Object Oriented Programming – Java 1

Lecture 11

Exception Handling

Dr. Obuhuma James

Description

This topic covers exception handling as a concept applicable is a number of object-oriented programming languages. The reasons behind the need for exception handling are discussed. Mechanisms for handling exceptions are then demonstrated using appropriate examples. Exception handling in programming prevents programs from crushing following occurrence of exceptions.

Learning Outcomes

By the end of this topic, you will be able to:

- Describe the reasons behind the need for exception handling in programming.
- Discuss the various types of exception errors likely to occur in a given program.
- Demonstrate the process of handling exceptions in programming.

Overview of Exceptions and Exception Handling

An exception is an unexpected or error conditions that occurs at a program's runtime [1]. Exceptions are unusual occurrences that cause the program to crush instantly when they are in an active running state. Some of the most common causes of exceptions include [1]:

1. A program issuing a call to a computer file that does not exist.
2. A program attempting to write to a full computer disk.
3. A user entering invalid data following a prompt at runtime.
4. A program attempting to divide a value by a zero (0).

Many other types of exceptions may also occur during the lifetime of a program. These may include instances where an array index is out of bound, and when a given element is not found, among others [1, 2].

Exceptions must be handled appropriately to avoid unnecessary crushing of the program at runtime. One of the crude approaches for handling exceptions is through the use of the if … else statement. A more standard approach for exception handling offered by the object-oriented programming paradigm is the try … catch … finally blocks that try codes for existence of exceptions then catch any detected exceptions.

Exceptions in Programming

An exception is an occurrence of an undesired situation detected during program execution [2]. Upon occurrence of an exception, an object of the specific exception class is created. Exceptions descend from the Throwable class. Thus Figure 1 shows the exception and error class inheritance hierarchy for the Java programming language.

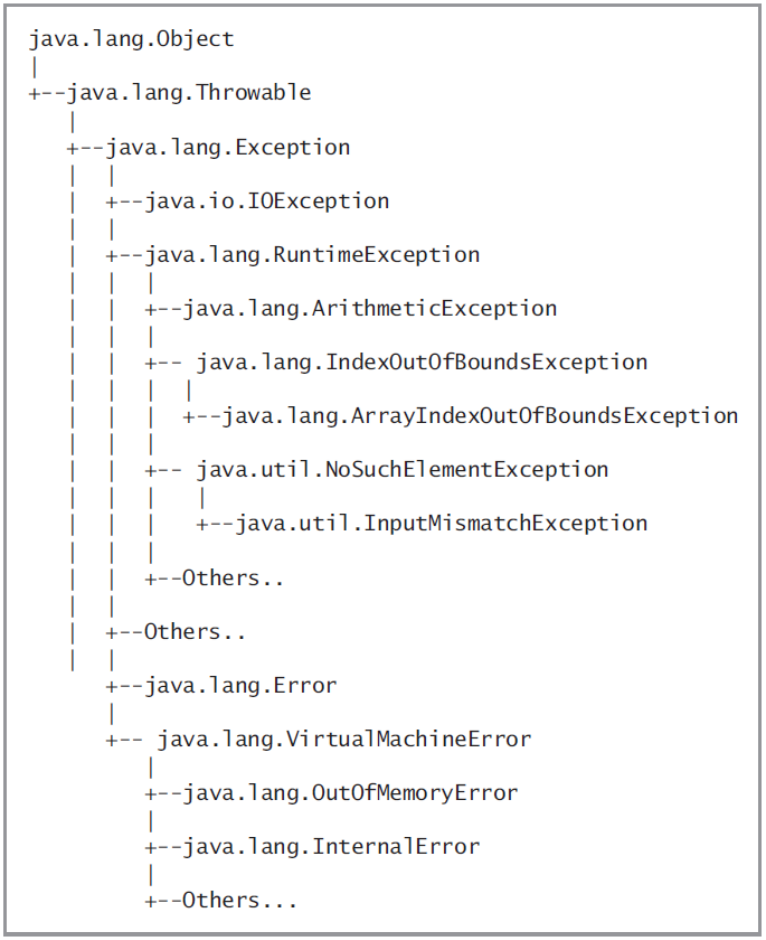

Figure 1. Java Exception and Error Class Inheritance Hierarchy [1]

The Error class as shown in Figure 1 represents serious errors that usually pose a challenge that makes the program unable to recover [1]. Such Error conditions include scenarios where a program runs out of memory and cases where a program fails to locate required files [1]. On the other hand, the Exception class represents less serious errors that occurs as unusual conditions [1]. Fortunately, a program is normally able to recover from such types of errors. The most common examples include arithmetic exceptions, array index out of bound exceptions, no such element exceptions, and input mismatch exceptions.

