Exception Handling in Java

Dr S Mahesan

01 May 2016

Abstract

This describes the Exception Handling in Java with suitable examples. Exercises are also given for students to try for better understanding the methods and their usage. This handout is useful for anyone using Java.

Exception Handling Basics

An error condition is treated as an exceptional condition. Java has a provision to handle error condition by means of exception handling. A Java exception is an object that describes error condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that has caused the error. The method may handle the exception itself, or pass it on to be caught and processed at some other point. Exceptions can be generated by the code or by the Java run-time system. Exceptions thrown by Java relate to fundamentals error that violate the rules of the Java Language or the constraints f the Java execution environment. Manually generated exceptions (by your code) are used to report some error condition to the caller of a method.

There are five keywords that involve in exception handling in Java : `try`, `catch`, `throw`, `throws` and `finally`. Program statements that we want to monitor exceptions are contained in `try` block. If an exception occurs within the try block, it is thrown. Thrown exception can be caught by `catch`, and handled in its block. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, the keyword `throw` is used.

Any exception that is thrown out of a method must be specified as such by a `throws` clause. Any code that absolutely must be executed before a method returns is put in a `finally` block.

The general form of the Exception handling:

```
 try{
 <block of code to monitor>
 }
 catch(ExceptionType1 exOb){
<block of code to handle the exception>
 }
 catch(ExceptionType2 exOb){
<block of code to handle the exception>
 }
:
:
 finally{
  <block of code that must be executed before try block ends>
 }
```

Here `ExceptionType` is the type of exception that has occurred - there are different exception types.

Exception Types

The `Throwable` class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause.

Two immediate child classes of `Throwable` class are `Exception` and `Error`:

The `Exception` class is used for exceptional conditions that user programs should catch. This is the class we can define subclasses of which to create our own exception types. `RuntimeException` is an important child class of `Exception`. Exceptions of this type are automatically defined for the programs that we write and include things such as *division by zero* and *invalid array indexing*.

The `Error` class defines exceptions that are not expected to be caught under normal circumstances by our program, but are used by the Java run-time system to indicate errors arising during run-time. Stack overflow is an example of such an error. Errors of this kind cannot be handled by our programs.

Let us see an example:

```
class Exc0{
    pubic static void main(String args[]){
        int d=0; int q=123/d;
    }
}
```

It is obvious that this would cause the "Division by zero error". Java run-time system would detect the attempt to divide by zero and construct a new exception object and then throws this exception. This causes the execution of `Exc0` to stop unless the exception is caught by an exception handler and dealt with immediately.

Exercise: Try this example as it is, and observe the output generated by the Java JDK run-time interpreter.

Also try another version of the same exception:

```
class Exc1{
    static void tryDivision(){
      int d; int q=123/d;
    }
    pubic static void main(String args[]){
       Exc1.tryDivision();
    }
}
```

Exercise: Try this example also, and observe the output generated by the Java JDK run-time interpreter. Note the difference in the output, and the line numbers indicated.

Note that the details in such output messages are very useful to pinpoint the places for debugging purposes.

Let us see how we can catch this exception:
We know that it is an arithmetic exception, and thus we use an object of `ArithmeticException` to catch the exception.

```
class Exc2{
  public static void main(String args[]){
    int d, q;
    try{d=0; q=123/d;
        System.out.println("This will never be printed");
       }
```

```
    catch(ArithmeticException aeObj){
       System.out.println("Division by Zero.");
      }
    System.out.println(">>After catch statement...");
   }
 }
```

After trying `q=123/d` no further instructions will be executed within `try` block, and the `println()` statement will never be executed, program control transfers out of the `try` block into the `catch` block.

On catching the exception, the statements in `catch` block will be executed, and execution continues as usual unless the program is interrupted from within `catch` block manually or by the Java run-time system.

The `try` and its `catch` statement form a unit. The scope of the `catch` clause is restricted to those statements specified by the immediately preceding `try` statement. A catch statement will *only* catch the exception thrown by its own matching try block or the try block enclosing it and its try block.

