1DV532 – Starting Out with Java

Lesson 12

Exception Handling

Dr. Nadeem Abbas
nadeem.abbas@lnu.se

What is an Exception

An **exception** is an abnormal event or condition that occurs at runtime and disrupts the normal flow of execution

- In other words, an exception is a *run-time error*

- **Example - Divide by Zero**

```
public class ExceptionDemo {
        public static void main(String[] args) {
         int p = 7, q = 0;
         int res = p/q;
         System.out.println(p + " divided by " + q + " = " + res);
        }
 }
```

- The above example program does not have any compile time error, however, it produces following Exception when we execute it.
  - `Exception in thread "main" java.lang.ArithmeticException: / by zero`

Exception Handling

- An exception can occur for many different reasons. Following are some example scenarios where an exception occurs.
  - A file that needs to be opened cannot be found.
  - A program is trying to access an array element using an index value beyond the array length.
  - A user has entered an invalid data, e.g., divide by zero.
- When an Exception occurs the normal flow of the program is disrupted and the program terminates abnormally, which is not recommended
  - Program shall be able to ***handle the exceptions***

Exception Handling

- Java provides a mechanism that signals when something unusual happens
  - This is called ***throwing an exception***

- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called ***handling the exception***

Java Exceptions – Fundamentals

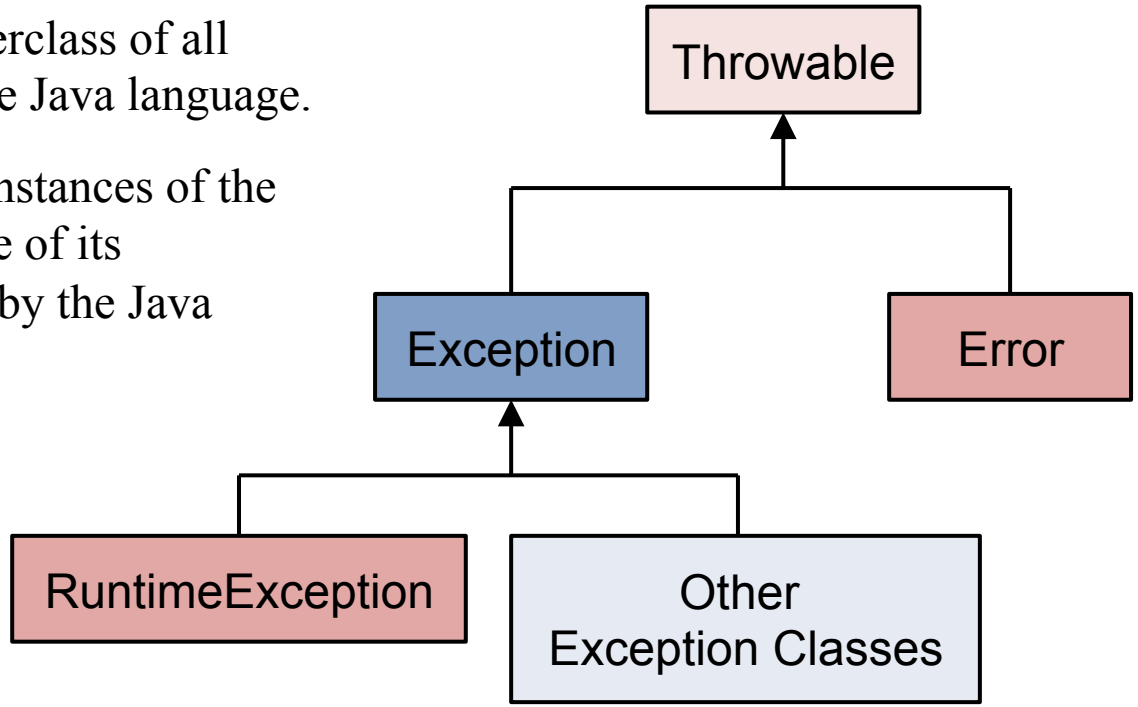
- **Java Exception is an object** that encapsulates information about an ***error*** or ***abnormal event***.
- When an error occurs within a method, the method creates an object, called an **Exception object,** and hands it off to the runtime system.
  - Creating an exception object and handing it to the runtime system is called **Throwing an Exception**.
    - For example, *IndexOutOfBoundsException* is thrown when a program tries to access an array element with *index* value which is less than 0 or more than the array's size.
- Program execution is halted immediately when an exception is thrown
- If a programmer does not provide an exception handler, the Java Virtual Machine (JVM) calls **Default Exception Handler** which
    - displays a string describing the exception,
    - prints a stack trace from the point at which the *exception* occurred, and
    - terminates the program
      - We did not provide an exception handler in the example program, *ExceptionDemo.java,* the JVM calls default exception handler.

