Lab 8: Exception Handling

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Background

Summary:

During the execution of a program, things can, and do go wrong. The user may enter inappropriate data, a device may go offline, necessary disk space for output may not be available, etc. Errors occurring at runtime are known as exceptions and exception handling is the process designed to handle exceptions. Failure to properly handle an exception can cause a program to abort and/or expose sensitive information about the program and the environment in which the program is executing.

Description:

When an exception occurs, the Java runtime creates an exception object containing information about the error and the program state at the time of the error. There are numerous (well over 50!) exception classes defined in the Java API.

Once the exception object has been constructed, it is handed to the Java runtime system. This is called ‘throwing’ an exception. The Java runtime system must then locate an exception handler. In searching for an exception handler, the runtime system begins with the method in which the exception occurred. If this method is not prepared to handle the exception, the runtime system will search the call stack in reverse in search of a suitable exception handler. For example, if methodA calls methodB which calls methodC and an exception occurs during the execution of methodC, the runtime system will search methodC first, then methodB, and lastly methodA for an exception handler. When an appropriate exception handler is found, the runtime system turns over the exception object to the handler. The selected exception handler is said to ‘catch’ the exception.

Risk – How Can It Happen?

Some exceptions are known as checked exceptions and exception handling, or catching checked exceptions, is required to be included in the program. Other exceptions, unchecked exceptions, do not require an exception handler. Most unchecked exceptions, however, cause the program to display an error message (which may be a security issue since it exposes the inner details of your application) and stop execution when not handled. Exception handling provides a way for the program to possibly recover from an error or at least terminate in a graceful manner.

*code responsibly – how to handle exceptions?*

1. Know your exceptions: Every method in the Java API lists any exceptions that might be thrown. Checked exceptions must be handled. Although unchecked exceptions do not require exception handling code, consider handling them or providing code checks such that the exception can never occur.
2. Order exception handlers correctly: When more than one exception handler is desired, code them in the order that they are more likely to occur. When a generic exception handler is also used, it should appear last.
3. Handle exceptions appropriately: Once an exception occurs, the remaining statements of the try block will not be executed. Therefore, it is important that the exception handler recover from the exception such that the program can continue. If recovery in not possible, exit the program.
4. An exception should not expose sensitive information: Some exception objects such as FileNotFoundException, define messages that expose information about the state of the program and/or the file system. Displaying the stack trace from the exception object may also leak information. Therefore, do not utilize the exception object’s printStackTrace() method and do not display the result of the exception object’s getMessage() method.
5. RuntimeException exceptions are thrown due to bugs in the program. These bugs should be corrected during development.

Lab Assignment

Program 1

import java.util.Scanner;

public class Exception1 {

public static void main(String[] args) {

Scanner scan = new Scanner(System.in);

int num = 0;

do {

  System.out.println("Enter a number between 1 and 10");

num = scan.nextInt();

if (num < 1 || num > 10)

System.out.println("\nIllegal value, " + num + " entered. Please try again.");

} while (num < 1 || num > 10);

System.out.println("\nValue correctly entered! Thank you.");

}

}

**Lab Questions:**

* 1. Type\* the program above and compile. Run and enter an integer between 1 and 10.
  2. The program is requesting a number between 1 and 10. Run the program again and enter 5.5.  Although this number is between 1 and 10, the program will abort.  Examine the error message.  You should see the word Exception, the method where the exception occurred (main), the class name of the exception (InputMismatchException), as well as the call stack listing the method calls.
  3. Add a try/catch block to catch and handle the InputMismatchException exception. Identify the statements that cause the error as well as the portions of the program that depend upon these statements.  Enclose these statements within the try block.  Follow the try block with the catch block given below.  Note, the InputMismatchException class is defined in java.util and must be imported.  Also, when the Scanner throws an InputMismatchException, the input token will remain in the buffer so that it can be examined by the program.  In our case, we will not be examining the token, but will simply clear out of the buffer to start over.

catch (InputMismatchException ime) {

   System.out.println("Enter whole numbers only, with no spaces or other characters");

   scan.next();        // clear the scanner buffer

}

4. Compile and run the program again, testing with a variety of input (integers, floats, characters) The program should not abort when floats or character data is given.

