RGPVNOTES.IN

Subject Name: **Principles of Programming Languages**

Subject Code: **IT-5002**

Semester: **5th**

Unit 5

Exception Handling

Exceptions - An exception (or exceptional event) is a problem that arises during the execution of a program. When an exception occurs the normal flow of the program is disrupted and the program/application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an 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.

Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Example

```
class ExceptionPropagation{
 void m(){
   int data = 10/0;
 }
 void n(){
   m();
 }
 void p(){
  try{
   n();
  }catch(Exception e){
     System.out.println("exception handled");
  }
 }
 public static void main(String args[]){
  ExceptionPropagation obj = new ExceptionPropagation();
  obj.p();
  System.out.println("normal flow...");
 }
}
```

Output:

```
exception handled
 normal flow...
```

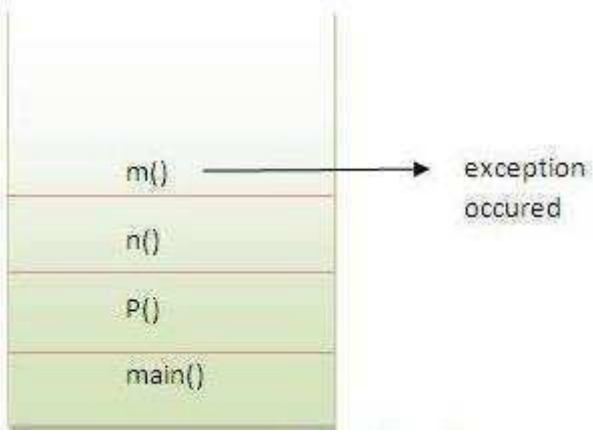

Figure 5.1 Call Stack

In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled. Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Exception handling in C++

Exceptions are runtime anomalies that a program encounters during execution. It is a situation where a program has an unusual condition and the section of code containing it can't handle the problem. Exception includes condition such as division by zero, accessing an array outside its bound, running out of memory, etc. In order to handle these exceptions, exception handling mechanism is used which identifies and deal with such condition. Exception handling mechanism consists of following parts:

1. Find the problem (Hit the exception)
2. Inform about its occurrence (Throw the exception)
3. Receive error information (Catch the exception)
4. Take proper action (Handle the exception)

C++ consists of 3 keywords for handling the exception. They are

1. **try:** Try block consists of the code that may generate exception. Exceptions are thrown from inside the try block.
2. **throw:** Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.
3. **catch:** Catch block catches the exception thrown by throw statement from try block. Then, exceptions are handled inside catch block.

Syntax

```
try
{
    statements;
    ... ... ...
    throw exception;
}

catch (type argument)
{
    statements;
    ... ... ...
}
```

Multiple catch exception

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

Syntax

```
try
{
   body of try block
}

catch (type1 argument1)
{
   statements;
   ... ... ...
}

catch (type2 argument2)
{
   statements;
   ... ... ...
}
... ... ...
... ... ...
catch (typeN argumentN)
{
   statements;
   ... ... ...
}
```


Example of exception handling

C++ program to divide two numbers using try catch block.

```
#include <iostream>
using namespace std;
int main()
{
   int a,b;
   cout << "Enter 2 numbers: ";
   cin >> a >> b;
   try
   {
      if (b != 0)
      {
         float div = (float)a/b;
         if (div < 0)
            throw 'e';
         cout << "a/b = " << div;
      }
      else
         throw b;

   }
```

```
    catch (int e)
    {
       cout << "Exception: Division by zero";
    }
    catch (char st)
    {
       cout << "Exception: Division is less than 1";
    }
    catch(...)
    {
       cout << "Exception: Unknown";
    }
    getch();
    return 0;
}
```

This program demonstrates how exceptions are handled in C++. This program performs division operation. Two numbers are entered by user for division operation. If the dividend is zero, then division by zero will cause exception which is thrown into catch block. If the answer is less than 0, then exception "Division is less than 1" is thrown. All other exceptions are handled by the last catch block throwing "Unknown" exception.

