Module: Core Java

Session 12: Exception Handling

* Exception: An Overview
* What is an Exception
* Exception-handling Fundamentals
* Exception Types
* Uncaught Exceptions
* Using Try and Catch
* Using Throw, Throws and Finally
* Multiple Catch Clauses
* Built-in Exceptions
* User-defined Exceptions
* Advantages of Exceptions

**Objective**

At the end of the chapter, you will be able to:

* Know about an Exception
* Understand Exception-handling Fundamentals
* Know different types of Exception
* Use Try and Catch
* Use Multiple Catch Clauses
* Use Throw, Throws and Finally
* Distinguish Built-in Exceptions and User-defined Exceptions
* Rule over advantages of Exceptions

**Exception: An Overview**

You are familiar with writing small Java programs. But whether it is a small or a large program, it is impossible that errors will not occur in programs at some points. Though the complier detects the syntax errors during compilation yet there are errors that are detected during execution.

Many programming languages like BASIC, PASCAL and C do not provide the facility to detect errors or take corrective actions during run-time. In the event of an error, the programs written in these languages terminate abruptly causing the control to be transferred to the operating system. It is the responsibility of the programmers to ensure that the programs are error-free. Errors should be checked and handled manually by using error codes. But this kind of programming is very cumbersome and leads to complex code, that is, a code that is very difficult to manage.

Exceptions are things that are not supposed to occur.

Java provides an excellent mechanism that enables us to easily handle errors or exceptions that occur at run-time so that the program does not terminate abruptly.

**What is an Exception?**

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

Consider an example in which we try to open a file and read from it.

Example:

1. Ask the user to enter the name of a file
2. Open the file
3. Read the contents
4. Close the file

The code for this program compiles without any error. However, during execution it can encounter an error if the file referred in the program, is not available. This causes the program to terminate abruptly. But this situation should not arise at run-time. Java is an Internet language and a program could be executed on different machines. A program terminating abruptly may leave the end user confused. Therefore, Java has provided this mechanism of Exception-Handling in order to handle run-time errors.

There are various situations when an exception could occur:

* Attempting to access a file that does not exist
* Inserting an element into an array at a position that is not in its bound
* Performing some mathematical operation that is not permitted, for example, a division by zero
* Declaring an array using negative values
* Using an un-initialized instance variable

**Exception-Handling Fundamentals**

You can provide ability to your program in order to intercept run-time errors, take corrective measures and also continue the execution without fail is referred to as '**Exception Handling**'.

A Java exception is an object automatically dispatched, that describes an exceptional (that is, error) condition that has occurred in a statement of your program. When an exceptional condition arises in any statement, an object representing that exception is created and is thrown or dispatched. If the statement is within a method, then it may choose to handle the exception itself, or pass it on to the default handler. Either way, at some point, the exception is caught within the method or default handler and it get processed. If the exception is caught by the default handler, then it provides a message at the console termed as StackTrace.

The Java run-time system is having the ability to generate an Exception, or your code can manually generate the Exception. Exceptions thrown by Java, relate to fundamental errors that violate the rules & regulations of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via following keywords:

**Blocks:**

* try { }
* catch () { }
* finally { }

**Clauses:**

* throw
* throws

The statements within your program, you anticipate to produce errors at runtime and want to monitor the same, is advised to be kept within a **try** block. If an exception occurs within the **try** block, then one object of corresponding type will be thrown by Java run-time environment. You can catch this exception (using **catch block**) and handle it in some rational manner. The exceptions generated by system are automatically thrown by the Java run-time system. To throw an exception manually within your program, you have to use the keyword **throw**.

Any exception that is being thrown within a method and is not handled within it, must be specified by a **throws** clause in the method signature. Any code that is to be executed absolutely at any cost, even if the control of the program goes to it or not, has to be put in a **finally** block.

This is the general form of an exception-handling block:

try {

// block of code to monitor for errors

}

catch (ExceptionType1 e1) {

// exception handler for ExceptionType1

}

catch (ExceptionType2 e2) {

// exception handler for ExceptionType2

}

// ...

finally {

// block of code to be executed before try block ends

}

Here, ExceptionType is a type of exception that has occurred and being dispatched at run time of the program, which need to be caught. The remaining chapter describes how to apply this framework.

**Exception Types**

All types of exceptions present in any hierarchy of Java are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Throwable class is followed by two subclasses that break exceptions into two distinct branches. One branch is headed by Exception and the other is Error. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of the types are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which says that you cannot expect to catch the Errors under normal circumstances within your program.