Exception Classes

- Java treats Exceptions as objects and provides number of classes, such as *FileNotFoundException*, *ArithmeticException IndexOutOfBoundsException*, to deal with different types of exceptions.

- All exception classes are subtypes of the java.lang.**Exception** class.
  - The Exception class itself is a subclass of the **Throwable** class

- The **Throwable** is the superclass of all *errors* and *exceptions* in the Java language.
  - Only objects that are instances of the Throwable class or one of its subclasses are thrown by the Java Virtual Machine.



Exception Types

There are two fundamental types of Exceptions

1. **Checked Exceptions**
   - These are the exceptions that can be anticipated and recovered from, e.g. java.io.FileNotFoundException, IOException, etc.
     - All Exception types are checked exceptions, except Error, RuntimeException, and their subclasses.
   - Must be checked at compile time, and must be handled using either
     - *try-catch* blocks, or *throws* clause
       - We will learn about *try-catch* and *throws* later in the lesson.
2. **Unchecked Exceptions**
   - These are the exceptions that are either external to the application (e.g., hardware failure) or cannot be anticipated and recovered from, e.g., NullPointerException, IOError
   - Are not subject to *try-catch* or the *throws* clause

Exception Handling

- Java provides with **try, catch**, and **finally** block statements to write an exception handler
  - **Exception handler** – a piece of code written to manage exceptions
- **try, catch, and finally – General Syntax**

```
try{
 // code that may throw an exception
}
catch (ExceptionType1 exOb){
 // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb){
 // exception handler for ExceptionType2
}
//… any number of catch blocks
finally{
 // block of code to be execcuted after try block ends
}
```

- See the example program, *TryCatchFinallyDemo.java*, which demonstrates how we can use try, catch and finally.

TryCatchFinallyDemo.java – Demonstrates try, catch and finally

```
public class TryCatchFinallyDemo {
	public static void main(String[] args) {
		int a = 0, b = 0, res = 0;
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter value of a: ");
		a = sc.nextInt();
		try {
				System.out.print("Enter value of b: ");
				b = sc.nextInt();
				res = a / b;
		} catch (ArithmeticException e) {
		    do {
				System.out.print("Zero is not allowed as value of b, Please enter a positive integer value: ");
				b = sc.nextInt();
		    } while (b == 0);                                                    }
		catch (InputMismatchException ex) {
					System.out.println("Non-numeric values are not allowed!");
					System.out.print("Enter positive integer value of b: ");
					sc.nextLine();
					b = sc.nextInt();
		} finally {
				res = a / b;
				System.out.println(a + " divided by " + b + " = " + res);
				sc.close();
		}
	}}
```

try Block

- As shown in the example program, **`try`** block encloses or contains code that may throw an *exception*

- The `try` block must be immediately followed by either a **`catch or finally`** or both of these two blocks,
  - The **`try` block cannot be used alone**.

- A single `try` block may have multiple `catch` blocks associated with it to handle multiple exceptions.

`try` Block

When a **`try`** block is executed, two things can happen:

1. No exception is thrown in the **`try`** block
   - The code in the **`try`** block is executed to the end of the block
   - The **`catch`** block is skipped
   - The execution continues with the code placed after the **`catch`** block
     - The code after the **`catch`** block can be **`finally`** block or any other statement(s).
2. An exception is thrown in the **`try`** block and caught in the **`catch`** block
   - The rest of the code in the **`try`** block is skipped
   - Control is transferred to a following **`catch`** block
   - The thrown exception object is plugged in for the **`catch`** block parameter
   - The code in the **`catch`** block is executed
   - The code that follows that **`catch`** block is executed (if any)

`catch` Block

- A **`catch`** block encloses or contains code to handle an *exception*

```
catch(ExceptionType e)
{
   ExceptionHandlingCode
}
```

- Each **`catch`** block is an **exception handler** that handles the type of exception indicated by its parameter.
  - The parameter type, **`ExceptionType`**, declares the type of exception that the catch block can handle.
    - For example, the below defined catch block can handle only the exceptions of type ArithmeticException, such as divide by zero.
    - We have to define another catch block if we want to handle an exception of some other type.

```
catch (ArithmeticException e) {
        // exception handling code
}
```

`catch` Block

- A program may have multiple **`catch`** blocks associated with singly **`try`** block to handle multiple exceptions, see the example program *TryCatchFinallyDemo.java*, for example.
  - For such programs, the first catch block that matches the type of the exception thrown is the one that is executed
  - Thus, while defining multiple catch blocks, we should be careful about order of the Exception type handled by each catch block.

