DUE DATE: **By the end of your lab session**

CSE-271: Object-Oriented Programming

Exercise #2

Max Points: 20

Name: Raobot

For your own convenient reference – You should first save/rename this document using the naming convention `MUid_Exercise2.docx` (example: `raodm_Exercise2.docx`) prior to proceeding with this exercise.

Objectives: The objectives of this exercise are to:

1. Understand the concept of exceptions
2. Explore the use of `throw` and `throws` keywords to generate exceptions
3. Explore the use of `try...catch...finally` blocks to handle exceptions
4. Trace the behavior of exceptions using a debugger
5. Practice the procedure for style checking and submitting programs

Fill in answers to all of the questions. For some of the questions you can simply copy-paste appropriate text from Eclipse output into this document. You may discuss the questions or seek help from your neighbor, TA, and/or your instructor.

Part #0: One time setup of Eclipse (IDE) – Only if needed

We already configured `Eclipse`'s source formatter and Checkstyle plug-in as part of Lab #1. If your `Eclipse` is not configured (because you are using a different computer) then use the instructions from Lab #1 to configure `Eclipse`.

Part #1: Basics of exceptions in Java

Estimate time: 20 minutes

Background: Clearly and concisely explaining concepts is a very important skills for your future job-interviews and in your jobs. Hence, the exams also involve such questions and in the labs we will practice this style of questions.

Exercise: Briefly (2-to-3 sentences each) respond to the following questions regarding the basic concepts of exceptions in Java.

1. What is an exception in the context of Java programming language?

   Exceptions are serious errors generated/reported by a Java program. The information about the error is contained in a suitable object. The unique property of exceptions is that they automatically propagate, aborting method calls (unwinding the call stack) until the exception is handled or the program is aborted.

2. Using a 1-line Java source code, illustrate how exceptions are generated in Java using the `throw` keyword.

In Java, exceptions are generated by throwing a new exception object as shown below:

```
throw new IllegalArgumentException("Not interested in arguments");
```

3. Briefly describe (English description is sufficient, but you can provide a short example source code too) how exceptions can be handled (or caught) and processed.

In Java, exceptions thrown by a method can be caught and processed using a `try…catch` block. Optionally, a `finally` block can be used to perform some clean-up immaterial of whether an exception is thrown by a method.

```
try {
  processFile("words.txt");
} catch (IOException ie) {
  System.out.println(ie);
} finally {
  System.out.println("Done");
}
```

4. What is the purpose of a `finally` block (used with a `try` statement) in Java?

The code in a `finally` block is always executed (after a `catch` block, if any) immaterial of whether an exception is thrown or not. This is useful to close any streams that may have been opened before or in the `try` block. A `try…finally` block can be simplified to a try-with-resource construct in Java.

5. In Java, what is the difference between a checked and unchecked exception in Java?

There are two types of exception objects that are supported in Java, namely:

1. *Unchecked exceptions*: These exceptions typically arise due to logic/programming errors. They don't need to be caught or reported. These exceptions are descendants of `RuntimeException` class in the JDK.
2. *Checked exceptions*: These exceptions are thrown due to user or I/O errors. These issues can be resolved by a user by retrying operations or correcting inputs. These exceptions must be caught by a method or it must report the exception(s) in its `throws` clause in its method signature. These exceptions are descendants of `Throwable` or the `Exception` class in the JDK.

3. Referring to the Javadoc (https://docs.oracle.com/javase/8/docs/api/), indicate if the following exception classes are checked or unchecked exception classes.
   - **You expected to memorize**: `RuntimeException` and its descendants are unchecked exceptions. All other `Throwable` and `Exception` classes and their descendants are checked exceptions and must be included in the throws clause of a method.

Exception class name	Checked/Unchecked?
a. `FileNotFoundException`	Checked
b. `ArrayIndexOutOfBoundsException`	Unchecked
c. `ClassCastException`	Unchecked
d. `CloneNotSupportedException`	Checked
e. `OutOfMemoryError`	Unchecked

6. Complete the signature of the methods shown below to include the `throws` clause with appropriate final exception types:

```
public void process(String s)
throws PrintException
{
  // Some java code goes here.
  throw new PrintException();
}
```

```
public void process(int i)
throws ParseException, PrintException
{
  // Some java code goes here.
  switch(i) {
    case 0: throw new ParseException();
    case 1: throw new PrintException();
  }
}
```

7. Write the Java code for a void method named `processInt` that accepts an `int` as a parameter and prints it, if the integer is not zero. If the parameter is zero, then it should throw an `InvalidPropertiesFormatException`, with the message `"Parameter was zero!"`. Don't forget the `throws` clause for the method.

```
public void processInt(int i) throws InvalidPropertiesFormatException {
  if (i == 0) {
```

```
        throw new InvalidPropertiesFormatException("Parameter was zero!");
    }
    System.out.println(i);
}
```

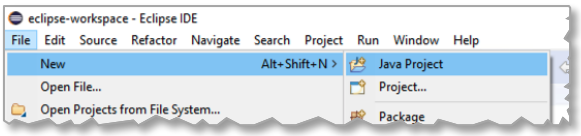
Part #2: Tracing exceptions via the debugger

Estimated time: < 20 minutes

Background: The best way to build a strong mental model of exception propagation is to observe (or trace) the operations using the debugger. Debugger is an invaluable tool to build a strong mental model, immaterial of the programming language you use.

