A Non-Parameterized Exception Handler Usage

Mr. Chetan Singh Khinchi

Assistance Professor, Department of FCA(Computer Applications)

Acropolis Institute of Technology and Research Indore

Madhya Pradesh , India.

cskhinchi@gmail.com

Abstract

Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional events requiring special processing – often changing the normal flow of program execution. It is provided by specialized programming language constructs or computer hardware mechanisms. In general, an exception is handled (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an exception handler. If exceptions are continuable , the handler may later resume the execution at the original location using the saved information. For example, a floating point divide by zero exception will typically, by default, allow the program to be resumed, while an out of memory condition might not be resolvable transparently .Till date this is done by providing the default handler which abnormally terminates the program and handles the exception on programmers behalf and the other way is provided in the form of a parameterized handler which always needs a specific parameter to handle the exception occurred. This paper provides a way to declare and define a "Non-Parameterized Exception Handler" without disturbing the conventions of Java but provides a new way to handle exceptions automatically upon their occurrence in a program. This adds to the efficiency of the code as well as the programmer is free from writing some specific code over and over in a catch block. This could be more helpful when there is the possibility of different types of exceptions to occur at the same time in a same program , because multiple catch statements will be used to handle such exceptions with different parameters to be passed.

The paper is organized as follows: **Section 1** Introduction to Exceptions. Types of Exceptions in Java . Exception Hierarchy . Methods related to Exceptions **. Section 2** Catching Exceptions . **Section 3** Using multiple catch Blocks in Java(to catch various types of exceptions) . **Section 4** The throws/throw Keywords . **Section5** The finally Keyword . **Section 6** The proposal - Providing a Non-Parameterized Exception Handler(Other than the Default and Parameterized Handler). **Section 7** The solution - Why this suggestion ? . **Section 8** Explaining the proposal through a program . **Section 9** Industrial advantages/benefits . **Section 10** Conclusion **. Section 11** Bibliography and References.

1.Introduction

This paper is all about providing a Non-Parameterized Exception Handler in JAVA. This work is done in order to facilitate the programmers in terms of code simplicity , code remembering and making the code execute a bit faster. This paper provides a view on how's the scenario now and what changes could be made and what will be the results. It has been observed that while exceptions occur a developer always seems in difficulty finding the specific name for the exception to be handled and then embedding the handler to handle the exception. The solution later on provided in this paper gives some relaxation to the programmer from the above tasks.

1.1.Exception Introduction

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

1.2.Types of Exceptions in Java

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

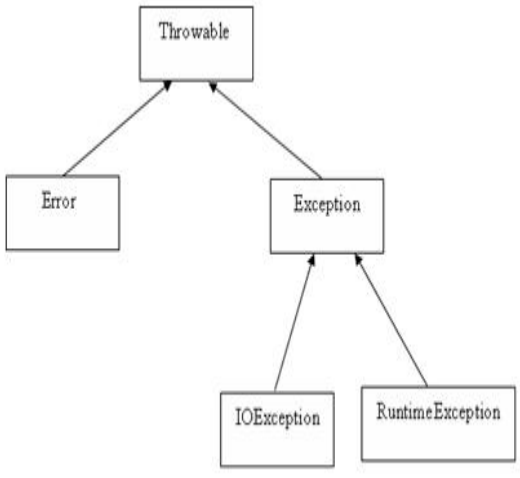
- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compliation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

1.3.Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses : IOException class and RuntimeException Class.



Here is a list of most common checked and unchecked Java's Built-in Exceptions.

Exceptions Methods:

1.4.Methods related to Exceptions

Following is the list of important medthods available in the Throwable class.

1) **Public String getMessage()**
Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

2) **Public Throwable getCause()**
Returns the cause of the exception as represented by a Throwable object.

3) **Public String toString()**
Returns the name of the class concatenated with the result of getMessage()

4) **Public void printStackTrace()**
Prints the result of toString() along with the stack trace to System.err, the error output stream.

5) **public StackTraceElement [] getStackTrace()**
Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

6) **public Throwable fillInStackTrace()**
Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

2.Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}catch(ExceptionName e1)
{
  //Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

  public static void main(String args[]){
    try{
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This would produce following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

3.Using multiple catch Blocks in Java(to catch various types of exceptions)

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
try
{
  file = new FileInputStream(fileName);
  x = (byte) file.read();
}catch(IOException i)
{
  i.printStackTrace();
  return -1;
}catch(FileNotFoundException f) //Not valid!
{
  f.printStackTrace();
  return -1;
}
```

4.The throws/throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws**keyword. The throws keyword appears at the end of a method's signature.
You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
  public void deposit(double amount) throws
RemoteException
  {
    // Method implementation
    throw new RemoteException();
  }
  //Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
  public void withdraw(double amount) throws
RemoteException,
                    InsufficientFundsException
  {
     // Method implementation
  }
  //Remainder of class definition
}
```