`catch` Block

- Since Java SE 7 and later, a single **`catch`** block can handle more than one type of exceptions.
  - The catch block shown below, for example, can catch and handle Exceptions of type *ArithmeticException* and *ArrayIndexOutOfBoundsException*

```
try{
        int array[] = new int[10];
        array[10] = 30/5;
}
catch(ArithmeticException |
      ArrayIndexOutOfBoundsException e){
            e.printStackTrace();
}
```


`finally` Block

- The **`finally`** block contains code to be executed whether or not an exception is thrown in a **`try`** block
  - If it is used, a **`finally`** block is placed after a **`try`** block and its following **`catch`** blocks

```
try{
   . . .
}catch(ExceptionClass1 e){
   . . .
}catch(ExceptionClassN e){
   . . .
}finally
{
   CodeToBeExecutedInAllCases
}
```


`finally` Block

- The finally block, which always gets executed regardless of an exception has been thrown or not, allows programmers to specify statements that must be executed before a program comes to an end.
  - It is often used to perform clean up operations that may result in resource leaks, e.g., closing the opened File Input/Output Streams, as shown in the example program, ReadFile.java.

Example Program – `ReadFile.java`

```
public class ReadFile {
    public static void main(String[] args) {
          /* Create object representing a file file */
          File file = new File("sample.txt");
          /* Create Scanner to read from file */
          Scanner fileScan =null;
          try {
                     fileScan = new Scanner(file);
                    /* Read one word at the time */
                    int count = 0;
                    while (fileScan.hasNext()) { // true ==> more text available
                              count++;
                              String str = fileScan.next(); // read next word
                              // String str = fileScan.nextLine(); // read next line
                              System.out.println(count + "\t" + str);
                    }
          } catch (IOException e) {e.printStackTrace();}
          finally {
            if(fileScan != null){
              try {
                     fileScan.close();
                     System.out.println("--- File End ---");
               } catch (Exception e) {  //do something clever with the exception}
         }
      }
    } }
```

try-with-resources statement

- *try-with-resources* is an improved form of the try statement that declares one or more resources
- A **resource** is an object that must be closed at the end of the try-with-resources statement.
  - The resource is closed automatically
    - No need of the **finally** block

Example:

```
public static void main(String[] args) {
        File file = new File("someFile.text");
        try (Scanner fileScan = new Scanner(file)){
            while (fileScan.hasNext())

        System.out.println(fileScan.nextLine());
        } catch (FileNotFoundException e) {

        e.printStackTrace();
        }
}
```


Exceptions thrown by a Method

If a method is capable of causing an exception that it does not handle, it must specify this behavior as a part of method's signature

- **General Form:**

  [access modifier] [static] type <methodName> (<parameters> **) throws Exception-List{**

  // method body

  }

  - The **Exception-List** is a comma-separated list of Exceptions that a method can throw.

- Specifying unhandled exceptions is important to ensure that those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them.

Exceptions thrown by a Method – Example

```
public static void main(String[] args) throws FileNotFoundException {
        /* Create object representing a file file */
        File file = new File("/Users/nadeem/eclipse/README.txt");
        /* Create Scanner to read from file */
        Scanner fileScan =null;
         fileScan = new Scanner(file);

        /* Read one word at the time */
        int count = 0;
        while (fileScan.hasNext()) { // true ==> more text available
                count++;
                String str = fileScan.next(); // read next word
                // String str = fileScan.nextLine(); // read next line
                System.out.println(count + "\t" + str);
        }
}
```

`throw` keyword

- Java provides us with a keyword `throw` to generate and throw exceptions explicitly
- General syntax to use the `throw` keyword:

```
throw someThrowableObject;
```

  - For example,
    - `throw new ArithmeticException("Divisor cannot be zero");`

- As shown above, the `throw` keword requires a single argument: a throwable object.
  - Throwable objects are instances of any subclass of the Throwable class.
- Example program *ThrowAnException.java* demonstrates use of the throw keyword.

Example Program – ThrowAnException.java

```
public class ThrowAnException {
 public static void main(String args[]) {
        int age;
        Scanner sc = new Scanner(System.in);
        System.out.print("To cast your vote you need to be
                        18+.\nPlease enter your age: ");
        age = sc.nextInt();
        if (age < 18)
          throw new ArithmeticException("Your age is less than
                18, you are not allowed to vote.");
        else
          System.out.println("Welcome to vote");
        }
}
```


Suggested Readings

- Absolute Java, Global Edition, 6/E by Walter J. Savitch, Chap 9 Exception Handling
- Introduction to Java Programming, Brief Version, Global Edition, 11/E Liang, Chapter 12 Exception Handling and Text I/O

- Java Tutorials
  - https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Lnu.se