Note that statements that are protected by try *must* be surrounded by curly braces.

The aim of a well constructed `catch` clause should be to resolve the exceptional condition and then continue on as if the error had not happened.

Let us look at the following example: (This makes use of the class `Random` from `java.util` package to produce random integers using `nextInt()` method.

```
import java.util.Random;
class HandleError{
  public static void main(String args[]){
    int d1=0,d2=0, q=0;
    Random rnd = new Random();
    for(int i=0; i<100000; i++) {
      try { d1=rnd.nextInt();   d2=rnd.nextInt();
            q=12345/(d1/d2);      //causes error when d1=0 and/or d2=0
          } catch (ArithmeticException e) {
               System.out.println("Division by Zero Error.");
               q=0; // set q to zero
               }
      System.out.println("q= "+q);
      }
    }
 }
```

Exercise: Include the statement `System.out.println("Exception: "+e);` and observe what is printed, see how useful the message is.

Multiple catch clause

In some cases, more than one exception could be raised by a single piece of code. We can specify two or more `catch` clauses, each catching a different type of exception. When an exception is thrown, each `catch` statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are by passed, and execution continues after the `try...catch` block. Let us look at the following example that demonstrates multiple `catch` statements:

```
class MultiCatch {
   public static void main(String args[]) {
      try{ int a = args.length;
           System.out.println("a = " + a);
           int b = 42 / a;
           int c[] = { 1 };
           c[42] = 99;
           }catch(ArithmeticException e) {
              System.out.println("Divide by 0: " + e);
              }
            catch(ArrayIndexOutOfBoundsException e) {
              System.out.println("Array index oob: " + e);
              }
      System.out.println("After try...catch blocks.");
      }
   }
```

Exercise: Try this code as follows after compiling:
`(a) java Multicatch`
`(b) java Multicatch aha`
and observe the output.
In the first case the value of `a` will be zero and thus the first catch statement will catch the error (division by zero), and the statements below `b=42/a` within the `try` block will not be executed.
In the second case: the value of a is 1, use of index > 0 with array `c` will cause an error (array out of bounds error, why?), and this will be caught by the second `catch` statement.

Note:

1. When you use multiple `catch` statements, it is important to remember that exception of subclasses must come before any of their superclasses (more specific first than more generic). This is because a `catch` statement that uses superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

2. Also remember that in Java, unreachable code is an error.

The following example illustrates this:

```
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println("Generic Exception catch.");
          }
        /* This catch is never reached because ArithmeticException is a subclass of Except
          catch(ArithmeticException e) { // ERROR - unreachable
            System.out.println("This is never reached.");
          }
    }
}
```

This program cannot be compiled successfully because `ArithmeticException` is a subclass of `Exception`. Any arithmetic errors will be captured by the first `catch` and thus the part of the second catch statement will never be reached - having an unreachable portion of code is illegal in Java.

Nested try Statements

A `try` statement can be inside the block of another `try`. Each time a `try` statement is entered, the context of that exception is pushed on the stack. If an inner `try` statement does not have a `catch` handler for a particular exception, the stack is unwound and the next `try` statement's `catch` handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested `try` statements are exhausted. If no `catch` statement matches then the Java run-time system handle the exception (as if there were no `try...catch` statements). Let us see the following example:

```
class NestTry {
  public static void main(String args[]) {
    try { //outer try
          int a = args.length;
          int b = 42 / a;   // division by zero error if a==0
          System.out.println("a = " + a);
          try { //inner try (nested try)
                a = a/(a-1); // division by zero error if a==1
                if(a==2) {
                int c[] = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
              } catch(ArrayIndexOutOfBoundsException e) {
                 //matching catch of the inner try
                 // but for array index out of bounds type
                  System.out.println("Array index out-of-bounds: " + e);
                }
```

```
          } catch(ArithmeticException e) { //matching catch of the outer  try
             System.out.println("Divide by 0: " + e);
             }
    }
  }
```

Exercise: Try the above code after compilation as follows:
(a) `java NestTry`
(b) `java NestTry aha`
(c) `java NestTry aha oho`
and observe the output.

