Session 17 – Exceptions

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Exceptions
- ✓ Propagations
- ✓ Exception Hierarchy
- ✓ System Defined Exceptions

Exceptions

Introduction to Exceptions

In Java to indicate to a calling method that an abnormal condition has occurred – is where we identify the scenario as 'Exception' being thrown or occurred. When a method encounters an abnormal condition (an *exception condition*) that it can't handle itself, it may *throw* an exception.

The errors we get with the programs can be classified as

- Design/Compile time errors - like syntax/semantic errors
- Runtime errors

Exception Handling is a technique used to monitor for exceptional conditions within your program and transfer the control to special exception-handling code (which you design) if an exceptional condition occurs.Exceptions are anticipatable technical errors/deviations that may occur at runtime. They could be file not found exception, unable to get connection exception and so on. On such conditions java throws an exception.

Java exception handling is used to handle error conditions in a program systematically by taking the necessary action. Exception handlers can be written to catch a specific exception such as "**Number Format exception**", or an entire group of exceptions by using a generic exception handlers. Any exceptions not specifically handled within a Java program are caught by the Java run time environment.

Exception Handling Syntax

```
try {
<code block
   ------------------
>
}catch (<Exception type1><identifier1>){
<Code block>
   ------------------
}catch(<Exception type 2><identifier>){
<Code block>
    ------------------
}
------------
finally { // finally block
<Code block>}
```

Exception Propagation:-

Traditional Way of Propagating Error:

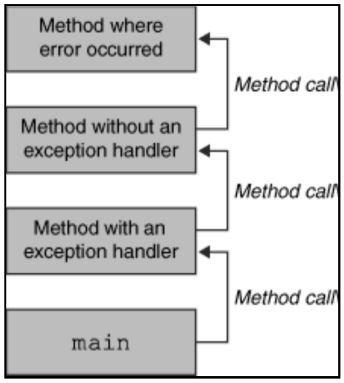
```
method1 {
   errorCodeType error;
   error = call method2;
   if (error)
      doErrorProcessing;
   else
      proceed;
}

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;
```

- Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

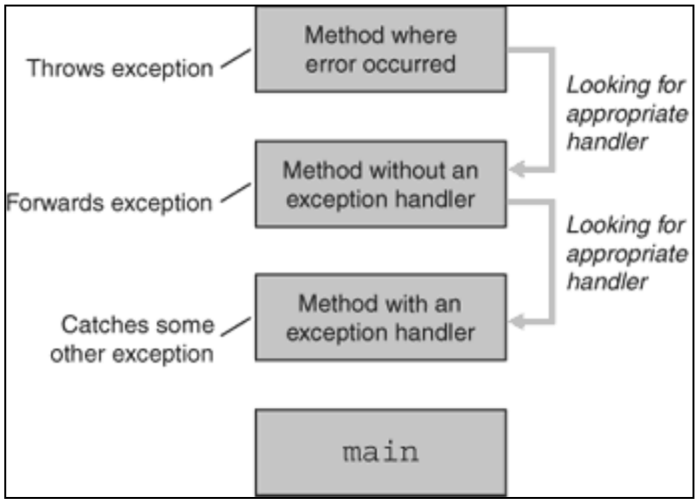
What happens when an Exception occurs – The runtime system searches the call stack for a method that contains an exception handler:
When an exception occurs within a method, themethod creates an exception object and hands it off to the runtime system
– Creating an exception object and handing it to the runtime system is called “throwing an exception”
– Exception object contains information about the error, including its type and the state of the program when the error occurred



