**lab-6: ExceptionsLab.java** instructions:

(a) For this problem, you will explore Java exceptions and how to handle them. Do the following steps, answering the questions marked **\*\*\*\*\***. Submit a copy of this handout with the answers inserted at these marked places, along with the final version of the code changes requested here.

Do each of the following steps – downloading this Word doc and adding your answers after the questions marked \*\*\*\*\*:

(b) Run the **Lab6** application and note its output: the JVM calls **main()**, which calls **method1()**, which calls **method2()**. When done, **method2()** returns to **method1()**, which in turn returns to **main()**, which finally ends and returns to the JVM, ending the app's execution (=> all of its executing threads 'die' – more later on threads).

We now will force an exception to be thrown: do so by editing **method2()** and changing "**1/1**" to "**1/0**" (**int** division by 0). Save all changes, and rerun. **\*\*\*\*\* What do you see in the output window (Console)?**Exception in thread "main" Before int division in method2().

java.lang.ArithmeticException: / by zero

at ExceptionsLab.method2(ExceptionsLab.java:46)

at ExceptionsLab.method1(ExceptionsLab.java:35)

at ExceptionsLab.main(ExceptionsLab.java:28)

An exception is shown.

When an exception occurs, it is thrown by the virtual machine. We'll see later how to do this explicitly in our code; for now, note that the attempt to do **int** divide-by-zero is detected by the JVM, which then creates an instance of an Exception class, followed by the JVM throwing it.

"Throw" means to signal the Java environment that normal sequential execution of code has been suspended, replaced by the exception handling and propagation mechanisms described here.

(c) All Java exceptions are instances of some exception class that is either an instance of **java.lang.Exception**, or an instance of some subclass (direct or indirect) thereof. Look up the Sun Javadoc for **Exception** (online at the Java/Oracle site).

When you are viewing the javadoc page for **Exception**, note the inheritance hierarchy of **Exception**, as well as its constructors and instance methods.

\*\*\*\*\* **What is the immediate superclass of Exception? What are the instance methods of Exception?**

**Immediate superclass is** [Throwable](https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html),instance methods of Exception are, Exception(), Exception(String message), Exception(String message, Throwable cause), Exception(String message, Throwable cause, Boolean enableSuppression, Boolean writableStackTrace) and Exception(Throwable cause)

(d) In step (b), the Console displayed the default error message that occurred when the **1/0** exception was thrown – along with the stack trace of the runtime environment when the exception occurred.

The stack trace is displayed, showing what method **m1** was executing when the exception was thrown. Below, it shows which method **m2** was waiting for a return from its call to **m1**, and so forth. This sequence of calls and pending returns is called the call chain, and starts with **main()** for Java applications. That is, the reported exception occurred with a call chain of **m1 <- m2 <- m3 <- … <- mk <-main()**.

Here, the JVM sent the stack trace to the console, which is part of its default handling of the thrown exception. We may also catch the exception in our code, so that it never propagates to the JVM – yet we can still print out the exception if we wish.

**\*\*\*\*\* What method in Exception prints out the stack trace to some location?** Hint: look in the superclass. Remember that a class inherits all methods from its superclass.

printStackTrace prints out the stacktrace

(e) In order to be able to call **Exception** methods, we need access to the **Exception** object that was originally thrown. To do this, we must catch the thrown exception – which we'll do in the next step or two.

For now, note that when the exception is thrown, it immediately skips over the rest of the following method statements, and "goes-to" to the calling method. (Think of the thrown **Exception** as resulting in an "object-oriented" go-to action.)

When control is returning to the calling method, the JVM checks if it that method might handle the exception itself (via exception handlers – later on this). If not (the case here), it then jumps immediately to its calling method, and so forth – back up the call chain.

If this control eventually reaches the JVM (we say the thrown exception propagates back to the JVM) then a default error message is sent to the Console, resulting in the exception type + other info + stack trace being displayed.

Now, change the **int** division in **method2()** back to **1/1**, and change the later indexing to **STOOGES[47]**. Run the app with the changed code.

**\*\*\*\*\* Which exception type is now thrown?**

Exception in thread "main" After int division in method2() .

Before array index in method2().

java.lang.ArrayIndexOutOfBoundsException: 47

at ExceptionsLab.method2(ExceptionsLab.java:50)

at ExceptionsLab.method1(ExceptionsLab.java:35)

at ExceptionsLab.main(ExceptionsLab.java:28)

Now change **1/1** back to **1/0**. Run the app again.

**\*\*\*\*\* What happens to the indexing exception? Explain.**

The indexing exception is not thrown because the arithmetic exception is found first and the exception skips the rest of the method and goes immediately to the calling method and travels back up to the JVM and the default error message is sent to console, skipping the array exception.

(f) Now we see how to catch and handle a thrown exception: change the call in **method1()** so that it calls **method3()** – instead of **method2()**. Run the app – it should run with no exceptions (though the output still says that **method1()** calls **method2()** – you can change this if you wish).

Now, in **method3()**, change **1/1** to **1/0**. Rerun the app.

**\*\*\*\*\* What do you see? Is the call stack displayed?**

ArithmeticException in method3(): java.lang.ArithmeticException: / by zero

Continuing...

The call stack is not displayed

The **try**-**catch** block in **method3()** caught the **1/0**-generated exception. When the exception was thrown, control immediately passed to the JVM, which noted the offending code was "wrapped" in a **try** block (delimited with **{...}**). It then tried to match the actual type of the thrown exception with the type of the exception listed in the **catch *(<eType>* aEx)** header.

