO.zip

...
```

Upon startup, the program must perform validation on the input that is read from the configuration file first.

If the settings.ini file doesn't exist, print the following message:

```
Could not find settings.ini file. Please supply one.
```

If this file does exist, then the program verifies each property in the following manner:

- If a property is missing, print the following message:

```
<property-name> is missing. The program couldn't start.
```

for example:

```
InputFile is missing. The program couldn't start.
```

- If a property has empty value, print the following message:

```
No value specified for <property-name>. The program couldn't start.
```

for example:

```
No value specified for DateFormat. The program couldn't start.
```

Next, verify `InputFile`: if the path doesn't exist, print the following message:

```
InputFile does not exist. The program couldn't start.
```

If `InputFile` does exist but is not a file, print the following message:

```
InputFile is not a file. The program couldn't start.
```

Next, verify `OutputDir`: If the path doesn't exist, print the following message:

```
OutputDir does not exist. The program couldn't start.
```

If `OutputDir` does exist, but is not a directory, print the following message:

```
OutputDir is not a directory. The program couldn't start.
```

Next, verify the date format by creating a new `SimpleDateFormat` object for the specified pattern. If an `IllegalArgumentException` is thrown, that means the date format is invalid.

Note that the program terminates after printing an error message.

Once all input is valid, the program will copy each file in the `InputFile` to the destination directory:

```
Destination Directory = OutputDir + DirectoryPrefix + Date
```

The program must create the destination directory if it doesn't exist. For example, with the given settings mentioned previously, and the current date is March 13th 2017, the destination will be:

```
F:\Backup\Backup_03_13_2017
```

For each file in the `InputFile`, the program performs the following validation before copying:

- If the file doesn't exist, print the following message and continue to the next one:

```
Skip one file that does not exist: <path>
```

- If the file does exist, but is not a file, print the following message and continue to the next one:

```
Skip one file that is not really a file: <path>
```

- If the file does exist and is actually a file, the program copies it to the destination directory, and print the following message:

```
Successfully backed up the file: <path>
```

The program continues proceed to the last file in the `InputFile`, and finally prints the following message:

```
Total <x> file(s) were backed up.

Total <y> file(s) are skipped.

Backup Completed.
```

Program #55: File Archiving Tool

Develop a Java program that allows the user to archive files of a specified format in a given directory to a compressed ZIP file. The user can run this program by typing the following command:

```
java Archiver <dir_path> <file_type> <zip_file_path> [-ks]
```

The parameters are described as follows:

- `<dir_path>`: the path of the directory where the program looks for files that need to be archived.
- `<file_type>`: specifies the extension of the file type to be archived. For example: `png`, `jpg`, `txt`, `mp3`, `mp4`, etc.
- `<zip_file_path>`: the path of the zip file that contains all the files of the specified type found in the given directory.
- `[ks]`: is an optional parameter. If specified, the program keeps the directory structure in the zip file as same as the source directory. If this parameter is omitted, all files will be put at the same level in the zip file.

Here's a couple of examples that show you how to run this program:

Example 1:

