RESEARCH ARTICLE OPEN ACCESS

An Exception Monitoring Using Java

Jyoti Kumari, Sanjula Singh, Ankur Saxena
Amity University Sector 125
Noida
Uttar Pradesh – India

ABSTRACT
Many programmers do not check for all possible errors for each method as this can make code unintelligible if each method invocation checks for all possible errors before executing next statement. This creates perplexity between correctness (checking for all possible errors) and clarity (not cluttering the basic flow of codes with many error checks).So, to cope up with this, a midway is required and this requirement is fulfilled by EXCEPTIONs. Java provides an efficient way to handle unexpected conditions which occur during execution of the program. When there is an error or bug in the program then the program terminates as soon as the error is encountered. This leaves the program in an inconsistent state. For avoiding such situation, Java makes use of Exception Handling which is of a great advantage so that whenever there is an exceptional condition, the program handles it in a manner and continues with the execution of the program without leaving it in an inconsistent way. java.lang. Throw able class handles the whole concept of exceptions and errors.
In this paper we are focused on the advantages of Exception handling and its implementation in programs to make it robust. Exception provides a clean way to check for errors without cluttering code. We also developed user defined exceptions to make programs more reliable. Exception handling is important feature for software fault tolerance that enables developers to produce reliable and robust software systems.
Keywords:- Exception, Errors, try, catch, finally.

I. INTRODUCTION

Java is a modern, evolutionary computing language that combines an elegant language design with powerful features that were previously available primarily in specialty languages. In addition to the core language components, Java software distributions include many powerful, supporting software libraries for tasks such as database, network, and graphical user interface (GUI) programming.

It provides the powerful exception handling and type checking mechanism as compare to other programming languages. [1] An exception is an abnormal condition which occurs during execution of program. A java exception is an object which describes the error which has evolved in program while execution. Java provides an efficient way to handle unexpected conditions that can occur in the program. When there is an error or bug in the program then the program terminates as soon as an error is encountered leaving the program in an in consistent state, to avoid this Java makes use of Exception. Exceptions are first-class objects in Java. Like normal objects, they can be defined by classes, instantiated, assigned to variables, passed as parameters, etc. [2] Exception Handling mechanism to a great advantage so that where ever there is an exceptional condition then the program handles it gracefully and continues program execution. [3]

Various reasons are responsible for generating exception, they are:

- A user has entered invalid data.
- A file that needs to be opened cannot be finding.
- A network connection has been lost in middle of communication.
- JVM had run out of memory.

Some exceptions are caused by user errors and some of programmer error. Exceptions can be also caused by the physical resources that have failed in some manner.

NOTE: Run time errors are handled through Exception handling routines of JAVA.

Some typical cases of exception handling where exception may occur

- int a=10/0; //**ArithematicException**
- String s=null;

  System.out.println(s.length());
  //**NullPointerException**
- String s="abc";

  int i=Integer.parseInt(s);
  //**NumberFormatException**
- int a[]=new int a[5];

  a[10]=5;
  //**ArrayIndexOutOfBoundsException**

Problem without exception:

```
class  jyoti
{
public static void main(String a[])
{
int d=10/0;
System.out.println("Hello");
}
}
```

output of simple.java

Java.lang.AirthmeticException: / by zero

Solution with exception handling

```
class  sanjula
{
public static void main(String a[])
{
try
{
int d=10/0;
}catch(ArithmeticException e)
{
System.out.println(e);
}
System.out.println("Hello");
}
}
```

output of sample.java

java.lang.AirthmeticException: /Zero

Hello

Exception Handling Mechanism

The main purpose of Exception handling mechanism is to provide a way of detecting and reporting "Exception circumstances" so that appropriate action can be taken. [4] The mechanism suggests incorporation of a separate error handling code that performs the following task:

1. Finding the problem i.e. **Hit the Exception**
2. Informing that error has occurred i.e. **Throw the Exception**
3. Receiving the error information i.e. **Catch the exception**
4. Take corrective action i.e. **Handle the exception**