Consider the Java program in Figure 2 that prompt for input of two integers, stored in variables x and y. The program then divides x by y with the results stored in variable z.

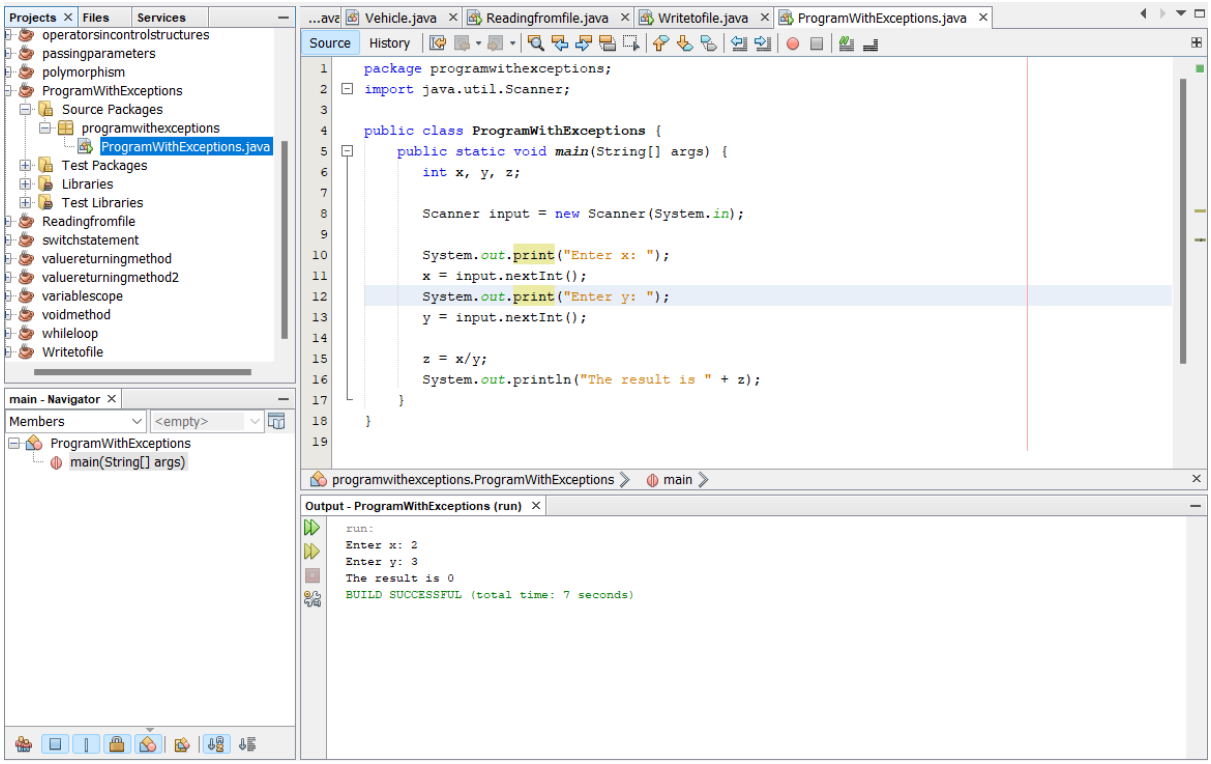

Figure 2. Java Susceptible to Exceptions

The program builds and runs successfully given two integer values for x and y as shown in the output window. The program is however susceptible to two kinds of exceptions that my cause it to terminate unexpectedly. These are the input mismatch exception and the arithmetic exception.

a) Input Mismatch Exception

This kind of exception is bound to occur in cases where the user inputs a value that is not an integer following prompts for values for either variable x and y. This is since the data types for variables x and y have been set to integers, hence, the expected data types for the values are integers. Figure 3 shows what happens when the data type for value x is violated.

