Treatment of Exceptions

Table of Contents

Java Exceptions 2

Rules for try, catch and finally Blocks..... 5

Defining New Exceptions..... 6

Handling Multiple Exceptions 8

The Java Exception Hierarchy..... 10

Exceptions in java are any abnormal, unexpected events or extraordinary conditions that may occur at runtime. They could be file not found exception, unable to get connection exception and so on. On such conditions java throws an exception object. Java Exceptions are basically Java objects. No Project can never escape a java error exception.

Java exception handling is used to handle error conditions in a program systematically by taking the necessary action. Exception handlers can be written to catch a specific exception such as Number Format exception, or an entire group of exceptions by using a generic exception handlers. Any exceptions not specifically handled within a Java program are caught by the Java run time environment

An exception is a subclass of the Exception/Error class, both of which are subclasses of the Throwable class. Java exceptions are raised with the throw keyword and handled within a catch block.

A Program Showing How the JVM throws an Exception at runtime

```
public class DivideException {

    public static void main(String[] args) {
        division(100,4);              // Line 1
        division(100,0);        // Line 2
        System.out.println("Exit main().");
    }

    public static void division(int totalSum, int totalNumber) {
        System.out.println("Computing Division.");
        int average  = totalSum/totalNumber;
        System.out.println("Average : "+ average);
    }
}
```

An ArithmeticException is thrown at runtime when Line 11 is executed because integer division by 0 is an illegal operation. The “Exit main()” message is never reached in the main method

Output

```
Computing Division.
java.lang.ArithmeticException: / by zero
Average : 25
Computing Division.
at DivideException.division(DivideException.java:11)
at DivideException.main(DivideException.java:5)
Exception in thread “main”
```

Java Exceptions

The Throwable class provides a String variable that can be set by the subclasses to provide a detail message that provides more information of the exception occurred. All classes of throwables define a one-parameter constructor that takes a string as the detail message.

The class Throwable provides getMessage() function to retrieve an exception. It has a printStackTrace() method to print the stack trace to the standard error stream. Lastly It also has a toString() method to print a short description of the exception. For more information on what is printed when the following messages are invoked, please refer the java docs.

Class Exception

The class Exception represents exceptions that a program faces due to abnormal or special conditions during execution. Exceptions can be of 2 types: Checked (Compile time Exceptions)/ Unchecked (Run time Exceptions).

Class RuntimeException

Runtime exceptions represent programming errors that manifest at runtime. For example ArrayIndexOutOfBounds, NullPointerException and so on are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class. These are basically business logic programming errors.

Class Error

Errors are irrecoverable condtions that can never be caught. Example: Memory leak, LinkageError etc. Errors are direct subclass of Throwable class.

Exception Statement Syntax

Exceptions are handled using a try-catch-finally construct, which has the Syntax

```
try {
<code>
} catch (<exception type1> <parameter1>) { // 0 or more
<statements>
}
} finally { // finally block
<statements>
}
```

try Block
The java code that you think may produce an exception is placed within a try block for a suitable catch block to handle the error.

If no exception occurs the execution proceeds with the finally block else it will look for the matching catch block to handle the error. Again if the matching catch handler is not found execution
proceeds with the finally block and the default exception handler throws an exception.. If an exception is
generated within the try block, the remaining statements in the try block are not executed.

catch Block
Exceptions thrown during execution of the try block can be caught and handled in a catch block. On exit from a catch block, normal execution continues and the finally block is executed
(Though the catch block throws an exception).

finally Block
A finally block is always executed, regardless of the cause of exit from the try block, or

whether any catch block was executed. Generally finally block is used for freeing resources, cleaning up, closing connections etc. If the finally clock executes a control transfer statement such as a return or a break statement, then this control statement determines how the execution will proceed regardless of any return or control statement present in the try or catch.

The following program illustrates the scenario.

```
try {
    <code>
} catch (<exception type1> <parameter1>) { // 0 or more
    <statements>

}
} finally {                               // finally block
    <statements>
}
```

Output

```
Computing Division.
Exception : / by zero
Finally Block Executes. Exception Occurred
result : -1
```

Below is a program showing the Normal Execution of the Program.