In the first case (`a=0`), division by zero occurs in the outer `try...catch` block. The division by zero that would arise when `a=1` will not be caught by the inner block, it is passed on to the outer try block, where it is handled. If `a=2`, an array boundary exception is generated from within the inner `try` block, and it is handled by the `catch` statement matching the inner `try` statement.
If a method called from within a `try...catch` block and if that method enclose a `try` statement then this is also nested inside the outer `try` block.

```
class MethodNestTry {
  static void nesttry(int a) {
    try {
         a = a/(a-1);  /* division by zero if a==1 */
         if(a==2) {
            int c[] = { 1 };
            c[42] = 99; // generate an out-of-bounds exception
            }
         } catch(ArrayIndexOutOfBoundsException e) {
              System.out.println("Array index out-of-bounds: " + e);
              }
    }

  public static void main(String args[]) {
    try {
          int a = args.length;
          int b = 42 / a;
          System.out.println("a = " + a);
          nesttry(a);/* this is not executed if a==0 */
          } catch(ArithmeticException e) {
              System.out.println("Divide by 0: " + e);
              }
    }
  }
```

Try this program with (a) no argument, (b) one argument, and (c) two arguments and compare the outputs with those of the previous version.

throw

It is possible for our program to throw an exception explicitly, using the throw statement. The general form of throw is show here:
`throw throwableInstance;`
Here `throwableInstance` must be an object of type `Throwable` or subclass of `Throwable`. Simple types such as `int` or `char`, as well as non-Throwable classes, such as `String` or `Object` cannot be used as exceptions. There are two ways to obtain a Throwable object:

1. using a parameter into a `catch` clause.
2. creating one with the `new` operator.

The flow of execution stops immediately after the `throw` statement; any subsequent statements are not executed. The nearest enclosing `try` block is inspected to see if it has a `catch` statement that matches that the type of the exception. If it finds a match, control is transferred to that statement. If not, the next enclosing `try` statement is inspected and so on. If no matching `catch` is found then the default exception handler halts the program and prints the stack trace.

```
class ThrowDemo {
  static void alliReign(String s) {
    try {
      if (s.equalsIgnoreCase("male")) throw new NullPointerException("Alli throws it!");
      else if (s.equalsIgnoreCase("female")) System.out.println("Alli welcomes..");
      else throw  new NullPointerException("Alli doesn't like it!");
    } catch(NullPointerException e) {
      System.out.println("Caught inside alliReign.");
      throw e; // re-throw the exception
    }
  }

  public static void main(String args[]) {
    try {
      alliReign(args[0]);
    } catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Give an argument - either male  or female");
     }
      catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

After compiling this program, try the following and make your notes:
`(a) java ThrowDemo (b) java ThrowDemo female (c) java ThrowDemo male (d) java ThrowDemo person`
Note that how the throwable instance is created using new with one parameter to constructor `NullPointerExcepti`
All of Java's built-in run-time exceptions have two constructors: one with no parameter and one that

takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print. It can also be obtained by call to `getMessage()` which is defined by `Throwable`. For example, in the above program in place of printing `e` we could have used `e.getMessage()`.

throws

A `throws` clause lists the types of exceptions that a method might throw. This is necessary for all exceptions except those type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw *must* be declared in the `throws` clause. Otherwise, a compile time error will result.
The general form of method declaration that includes a `throws` clause:

```
   type method-name(parameter-list) throws exception-lists
{ \\body
}
```

Here *exception-list* is a comma separated list of the exceptions that a method can throw.
Let us look at the following example:

```
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Exercise: Try this to see how it works!