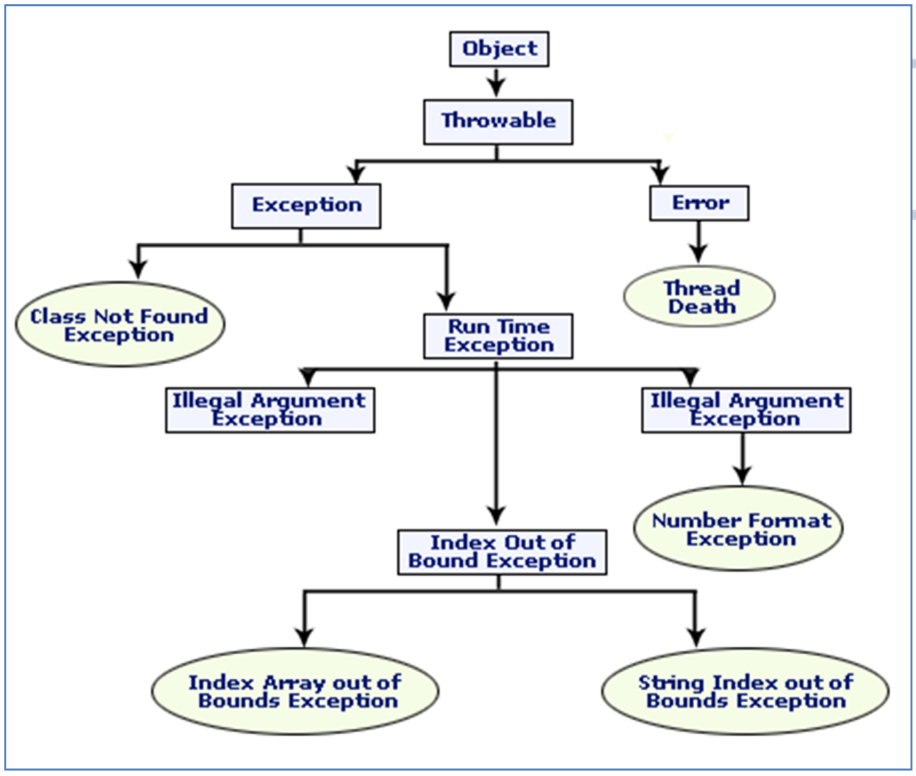
● When an appropriate handler is found, the runtimesystem passes the exception to the handler
– An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler
– The exception handler chosen is said to catch the exception.

- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates and uses the default exception handler

Searching the Call Stack for an Exception Handler



Exception Hierarchy



Throwable Class

"The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause."

Sun says:

"Instances of two subclasses, Error and Exception, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data)."

The Throwable class provides a String variable that can be set by the subclasses to provide a detail message that provides more information of the exception occurred. All classes of throwables define a one-parameter constructor that takes a string as the detail message.

Imp Methods Throwable class
String getMessage() -retrieves the message of exception
void printStackTrace() - it prints the stack trace to the standard error stream
String toString()

System Defined Exceptions
We can classify exceptions as

- System defined Exceptions -Pre defined Exceptions which are thrown by the system.
- User defined Exceptions - Defined,raised and handled by us.

Let us now look into the Important System defined Exceptions
NumberFormatException - Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.
Arithmetic Exception - Thrown when an exceptional arithmetic condition
has occurred. For example, an integer "divide by zero" throws an instance of this class.
ClassCastException - Thrown to indicate that the code has attempted to
cast an object to a subclass of which it is not an instance. For example, the following code generates a ClassCastException:
Ex: Object x = new Integer(0);
Sysem.out.println((String)x);
IndexOutOfBoundsException - Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range. Direct Subclasses of "IndexOutOfBoundsException" are ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException.
ArrayIndexOutOfBoundsException - Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
StringIndexOutOfBoundsException -Thrown by String methods to indicate that an index is either negative or greater than the size of the string. For some methods such as the charAt method, this exception also is thrown when the index is equal to the size of the string.
NullPointerException - Thrown when an application attempts to use null in a case where an object is required. These include:

- ✓ Calling the instance method of a null object.
- ✓ Accessing or modifying the field of a null object.
- ✓ Taking the length of null as if it were an array.
- ✓ Accessing or modifying the slots of null as if it were an array.
- ✓ Throwing null as if it were a Throwable value.

Simple Example to demonstrate the System defined Exceptions.
Let us look at the example to handle the different system defined exceptions as shown below. The example is written to take the inputs as command line arguments.

```
//Divide.java
class Divide{
  public static void main(String args[]){
        int x,y,z;
     try{
        x=Integer.parseInt(args[0]);
```

```
        y=Integer.parseInt(args[1]);
        z=x/y;
        System.out.println(z);
        }catch(NumberFormatException nef){
           System.out.println("Invalid input type");
        }catch(ArithmeticException ae){
           System.out.println("Denominator cannot be zero");
        }catch(ArrayIndexOutOfBoundsException e){
           System.out.println("Sorry inputs are missing");
        }
  }}
```

Run the above program as shown below and watch the outputs
>java Divide 25 5
5
>java Divide 25 0
Denominator cannot be zero
>java Divide 25 A
Invalid input type
>java Divide 25
Sorry inputs are missing

Session 17 – Exceptions Contd..