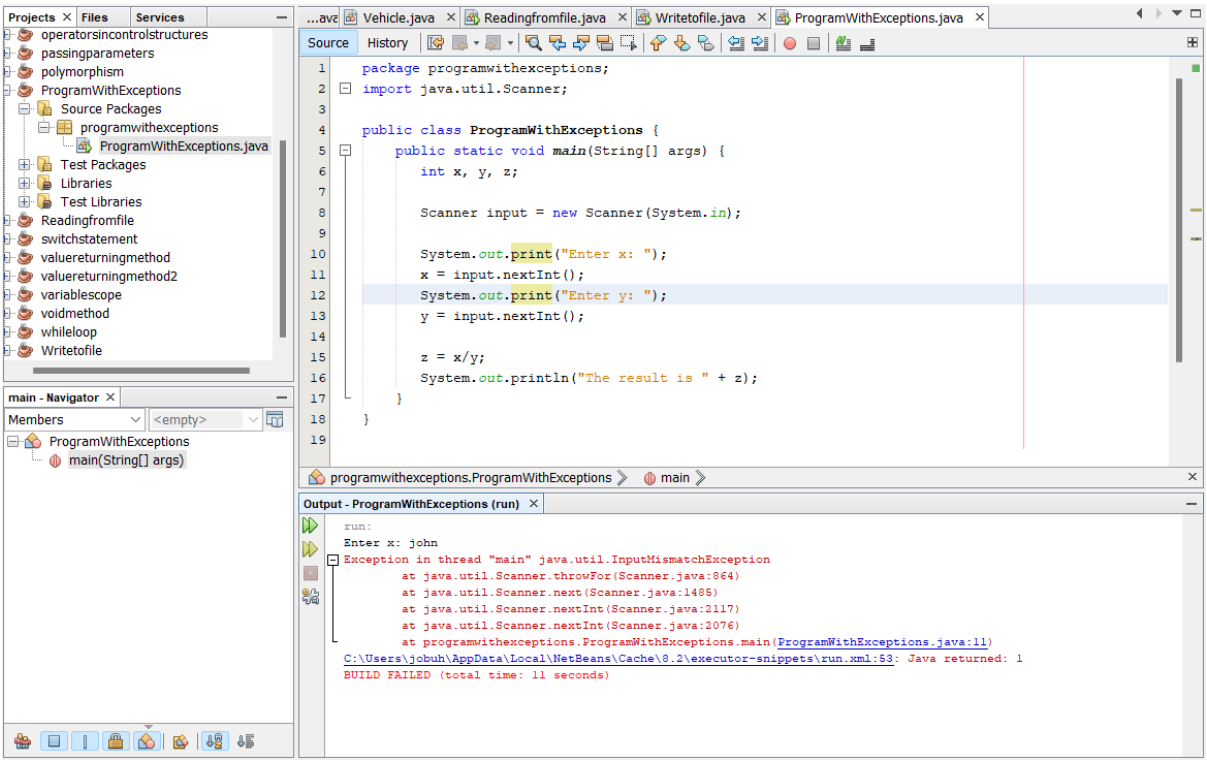

Figure 3. Input Mismatch Exception Example

Based on the output window, the program terminated unexpectedly since the user entered he String “john” instead of an integer value. The main error message says that “Exception in thread main java.util.InputMismatchException”. This is a clear indicator that an input mismatch exception was detected causing the program to terminate unexpectedly. The same is bound to happen if a non-integer value is supplied following a prompt for value y.

b) Arithmetic Exception

This kind of exception is bound to occur in cases where the user inputs a zero (0) for the prompt for value y. This is based on the fact that any given value cannot be divided by a zero (0), hence, the expected data types for y should be an integer that is not a zero (0). Figure 4 show what happens when a zero (0) is inputted as the value for variable y.