5. Complete the security checklist for Program 1.

*\*Copying and pasting programs may result in syntax errors and other inconsistencies. It is recommended you type each program*.

Program 2

import java.util.Scanner;

import java.io.File;

public class Exception2 {

public static void main(String[] args) {

int total = 0;

int num = 0;

File myFile = null;

Scanner inputFile = null;

myFile = new File("inFile.txt");

inputFile = new Scanner(myFile);

while (inputFile.hasNext()) {

num = inputFile.nextInt();

total += num;

}

System.out.println("The total value is " + total);

}

}

**Lab Questions:**

1. As we saw in Lab Program 1, the Scanner may throw an *InputMismatchException*. This is an unchecked exception and we are not required to handle unchecked exceptions. The Java runtime system will handle the exception by aborting the program and displaying the exception information. Other methods may throw checked exceptions. A checked exception must be handled. Type\* the program above and compile. You should receive a compilation error stating that the *FileNotFoundException* must be caught or declared to be thrown.
2. We will first test the program knowing that the input file does not exist. Add a try/catch block such that if the input file is not found, report the problem and display a total of 0. Use the getMessage() method of the exception object to report the problem.
3. } catch (FileNotFoundException fnf)
4. {

//display total of 0

System.out.println(fnf.getMessage())

}

1. Create a file named inFile.txt using a plain text editor. Enter only integer values into the file and save the file in the same folder as the program (if you are using a IDE, then put it in the main project folder). Run the program to verify that the program functions correctly with a valid input file.
2. Any number of catch clauses may be coded for a single try block. Add another catch clause to catch *InputMismatchException* immediately before or after the *FileNotFoundException* clause. Remember to import the exception class and to clear the scanner buffer. Modify the inFile.txt file to include an error. Run and test the program. As soon as an exception occurs, the catch clause is executed. Since the *InputMismatchException* catch clause is outside of the read loop, the first invalid piece of data will cause the program to end without reading the remainder of the file.
3. Any number of try blocks may be coded with each having a least one catch clause. Rather than catching the *InputMismatchException* outside the loop, let’s move it inside the loop. This will allow our program to report the invalid data, resume processing, and produce correct results for the valid data given. Add a new try block within the while loop and move the *InputMismatchException* catch block such that the while loop is as shown below.
4. while (inputFile.hasNext()) {
5. try {
6. num = inputFile.nextInt();
7. total += num;
8. } catch (InputMismatchException ime) {
9. System.out.print("Illegal value found");
10. inputFile.next();
11. }
12. } // end while
13. Complete the security checklist for Program 2.

*\*Copying and pasting programs may result in syntax errors and other inconsistencies. It is recommended you type each program*.

Security Checklist

|  |  |
| --- | --- |
| **Security Checklist** | |
| **Vulnerability:** *Exception* |  |
| **Check each line of code** | **Line number** |
| *1.*line number of statement that may cause an exception to occur. |  |
| *2.*line number of any statement in the exception handler that may expose information about the program. |  |
| *3.*line number of any statement in the exception handler that may expose information about the file system. |  |
| **Highlighted areas indicate vulnerabilities!** |  |

Discussion Questions

[top](http://cis1.towson.edu/~cssecinj/modules/other-modules/build-the-lab/build-the-lab-lab-module-2/#top)

1. With keyboard input, what types of errors may occur?
2. An *InputMismatchException* is an unchecked exception. Since exception handling of an unchecked exception in not required, why should the programmer add this checking?
3. What types of exceptions may occur when working with a file? Hint, visit the Java API and select the java.io library. Review the list of exceptions included in this library.
4. Describe other types of exceptions (non-IO related) that may occur at runtime. Hint: view the Exceptions category of various libraries such as java.lang and java.util in the Java API.
5. If a method throws a checked exception, how can the calling method avoid coding a try/catch block to handle the exception? In what cases might the calling method use this option?
6. Exception classes are derived from other exception classes. For example, the *FileNotFoundException* class is derived from *IOException* class. And, the *IOException* class is derived from the *Exception* class. Rather than check for the exact exception class, the catch clause could specify a super class. What are the advantages and disadvantages of specifying a super class in a catch clause?