5.The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

Example:

```
public class ExcepTest{

  public static void main(String args[]){
    int a[] = new int[2];
    try{
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown :" + e);
    }
    finally{
      a[0] = 6;
      System.out.println("First element value: " +a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

This would produce following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Some special Notes regarding Exception Handling in Java

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

6.The proposal - Providing a Non-Parameterized Exception Handler(Other than the Default and Parameterized Handler)

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}
```

This is the code which you have already gone through . Various catch statements could be written for the code in a single try block. It means that the try block may throw various types of exceptions depending upon the situation. If no matching catch block is found , the exception is handled by the catch statement which contains an object reference of class **"Exception"** . This time it is necessary for the catch block to contain the Exception class object reference variable.

What I am proposing is that it should not be necessary for a programmer to put in the parameter in the catch block if he is going to catch the exception for which the catch block is not written(i.e to provide a non – parameterized exception handler) .To make the scenario more clear go through the following :

Till today it is used as :

Catch(Exception e) **//Exception class is at the top of the exception handling hierarchy**

/*If the try catch throws an exception for which no handler is written(Either intentionally or by mistake) */

```
{
e.printStackTrace();
}
```

So in the above code you have written "Exception e" as the parameter. Since it is the parent class for every exception class it can handle any exception except "Exceptions belonging the class Error".

7.The solution - Why this suggestion ?

My solution is that , this catch must not take any parameter and must be designed so as to accept any non handled exception object as an implicit argument(No display to user), and handle it accordingly. But the condition should be that it must handle exceptions those are subclasses of class Exception.

Some people might consider it same as that of the default handler, but it is not the same , because the default handler in execution takes time and uses various system resources , in order to free the system from the task of executing the default handler and thus limiting the resources of system to be used at the same time, a non parameterized handler is used which will do two things :

1)Keeps resource utilization to a minimum

2)Safeguard your program from terminating abnormally.

I have brought this concept based on **"Constructors"** but it is not completely same as that of the concept of a constructor. If some one needs an immense need of keeping the previous version of catch i.e.

```
Catch(Exception e)
{
//Some code
}
```

One may keep it for some other uses or for the same , but implementing something which I have suggested is not at all disturbing the conventions of java , in fact it adds to the facilities provided to a programmer.

The syntax which I have suggested will look like this if made available to use :

```
Catch()   //Notice that no parameters are used
{
/* Some code – implicitly receives the object of the unhandled exception like (This pointer). */
e.printStackTrace();
}
```

8.Explaining the proposal through a program

```
Class exceptionDemo
{
Try
{
Throw  new NullPointerException("Demo");
}
Catch(ArithmeticException  e)
{
e.printStackTrace();
}
Catch(ArrayIndexOutOfBoundsException e)
{
e.printStackTrace();
}
Catch()
{
e.printStackTrace();
}
}
```

In the above program the try block throws NullPointerException object and searches for the specific catch statement which handles the exception. But no such handler is written specifically for NullPointerException object. So previously the programmer has to write a catch as :

```
Catch(Exception e)
{
e.printStackTrace();
}
```

Or

A default handler was used by the program , but according to my suggestion what could be done now is , that we can write a default handler for the same , as:

```
Catch()
{
e.printStackTrace();
}
```

Where e should be passed implicitly to the catch block in the same way as a this pointer is passed to a method or the constructor implicitly.This obviously will save the extra work done by the machine previously.

9.Industrial(Futuristic-Applications)
Advantages/Benefits

1)Programmers will be free from remembering the types of exception names.
2)Programmers do not have to focus on the parameters to be passed to a handler which will now be done by the machine automatically
3)This concept adds to the already existing conventions of java and does not effect any pre-existing conventions of java, so programmers do not have to bother about any other conventions in specific.
4)Easy to understand and use.
5)Correct use of existing facilities and technology.
6)Makes the code more simple and increases readability and also increases understanding of the program.
7)Provides a choice to the programmer (Adds to the number of choices)

10.Conclusion

In today's scenario it is very important to keep things as simple as possible . Because programmers are already overburdened with a lots of work at workplace. So keeping that in mind above solution to a simple issue makes code simpler and easy to understand without changing or disturbing the other conventions of java .

11.Bibliography and References

- Herbert Scildt (The Complete Reference)-Updated to cover J2SE 1.4
- Thinking in JAVA by Bruce Eckel
- JAVA PUZZLERS TRAPS PITFALLS AND CORNER CASES by Joshua Bloch and Neal Gafter.
- Head First Java by Kathy Sierra and Bert Bates - Second Edition covers Java 5.0
- Wikipedia (Internet)
- Rose.India .com (Internet) and many more java related sites (Internet)
- Java Enterprise Best Practices – By O'Reilly Java Authors(Tim O'Reilly).