Exception Handling

Session 8

Introduction

- Rarely does a program runs successfully at its very first attempt.
- It is common to make mistakes while developing as well as typing a program.
- Such mistakes are categorised as:
  1. syntax errors - compilation errors.
  2. semantic errors– leads to programs producing unexpected outputs.
  3. runtime errors – most often lead to abnormal termination of programs or even cause the system to crash.

Introduction to Exception

- Is a special type of error
- It occurs at runtime in a code sequence
- Abnormal conditions that occur while executing the program cause exceptions
- If these conditions are not dealt with, then the execution can be terminated abruptly

Purpose of Exception handling

- Minimize the chances of a system crash, or abrupt program termination
- For example,

  In an I/O operation in a file. If the data type conversion is not properly done, an exception occurs, and the program aborts, without closing the file. This may damage the file, and the resources allocated to the file may not return to the system

Handling Exceptions

- When an exception occurs, an object that represents that exception is created
- This object is then passed to the method where the exception has occurred
- The object contains detailed information about the exception. This information can be retrieved and processed
- The 'throwable' class that Java provides is the superclass of the Exception class, which is, in turn, the superclass of individual exceptions

Exception handling Model

- Is also known as the ‘catch and throw’ model
- When an error occurs, an ‘exception’ is thrown, and caught in a block
- Keywords to handle exceptions
  - try
  - catch
  - throw
  - throws
  - finally

Structure of the exception handling model

- **Syntax**

```
try { ….  }
catch(Exception e1) { …. }
catch(Exception e2) { …. }
catch(Exception eN) { …. }
finally { …. }
```

Advantages of the ‘Catch and Throw’ Model

- The programmer has to deal with an error condition only where necessary. It need not be dealt with at every level
- An error message can be provided in the exception-handler

‘try’ and ‘catch’ Blocks

- Is used to implement the ‘catch and throw’ model of exception handling
- A ‘try’ block consists of a set of executable statements
- A method, which may throw an exception, can also be included in the ‘try’ block
- One or more ‘catch’ blocks can follow a ‘try’ block
- These ‘catch’ blocks catch exceptions thrown in the ‘try’ block

try' and 'catch' Blocks (Contd…)

- To catch any type of exception, specify the exception type as 'Exception'

  catch(Exception e)

- When the type of exception being thrown is not known, the class 'Exception' can be used to catch that exception
- The error passes through the 'try catch' block, until it encounters a 'catch' that matches it, or the program terminates

Multiple Catch Blocks

- Multiple ‘catch()’ blocks process various exception types separately
- Example

```
try
{   doFileProcessing();
    displayResults();    }
catch(LookupException e)
{   handleLookupException(e);    }
catch(Exception e)
{  System.err.println(“Error:”+e.printStackTrace());   }
```

Multiple Catch Blocks (Contd…)

- When nested ‘try’ blocks are used, the inner ‘try’ block is executed first
- Any exception thrown in the inner ‘try’ block is caught in the following ‘catch’ blocks
- If a matching ‘catch’ block is not found, then ‘catch’ blocks of the outer ‘try’ blocks are inspected
- Otherwise, the Java Runtime Environment handles the exception

Without Error Handling – Example 1

```
class NoErrorHandling{
        public static void main(String[] args){
                int a, b;
                a = 7;
                b = 0;
                System.out.println("Result is " + a/b);
                System.out.println("Program reached this line");
        }
}
```

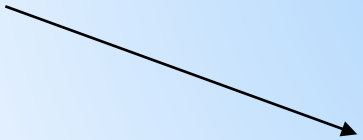
Program does not reach here

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at javaapplication2.JavaApplication2.main(JavaApplication2.java:63)
Java Result: 1
```

Traditional way of Error Handling - Example 2