Output

```
Enter 2 numbers: 8 5
a/b = 1.6
Enter 2 numbers: 9 0
Exception: Division by zero
Enter 2 numbers: -1 10
Exception: Division is less than 1
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below −

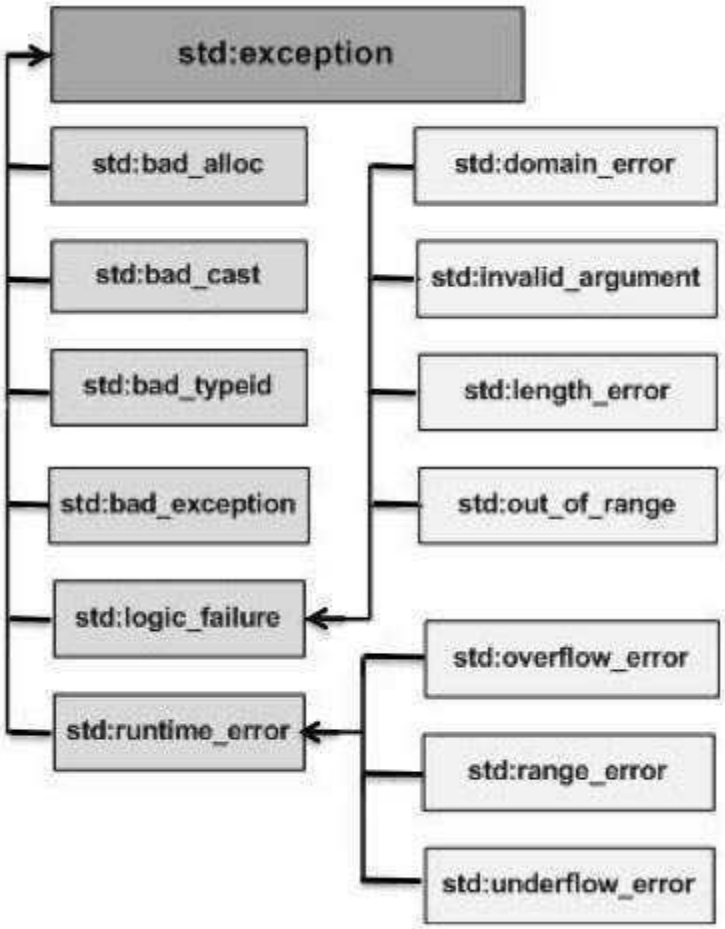

Figure 5.2 C++ Standard Exceptions

Advantages of Exception Handling

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of exceptions
- Exception Reporting
- Identifying Exception Types

Exception Handling in Java

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 abnormal conditions that happen in case of severe failures, these 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.

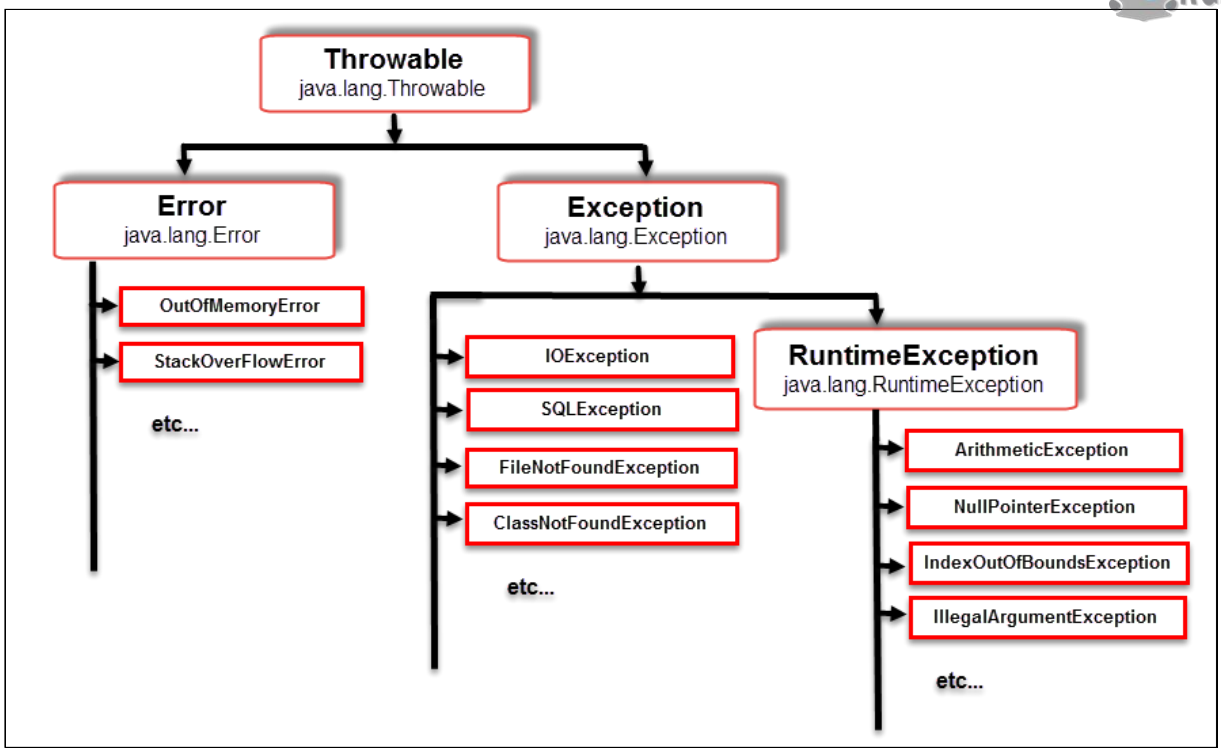

Figure 5.3 Exceptions in Java

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 –

Syntax

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

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally 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 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 will produce the following result −

Output

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

Multiple Catch Blocks

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

Syntax