Since the first **catch** header had ***<eType>*** == **ArithmeticException**, and the thrown exception was also of this type, there's a match. This triggers the JVM to call this **catch** handler: a reference to the thrown exception object is copied into the **aEx** reference variable (which is like a specially-initialized local variable), and the **catch** block delimited by **{…}** is run. This block may then directly access the thrown exception object through **aEx** (as it does via **aEx.toString()**).

This is analogous to parameter passing when an ordinary method is called – but here, we do not explicitly call the **catch** handler: the JVM does this for us, and only if there is a thrown exception.

After catching the thrown exception, and executing the corresponding **catch** handler, code execution continues as before – jumping over any other **catch** blocks, then continuing to execute the statements after the entire **try-catch** code.

Now try switching the order of the two **catch** header + handlers, so that the handler for **ArithmeticException** is the second one after the **try** block. Rerun the app.

**\*\*\*\*\* What happens? Is there any change? Explain why.**

**No, there is no change because the 1\0 is still before the array exception, so once the method sees 1\0, it checks all the catch headers + handlers and still finds the ArithmeticException, executes the catch handler and jumps over the rest of the catch blocks.**

Change both the **int** division and array indexing statements, so neither throws any exceptions.

(g) In the current app, the compiler does not force us to add a **try-catch** block around the code – but if we do, it uses it as described before. But sometimes the compiler does force handling – depending on the type of the exception that might be thrown by the executed code.

To explore this forced use of a **try-catch**, remove the comments around the **Thread.sleep(1000)** statement at the end of **method3()**.

**\*\*\*\*\* What happens? What does the compiler report? What exception is flagged as 'not handled'?**

Exception in thread "main" Before call to method2() from within method1().

java.lang.Error: Unresolved compilation problem:

Unhandled exception type InterruptedException, The interruptedException is “not handled”

Java divides exceptions into two kinds: those for which **try-catch** handling is mandatory, and those for which it is optional. The **Thread.sleep(1000)** can throw the former kind – and hence the compiler forces us to add a **try-catch** block… or else requires us to add additional code that forces the calling method to handle this exception, instead.

Java distinguishes between these two kinds, via the class type of the exception and its inheritance hierarchy. That is, non-mandatory exceptions all share a common superclass, which then signals the compiler to relax its handler checks. If a possible thrown exception does not have this class as an 'ancestor', then the compiler forces you to handle it.

**\*\*\*\*\* Name this 'non-mandatory handling' superclass.** RuntimeException is the superclassHint: it is mentioned in a comment somewhere in **method3()**. Also look up the javadoc for **Thread.sleep(…)** and see what exception it may throw – then look up this exception, and verify that it not a subclass of the former. Finally, list the thrown exception's inheritance hierarchy.

(h) Continuing with the previous step's explorations: in order to compile the **Thread.sleep(1000)** code, we may indicate that its **method3()** will not handle it internally, but instead expect any **caller** of **method3()** to provide the exception handling, instead. (We say that **method3()** thus delegates the handling of such exceptions to its caller, instead of doing it itself.)

Uncomment the header code in **method3()**'s header, which indicates this delegation. Save the resulting code.

**\*\*\*\*\* What error is now reported? Briefly explain why, and how you might fix this problem. Do so** (hint: you may need to change both **method1()** and **main()**).

Thus, **throws Etype1** can be added to the end of a method's header, to signal the compiler re this alternative handling. You can have multiple such exceptions listed:

**throws InterruptedException, AnotherException {…**

(i) Now uncomment the **throw…** statement within **method3()** – this creates and throws a new exception. Run the resulting program.

**\*\*\*\*\* What happens? Why wasn't this thrown exception caught?**

If a thrown exception does not match any of the associated **catch** headers, it is then propagated ("rethrown") back to the method's caller. This is true for any uncaught exception within some method – and this "handle or propagate" is repeated, back up the call chain.

(j) Return to **method1()**, and replace the call to **method3()** with one to **method4()** (you can comment/uncomment existing code). Save, and rerun the app – it should run to completion with no exceptions.

**method4()** contains two changes to **method3()**. One shows the ability to return from a **catch** handler via the **return** statement (here with no arguments, since **method4()** has **void** return type). Now change **method4()** so that this return is actually made.

**\*\*\*\*\* Briefly describe your changes, and the resulting differences in the output.**

(k) The other change that **method4()** makes involves output within a **catch** handler, illustrating another **Exception** instance method (actually a **Throwable** method).

**\*\*\*\*\* What is the name of this method? Look it up in the javadoc, and briefly describe what it does.**

There are actually multiple overloaded versions of this method – the others send the information to a non-console location.

(l) Finally, change the call to **method4()** within **method1()** to **method5()**, and rerun the app. Note the new output, created by the new **finally** block, added at the end of the **try-catch** block.

Experiment with **method5()**, making modifications that: (i) throw an **ArithmeticException**; (ii) throw an **ArrayIndexOutOfBoundsException**; (iii) throw a **RuntimeException** inside some **catch** handler.

**\*\*\*\*\* State what the finally block seems to be doing, in all of these cases.**

**\*\*\*\*\* Extra credit: there are a few situations where the previous finally block behavior doesn't occur. Research these situations, and describe one or more of them.**

(m) Submit the copy of this document with your answers inserted after the above \*\*\*\*\* questions. Also submit your changed code, along with the other problems in this Lab.