```
class WithErrorHandling{
        public static void main(String[] args){
                int a, b;
                a = 7;   b = 0;
                if (b != 0){
                        System.out.println("Result is " + a/b);
                }
                else{
                        System.out.println(" B is zero);
                }
                System.out.println("Program is complete");
        }
}
```

Program reaches here



Exceptions

- An exception is a condition that is caused by a runtime error in the program.
- Provide a mechanism to signal errors directly without using flags.
- Allow errors to be handled in one central part of the code without cluttering code.

Exceptions and their Handling

- When the JVM encounters an error such as divide by zero, it creates an exception object and throws it – as a notification that an error has occurred.

- If the exception object is not caught and handled properly, the interpreter will display an error and terminate the program.

- If we want the program to continue with execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then take appropriate corrective actions. This task is known as *exception handling*.

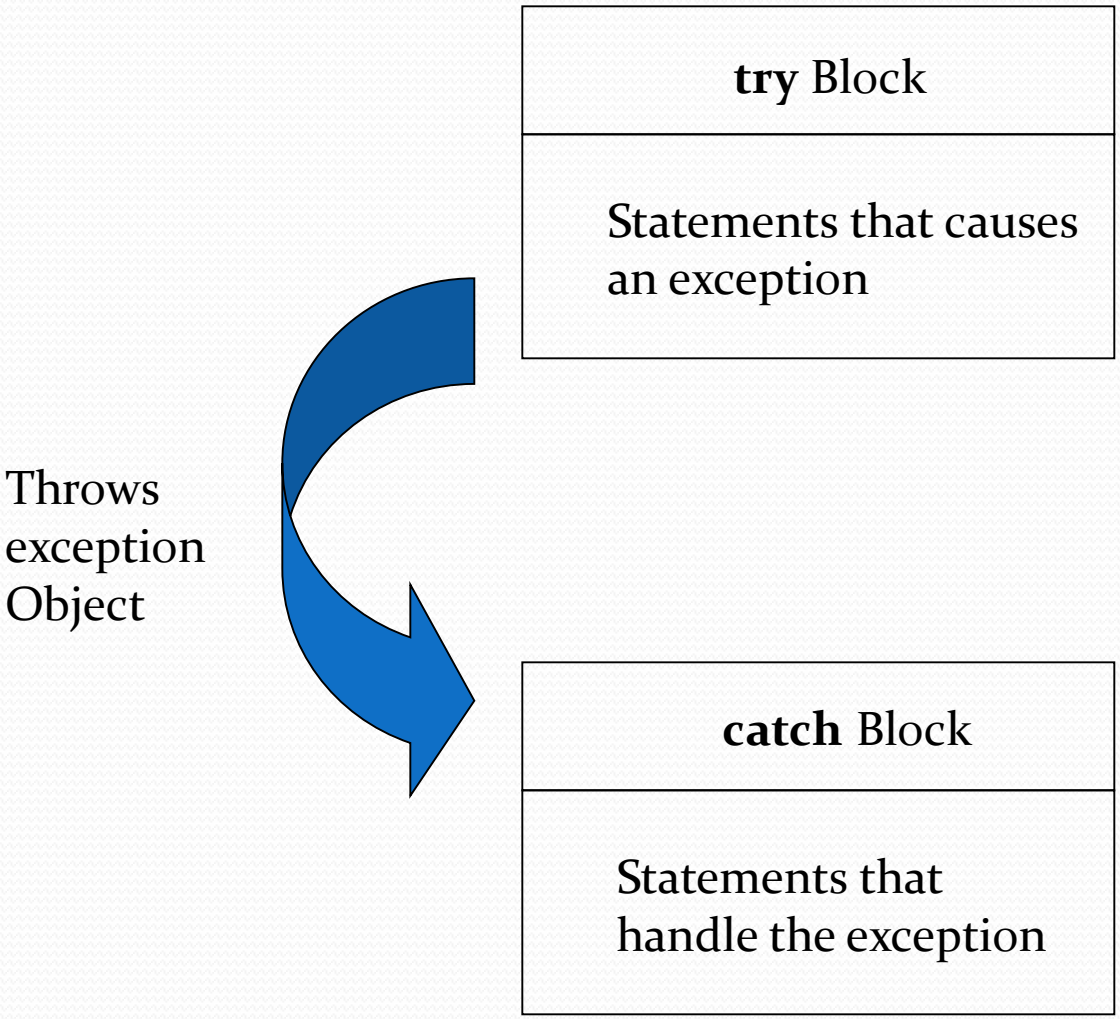
Common Java Exceptions

- ArithmeticException
- ArrayIndexOutOfBoundException
- FileNotFoundException
- IOException – general I/O failure
- NullPointerException – referencing a null object
- OutOfMemoryException
- SecurityException – when applet tries to perform an action not allowed by the browser's security setting.
- StringIndexOutOfBoundException

Exceptions in Java

- A method can signal an error condition by throwing an exception – *throws*
- The calling method can transfer control to a exception handler by catching an exception - *try, catch*
- Clean up can be done by - *finally*

Exception Handling Mechanism



Syntax of Exception Handling Code

```
…
…
try {
      // statements
}
catch( Exception-Type e)
{
      // statements to process exception
}
..
..
```

With Exception Handling - Example 3

```
class WithExceptionHandling{
        public static void main(String[] args){
                int a,b;  float r;
                a = 7;   b = 0;
                try{
                    r = a/b;
                    System.out.println("Result is " + r);
                }
                catch(ArithmeticException e){
                        System.out.println(" B is zero);
                }
                System.out.println("Program reached this line");
        }
}
```

Program will not Reaches here

Program Reaches here

Finding a Sum of Values Passed as Command Line Arguments