Please note that no NullPointerException is generated as was expected by most people

```
public class DivideException2 {

    public static void main(String[] args) {
        int result  = division(100,0);        // Line 2
        System.out.println("result : "+result);
    }

    public static int division(int totalSum, int totalNumber) {
        int quotient = -1;
        System.out.println("Computing Division.");
        try{
                quotient  = totalSum/totalNumber;

        }
        catch(Exception e){
                System.out.println("Exception : "+ e.getMessage());
        }
        finally{
                if(quotient != -1){
                        System.out.println("Finally Block Executes");
                        System.out.println("Result : "+ quotient);
                }else{
                        System.out.println("Finally Block Executes.
Exception Occurred");
                        return quotient;
                }

        }
        return quotient;
    }
}
```

Output

null (And not NullPointerException)

Rules for try, catch and finally Blocks

1. For each try block there can be zero or more catch blocks, but only one finally block.

2. The catch blocks and finally block must always appear in conjunction with a try block.

3. A try block must be followed by either at least one catch block or one finally block.

4. The order exception handlers in the catch block must be from the most specific exception

Java exception handling mechanism enables you to catch exceptions in java using try, catch, finally block. be An exception consists of a block of code called a try block, a block of code called a catch block, and the finally block. Let’s examine each of these in detail.

```
public class DivideException1 {

    public static void main(String[] args) {
        division(100,0);        // Line 2
        System.out.println("Main Program Terminating");
    }

    public static void division(int totalSum, int totalNumber) {
        int quotient = -1;
        System.out.println("Computing Division.");
        try{
                quotient  = totalSum/totalNumber;
                System.out.println("Result is : "+quotient);
        }
        catch(Exception e){
                System.out.println("Exception : "+ e.getMessage());
        }
        finally{
                if(quotient != -1){
                        System.out.println("Finally Block Executes");
                        System.out.println("Result : "+ quotient);
                }else{
                        System.out.println("Finally Block Executes.
Exception Occurred");
                }

        }
    }
}
```

Output

```
Computing Division.
Exception : / by zero
Finally Block Executes. Exception Occurred
Main Program Terminating
```

As shown above when the divide by zero calculation is attempted, an ArithmeticException is thrown. and program execution is transferred to the catch statement. Because the exception is thrown from the try block, the remaining statements of the try block are skipped. The finally block executes.

Defining New Exceptions

We can have our own custom Exception handler to deal with special exception conditions instead of using existing exception classes. Custom exceptions usually extend the Exception class directly or any subclass of Exception (making it checked). The super() call can be used to set a detail message in the throwable. Below is an example that shows the use of Custom exception’s along with how the throw and throws clause are used.

Output

```
Very Hot
Very Cold
Perfect Temperature
```

throw, throws statement

A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.

The general format of the **throw** statement is as follows:

throw <exception reference>;

The Exception reference must be of type Throwable class or one of its subclasses. A detail message can be passed to the constructor when the exception object is created.

throw new TemperatureException(“Too hot”);

A **throws** clause can be used in the method prototype.

Method() throws <ExceptionType1>,…, <ExceptionTypen> {

}

Each <ExceptionTypei> can be a checked or unchecked or sometimes even a custom Exception. The exception type specified in the throws clause in the method prototype can be a super class type of the actual exceptions thrown. Also an overriding method cannot allow more checked exceptions in its throws clause than the inherited method does.

When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a matching catch block that can handle the exception. Any associated finally block of a try block encountered along the search path is executed. If no handler is found, then the exception is dealt with by the default exception handler at the top level. If a handler is found, execution resumes with the code in its catch block. Below is an example to show the use of a throws and a throw statement.

```
public class DivideException3 {

    public static void main(String[] args) {
        try{
                int result  = division(100,10);
                result  = division(100,0);
                System.out.println("result : "+result);
        }
        catch(ArithmeticException e){
                System.out.println("Exception : "+ e.getMessage());
        }
    }

    public static int division(int totalSum, int totalNumber) throws
ArithmeticException {
        int quotient = -1;
        System.out.println("Computing Division.");
        try{
                if(totalNumber == 0){
                        throw new ArithmeticException("Division attempt by
0");
                }
                quotient  = totalSum/totalNumber;
        }
        finally{
                if(quotient != -1){
                        System.out.println("Finally Block Executes");
                        System.out.println("Result : "+ quotient);
                }else{
                        System.out.println("Finally Block Executes.
Exception Occurred");
                }

        }
        return quotient;
    }
}
```