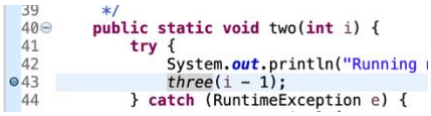
Exercise: Trace the operation of exception propagation using the following procedure:

1. Launch Eclipse (IDE) using the default workspace.
2. Next, from Eclipse's main menu use the option File → New → Java project to create a new Java project titled `lab2`. Don't create any modules.



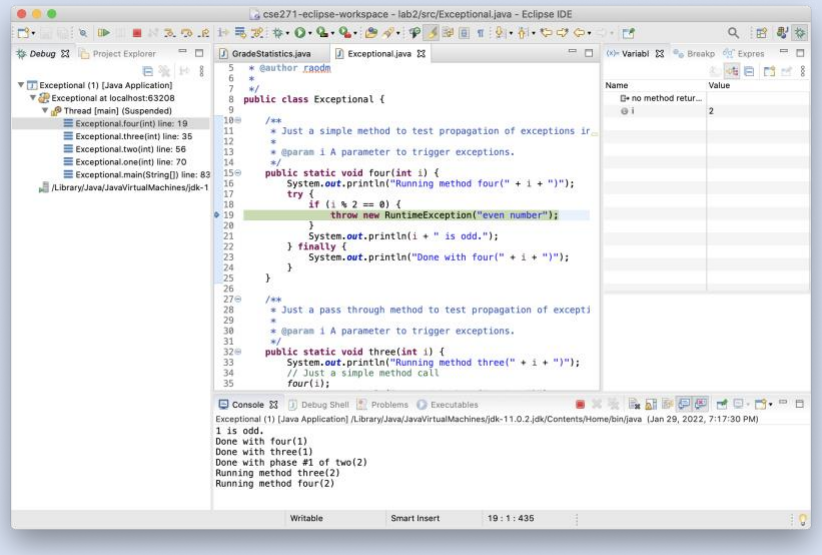
3. From Canvas, download the supplied starter code for this lab directly to the directory of your newly created Java project. Refresh your project after you have downloaded the files so that it shown in Eclipse.

4. In order to use the debugger, you first need to set a breakpoint. Set a breakpoint in method `two`, on the line of code where `three(i - 1)` method is called as shown in the adjacent figure.



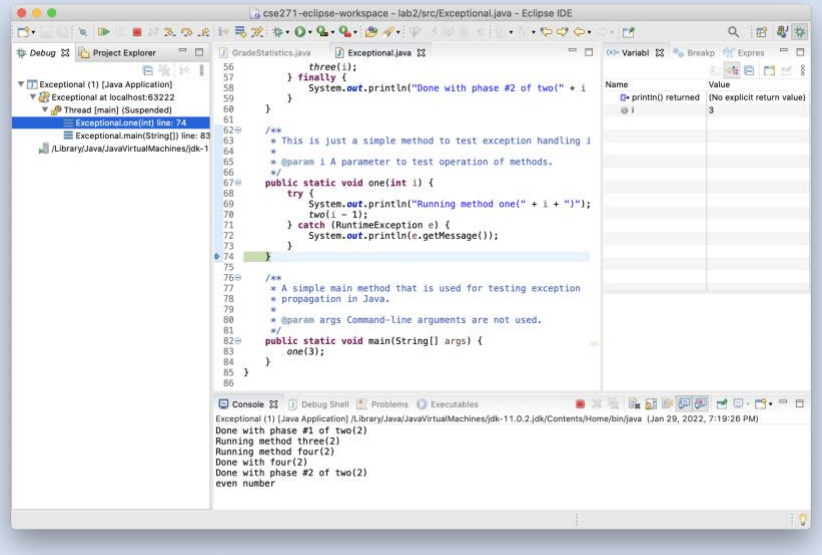
5. Run the program via the debugger. The debugger will stop at the breakpoint you have set. When prompted switch over to the debug perspective.
6. Step into method calls for `three` and `four` to the line code just before an exception is thrown at the line "`throw new RuntimeException("even number");`". Now, make a screenshot of the entire Eclipse window (ensure the call stack is visible) and place it in the space below:

Place screenshot of debugger here:



7. Next, continue stepping through the code until the call stack unwinds all the way to the `main` method. As you step through the code make a mental model of how the stack trace is unwinding.
8. Once the stack trace unwinds to the end of `one` method make another screenshot of your Eclipse window and place it in the space below:

Place screenshot of debugger here:



9. Using information from lecture and experience from the debugger tracing you just did, briefly (2-to-3 sentences) state the rules that Java uses to propagate an exception as the call stack to unwinds.