```
java Archiver D:\Photos png F:\MyPhotos.zip
```

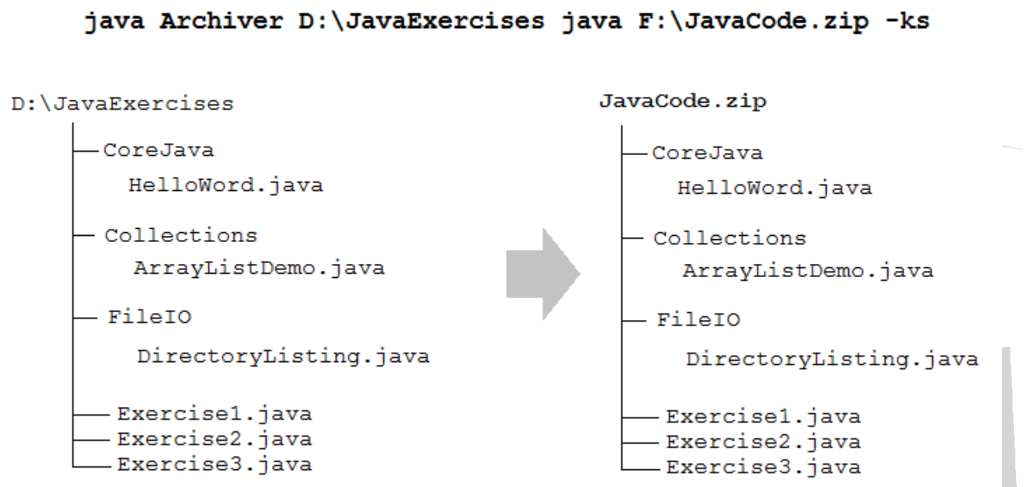
This tells the program to search for all `*.png` files in the directory `D:\Photos` recursively and compress the matching files to a zip file called `MyPhotos.zip` in the drive `F`.

Example 2:

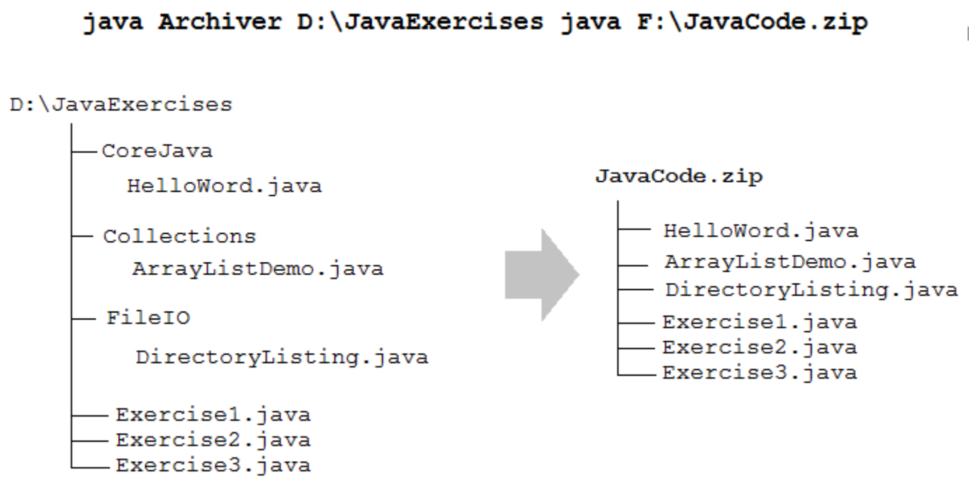
```
java Archiver D:\JavaExercises java F:\JavaCode.zip -ks
```

This tells the program to search for all `*.java` files in the directory `D:\JavaExercises` and add the matching files to a zip file named `JavaCode.zip` in the drive `F`. The `–ks` option tells the program to keep the directory structure of the source directory in the zip file.

The following picture depicts the output with the `–ks` option specified:



And the following picture explains the output without the option `–ks`:



But before searching and compressing the files, the program must verify the parameters to be correct, in the following manner:

- If the number of arguments is less than 3, print the syntax of the program.

- Verify the first argument (the source directory):

+ If it doesn't exist, print this message: `The source directory does not exist.`

+ If it does exist but is not a directory, print this message: `The source is not a directory. Please specify a directory.`

- Verify the 3rd argument (zip file path):

+ If it does exist, print this message: `The specified archive file already exists.`

+ If it doesn't end with .zip extension, print this message: `The archive file name must bend with .zip extension.`

+ If one of the parent directories of the zip file path doesn't exist, print this message: `A parent directory does not exist. Please specify a valid path name.`

- Verity the 4th argument: if it is present, it must be `–ks`, otherwise print this message: `Unknown option <argument_name>.`

Once all input is correct, the program searches the source directory recursively for files that match the specified extension. If no files are found, print the following message:

```
Could not find any files of type <file_type>. No file was archived.
```

For example:

```
Could not find any files of type .xls. No file was archived.
```

If the program found matching files, it compresses these files in a zip file specified by the `<zip_file_path>` argument. And depending on the presence of the `–ks` option, the program replicates the source directory structure n the zip file, or put all files in one place. And finally the program prints the result like this:

```
x files were archived in <zip_file_path>.
```

For example:

```
100 files were archived in MyPhotos.zip
```

In case of an I/O error occurs, the program prints the following message:

```
There was an error occurs <error_message>.
```

Program #56: Simple Quiz Program

Develop a simple quiz program that allows the user to take a number of quizzes which are chosen randomly from text files. The user can choose how many quizzes he or she wants to take. Each quiz is presented with a question and some possible answers. The user takes a quiz at a time, one after another. After the last quiz has been taken, the program shows the result with the score in percentage and number of correctly answered questions.

The program loads quiz data from a list of text files in the current directory:

```
quiz1.txt

quiz2.txt

quiz3.txt

…
```

Each text file contains information of a quiz in the following format:

```
Question content

----------

Possible answers

----------

Correct answer
```

There are 3 parts: question content, possible answers and correct answer, each is separated by hyphens (10 characters) in one line. For example, here's the content of `quiz1.txt` file:

```
What is the output of the following program?

      public class Quiz1 {


            public static void main(String[] args) {

                  byte x = 127;

                  byte y = 128;


                  Byte z = new Byte("255");
```

```
                              System.out.println((x + y) == z);


                        }

                }

        ----------

        Choose a correct answer:

           a) Print: true
           b) Print: false
           c) Runtime error
           d) Compile error

        ----------

           d) Compile error
```

There must be at least 3 quiz files. If not, the program prints the following message:

```
        Not enough quiz data.
```

In case there's no quiz file found, print the following message:

```
        Could not find quiz data.
```

Once the quiz data is loaded, the program asks the user to choose a number of quizzes he or she wishes to take:

```
        Choose number of quizzes (3 -> n): _
```

(`n` is the total number of quizzes).

The program must validate the user's input, which must be a number ranging from 3 to n. If not valid, the program asks the user to enter the number again.

Once the user enters the number of quizzes, the program shows the questions randomly, one at a time, in the following format:

```
        Quiz #x/n:

        <question content>

        <possible answers>

        Enter your answer: _
```

Where:

- `x` is the quiz number.

- `n` is the total of quizzes chosen by the user.

- `<question_content>`: is the question content read from a quiz file.

- `<possible answers>`: is the possible answers read from a quiz file. Each question has 4 possible answers.

The program must check to answer is within the possible answers (a, b, c and d). If not, ask the user to enter the answer again.

Then the program records the answer, and proceeds to the next quiz, until the last quiz is encountered. And after the user answered the last quiz, the program prints the result in the following format:

```
Your Score: <percentage>

You have x correct answers out of n.
```

For example:

```
Your Score: 80%

You have 8 correct answers out of 10.
```

Then the program terminates.

Program #57: Simple Text File Creator

Develop a Java program that allows the user to create a simple text file in command line mode (something like `vi` on Linux and `copy con` statement in Windows, but even simpler). The user can create a text file by typing the following command:

```
java Texter <file_name>
```

For example:

```
java Texter Notes.txt
```

This will create a text file named `Notes.txt` in the current directory. The program allows the user to enter any characters in the command line (press `Enter` key to create a new line). And to save the file, the last line must contain only the text “save”. For example:

```
The quick brown fox jumps over the lazy dog
The quick brown fox jumps over the lazy dog

The quick brown fox jumps over the lazy dog

save
```

Then the program creates the `Notes.txt` file with content is whatever the user types in, except the word "save" serves as a command in the last line.

Program #58: Country Lookup

Develop a program that allows the user to search for information of specified country in the world. The database of country information is stored in a text file called `worlddb.txt`, with each country is described on one line in the following format:

```
country name, capital city, continent, population, GDP
```

Each country is described by 5 fields separated by commas. Here's an example:

```
United States, Washington D.C., America, 322 million, 55836 USD

Japan, Tokyo, Asia, 127 million, 32477 USD

Germany, Berlin, Europe, 82 million, 47000 USD

Australia, Canberra, Oceania, 24 million, 48800 USD

Egypt, Cairo, Africa, 92 million, 12560 USD

…
```

The user can run this program by typing the following command:

```
java WorldSearch
```

Upon startup, the program loads the database in the `worlddb.txt` file. If this file doesn't exist, print the following message:

```
Could not find the database.
```

Once the database is loaded, the program prints the following message:

```
Welcome to World Database! (type exit to quit).

Enter country name: _
```

The program must validate the user's input. e.g. the country name cannot be blank. The program asks the user to enter country name again if it is blank.

Once the input is valid, the program searches for exact match in the database. If the specified country is found in the database, print the result in the following format:

```
Country: <country_name>

Capital: <capital_city>

Population: <population>

GDP: <GDP>
```

For example:

```
Country: Egypt

Capital: Cairo

Population: 92 million

GDP: 12560 USD
```

If the specified country is not found in the database, print the following message:

```
Could not find information for <country_name>
```

In both cases, the program continues asking the user to search for another country:

```
Enter country name: _
```

If the user types `exit`, the program terminates.