# STUDENT LESSON

## A13 – Exceptions and File I/O

# STUDENT LESSON

# A13 – Exceptions and File I/O

**INTRODUCTION:** Java provides a structured approach for dealing with errors that can occur while a program is running. This approach is referred to as "exception-handling." The word "exception" is meant to be more general than "error." Exception-handling is used to keep a program running even though an error is encountered that would normally stop the program. This lesson will explore file input and output (I/O) as an example of how exceptions are used.

The key topics for this lesson are:

A. Exceptions
B. Handling Exceptions
C. Exception Messages
D. Throwing Exceptions
E. Reading From a File
F. Writing to a File

**VOCABULARY:**

| | |
|---|---|
| **`catch`** | ERROR |
| EXCEPTION | **`try`** |

**DISCUSSION:** A. <u>Exceptions</u>

1. When a Java program performs an illegal operation, a special event known as an *exception* occurs. An exception represents a problem that the compiler was unable to detect before the execution of the program. This is called a run-time error. An example of this would be dividing by zero. The compiler often cannot tell before the program runs that a denominator would be zero at some later point and therefore cannot give an error before the program is run.

2. An exception is an object that holds information about a run-time error. The programmer can choose to ignore the exception, fix the problem and continue processing, or abort the execution of the code. An *error* is when a program does not do what it was intended to do. Compile time errors occur when the code entered into the computer is not valid. Logic errors are when all the code compiles correctly but the logic behind the code is flawed. Run-time errors happen when Java realizes during execution of the program that it cannot perform an operation.

3. Java provides a way for a program to detect that an exception has occurred and execute statements that are designed to deal with the problem. This process is called *exception handling*. If you do not deal with the exception, the program will stop execution completely and send an exception message to the console.

4. Common exceptions include:
   - `ArithmeticException`
   - `NullPointerException`
   - `ArrayIndexOutOfBoundsException`
   - `ClassCastException`
   - `IOException`

   For example, if you try to divide by zero, this causes an `ArithmeticException`. Note that in the code section below, the second `println()` statement will not execute. Once the program reaches the divide by zero, the execution will be halted completely and a message will be sent to the console:

```
int numerator = 23;
int denominator = 0;

// the following line produces an ArithmeticException
System.out.println(numerator/denominator);

System.out.println("This text will not print");
```

   A `NullPointerException` occurs if you use a null reference where you need an object reference. For example,

```
String name = null;

// the following line produces a NullPointerException
int i = name.length();
System.out.println("This text will not print");
```

   Since `name` has been declared to be a reference to a `String` and has the value `null`, indicating that it is not referring to any `String` at this time, an attempt to call a method within `name`, such as `name.length()`, will cause a `NullPointerException`. If you encounter this exception, look for an object that has been declared but has not been instantiated.

## B. Handling Exceptions

1. There are three ways of handling a possible exception occurrence (we say that the exception is *thrown*). In some cases, such as when there is a possibility that a divide by zero exception might occur, the programmer has the option of not dealing with the exception at all. This can be useful in situations where the programmer has already taken steps to ensure that a denominator never becomes zero. The programmer may also attempt to fix the problem or skip over the problem. To do either of these, the programmer needs to catch any exception that may be thrown. To *catch* an exception, one must anticipate where the exception might occur and enclose that code in a `try` block. The try block is followed by a `catch` block that catches the exception (if it occurs) and performs the desired action.

2. The general form of a `try-catch` statement is:

```
try{
  try-block
} catch (exception-type identifier){
  catch-block
}
```

a. The `try-block` refers to a statement or series of statements that might throw an exception. If no exception is thrown, all of the statements within the try-block will be executed. Once an exception is thrown, however, all of the statements following the exception in the try-block will be skipped.

b. The `catch-block` refers to a statement or series of statements to be executed if the exception is thrown. A `try` block can be followed by more than one `catch` block. When an exception is thrown inside a `try` block, the first matching `catch` block will handle the exception.

c. `exception-type` specifies what kind of exception object the `catch` block should handle. This can be specific or it can be general, i.e. `IOException` or just `Exception`. `Exception` by itself will catch any type of exception that comes its way.

d. `identifier` is an arbitrary variable name used to refer to the `exception-type` object. Any operations done on the exception object or any methods called will use this identifier.

3. The `try` and `catch` blocks work in a very similar manner to the `if-else` statement and can be placed anywhere that normal code can be placed.

4. If an exception is thrown anywhere in the `try` block which matches one of the exception-types named in a `catch` block, then the code in the appropriate catch block is executed. If the `try` block executes normally, without an exception, the `catch` block is ignored.