Java keeps propagating an exception, unwinding the call stack based on the following rules:

1. For the current method call (at the top of the call stack) –
    a. If the current method is surrounded by a `try…catch` block that handles the exception, then exception propagation ends and the statements in the `catch` block start executing.
    b. If not, the `finally` block, if any, surrounding the method call is first executed.
        i. If the current method is the main method, then the program is aborted and exception stack trace is printed.
        ii. Otherwise, then the current method is aborted. The exception is propagated to the calling method and the process repeats from step 1 (above).

Part #3: Manually tracing programs involving exceptions

Estimated time: < 30 minutes

1. Given the following implementation for a `mystery` method, illustrate the output generated by the calls to the `mystery` method shown below:

```
public static final String Stars = "********************";

public static void other(int n) throws Exception {
    if (n % 2 == 0) {
        throw new Exception(n + " is even.");
    }
}

public static void mystery(int n) throws Exception {
    if (n < 0) {
        throw new Exception("negative!");
    }
    try {
        other(n - 1);
        System.out.println(Stars.substring(0, n));
    } catch (Exception e) {
        System.out.println(e.getMessage());
    } finally {
        System.out.println("Done.");
    }
}
```

    a. What is the output generated by the method call: `mystery(4)`?

```
****
Done.
```

b. What is the output generated by the method call: `mystery(3)`?

```
2 is even.
Done.
```

c. What is the output generated by the method call: `mystery(-2)`?

```
Exception("negative!")
```

2. Given the following implementation for `method1` and `helper` methods, illustrate the output generated by the following calls to `method1`. Assume `Exp1`, `Exp2`, `Exp3`, and `Exp4` are checked exception classes.

```
public static void method1(int i) throws
Exp4 {
  try {
    helper(i);
    System.out.println("method1 OK");
  } catch (Exp1 e1) {
    System.out.println("method1 error");
  }
}

public static void helper(int i) throws
Exp1, Exp4 {
  try {
    switch(i) {
    case 0: throw new Exp1();
    case 1: throw new Exp4();
    case 2: throw new Exp2();
    case 3: System.out.println("helper
OK");
    }
  } catch (Exp2 e2) {
    System.out.println("helper error");
  } finally {
    System.out.println("helper done");
  }
}
```

What is the output from `method1(3)`?

```
helper OK
helper done
method1 OK`
```

What is the output from `method1(2)`?

```
helper error
helper done
method1 OK
```

What is the output from `method1(0)`?

```
helper done
mehtod1 error
```

Part #4: Develop a simple Java program

Estimated time: < 20 minutes

Background: Handling exceptions in a Java program can be used for developing a "trial-and-error" approach for problem-solving. In this "trial-and-error" approach, a program can attempt a simple operation. If the operation generates an exception (i.e., it fails) then we can infer certain properties. For example, the `Integer.parseInt` method can be used to convert a string to an integer. This method throws a `NumberFormatException` if the parameter is not an integer. Using the presence or absence of an exception, we can detect if an input String is an integer.

Exercise: In this part of the exercise, you are expected to develop a simple Java program to detect if a word (entered by the user) is an `int`, `double`, or `String`. Develop the program in the following manner:

1. You can reuse your current Eclipse project or create a new Java project in Eclipse.
2. Create a new class called `Numero` with a `main` method. You will be developing the `main` method in the following manner:
    a. Prompt the user to enter a word via `System.out.println("Enter a word or a number: ")`
    b. Read a `word` from the user from `System.in` using a suitable `Scanner` object.
    c. Next use two `try...catch` blocks to detect the input `word`'s type. First, use `Integer.parseInt` to convert the word to `int`. If that succeeds the input word is an integer. If not, use `Double.parseDouble` method to try and convert the word to double. If that succeeds the word is a `double`. Otherwise assume the type of `String`.
    d. Print the word and its type via: `System.out.println(word + " is a " + type);`

Sample outputs:
Sample outputs from independent runs of the completed program are shown below. User inputs are in **green** color.

```
Enter a word or a number:
123
123 is a int
```

```
Enter a word or a number:
12.3
12.3 is a double
```

```
Enter a word or a number:
12,3
12,3 is a String
```

Part #5: Submit to Canvas via CODE plug-in

Estimated time: < 5 minutes

Exercise: You will be submitting the following files via the Canvas CODE plug-in:

DUE DATE: **By the end of your lab session**

1. This MS-Word document saved as a PDF file – **Do not upload Word documents. Only submit PDF file**.
2. The Java program (`Numero`) that you developed in this exercise

Ensure you actually complete the submission on Canvas.