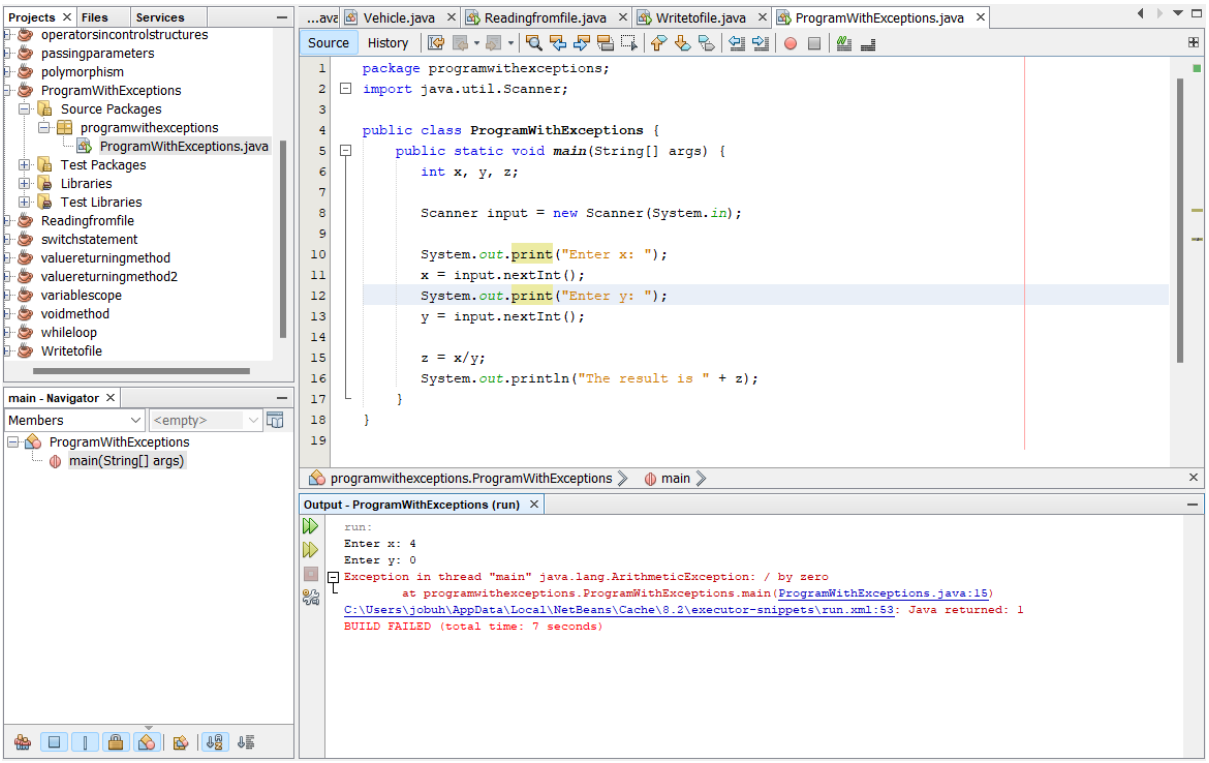

Figure 3. Input Mismatch Exception Example

Based on the output window, the program terminated unexpectedly since the user entered a zero (0) following a prompt for y. The main error message says that “Exception in thread main java.lang.ArithmeticException”. This is a clear indicator that an arithmetic exception was detected, in this case, an attempt to divide by a zero (0), causing the program to terminate unexpectedly.

Exception Handling in Object-Oriented Programming

Programmers may opt to ignore exceptions by letting the offending program to terminate whenever they run into exception conditions. However, doing so may turn out to be abrupt and unforgiving [1] to program users. There are two ways to handle exceptions

1. Using a decision-making statement to avoid occurrence of errors
2. Using the standard object-oriented exception handling technique

In as much as the if … else decision-making approach may be used to avoid occurrence of exceptions, it is a very crude way of handling exceptions. The standard exception handling technique provide a more elegant approach for handling error conditions [1]. This is since they are deemed to be more fault tolerant and robust [1]. The try … catch … finally blocks are

used in this case, which entails trying pieces of code followed by catching resulting exceptions. The following is the syntax for the try … catch … finally blocks as applied to exception handling.

```
try{
   //statements likely to cause exceptions
}catch(ExceptionClassName1 objectreference1){
   //code to handle the detected exception
}catch(ExceptionClassName2 objectreference2){
   //code to handle the detected exception
}catch(ExceptionClassNameN objectreferenceN){
   //code to handle the detected exception
}finally{
   //statements to execute afterwards
}
```

The try block contains statements of program code that might generate an exception [2]. In addition, statements that should not execute in case an exception occurs may also be placed in the try block. The try block is followed by zero or more catch blocks each of which specifies the type of exception it is targeting to catch and contains pieces of code for handling the exception [2]. The finally block may optionally follow the last catch block, in which case, it contains statements that always execute regardless of whether an exception occurs or not [2]. However, in cases when a program exits early from a try block by calling the System.exit, statements placed in the finally block will not execute in this case.