The Java run-time system uses exceptions of the type Error to indicate errors that are related to the run-time environment itself. An example of this kind of an error is stack overflow. Note that the Error category could not be handled within your programs.

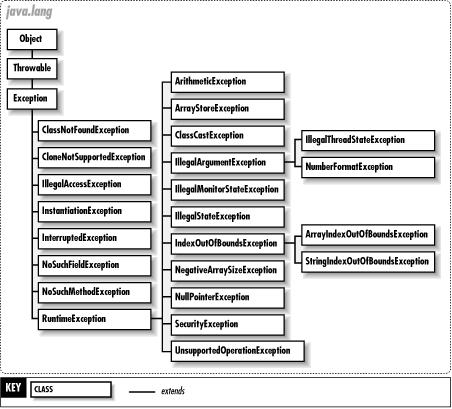


Fig.1: Exception Hierarchy

**Uncaught Exceptions**

Before you learn how to implement the exception handling mechanisms in your program, it is advisable to see what happens when you don’t handle them. The following is a small program that includes an expression that intentionally causes a divide-by-zero error.

Example: ExceptionTestDemo.java:

public class ExceptionTestDemo{

public static void main(String args[]) {

int d = 0;

int a = 42 / d;

}

}

What happens when a statement attempting to divide a number by zero is detected by the Java run-time system? An object of the corresponding type is created by it and that throws the exception object. The execution of ExceptionTestDemo is stopped by this. It happens like this because once an exception has been dispatched, the default or user defined exception handler must *catch* and deal with it immediately. In the example here, the default handler handles the exception since no user defined exception handlers have been supplied. Any exception, not caught by your program will be ultimately processed by the default handler. The thrown objected is accepted by the default handler, which in turn, attaches some description to the object. The default handler also prints a stack trace from the point where the execution took place and thus terminates the program.

The above example generates the following output when executed.

java.lang.ArithmeticException: / by zero

at ExceptionTestDemo.main(ExceptionDemo.java:4)

You can notice how the class name, ExceptionTestDemo; the method name, main; the filename, ExceptionTestDemo; and the line number, 4, are all included in the simple stack trace. Also, you can notice that the type of the exception thrown is a subclass of Exception class called ArithmeticException, which describes what type of error happened. Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

The sequence of method invocations that led upto the error will always be shown by the stack trace. Consider the example of RevisedExceptionTestDemo.java. This is another version of the ExceptionTestDemo.java program. The same error is introduced here, however, in a separate method from main( ):

Example: RevisedExceptionDemo.java:

public class RevisedExceptionTestDemo{

static void subroutine() {

int d = 0;

int a = 10 / d;

}

public static void main(String args[]) {

RevisedExceptionTestDemo.subroutine();

}

}

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

java.lang.ArithmeticException: / by zero

at RevisedExceptionTestDemo.subroutine(RevisedExceptionTestDemo.java:4)

at RevisedExceptionTestDemo.main(RevisedExceptionTestDemo.java:7)

Now after execution, you can see, the bottom of the stack is main’s line 7, which is the call to subroutine(), which caused the exception at line 4. The call stack is quite useful for debugging because it pinpoints the precise sequence of steps that led to the error.

**Using try and catch**

As you know Exception Handling can be performed by the default exception handler provided by the Java run-time system, you can also handle an exception by yourself. This benefits you in two different ways. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most of the end users would be confused if your program stopped running and prints a stack trace whenever an error occurred. You need not worry. It is quite easy to prevent this.

In order to guard against this and handle a run-time error, simply enclose the code that you anticipate to throw errors at run time and want to monitor the same; inside a try block. A catch block must follow the try block that specifies the exception type that you wish to catch. The following program illustrates how easily this can be done. The ExceptionDemoTryCatch.java program includes a try block and a catch clause, which processes the ArithmeticException generated by the division-by-zero error:

Example: ExceptionDemoTryCatch.java

public class ExceptionDemoTryCatch{

public static void main(String args[]) {

int d, a;

try { // monitor a block of code.

d = 0;

a = 42 / d;

System.out.println("This will not be printed.");

} catch (ArithmeticException e) { // catch divide-by-zero error

System.out.println("Division by zero.");

}

System.out.println("After catch statement.");

}

}

This program generates the following output:

Division by zero.

After catch statement.