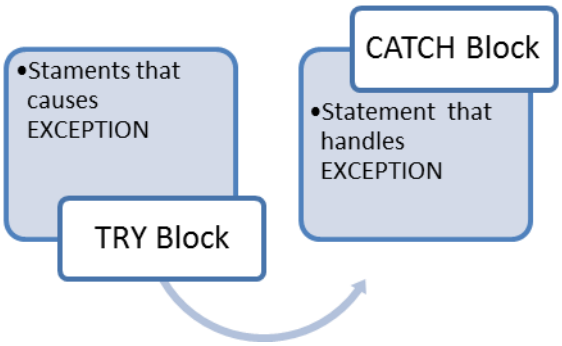

Figure 1: Exception Handling Mechanism

Types of exception:

- **Checked exceptions:** The checked exception is typically generated by the user. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot be ignored at compilation time.
- **Runtime exceptions:** The runtime exception occurs when the programmer probably avoided. Unlike checked exceptions, runtime exceptions are usually ignored at the time of compilation.
- **Errors:** Errors are not exceptions , but problems that arise beyond the control of the programmer. These are ignored in the code because you can rarely do anything about an error. Example, if a stack overflow occurs, an error can arise. These are also ignored at the time of compilation.

Mainly there are two types of errors:

1. **Compile time errors**: These errors occur due to violation of programming language's syntax rules.
2. **Run-time errors:** These errors occur during execution of program. When the

JAVA interpreter encounters an error during runtime it throws an exception. [5]

try and catch

When there is any possibility of an exception being generated in a program, it is better to handle it explicit manner. This can be achieve with the *try* and *catch* keywords. The code sequence, which needs to be guarded, should be placed inside the *try* block. The *catch* clause should immediately follow the *try* block. Each catch block is an exception handler that handles the type of exception indicated by its argument. [6] The *catch* clause can consist of statements explaining the cause of the exception generated. The scope of each catch is restricted to a *try* block. The execution of the program can be continued once the exception has been handled.

Some points of exception:

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

Syntax:

try

{

<code>

} catch (<exception type> <parameter>)

{

<statements>

}

}

try with Multiple catch

Multiple *catch* statements are required in the program when one than one exception is generated by a program. These *catch* statements are searched by the exception thrown in order. The first matches *catch* clause is executed. Finally, if the exception does not find a multiple catch clause, it is passed on the default handler.

Syntax:

try

{

<code>

} catch (<exception type1> <parameter1>)

{

<statements>

}

}

catch (<exception type2> <parameter2>)

{

<statements>

}

}

At a time only one Exception is occurred and at a time only one catch is executed.

finally Block

The finally block is used for freeing resources, cleaning up, closing connections. The finally block contains code that will be run whether or not an exception is thrown [7] in a try block. 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.

Syntax:

try

{

<code>

} catch (<exception type1> <parameter1>)

{

<statements>

}

}

finally

{ // finally block

<statements>

```
}
```

Nested try catch

Some situation may arise where a part of a block may cause one error and the entire block itself may cause another error. [8] These cases, Exception handlers have to be nested.

Syntax:

```
try
{
statement1;
statement2;
        try
        {
statement1;
statement2;
}
catch(Exception e)
{
}
}
catch (Exception e)
{
}
```

II. IMPLEMENTATION

Customized exception is necessary to handle abnormal conditions of application created by the programmer. The advantage of creating such an exception class is that, according to situation defined by the user an exception can be thrown. That is possible to set any condition or value to a variable and generate user-defined exception.

Declaring you own Exception: You can create your own exceptions in Java. Follow the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable class.
- When we want to write a checked exception that is automatically forced by the Handle or Declare Rule, you need to extend the Exception class.
- When we want to write a runtime exception, you need to extend the RuntimeException class.

You can define our own Exception class as below:

```
class MyException extends Exception
{
}
```

You need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions.

Following example InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception.