Considering the program example in Figure 2, statements represented by lines 11 and 13 are likely to cause input mismatch exceptions while line 15 is likely to cause an arithmetic exception. Thus, to handle the two kinds of exceptions, the statements must be placed in the try block. Two catch blocks should then be used, each targeting to handle the two exceptions separately. Hence, Figure 4 show the full program, this time round, exceptions handled appropriately.

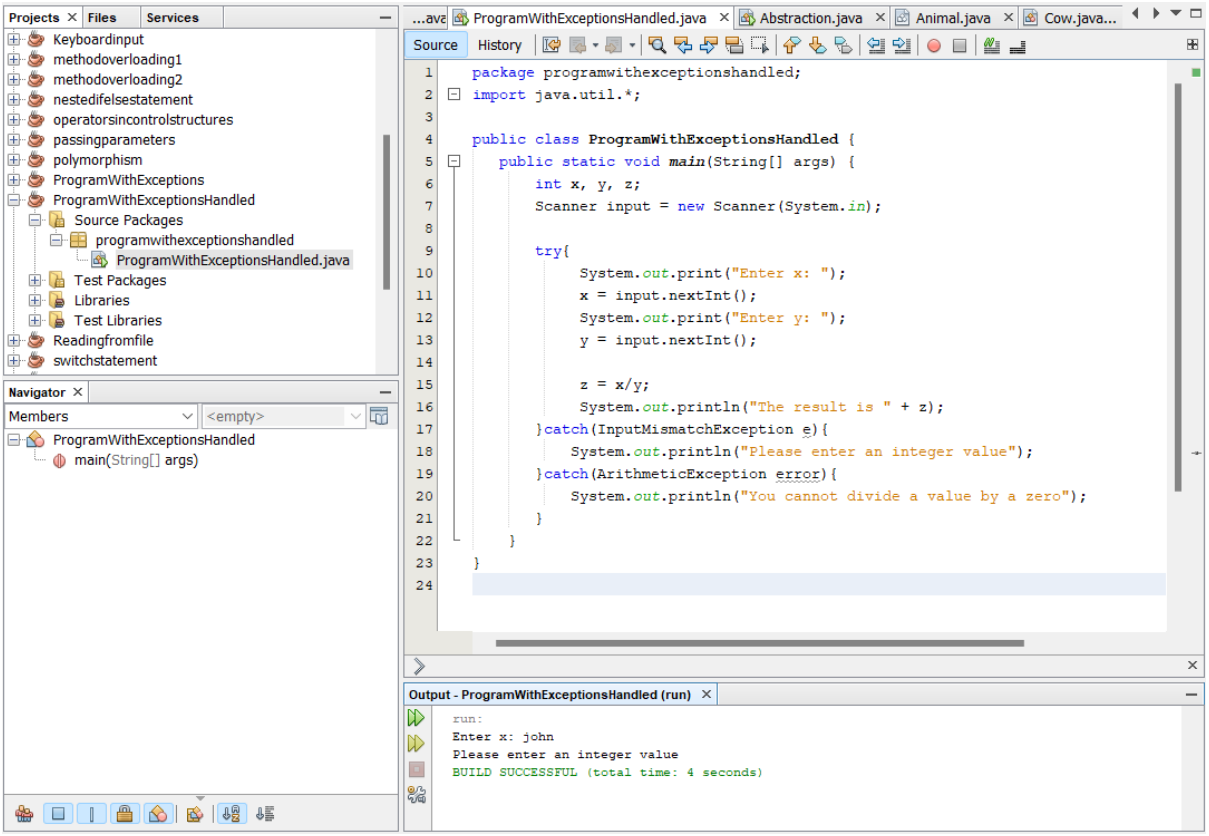

Figure 4. Program with Input Mismatch Exception Handled

According to Figure 4, the try block contains statements of the code that are likely to cause exceptions. These are the two statements allowing console inputs for two integers to be stored in variables x and y, and the statements that divides x by y while storing the results in variable z. The two catch blocks in lines 17 to 21 are independently handling each of the two exceptions detected and thrown by the try block. Based on the output results, an attempt to enter String “john” for the value x prompt is detected and caught by the catch block. Notice that this time round the program does not terminate unexpectedly, it instead notifies the user about the expected value. The same will happen for a similar violation for value y. In the same manner, Figure 5 shows the output for an attempt to divide by a zero (0) which is flagged as an arithmetic exception with the user notified about the occurrence.