What is happening here? The call to println( ) inside the try block will never be executed. Now throw an exception. With this, the control of the program will transfer itself out of the try block and will enter in the catch block that follows it. You have noticed that the execution never “returns” to the try block from a catch since the catch itself is not “called”. Therefore, “This will not be printed.” is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

A unit is formed by a try and its catch statement. The immediately preceding try statement restricts the scope of the catch clause to those statements. The only exception is in the case of the nested try statements, where an exception thrown by another try statement cannot be caught by a catch block. Remember that the braces that are protected by try, must be surrounded by curly braces.

## Multiple catch Clauses

In some scenarios, more than one exception could be raised by a single piece of code. To handle this type of scenarios, you can specify two or more catch blocks, 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 with that of the exception is executed. After one catch statement is executed, the others are bypassed and execution continues after the try/catch block. The MultiCatchTestDemo.java traps two different exception types:

Example: MultiCatchTestDemo.java:

// Demonstrate multiple catch statements.

public class MultiCatchTestDemo {

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.");

}

}

This program will cause a division-by-zero exception, if it is started with no command line parameters, since variable ‘a’ will be equal to zero. It will survive the division, if you provide a command-line argument, setting the varibale ‘a’ to something larger than zero. But, it will cause an ArrayIndexOutOfBoundsException, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].

The above program generates the following output in both ways:

C:\>java MultiCatchTestDemo

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultiCatchTestDemo TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException

After try/catch blocks.

When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type, plus any of its subclasses. Thus, the control would never be reached if it came after its superclass.

Further, in Java, unreachable code is an error. For example, consider the SuperSubCatchTestDemo.java:

Example: SuperSubCatchTestDemo.java:

/\* This program contains an error.

A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.

\*/

public class SuperSubCatchTestDemo {

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 Exception. \*/

catch(ArithmeticException e) { // ERROR - unreachable

System.out.println("This is never reached.");

}

}

}

If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

## Using Throw

So, far you have seen exceptions that are thrown by the Java run-time system. You can also make your program explicitly throw an exception by using the throw statement.

throw ThrowableObject;

Notice that the above statement throws an object that represents an exception. It is an object of class Exception or class Throwable or any of its subclasses.

Example: ThrowDemo.java demonstrates this concept:

public class ThrowDemo{

static void testMethod(){

try{

throw new Exception(“test”);

}catch (Exception e){

System.out.println(“Exception caught...”);

}

}

public static void main(String args[]) {

testMethod();

}

}

Pay attention to:

throw new Exception(“test”);

This statement throws an object of the Exception class. The object is created using new operator. Remember that the object that is thrown should be of type **Throwable** or any of its subtypes, not any generic object. The constructor that is used to create the exception object can be an empty constructor or it can take a string as a parameter. The string inside the constructor describes the exception. When you use this exception object in the println() method, it displays the string in the constructor.

**Using Throws**

Sometimes, a method may throw an exception but does not handle it itself. Instead, the callers of the method should provide an exception context in the form of a try block and a corresponding catch handler for the exception. The method only specifies the exception it is throwing. A method can throw more than one exception. For example:

If a code contains statements to open and read a file, the corresponding method could throw:

* FileNotFoundException
* IOException
* EOFException

Hence, wherever this method is used, you should guard against these exceptions. That is, this method should be called in a try block and the respective exceptions should be caught.

Example: ThrowsDemo.java:

import java.io.\*;

public class ThrowsDemo{

static void processFile(String str) throws FileNotFoundException {

System.out.println("Inside processFile method...");

throw new FileNotFoundException ("test");

}

public static void main (String args[]) {

//

try{

processFile(*filename*);

} catch (FileNotFoundException e){

System.out.println("Cannot find file : " + filename);

}

}

}

When you run ThrowsDemo.java, with a filename as the parameter that does not exist, the following would be printed.

Inside processFileMethod...

Cannot find file (filename)

**Using finally**

In a program whenever an exception occurs, the execution of the program stops. There could be situations where you establish a connection with a database and an exception occurs. The program terminates, but the connection with the database is still open. This is not desirable from security point of view. To handle such situations, finally blocks should be used.

Irrespective of whether the exception occurs or not, the code inside the finally blocks should be used. This is because both under the situation of exception and non-exception, the control will go to the finally block. The finally block is optional, though usually the cleanup code like closing the files, or closing the database connections are kept in the finally block.