OBJECTIVES

After completing this session you will be able to understand:

- ✓ Checked & UncheckedExceptions
- ✓ User Defined Exceptions
- ✓ Exception Chaining
- ✓ Multi Catch

Unchecked & Checked Exceptions
Exceptions are also classified in another way as

- ✓ Compiler-enforced exceptions, or checked exceptions
- ✓ Runtime Exceptions, or unchecked exceptions

Unchecked exceptions
Basically, an unchecked exception is a type of exception that you can optionally handle, or ignore. If you elect to ignore the possibility of an unchecked exception, and one occurs, your program will terminate as a result. If you elect to handle an unchecked exception and one occurs, the result will depend on the code that you have written to handle the exception.
Exceptions instantiated from RuntimeException and its subclasses as unchecked exceptions.
Various Unchecked Exceptions are,

- ✓ ArithmeticException
- ✓ ArrayIndexOutOfBoundsException
- ✓ ArrayStoreException
- ✓ ClassCastException

- ✓ IllegalArgumentException
- ✓ NegativeArraySizeException
- ✓ NullPointerException
- ✓ NumberFormatException

Checked exceptions

The Checked exception represents routine abnormal conditions that should be anticipated and caught to prevent program termination. These must be either handled with a try block followed by a catch block, or declared in a throws clause of any method that can throw them In other words, checked exceptions cannot be ignored when you write the code in your methods.

All exceptions instantiated from the Exception class, or from subclasses of Exception other than Runtime Exception and its subclasses are Checked Exceptions.

Various checked exceptions defined in the java.lang.package are,

- ✓ ClassNotFoundException
- ✓ IllegalAccessException
- ✓ InstantiationException
- ✓ NoSuchMethodException

Custom/User Defined Exceptions

User defined exceptions are used to handle logical errors such as functional deviations.

User Defined Exceptions Theme:

User defined exceptions are used to transfer the control from the called block to calling block with all the details functional deviations. For this we use the theme of standard exception mechanism.

Steps for Executing User defined exception

Step 1: Defining an Exception

1. Identify the functional deviation possible and also identify the details of the deviation need to be captured.
2. Write a custom exception class(by inheriting the Exception class) with suitable members to capture the above identified details. The super() call can be used to set a detail message in the throwable.

Step 2: Throwing the exception

1. Identify the functional deviation code and throw the above defined exception. First create the object of the Exception class written with the details encapsulated then throw the exception using "throw" statement.
2. And also define the exception in method signature with "throws" clause for typesafe ness.

Step 3: Handling the exception

Often people write the handling code(catch) in the same method where the exception is raised. Usually we use "User defined Exceptions" to transfer the control with all the details of deviation from called to calling block. So we handle the exception in calling block.

throw -used to explicitly throw an exception

Syntax:

throw <exception object>;

The Exception object must be of type Throwable class or one of its subclasses. A detail message can be passed to the constructor when the exception object is created.

throw new TemperatureException("Too hot");

throws

A throws clause can be used in the method prototype.

method() throws <ExceptionType1>,..., <ExceptionTypen> {

}

Each <ExceptionTypei> can be a checked or unchecked or sometimes even a custom Exception. The exception type specified in the throws clause in the method prototype can be a super class type of the actual exceptions thrown.

Note: An overriding method cannot allow more checked exceptions in its throws clause than the inherited method does.

Example to demonstrate the user defined exceptions

```
//NoFundsException.java
public class NoFundsException extends Exception{
   double eBal,aAmt;
   public NoFundsException(double eb,double aa){
            eBal=eb;
            aAmt=aa;
   }
  public void setEBal(double b){
            eBal=b;
   }
   public double getEBal(){
            return eBal;
   }
   public void setAAmt(double a){
            aAmt=a;
   }
   public double getAAmt(){
            return aAmt;
   }
}
```



```
//Account.java
class Account{
   protected int ano;
   protected String title;
   protected double balance;
   public Account(int a,String t,double b){
ano=a;
            title=t;
            balance=b;
   }
 public int getAno(){
            return ano;
   }
   public void setTitle(String t){
            title=t;
   }
   public String getTitle(){
            return title;
   }
   public double getBalance(){
```

```
        return balance;
    }
    public void deposit(double amt){
        balance+=amt;
        System.out.println("Balance after deposit is "+balance);
    }
    public void withdraw(double amt) throws NoFundsException{
        if ((balance-amt)>=1000){
            balance-=amt;
            System.out.println("Balance after withdrawl is "+balance);
        }else{
            NoFundsException nf=new NoFundsException(balance,amt);
            throw nf;
        }           }}

//TestUserExceptions.java
class TestUserExceptions{
  public static void main(String args[]){
        Account a=null;
      try{
        a=new Account(101,"krsna",50000);
        a.deposit(5000);
        a.withdraw(2300);
        a.withdraw(353500);
        System.out.println("Sucessfully withdrawn");
        }catch(NoFundsException nf){
            double eb,aa;
            eb=nf.getEBal();
            aa=nf.getAAmt();
System.out.println("While a/c is having the balance of"+eb+" you have asked the amount of "+aa);
        }
  }
```