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends Exception
{
  private double amount;
  public InsufficientFundsException(double amount)
  {
    this.amount = amount;
  }
  public double getAmount()
  {
    return amount;
  }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java

import java.io.*;

public class CheckingAccount

{

  private double balance;

  private int number;

  public CheckingAccount(int number)

  {

    this.number = number;

  }

  public void deposit(double amount)

  {

    balance += amount;

  }

  public void withdraw(double amount) throws
InsufficientFundsException

  {

    if(amount <= balance)

    {

      balance -= amount;

    }

    else

    {

      double needs = amount - balance;

      throw new
InsufficientFundsException(needs);

    }

  }

  public double getBalance()

  {

    return balance;

  }

  public int getNumber()

  {

    return number;

  }

}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java

public class BankDemo

{

  public static void main(String [] args)

  {

    CheckingAccount c = new
CheckingAccount(101);

    System.out.println("Depositing 5000...");

    c.deposit(500.00);

    try

    {

      System.out.println("\nWithdrawing 1000...");

      c.withdraw(100.00);

      System.out.println("\nWithdrawing 6000...");

      c.withdraw(600.00);

    }catch(InsufficientFundsException e)

    {

      System.out.println("Sorry, but you are short "
+ e.getAmount());

      e.printStackTrace();

    }

  }

}
```

```
<terminated> BankDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Feb 20, 2013 8:56:13 AM)
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:22)
        at BankDemo.main(BankDemo.java:13)
Depositing 5000...

Withdrawing 1000...

Withdrawing 6000...
Sorry, but you are short 200.0
```

Figure 2: Output of BankDemo .java

III. RELATED WORKS

The exception monitoring system is implemented in Java based on Barat [9] ,which is a front-end for a Java compiler. Barat builds an abstract syntax tree for an input Java program and enriches it with type and name analysis information. It also provides interfaces for traversing abstract syntax trees, based on visitor design pattern in [10]. Barat provides several visitors as basic visitors: Descending Visitor which traverses every AST node in depth-first order and Ouput Visitor which outputs input programs by traversing AST nodes. We can develop a static analyzer by implementing visitors to do necessary actions or operations at visiting AST nodes by extending basic visitors.

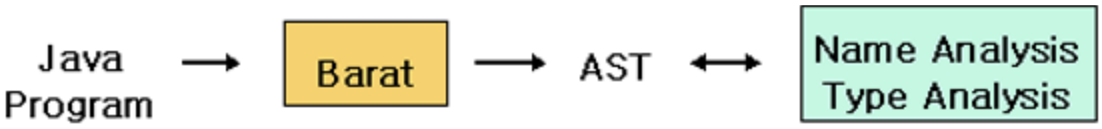

Figure 3: Architecture of BARAT

Users can select files from the package, the window displays a list of exceptions, handlers and methods based on the static analysis information from the menu window. Then, users can select only focus on interesting exceptions when debugging. interesting exceptions, handlers and methods among them. By selecting options, users can get only interesting trace information an

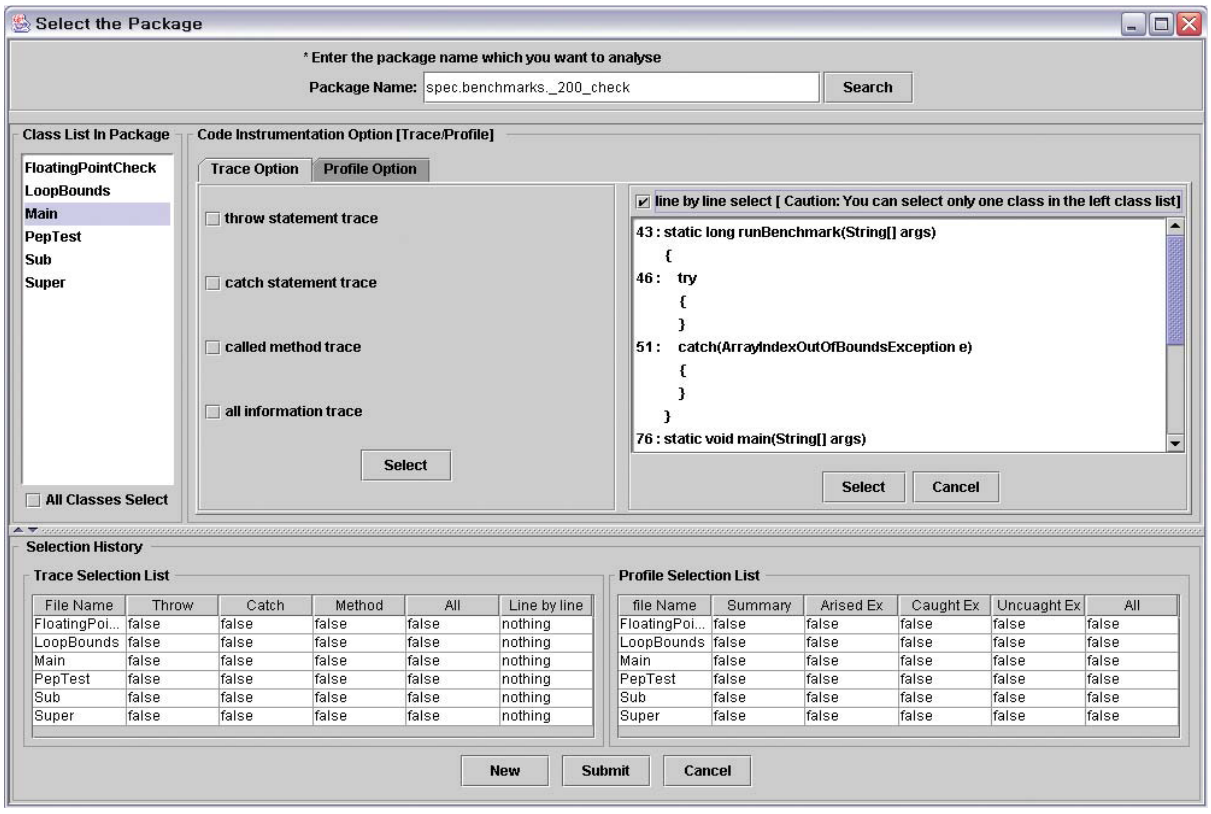

Figure 4: Menu Window

To provide users with options, they implement a static analyzer to extract exception-related constructs by extending DescendingVisitor. It extracts static information about possible exceptions and methods by analyzing input programs statically. In particular, it extracts static program constructs on exception raising, handling and methods. They also implement a program transformer called TransformVisitor by extending OuputVisitor, which transforms an input program *P* into a program *P'* by inlining codes so as to trace handling and propagation of thrown exceptions according to selected options.

Structure of Transform Visitor