Output

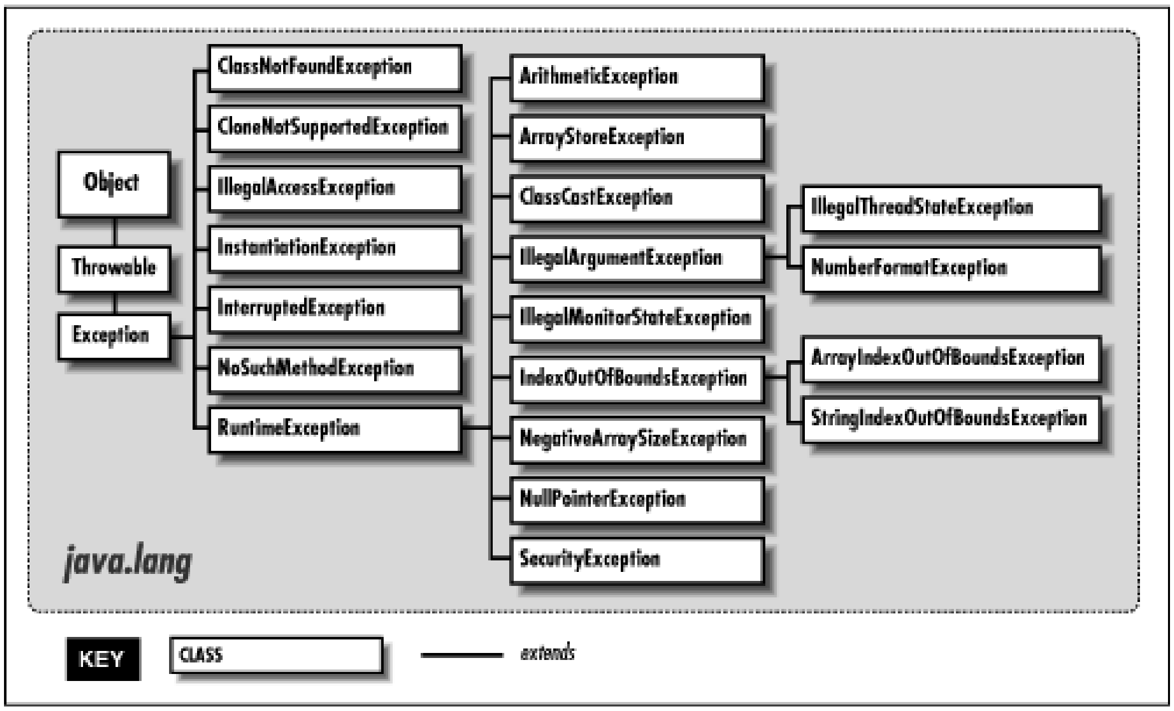
```
Computing Division.
Finally Block Executes
Result : 10
Computing Division.
Finally Block Executes. Exception Occurred
Exception : Division attempt by 0
```

Handling Multiple Exceptions

It should be known by now that we can have multiple catch blocks for a particular try block to handle many different kind of exceptions that can be generated. Below is a program to demonstrate the use of multiple catch blocks.

```
import java.io.DataInputStream;
import java.io.IOException;

import javax.swing.JOptionPane;
public class ExceptionExample7{
    static int numerator, denominator;

    public ExceptionExample7( int t, int b ){
        numerator = t;
        denominator = b;
    }

    public int divide( ) throws ArithmeticException{
        return numerator/denominator;
    }

    public static void main( String args[] ){

        String num, denom;

        num = JOptionPane.showInputDialog(null, "Enter the Numerator");
        denom = JOptionPane.showInputDialog(null, "Enter the 
Denominator");

                try{
                    numerator = Integer.parseInt( num );
                    denominator = Integer.parseInt( denom );
                }
                catch ( NumberFormatException nfe ){
                    System.out.println( "One of the inputs is not an 
integer" );
                    return;
                }
        catch ( Exception e ){
                    System.out.println( "Exception: " + e.getMessage( ) );
                    return;
        }

                ExceptionExample7 d = new ExceptionExample7( numerator, 
denominator );
                try{
                    double result = d.divide( );
                    JOptionPane.showMessageDialog(null, "Result : " + 
result);
                }
                catch ( ArithmeticException ae ){
                        System.out.println( "You can't divide by zero" );
                }
                finally{
                        System.out.println( "Finally Block is always 
Executed" );
            }
    }
}
```

The Java Exception Hierarchy


Object
Throwable
Exception
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
IndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
IllegalThreadStateException
NumberFormatException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
java.lang
KEY
CLASS
extends