```
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;
}
```

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.

throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

Example

```
import java.io.*;
public class className {

  public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException();
  }
  // Remainder of class definition
}
```

Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

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 −

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 will produce the following result −

Output

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

Important points

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever 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.

Logic Programming

Logic programming is a computer programming paradigm in which program statements expresses facts and rules about problems within a system of formal logic. Rules are written as logical clauses with a head and a body; for instance, "H is true if B1, B2, and B3 are true." Facts are expressed similar to rules, but without a body; for instance, "H is true."

Some logic programming languages such as Data log and Answer Set Programming (ASP) are purely declarative they allow for statements about what the program should accomplish, with no explicit step-by-step instructions about how to do so. Others, such as Prolog, are a combination of declarative and imperative — they may also include procedural statements such as "To solve H, solve B1, B2, and B3."

- Programs are written in the language of some logic.
- Execution of a logic program is a theorem proving process.
- Prolog, Programming in Logics, is a representative LP language, based on a subset of first order predicate logic. However, logic programming does not equal programming in Prolog, there can be different logic programming languages based on different logics.

Introduction and overview of Logic programming

- A logic program is a specification of a solution to a problem; in addition, it is an executable specification.
- Like LISP, LP is about manipulation of symbols, and thus has potential in AI applications.
- Unlike LISP, computations in LP are reasoning processes.

Logic programming is widely used in parsing, both in natural languages and programming languages. Since the creator of logic programming is also an linguist, it once was widely used in natural language processing (NLP), mostly in parsing natural language processing and generating natural language. Prolog and Natural language analysis is a book about this topic. But nowadays, NLP is mostly occupied by statistic approach. It is really easy to write a parser in Prolog, it is also quite often used to implement new programming. The first interpreter of Erlang Programming Language is written in Prolog. It is also used widely when there are a lot of relations, like in semantic web's RDF manipulation (which you can view it as the root of Knowledge Graph). There are also some libraries to do constrained logic programming which is quite interesting and exciting.

Basic Elements of prolog

Prolog is a logic language that is particularly suited to programs that involve symbolic or non-numeric computation. For this reason it is a frequently used language in Artificial Intelligence where manipulation of symbols and inference about them is a common task.

Prolog consists of a series of rules and facts. A program is run by presenting some query and seeing if this can be proved against these known rules and facts.

Simple Facts

In Prolog we can make some statements by using facts. Facts either consist of a particular item or a relation between items. For example we can represent the fact that it is sunny by writing the program:

sunny.

We can now ask a query of Prolog by asking

?- sunny.

?- is the Prolog prompt. To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.

Facts have some simple rules of syntax. Facts should always begin with a lowercase letter and end with a full stop. The facts themselves can consist of any letter or number combination, as well as the underscore _ character. However, names containing the characters -,+,*,/, or other mathematical operators should be avoided.

Examples of Simple Facts

Here are some simple facts about an imaginary world. /* and */ are comment delimiters