Example: FinallyDemo.java

class FinallyDemo{

public static void main(String args[])

{

try{

int x=Integer.parseInt(args[0]);

int y=Integer.parseInt(args[1]);

System.out.println("Dividing two numbers");

int z=x/y;

//Following line will be skipped if exception occurs

System.out.println("The result is " + z);

}

catch (ArithmeticException e)

{

System.out.println("The denominator was zero");

}

finally

{

System.out.println("In the finally block");

}

}

}

When you will run FinallyDemo.java program along with command line parameter 10 and 0, it will throw ArithmeticException. If the parameters are given as 10 and 5 then Exception will not be thrown. But in both the cases, the finally block will be executed.

**Built-in Exceptions**

You know that the package java.lang is implicitly imported into all Java programs. The java.lang package contains the exception classes, most of which are of type RuntimeException. Therefore, all these exception classes are automatically available to our programs. The run time exceptions are further divided into two categories: 'checked exceptions' and 'unchecked exceptions'.

Checked exceptions are exceptions that should be specified in the throws list of the methods that may throw these exceptions. The complier checks to see whether any exceptions are specified or not. If they are not specified, then the compiler gives a compilation error. Following are some of the checked exceptions:

* ClassNotFoundException
* NoSuchMethodException
* InterruptedException
* IllegalException
* InstantiationException

Unchecked exceptions are exceptions that may not be included in a method's throws list. A method may or may not throw these exceptions. The complier does not check for these exceptions. Hence, these exceptions are called unchecked exceptions.

Following are some of the unchecked exceptions:

* ArithmeticException
* ArrayIndexOutOfBoundsException
* NegativeArraySizeException
* ClassCastException
* NullPointerException
* SecurityException

**User-defined Exceptions**

Although the Java language had provided a list of exceptions to handle various situations, it has also provided the facility of creating your own exceptions. There can be situations when you want to handle a specific situation and the available exceptions cannot fulfill your need. For example, you are writing a java program for a company, which recruits people within the age group of 21 and 60 only. In such a case you can create your own exception class and cause an object of this exception class to be thrown in case the age of the incumbent is either less than 21 or greater than 60. You have to subclass your exception class from the class Exception.

In the TestException.java, first a user-defined exception called, MyException is created by extending the class Exception. Then a class TestException is created which contains a method called checkAge(), that checks whether the age is between 21 and 60. The checkAge() method is capable of throwing the exception of type MyException. If checkAge() method is invoked with an invalid age, it throws an object of the MyException class and the exception is later handled, thus displaying the error message.

Example: TestException.java

class MyException extends Exception {

String s;

MyException() {}

MyException(String str) {

super(str);

}

}

public class TestException {

public void checkAge(int age) throws MyException {

if(age>=21 && age<=60){

System.out.println(“valid age”);

}

else

{

throw new MyException(“Invalid age”);

}

}

public static void main(String args[]){

TestException e = new TestException();

try{

e.checkAge(25);

e.checkAge(18);

}

catch (MyException me) {

System.out.println(“caught “+ me);

}

}

}

Thus, you have seen how you can create your own exceptions and apply them in your programs to handle very specific conditions.

## Another example of User Defined Exception

##### Called Program

class AgeException extends Exception {

int age;

AgeException(String message) {

super(message);

}

AgeException() {

super();

}

}

class OutOfAgeLimitException extends AgeException {

int ageLimit;

OutOfAgeLimitException(int ageLimit, String message) {

super(message);

this.ageLimit = ageLimit;

}

OutOfAgeLimitException(String message) {

super(message);

}

}

class TooYoungException extends OutOfAgeLimitException {

TooYoungException(int age, int ageLimit, String message) {

super(ageLimit, "You are too young to " + message + ".");

this.age = age;

}

TooYoungException() {

super("too young");

}

}

class TooOldException extends OutOfAgeLimitException {

TooOldException(int age, int ageLimit, String message) {

super(ageLimit, "You are too old to " + message + ".");

this.age = age;

}

TooOldException() {

super("too old");

}

}

class IllegalAgeFormatException extends AgeException {

IllegalAgeFormatException(String message) {

super(message);

}

IllegalAgeFormatException() {

super("Illegal age format");

}

}

class NegativeAgeException extends IllegalAgeFormatException {

NegativeAgeException(String message) {

super(message);

}

NegativeAgeException(int age) {

super("Age must be nonnegative.");

this.age = age;

}

}

##### Callable Program

import java.io.\*;

public class AgeExceptionTest {

static PrintWriter out = new PrintWriter(System.out, true);

static void rideRollerCoasterAtAge(int age)

throws NegativeAgeException, OutOfAgeLimitException {

out.println("Trying to ride a roller coaster at age " +

age + "...");

if (age < 0)

throw new NegativeAgeException(age);

else if (age < 5)

throw new TooYoungException(age, 5,

"ride a roller coaster");

else if (age > 45)

throw new TooOldException(age, 45,

"ride a roller coaster");

out.println("Riding the roller coaster....");