Note: If we omit `throws IllegalAccessException` the program will not compilable. The same problem will arise if we ignore `try ...catch block`.

```
class ThrowsDemo {
  static void throwOne()  {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
      throwOne();
    }
}//this program cannot be compiled. Why?
```

finally

It creates a block of code that will be executed after a `try...catch` block. The `finally` block will execute even if no `catch` statement matches the exception. Anytime a method is about to return to the caller from inside a `try...catch` block, via an uncaught exception or an explicit return statement, the `finally` clause is executed just before the method returns. The finally clause is optional. However, this can be useful in situations when there is a need for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. Each `try` statement will have at least one `catch` or a `finally` clause.
Look at the following example:

```
 class FinallyDemo {
// Through an exception out of the method.
static void procA() {
  try {
    System.out.println("inside procA");
    throw new RuntimeException("demo"); // after throwing exception,
                                        //  the finally clause will be executed!
  } finally {
    System.out.println("procA's finally");
  }
}

// Return from within a try block.
static void procB() {
  try {
    System.out.println("inside procB");
    return;  //yet, before returning from procB, the finally clause will be executed!

  } finally {
    System.out.println("procB's finally");
  }
}

// Execute a try block normally.
static void procC() {
  try {
    System.out.println("inside procC"); //after printing,  the finally clause will be execut
  } finally {
    System.out.println("procC's finally");
  }
}
public static void main(String args[]) {
  try {
    procA();
  } catch (Exception e) {
```

```
      System.out.println("Exception caught");
    }
    procB();
    procC();
  }
}
```

Exercise: Try the above and observe the output.
Exercise: Write a program to print the solutions to the equations of the form $ax^2 + bx + c = 0$ where $a^2 + b^2 \neq 0$.

Java's Built-in Exceptions

Inside the standard package `java.lang`, Java defines several exception classes. Only a few have been used in the preceding examples. The most general of these exceptions are subclasses of the standard type `RuntimeException`. Since `java.lang` is implicitly imported into all Java programs, most exceptions derived from `RuntimeException` are automatically available. They need not be included in any method's `throws` list. In the language of Java, they are called *unchecked exceptions*. The following table lists the unchecked exceptions defined in `java.lang`.

Java's unchecked RuntimeException Subclasses	
Exception	**Meaning**
ArithmeticException	Arithmetic error, such as division by zero
ArrayIndexOutOfBoundsException	Array index is out-of-bounds
ArrayStoreException	Assignment to an array element of an incompatible type
ClassCastException	Invalid cast
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value
IllegalArgumentException	Illegal argument used to invoke a method
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread
IllegalStateException	Environment or application is in incorrect state
IllegalThreadStateException	Requested operation not compatible with current thread state
IndexOutOfBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference
NumberFormatException	Invalid conversion of a string to a numberic format
SecurityException	Attempt to violate security
StringIndexOutOfBounds	Attempt to index outside the bounds of a string
UnsupportedOperationException	An unsupported operation was encountered

The following table lists Java's checked exceptions defined in `java.lang`.

Java's Checked Exceptions defined in `java.lang`	
Exception	**Meaning**
ClassNotFoundException	Class not found
CloneNotSupportedException	Attempt to clone an object that does not implement the `Cloneable` interface
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface
InterruptedException	One thread has been interrupted by another thread
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist
ReflectiveOperationException	Superclass of reflection-related exceptions

Creating Exception Subclasses

Although Java has built-in exceptions to handle most common errors, we can create our own exception types to handle situations specific to our applications. We define such a class as a subclass of `Exception` (which is a subclass of `Throwable`. The `Exception` class does not define any methods of its own, it only inherits those methods provided by `Throwable`.
The following table shows the methods defined by the `Throwable` class:

The Methods Defined by Throwable	
Method	**Description**
final void addSuppressed(Throwable exc)	Adds exc to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try-with-resources statement.
Throwable FillInStackTrace()	Returns a `Throwable` object that contains a completed stack trace. This object can be rethrown
String getLocalizedMessage()	Returns a localised description of the exception
String getMessage()	Returns a description of the exception
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of `StackTraceElement`. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The `StackTraceElement` class gives your program access to information about each element in the trace, such as its method name.
final Throwable[ ] getSuppressed( )	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try-with-resources statement.
Throwable initCause(Throwable causeExc)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception
void printStackTrace()	Displays the stack trace

void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream
void setStackTrace (StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.
String toString()	Returns a String object containing a description of the exception. This method is called by println() when outputting a `Throwable` object