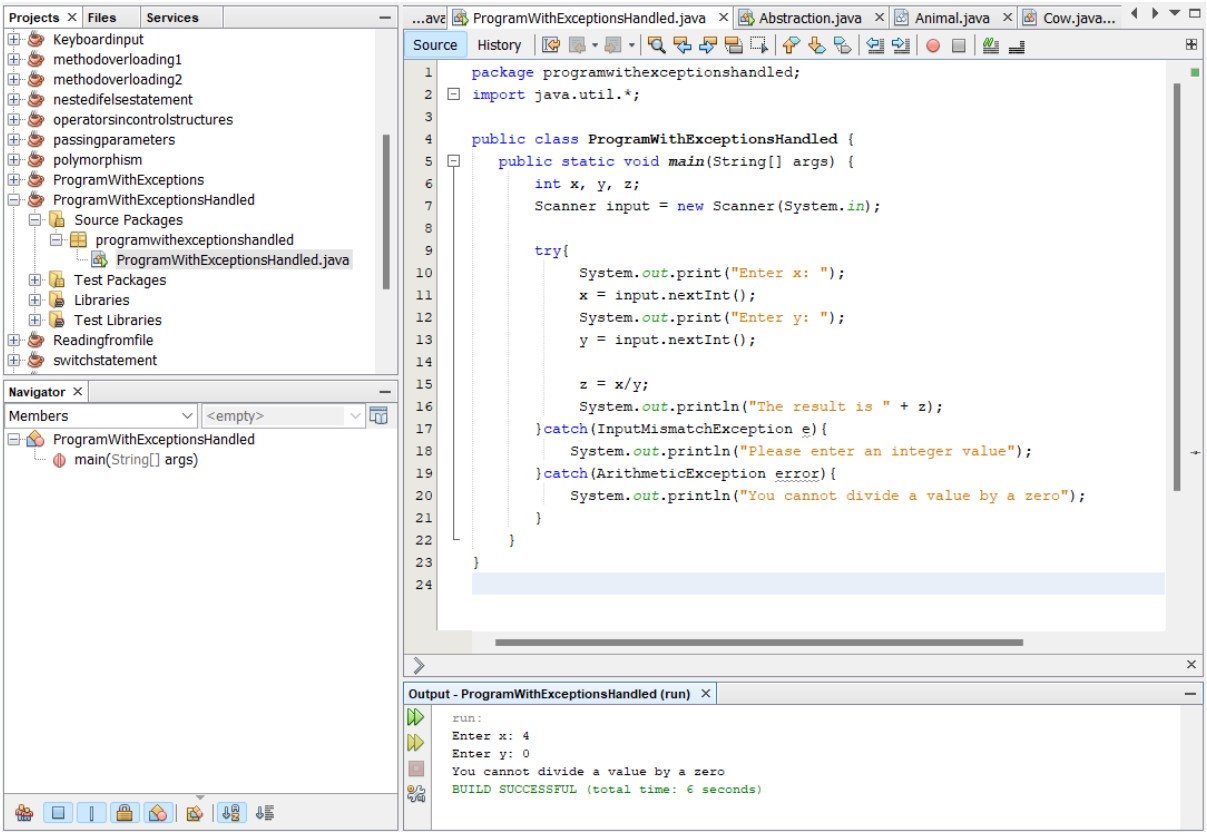

Figure 5. Program with Arithmetic Exception Handled

Figure 6 and 7 shows the fact that inclusion of the finally block does not really change anything in program execution. Instead, the statements located in the finally block are executed whether the try block detects an exception or not.