5. Here is an example of `try` and `catch`:

```
int quotient;
int numerator = 23;
int denominator = 0;
try{
   quotient = numerator/denominator;
   System.out.println("The answer is: " + quotient);
} catch (ArithmeticException e){
   System.out.println("Error: Division by zero");
}
```

The value of `denominator` is zero so an `ArithmeticException` will be thrown whenever `numerator` is divided by `denominator`. The `catch` block will catch the exception and print an error message. The `println()` statement in the `try` block will not be executed because the exception occurs before the program reaches that line of code. Once an exception is encountered, the rest of the lines of code in the try-block will be ignored. If the value of `denominator` is not zero, the code in the `catch` block will be ignored and the `println()` statement will output the result of the division. Either way, the program continues executing at the next statement after the `catch` block.

C. Exception Messages

1. If a program does not handle exceptions at all, it will stop the program and produce a message that describes the exception and where it happened. This information can be used to help track down the cause of a problem.

2. The code shown below throws an `ArithmeticException` when the program tries to divide by zero. The program crashes and prints out information about the exception:

```
int numerator = 23;
int denominator = 0;

// the following line produces an ArithmeticException
System.out.println(numerator/denominator);

System.out.println("This text will not print");
```

*Run Output:*

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZero.main(DivideByZero.java:10)
```

The first line of the output tells which exception was thrown and gives some information about why it was thrown. “DivideByZero.main” indicates that the exception occurred in the main method of the DivideByZero class. In the parentheses, the specific file name and line number are given so that the programmer can find where their code went wrong (in the example above, the exception occurred on line 10 of a Java file, DivideByZero.java). This is the line where Java found a problem. The actual root cause of the problem may be a line or two ahead of line 10.

The rest of the output tells where the exception occurred and is referred to as a *call stack trace*. In this case, there is only one line in the call stack trace, but there could be several, depending on where the exception originated.

3. When exceptions are handled by a program, it is possible to obtain information about an exception by referring to the “exception object” that Java creates in response to an exception condition. Every exception object contains a String that can be accessed using the `getMessage` method as follows:

```
try{
  quotient = numerator/denominator;
  System.out.println("The answer is: " + quotient);
} catch (ArithmeticException e){
  System.out.println(e.getMessage());
}
```

If a divide by zero error occurs, an exception is thrown and the following message is displayed:

```
/ by zero
```

4. Printing the value returned by `getMessage` can be useful in situations where we are unsure of the type of error or its cause.

5. If an exception is unknown by your Java class, you may need to import the appropriate exception (just like you would import any other class you wanted your class to know about). For example, the `ArithmeticException` is a standard Java class and no import statement is needed for it. However, `IOException` is part of the `java.io` package and must therefore be imported before it is used.

## D. Throwing Exceptions

1. There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exception or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception.

2. To throw an exception, use a **`throw`** statement. This is usually done with an `if` statement. The syntax of the `throw` statement is:

```
throw exception-object;
```

3. For example, the following statement throws an `ArithmeticException`:

```
if(number == 0){
   throw new ArithmeticException("Division by zero");
}
```

The exception object is created with the `new` operator right in the `throw` statement. Exception classes in Java have a default constructor that takes no arguments and a constructor that takes a single `String` argument. If provided, this `String` appears in the exception message when the exception occurs.

## E. Reading From a File

1. Reading textual data from a file is very similar in many ways to reading input from the keyboard. The Scanner class and all of its methods remain the same. However, you must also import the classes `java.io.File` and `java.io.IOException`. Also, the way in which you use the `Scanner` class constructor changes.

2. The `java.io.File` is a holder class that can take a `String` representing the path to a file on your computer. Creating a Scanner object that reads from a file is as simple as:

   ```
   Scanner in = new Scanner(new File("test.txt"));
   File f = new File("C:\MyDocuments\Java\tester.txt");
   Scanner in2 = new Scanner(f);
   ```

   The first line assumes that there is a file named "test.txt" in the directory or folder where you run your java files. The second line shows an example of creating the File object in its own line of code. It also shows the File being created with the full path to the file. This would be used if your text file was not in the same directory as your Java files. But what happens if the file that you are looking for doesn't exist or has some other status that prevents it from being read? This causes an exception to be thrown!

3. Scanner will take no responsibility in handling the exception, so every time that you want to use Scanner with a file, you will have to use a try-catch block. A full example is shown below:

   ```
   Scanner in;
   try{
         in = new Scanner(new File("test.txt"));
         String test = in.nextLine();
         System.out.println(test);
   }catch(IOException i){
         System.out.println("Error: " + i.getMessage());
   }
   ```

4. When reading a large amount of data from a file, it is often useful to know whether there is any more data in the file to read. Reading from a file which has no more data will give you a `NoSuchElementException` and stop your program. The Scanner class has several methods for determining if there is any more data in the file to be read: `hasNext()`, `hasNextDouble()`, and `hasNextInt()` will be the methods most useful for you. If there is anything still to come, `hasNext()` will return true, while `hasNextDouble()` will return true only if a valid double is next and `hasNextInt()` will return true only if an int value is next. Using a simple while loop, you can easily read in data until the end of a file.

   ```
   while(in.hasNext()){
         System.out.println(in.next());
   }
   ```

F. <u>Writing to a File</u>

1. Creating your own files is a little bit trickier than reading from them. There are two basic ways to write data to files: raw bytes and character streams. Raw byte writing is useful for items such as pictures. Character streams are used for writing plain text. This curriculum will focus only on character streams.

2. This curriculum uses the java.io.FileWriter class. It has two basic constructors:

   ```
   FileWriter file = new FileWriter("test.txt");
   FileWriter file2 = new FileWriter("test2.txt", true);
   ```

   The first constructor opens a basic FileWriter object that points at the file "test.txt" in the same directory where the Java files are being run. When this file is first opened and written to, all of the data that was previously stored in the file will be erased. The second constructor indicates that the new data being sent to the file will be appended to the end of the file. In either case, if the file does not exist, then Java will attempt to create a new file with the indicated name and at the indicated location.

3. Writing data to the file is done by using the FileWriter.write(`String`, `int`, `int`) method. The `String` is the data that will be written to the file. The first `int` is where to start writing the data in the `String`. The second `int` indicates how many characters of the `String` to actually write. For example:

   ```
   String one = "#Hello!!!";
   FileWriter out = new FileWriter("test.txt");
   out.write(one, 1, 5);
   ```

   This will write only "Hello" to the file "test.txt."

4. Merely opening a file and writing to it is not enough to store your data in most cases. You know from personal experience that if you don't save your work in a word processor, your work will not be there the next time you start up your computer. Data must be saved. This is done with FileWriter by calling the close() method when you are done writing data. This "closes" the output stream to the file and saves your data.

   ```
   Out.write(one, 1, 5);
   Out.close();
   ```

5. What if there is some error in opening the file? That's right - an exception is thrown and it must be dealt with just like in the Scanner class.

   ```
   String one = "Hello World!!!";
   FileWriter out;
   try{
         out = new FileWriter("test.txt");
         out.write(one, 0, one.length());
         out.close();
   }catch(IOException i){
         System.out.println("Error: " + i.getMessage());
   }
   ```

6. There is no equivalent to `println()` with the `FileWriter` class, so any newlines that you wish to create must be done with the `'\n'` character.

7. FileWriter only deals with writing Strings to the text files, which creates a little bit of a problem with writing numeric data. However, we can use the shortcut learned earlier in Lesson A10, *Strings* to change our other data types to Strings.

```
String temp;
int a = 5;
temp = “” + a + “\n”;
out.write(temp, 0, temp.length());
double p = 3.14;
temp = “” + p + “\n”;
out.write(temp, 0, temp.length());
boolean test = true;
temp = “” + test + “\n”;
out.write(temp, 0, temp.length());
```

8. Because `FileWriter` requires you to specify how many characters of the given String to print out, you must be careful with the values that you give it. If the `int` value that you send is bigger than the `String` itself, you will get a `StringIndexOutOfBoundsException` when the `FileWriter` object tries to access characters in the String which do not exist. An easy way to prevent this from ever occurring is to always create a `String` object before the write method is called with the data you wish to output, place that `String` in the call to write, and use that `String’s length()` method for how many characters to print.

```
String one = “Hello World!!!\n”;
Out.write(one, 0, one.length());
```

**SUMMARY/ REVIEW:**

Exceptions provide a clean way to detect and handle unexpected situations. When a program detects an error, it throws an exception. When an exception is thrown, control is transferred to the appropriate exception handler. By defining a method that catches the exception, the programmer can write the code to handle the error. Exceptions are used heavily when dealing with File I/O because there are many situations that can turn dangerous when reading and writing data directly from a hard drive.

**ASSIGNMENT:**

Lab Assignment A13.1, *ErrorCheck*
Lab Assignment A13.1, Starter Code – *CheckingAccount.java*
Lab Assignment A13.2, *Average*
Lab Assignment A13.2, Data File, *numbers.txt*
Lab Assignment A13.3, *Squeeze*
Lab Assignment A13.3, Data File, *squeeze.txt*
Worksheet A13.1, *Exceptions Review*