The following example declares a new subclass of `Exception` and then uses that subclass to signal an error condition in a method. It overrides the `toString()` method, allowing the description of the exception to be displayed using `println()`.

```
class AlliException extends Exception {
  private String sexDetail;
  AlliException(String a) {
    sexDetail = a;
  }
  public String toString() {
    return "AlliException[" + sexDetail + "]";
  }
}
```

The following example shows a method that throws this exception, and also shows how a thrown exception is caught to be handled.

```
 class AlliExceptionDemo {
  static void filter(String sx) throws AlliException {
    System.out.println("Called filter(" + sx + ")");
    if(!sx.equalsIgnoreCase("female"))
      throw new AlliException(sx);
      //method stops here when the if condition is true
    System.out.println("Normal exit");
  }
  public static void main(String args[]) {
    try {
      filter("Female");
      filter("Male");
    } catch (AlliException e) {
      System.out.println("Caught " + e);
    }
  }
```

Exercises:

1. Try the above and make your observation about how exception handling works.
2. Try and see the difference after swapping the filter statements in the main.
3. Define your own class to handle situation when a square root is to be found for a negative real number.
4. Show how you would use this class in a program that finds solutions to quadratic equations.

Look at the following example in which a provision is made to get the cause for the exception:

```
class ChainExcDemo {
  static void demoproc() {
    // create an exception
    NullPointerException
     e =  new NullPointerException("top layer");
    // add a cause
    e.initCause(new ArithmeticException("cause"));
    throw e;
  }
  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      // display top level exception
      System.out.println("Caught: " + e);
      // display cause exception
      System.out.println("Original cause: " +
                         e.getCause());
    }
  }
}
```

Exercises:

1. Try and see the output.
2. Replace `e.initCause(new ArithmeticException("cause"))` by `e.initCause(new Exception("cause"))` and try to see what difference it would make.
3. Modify `AlliExceptionDemo` given above to set the cause as `"Alli Doesn't like other Sex"` and to get the cause.
4. What have you learnt from this handout?

Three Recently Added Exception Features

Since JDK7, three useful features have been added to the existing system.

try-with-resources to automate the process of releasing a resource, such as a file, when it is no longer needed.

Multi-catch feature allows two or more exceptions to be caught by the same catch clause, and handled by the same code sequence even though the exception types may be different.

To use a multi-catch, separate each exception type in the catch clause with the OR-operatory. Each multi-catch parameter is implicitly final, it cannot be assigned a new value.

```
catch(ArithmeticException | ArrayOutOfBoundsException e) {...}
```

The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.
class MultiCatch {
  public static void main(String args[]) {
    int a=10, b=0;
    int vals[] = { 1, 2, 3 };
    try {
           int result = a / b; // generate an ArithmeticException
           vals[10] = 19; // generate an ArrayIndexOutOfBoundsException
           // This catch clause catches both exceptions.
         } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
           System.out.println("Exception caught: " + e);
         }
         System.out.println("After multi-catch.");
       }
}
```

Exercise:

1. Compile and run the program and observe the output.
2. Replace the assignment `b=0` with `b=1`, and try the program again and then complie and run again, and observe the difference but output by the same code in catch-clause.

More precise rethrow feature restricts the type of exceptions that can be rethrown only to those checked exceptions that the associated `try` block throws, that are not handled by a preceding `catch` clause, and that are a subtype or supertype of the parameter.

For the more-precise-rethrow feature to be in force, the `catch` parameter must be either effectively `fina` inside the catch block, or explicitly declared `final`.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of `try`, `throw`, and `catch` as clean ways to handle errors and unusual boundary conditions in your programs logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses *exceptions*. Thus, when a method can fail, have it *throw* an exception. This is a cleaner way to handle failure modes.

One last point: Javas exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

-x-