```
john_is_cold.                  /* john is cold */
raining.                       /* it is raining */
john_Forgot_His_Raincoat.      /*  john forgot his raincoat */
```

Rule Statements

- Used for the hypotheses
- Headed Horn clause
- Right side: antecedent (if part)
  - May be single term or conjunction
- Left side: consequent (then part)
  - Must be single term
- Conjunction: multiple terms separated by logical AND operations (implied)

Example Rules

Ancestor (mary,shelley):- mother(mary,shelley).

- Can use variables (universal objects) to generalize meaning:

```
parent(X,Y):- mother(X,Y).
 parent(X,Y):- father(X,Y).
 grandparent(X,Z):- parent(X,Y), parent(Y,Z).
 sibling(X,Y):- mother(M,X), mother(M,Y),
 father(F,X), father(F,Y).
```

Application of Logic Programming

1. Relational database management system.
2. Expert System.
3. Natural language processing.
4. Symbolic Equation solving.
5. Planning
6. Prototyping.
7. Simulation.
8. Programming Language Implementation.

Functional Programming Language Introduction

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional

programming languages include: LISP, Python etc. Functional programming languages are categorized into two groups, i.e. –

- **Pure Functional Languages** – These types of functional languages support only the functional paradigms. For example – Haskell.
- **Impure Functional Languages** – These types of functional languages support the functional paradigms and imperative style programming. For example – LISP.

Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows –

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports **higher-order functions** and **lazy evaluation** features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Functional Programming – Advantages

Functional programming offers the following advantages –

- **Bugs-Free Code** – Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.
- **Efficient Parallel Programming** – Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- **Efficiency** – Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- **Supports Nested Functions** – Functional programming supports Nested Functions.
- **Lazy Evaluation** – Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible.
- The basic process of computation is fundamentally different in a FPL than in an imperative language.
  - In an imperative language, operations are done and the results are stored in variables for later use.
  - Management of variables is a constant concern and source of complexity for imperative programming.
- In an FPL, variables are not necessary, as is the case in mathematics.
- Referential Transparency - In an FPL, the evaluation of a function always produces the same result given the same parameters.

Introduction to 4GL

A fourth-generation programming language (4GL) is any computer programming language that belongs to a class of languages envisioned as advancement upon third-generation programming languages (3GL). Each of the programming language generations aims to provide a higher level of abstraction of the internal computer hardware details, making the language more programmer-friendly, powerful and versatile. While the definition of 4GL has changed over time, it can be typified by operating more with large collections of information at once rather than focusing on just bits and bytes. Languages claimed to be 4GL may include

support for database management, report generation, mathematical optimization, GUI development, or web development. Some researchers state that 4GLs are a subset of domain-specific languages

Advantages of 4GL

1. Programming productivity is increased. One line of 4GL code is equivalent to several lines of 3GL code.
2. System development is faster.
3. Program maintenance is easier.
4. The finished system is more likely to be what the user envisaged, if a prototype is used and the user is involved throughout the development.
5. End user can often develop their own applications.
6. Programs developed in 4GLs are more portable than those developed in other generation of languages.
7. Documentation is improved because many 4GLs are self documenting.

Disadvantages of 4GL

1. The programs developed in the 4GLs are executed at a slower speed by the CPU.
2. The programs developed in these programming languages need more space in the memory of the computer system.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in