```
Class TransformVisitor extends OutputVisitor{
visitThrow{
// Print the location of thrown exception and its
type
}
visitTry{
// Print the location of try statement
}
visitCatch{
// Print the location of catch statement, the type of
caught exception
// call printStackTrace() method
}
visitMethod{
// Print the method information and record
exception propagation
// via this method
}
}
```

IV. CONCLUSION AND FUTURE WORK

In this paper we focused on advantages of Exception Handling and how it should be implemented. The basic purpose of exception handling is to maintain the normal flow of the application, irrespective of errors or exceptions which might occur during execution. If these exceptions are not handled properly then the application flow will be disrupted during its execution.

Although exception handling is an important aspect of application development but it is often neglected during development time which results into unknown errors during production time which can kill the system and produces results which are unexpected. So, improper exception handling is one of the major causes for real time applications failure and at that point it is very difficult to redesign the application and implement exception handling mechanism.

REFERENCES

[1] A. Trottier, Java 2 Core Language Little Black Book, Paraglyph Press, 2002.

[2] K. Y. Sukyoung Ryu, "Exception Analysis for Multithreaded Java Programs," Daejeon,Korea.

[3] H. O. a. B.-M. Chang, "An Exception Monitoring System for Java," *RISE,* pp. 71-81, 2005.

[4] A Saxena, TechnoWorld II, Noida: GRAM, 2014.

[5] M. (. Ben-Ari, "Compile and Runtime errors in Java," Rehovot, 2007.

[6] "The Java Tutorials," Oracle, [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html.

[7] "Use of 'finally' in Java," JavaSamples, [Online]. Available: http://www.java-samples.com/showtutorial.php?tutorialid=296.

[8] "Java Exception Handling: Nested try block example - javatpoint," JavaTpoint, [Online]. Available: http://www.javatpoint.com/nested-try-block. [Accessed 25 12 2013].

[9] A. S. B. Bokowski, "Barat A Front-End for Java," 1998.

[10] R. H. R. J. a. J. V. E. Gamma, Design Patterns:Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.