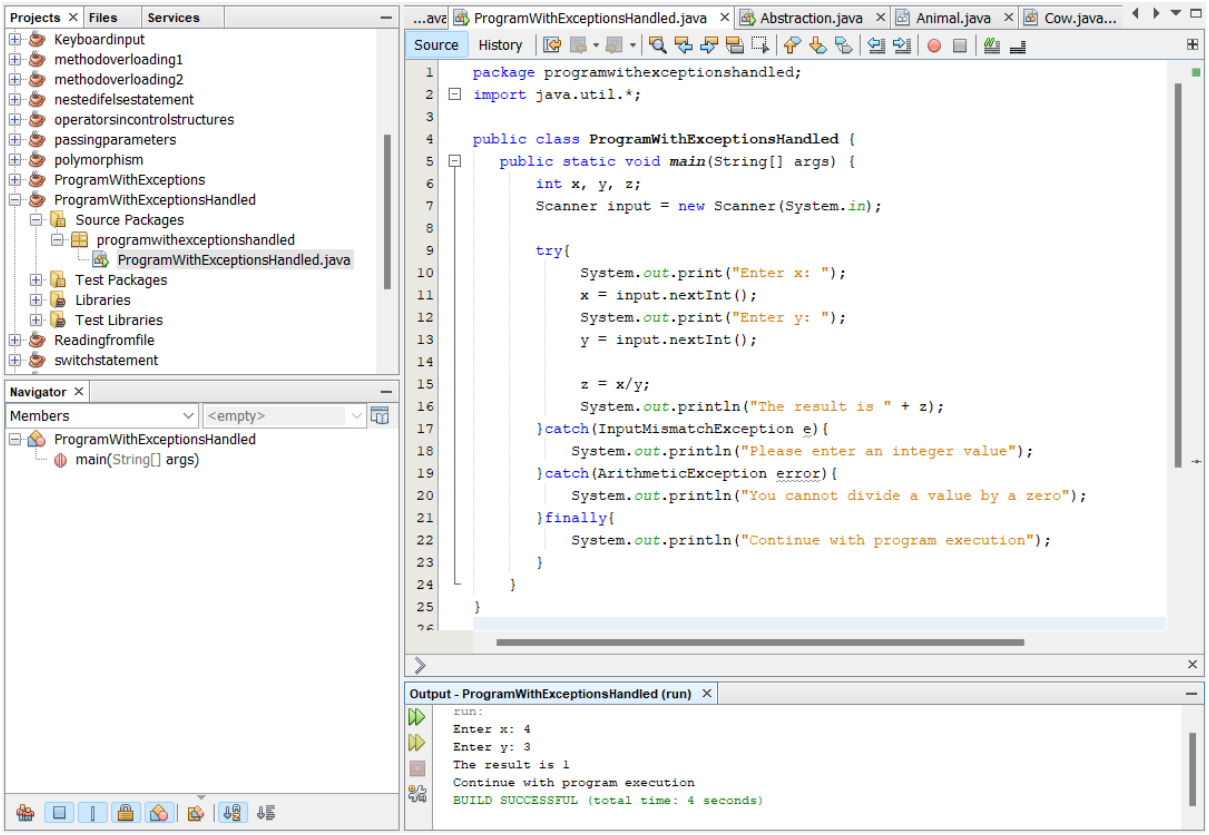

Figure 6. Finally Block with no Exception Detected

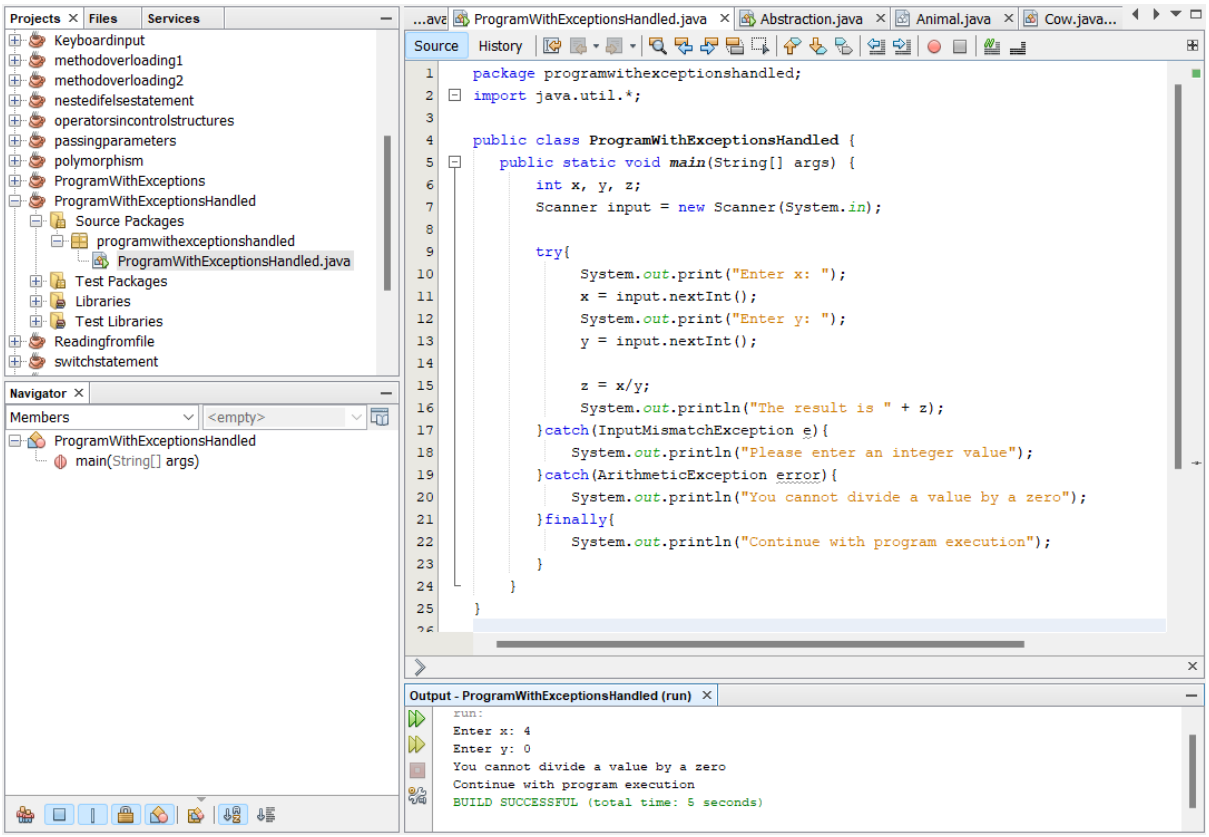

Figure 7. Finally Block with Exception Detected

Using the same approach, any type of exception can be handled in Java with the main aim of preventing unexpected termination of programs following occurrences of unusual conditions.

Summary

The topic has discussed a useful concept of exception handling by first describing exceptions as unusual occurrences during the point when a program is running. Differences between the Java Exception class and Error class have been outlines with example given in each case. The try … catch … finally blocks have been discussed and demonstrated as applied in object-oriented programming for exception handling. Occurrence of the input mismatch exception and the arithmetic exception has been demonstrated with the mechanism used to handle each of them discussed and demonstrated appropriately. A similar approach is applicable for any other type of exception not demonstrated in this topic. The next topic will demonstrate another approach that could be used to handle exceptions with a focus on situations where a computer file being referenced may not be found.

Check Points

1. Differentiate between exceptions and normal errors as experienced in computer programming.
2. State and explain the four main types of exceptions likely to occur in a program.