Exception Chaining

Whenever in a program the first exception causes an another exception, that is termed as **Chained Exception**. Java provides new functionality for chaining exceptions. Exception chaining (also known as "nesting exception") is a technique for handling the exception, which occur one after another i.e. most of the time is given by an application to response to an exception by throwing another exception. Typically the second exception is caused by the first exception. Therefore chained exceptions help the programmer to know when one exception causes another.

The **constructors** that support chained exceptions in **Throwable** class are:

- ✓ Throwable initCause(Throwable)
- ✓ Throwable(Throwable)
- ✓ Throwable(String, Throwable)
- ✓ Throwable getCause()

The methods of the Throwable class are:

METHOD	DESCRIPTION
toString()	Returns the exception followed by a message string (if one exit) .

getMessage()	Returns the message string of the Throwable object.
printStackTrace()	Returns the full name of the exception class and some additional information apart from the information of first two method.
getCause()	Returns the exception that caused the occurrence of current exception.
initCause()	Returns the current exception due to the Throwable constructors and the Throwable argument to initCause.

The syntax for using a chained exception is as follows in which a new TestException exception is created with the attached cause when an IOException is caught. Thus the chain exception is thrown to next level of exception handler.

```
try {
} catch (IOException e) {
throw new TestException("Other IOException", e);
}
```

Let us see an example having the implementation of chained exceptions:


```
//MyException.java
import java.io.*;
import java.util.*;
class MyException extends Exception{
MyException(String msg){
  super(msg);
  }
}
public class ChainExcep{
 public static void main(String args[])throws MyException, IOException{
 try{
  int rs=10/0;
  }
catch(Exception e){
 System.err.println(e.getMessage());
  System.err.println(e.getCause());
 throw new MyException("Chained ArithmeticException");
  }
 }
 }
```

Output
>javac ChainExcep.java
>java javac ChainExcep
/ by zero
null

```
Exception in thread "main" MyException: Chained ArithmeticException
at ChainExcep.main(ChainExcep.java:21)
```

Note: This example has an user defined exception that throws an **ArithmeticException** and has been thrown under the **catch** handler. After throwing an exception, the handler will execute the statement of the catch block and then invoke the user defined constructor. Thus the implementation of chained exception is very helpful to the user to make a program or an application error and exception free.

Multi Catch

So far we have seen how to use a single catch block, now we will see how to use more than one catch blocks in a single try block.In java when we handle the exceptions then we can have multiple catch blocks for a particular try block to handle many different kind of exceptions that may be generated while running the program i.e. you can use more than one catch clause in a single try block however every catch block can handle only one type of exception. this mechanism is necessary when the try block has statement that raise different type of exceptions.

The syntax for using this clause is given below:-

```
try{
???
???
}
catch(<exceptionclass_1> <obj1>){
//statements to handle the exception
}
catch(<exceptionclass_2> <obj2>){
//statements to handle the exception
}
catch(<exceptionclass_N> <objN>){
//statements to handle the exception
}
```


When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a matching catch block that can handle the exception. If no handler is found, then the exception is dealt with by the default exception handler at the top level.

Let us see an example given below which shows the implementation of multiple catch blocks for a single try block.

```
//Multi_Catch.java
public class Multi_Catch
{
    public static void main (String args[])
     {
      int array[]={20,10,30};
      int num1=15,num2=0;
      int res=0;
     try
     {
       res = num1/num2;
      System.out.println("The result is" +res);

      for(int ct =2;ct >=0; ct--)
      {
     System.out.println("The value of array are" +array[ct]);
```

```
    }
  }
       catch (ArrayIndexOutOfBoundsException e)
    {
      System.out.println("Error…. Array is out of Bounds");
     }

   catch (ArithmeticException e)
     {
       System.out.println ("Can't be divided by Zero");
     }
  }
 }
```

Output of the program:

```
>javac Multi_Catch.java
>java Multi_Catch
Can't be divided by Zero
```