```
// ComLineSum.java: adding command line parameters
class ComLineSum
{
  public static void main(String args[])
  {
      int InvalidCount = 0;
      int number, sum = 0;

      for( int i = 0; i < args.length; i++)
      {
            try {
                  number = Integer.parseInt(args[i]);
            }
            catch(NumberFormatException e)
            {
                  InvalidCount++;
                  System.out.println("Invalid Number: "+args[i]);
                  continue;//skip the remaining part of loop
            }
            sum += number;
      }
      System.out.println("Number of Invalid Arguments = "+InvalidCount);
      System.out.println("Number of Valid Arguments = "+(args.length-InvalidCount));
      System.out.println("Sum of Valid Arguments = "+sum);
  }
}
```

Sample Runs

C:>java ComLineSum 1 2

```
Number of Invalid Arguments = 0
Number of Valid Arguments = 2
Sum of Valid Arguments = 3
```

C:>java ComLineSum 1 2 abc

```
Invalid Number: abc
Number of Invalid Arguments = 1
Number of Valid Arguments = 2
Sum of Valid Arguments = 3
```

Multiple Catch Statements

- If a try block is likely to raise more than one type of exceptions, then multiple catch blocks can be defined as follows:

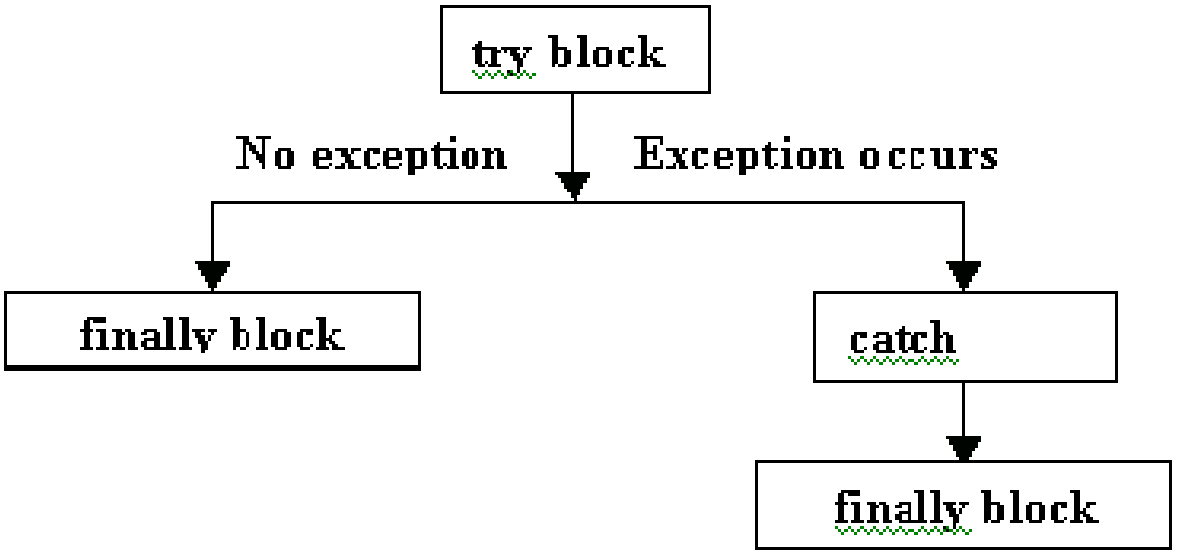
```
...
...
try {
      // statements
}
catch( Exception-Type1 e)
{
     // statements to process exception 1
}
..
..
catch( Exception-TypeN e)
{
     // statements to process exception N
}
...
```

‘finally’ Block

- Takes care of all the cleanup work when an exception occurs
- Can be used in conjunction with a ‘try’ block
- Contains statements that either return resources to the system, or print messages
  - Closing a file
  - Closing a result set (used in Database programming)
  - Closing the connection established with the database

‘finally’ Block (Contd…)

- Is optional
- Is placed after the last ‘catch’ block
- The ‘finally’ block is guaranteed to run, whether or not an exception occurs



Flow of the ‘try’, ‘catch’ and ‘finally’ blocks

finally block

- When a finally is defined, it is executed regardless of whether or not an exception is thrown. Therefore, it is also used to perform certain house keeping operations such as closing files and releasing system resources.