}

public static void main(String args[]) {

int ages[] = {-3, 2, 10, 35, 65};

for (int i = 0; i < ages.length; i++)

try {

rideRollerCoasterAtAge(ages[i]);

out.println("Wow! What an experience!");

} catch (OutOfAgeLimitException e) {

out.println(e.getMessage());

if (ages[i] < e.ageLimit)

out.println((e.ageLimit - ages[i]) +

" more years and you'll be able to try it.");

else

out.println((ages[i] - e.ageLimit) +

" years ago riding it was like a piece of cake.");

} catch (NegativeAgeException e) {

out.println(e.getMessage());

} finally {

out.println();

}

}

}

## Advantages of Exceptions

Now that you know what are exceptions and how to use them, your next step should be to know the advantages of using exceptions in programming.

**Advantage 1: Separating Error-Handling Code From "Regular" Code**

All is well and good if the program runs normally. But what if you suddenly find certain things start happening outside the main programming logic? Naturally, it becomes a little tiresome working with the program since detecting error, reporting and handling tend to lead to codes that are confusing if the programming is done in a traditional fashion. Exceptions help in separating the details of what to do in such cases. Take a close look at the following example. Note that the pseudocode method has been used here to read an entire file into memory:

readFile {

open the file;

determine its size;

allocate that much memory;

read the file into memory;

close the file;

}

What is the problem with this function? The problem is that some potential errors are ignored by this function:

* Suppose the file does not open. What happens then?
* Suppose a situation where you cannot determine the length of the file. What happens then?
* Suppose a situation where enough memory cannot be allocated. What happens then?
* What if the read fails?
* What if the file cannot be closed?

More codes must be contained in the readFile function. Why? Because otherwise, error detection, reporting and handling become very tedious processes. Take a close look at the following ReadFile.java example. This illustrates what the function might actually look like:

class ReadFile{

errorCodeType readFile {

initialize errorCode = 0;

open the file;

if (theFileIsOpen) {

determine the length of the file;

if (gotTheFileLength) {

allocate that much memory;

if (gotEnoughMemory) {

read the file into memory;

if (readFailed) {

errorCode = -1;

}

} else {

errorCode = -2;

}

} else {

errorCode = -3;

}

close the file;

if (theFileDidntClose && errorCode == 0) {

errorCode = -4;

} else {

errorCode = errorCode and -4;

}

} else {

errorCode = -5;

}

return errorCode;}}

}

What is happening here? Take a close look at the readLine() function. This function contains the original seven lines, error detection and reporting codes. This is fine except the fact that it leads to some confusion regarding determining whether the code does the right thing. This is because the logical flow of the code is lost. Also, can you tell for certain whether the file is really being closed if the function fails to allocate enough memory? Or can you ensure that the code still continues to do the right things even several months after it was written? Most of the times these little nuggets are ignored and as a result, the program itself ultimately crashes!

The advantage of using exceptions is that you can write the main flow of your program and yet deal with the exceptions in other places. Let us suppose the readFile function uses “except” (in place of the error-management techniques that are traditionally used). Now take a close look at the following:

readFile {

try {

open the file;

determine its size;

allocate that much memory;

read the file into memory;

close the file;

} catch (fileOpenFailed) {

doSomething;

} catch (sizeDeterminationFailed) {

doSomething;

} catch (memoryAllocationFailed) {

doSomething;

} catch (readFailed) {

doSomething;

} catch (fileCloseFailed) {

doSomething;

}

}

**Note:** Exceptions help you in organizing your work more effectively, not in the actual process of detecting, reporting or handling errors.

**Advantage 2: Propagating Errors Up the Call Stack**

Error-reporting up the call stack of methods can be distributed by exceptions. Consider the following:

The main program makes calls in a series of nested methods where the readFile is the third method.