3. Using examples, differentiate between the Java Exception class and Error class.
4. Describe the standard approach used to handle exceptions in object-oriented programming languages.
5. Using an appropriate example, demonstrate how arithmetic exceptions can be handled in Java.
6. Using an appropriate example, demonstrate how input mismatch exceptions can be handled in Java.

Core Textbooks

1. Joyce Farrell, Java Programming, 7th Edition. Course Technology, Cengage Learning, 2014, ISBN-13 978-1-285-08195-3.
2. Malik, Davender S. JavaTM Programming: From Problem Analysis to Program Design, International Edition, 5th Edition, Cengage Learning.

Other Resources

3. Daniel Liang, Y. "Introduction to Java Programming, Comprehensive." (2011).
4. Malik, Davender S. JavaTM Programming: From Problem Analysis to Program Design, International Edition, 4th Edition, Cengage Learning, 2011.
5. Shelly, Gary B., et al. Java programming: comprehensive concepts and techniques. Cengage Learning, 2012.

References

[1] Farrell, J., Java Programming, 7th Edition. Course Technology, Cengage Learning, 2014, ISBN-13 978-1-285-08195-3.

[2] Malik, D. S., JavaTM Programming: From Problem Analysis to Program Design, International Edition, 5th Edition, Cengage Learning.

[3] Sebester, R. W., Concepts of Programming Languages, 12th Edition, Pearson, 2018, ISBN 0-321-49362-1.