```
...
try {
      // statements
}
catch( Exception-Type1 e)
{
     // statements to process exception 1
}
..
..
finally {
  ....
}
```

User-defined Exceptions with ‘throw’ and ‘throws’ statements

- Exceptions are thrown with the help of the ‘throw’ keyword
- The ‘throw’ keyword indicates that an exception has occurred
- The operand of throw is an object of a class, which is derived from the class ‘Throwable’
- Example of the ‘throw’ statement

```
try{
    if  (flag < 0)
    {
        throw new MyException( ) ;  // user-defined
    }
}
```

User-defined Exceptions with ‘throw’ and ‘throws’ statements (Contd…)

- A single method may throw more than one exception
- Example of the ‘throw’ keyword to handle multiple exceptions

```
public class Example {
    public void exceptionExample( ) throws ExException,
    LookupException  {
        try
        {   // statements     }
        catch(ExException exmp)
        { …. }
        catch(LookupException lkpex)
        { …. }      }       }
```

User-defined Exceptions with ‘throw’ and ‘throws’ statements (Contd…)

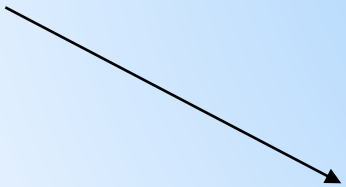
- The ‘Exception’ class implements the ‘Throwable’ interface, and provides some useful features for dealing with exceptions
- Advantage of subclassing the Exception class is that the new exception type can be caught separately from other Throwable types

With Exception Handling - Example 4

```
class WithExceptionCatchThrow{
        public static void main(String[] args){
                int a,b;  float r;  a = 7;   b = 0;
                try{
                    r = a/b;
                    System.out.println("Result is " + r);
                }
                catch(ArithmeticException e){
                        System.out.println(" B is zero);
                        throw e;
                }
                System.out.println("Program is complete");

        }
}
```

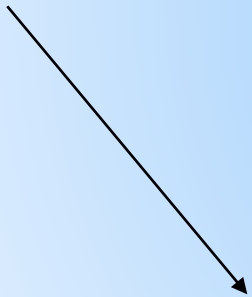
Program Does Not reach here when exception occurs



With Exception Handling - Example 5

```
class WithExceptionCatchThrowFinally{
        public static void main(String[] args){
                int a,b;  float r;  a = 7;   b = 0;
                try{
                    r = a/b;
                    System.out.println("Result is " + r);
                }
                catch(ArithmeticException e){
                        System.out.println(" B is zero);
                        throw e;
                }
                finally{
                 System.out.println("Program is complete");
                }
        }
}
```

Program reaches here



User-Defined Exceptions

- Problem Statement :
  - Consider the example of the Circle class
  - Circle class had the following constructor

```
public Circle(double centreX, double centreY,
                              double radius){
        x  = centreX; y = centreY;  r = radius;
}
```

  - How would we ensure that the radius is not zero or negative?

Defining your own exceptions

```
import java.lang.Exception;
class InvalidRadiusException extends Exception {

        private double r;

        public InvalidRadiusException(double radius){
                r = radius;
        }
        public void printError(){
              System.out.println("Radius [" +  r + "] is not valid");
        }
}
```

Throwing the exception

```
class Circle    {
        double x, y, r;

       public Circle (double centreX, double centreY, double
radius ) throws InvalidRadiusException {
        if (r <= 0 ) {
                throw new InvalidRadiusException(radius);
        }
        else {
                x = centreX ; y = centreY;  r = radius;
        }
      }
}
```

Catching the exception

```
class CircleTest {
    public static void main(String[] args){
     try{
            Circle c1 = new Circle(10, 10, -1);
            System.out.println("Circle created");
     }
     catch(InvalidRadiusException e)
     {
            e.printError();
     }
    }
}
```

User-Defined Exceptions in standard format

```
class MyException extends Exception
{
        MyException(String message)
        {
            super(message); // pass to superclass if parameter is not handled by used defined exception
        }
}
class TestMyException {
…
  try {
        ..
        throw new MyException("This is error message");
        }
        catch(MyException e)
        {
            System.out.println("Message is: "+e.getMessage());
        }
  }
}
```

Get Message is a method defined in a standard Exception class.

Summary

- A good programs does not produce unexpected results.
- It is always a good practice to check for potential problem spots in programs and guard against program failures.
- Exceptions are mainly used to deal with runtime errors.
- Exceptions also aid in debugging programs.
- Exception handling mechanisms can effectively used to locate the type and place of errors.

Summary

- *Try* block, code that could have exceptions / errors
- *Catch* block(s), specify code to handle various types of exceptions. First block to have appropriate type of exception is invoked.
- If no 'local' catch found, exception propagates up the method call stack, all the way to main()
- Any execution of try, normal completion, or catch then transfers control on to *finally* block