Method1 calls method2 and finally the call to readFile is made by method3.

method1 {

call method2;

}

method2 {

call method3;

}

method3 {

call readFile;

}

Now think of method1 as the only relevant method in the errors that might take place within the readFile. Method 2 and Method 3 would have been forced by the error-notification techniques that are traditional in nature. These techniques will send the error codes returned by the readFile. This will be sent up the call stack. This will continue till the error codes finally reach method1, which is the interested method.

method1 {

errorCodeType error;

error = call method2;

if (error)

doErrorProcessing;

else

proceed;

}

errorCodeType method2 {

errorCodeType error;

error = call method3;

if (error)

return error;

else

proceed;

}

errorCodeType method3 {

errorCodeType error;

error = call readFile;

if (error)

return error;

else

proceed;

}

In order to find any methods interested in handling a particular exception, the JRE searches backward through the call stack. Any exceptions thrown within a method can be ducked by the method itself. This allows a method further up the call stack to catch it. The advantage is that this allows only those methods that are interested in detecting and catching errors to do so, leaving the others aside.

method1 {

try {

call method2;

} catch (exception e) {

doErrorProcessing;

}

}

method2 throws exception {

call method3;

}

method3 throws exception {

call readFile;

}

Ducking an expression requires some effort on part of the methods in-between (consider the pseudocode example). The “throws” clause must specify any checked exceptions that can be thrown within a method.

**Advantage 3: Grouping and Differentiating Error Types**

The grouping or categorizing of exceptions is a natural outcome of the class hierarchy since all exceptions thrown within a program are objects. The exception classes are defined in the Java platform. Note that these are examples of group or related exception classes. They are defined in the java.io – IOException and its descendants. The most general exception is the IOException. It represents any type of errors that can take place while an I/O operation is being performed. More specific errors are represented by its descendants. Consider how you will represent the case of a file that cannot be located on the disk. Use the FileNotFoundException.

What about a very specific exception? The specific handler/s can handle this while a method will be used to write it. The following handler can only handle only one type of exception, since the FileNotFoundException class does not have any descendants.

Specific handlers can be written by a method. These handlers can then handle a very specific expression. No descendants are available for the FileNotFoundException class. Therefore, you handle just one type of exception with the usage of the following handler in this example:

catch (FileNotFoundException e) {

...

}

An exception can be caught by a method. This will be caught on the basis of its group or general type, or by specifying any of the exception’s superclasses in the catch statement. To catch all the I/O exceptions, an exception handler will specify an IOException, irrespective of their specific types.

catch (IOException e) {

...

}

All the I/O exceptions will be caught by the handler. The FileNotFoundException, EOFException will also be caught. How can you find the details of what happenned when you are querying the argument while it was being passed through the exception handler. Consider the following case.

catch (IOException e) {

e.printStackTrace(); //Output goes to System.err.

e.printStackTrace(System.out); //Send trace to stdout.

}

Set up an exception handler. This exception handler will handle any exception with the handler.

catch (Exception e) { //A (too) general exception handler

...

}

The Exception class is located at the top of the Throwable class hierarchy. Hence, the handler will catch both the exceptions as well the ones, it is intended to catch. This way of handling exceptions is useful when the program is intended for simple usage like printing out error messages for the user and then exit.

For general (and usually more complicated) cases, the exception handlers should be more specific. This is so because the first thing a handler must do is to determine what type of exception took place. The handler than decides the best recovery strategy. Any possibility must be accommodated by the handler and by not catching specific errors. Note that the code itself tends to become very error-prone if the handlers themselves are too general in nature. Why does this happen? It is because these handlers catch and handle exceptions for which they were not handed, since the programmer did not anticipate them.

Groups of exceptions can be created and handled in a general fashion. The specific exception type to differentiate and handle exceptions in exact manner can also be used.

**Summary**

This session has covered many of the details having to do with exception handling and Assertion in Java.  By now, you should know that the use of exception handling is not optional in Java, and you should have a pretty good idea how to use exception handling in a beneficial way. Along the way, the chapter has included the following topics:

* Exceptions come in two flavors: checked and unchecked.
* Checked exceptions include all subtypes of Exception, excluding classes that extend Runtime Exception.
* Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using the throws keyword, or handle the exception with an appropriate try/catch.
* Subtypes of Error or Runtime Exception are unchecked, so the compiler does not enforce the handle or declare rule. You are free to handle them, and you are free to declare them, but the compiler does not care one way or the other.
* If you use an optional finally block, it will always be invoked, regardless of whether an exception in the corresponding try is thrown or not, and regardless of whether a thrown exception is caught or not.
* You can create your own exceptions, normally by extending Exception or one of its subtypes. Your exception will then be considered a checked exception and the compiler will enforce the handle or declare rule for that exception.
* All catch blocks must be ordered from most specific to most general. For example, if you have a catch clause for both IO Exception and Exception, you must put the catch for IO Exception first (in order from top to bottom in your code). Otherwise, the IO Exception would be caught by catch(Exception e), because a catch argument can catch the specified exception or any of its subtypes!