Exception Handling

Exception is a run-time error which arises during the execution of java program. The term exception in java stands for an **"exceptional event"**.

So Exceptions are nothing but some abnormal and typically an event or conditions that arise during the execution which may interrupt the normal flow of 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.

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

Now if we want to continue the program with the remaining code, then we should write the part of the program which generate the error in the try{ } block and catch the errors using catch() block.

Exception turns the direction of normal flow of the program control and send to the related catch() block and should display error message for taking proper action. This process is known as."

Exception handling

The purpose of exception handling is to detect and report an exception so that proper action can be taken and prevent the program which is automatically terminate or stop the execution because of that exception.

Java exception handling is managed by using five keywords: **try, catch, throw, throws and finally**.

Try: Piece of code of your program that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.

Catch: Catch block can catch this exception and handle it in some logical manner.

```
Scanner scan= new Scanner(System.in);
int marks;
System.out.println(“Enter marks:”)
marks = scan.nextInt();
System.out.println(“Please enter marks as int”);
```

```
boolean flag = true;
Scanner scan= new Scanner(System.in);
int marks;
while(flag){
        flag = false;
        try {
             System.out.println(“Enter marks:”);
             marks = scan.nextInt();
        }
        catch(Exception e){
            System.out.println(“Please enter marks as int”);
            flag = true;
        }
}
```

Line No of Code – Multiple line try

```
1
2
3
try{    4
        5
        6 }
catch(…){
        7
        8 }
9
10
```

Throw: System-generated exceptions are automatically thrown by the Java run-time system. Now if we want to manually throw an exception, we have to use the throw keyword.

Throws: If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a throws clause in the method's declaration. Basically it is used for IOException. A throws clause lists the types of exceptions that a method might throw.

This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

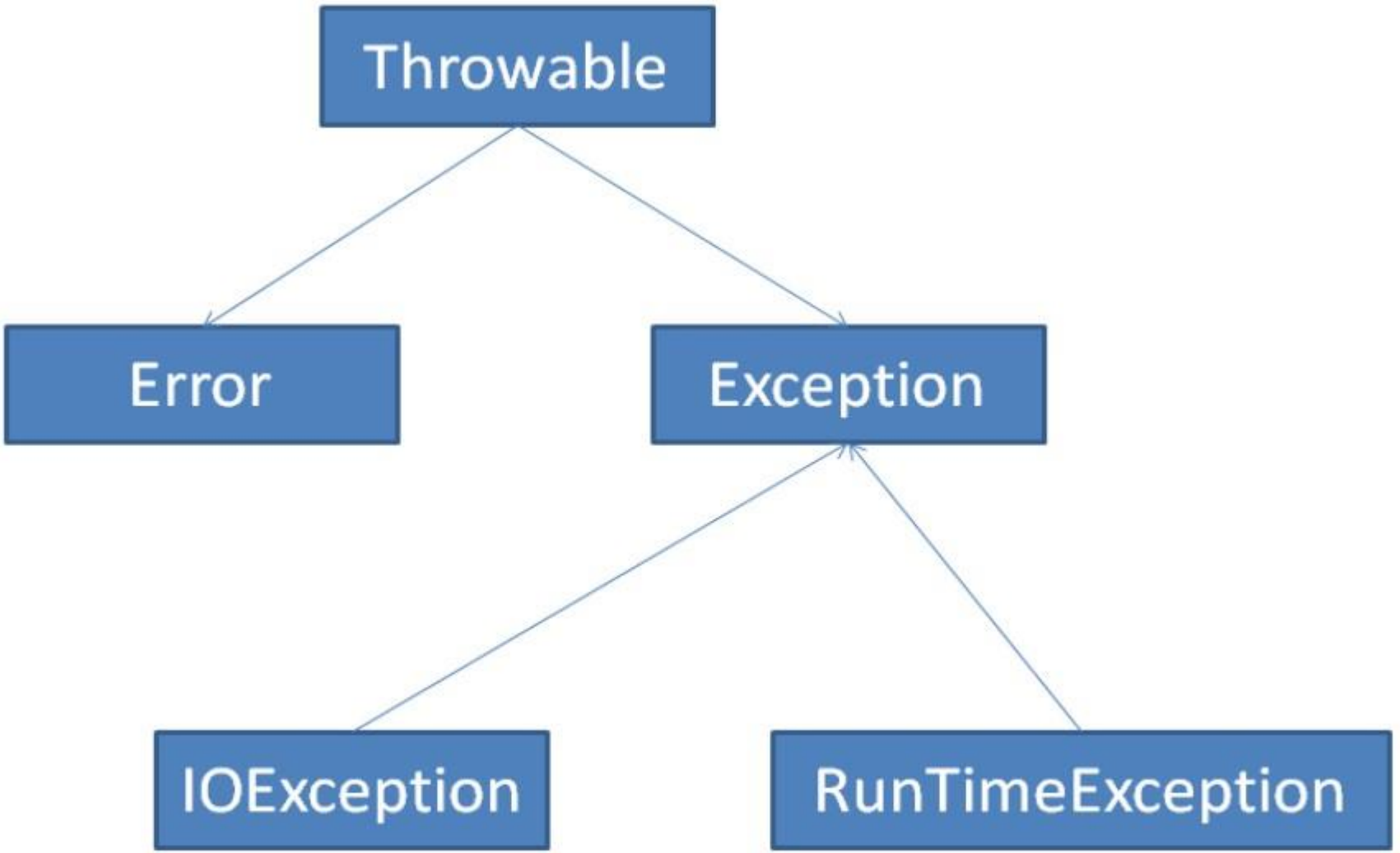
All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

Finally: Any code that absolutely must be executed before a method returns, is put in a finally block.

General form:

```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1  e1) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 e2) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed before try block
ends
}
```

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 :

These are not normally trapped form the Java programs.

Errors are typically ignored in your code because you can rarely do anything about an error.

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.

For Example :

(1) JVM is out of Memory. Normally programs cannot recover from errors.

(2) If a stack overflow occurs then an error will arise. They are also ignored at the time of compilation.

The Exception class has two main subclasses:

(1) IOException or Checked Exceptions class and

(2) RuntimeException or Unchecked Exception class

(1) IOException or Checked Exceptions :

Exceptions that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.*

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.

Java's Checked Exceptions Defined in java.lang

```
File file = new File("not_existing_file.txt");
FileInputStream stream = new FileInputStream(file);
```

```
File file = new File("not_existing_file.txt");
try {
        FileInputStream stream = new FileInputStream(file);
    }
catch (FileNotFoundException e)
        { e.printStackTrace();
        }
```

(2) RuntimeException or Unchecked Exception :

Exceptions need not be included in any method's **throws** list. These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.

As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

Java's Unchecked RuntimeException Subclasses

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.

IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Try And Catch

We have already seen introduction about try and catch block in java exception handling.

Now here is the some examples of try and catch block.

EX :

```
public class TC_Demo
{
   public static void main(String[] args)
   {
      int a=10;       int b=5,c=5;       int x,y;
      try    {
         x = a / (b-c);
      }
      catch(ArithmeticException e){
         System.out.println("Divide by zero");
      }
      y = a / (b+c);
      System.out.println("y = " + y);

   }         }
```

Output :

Divide by zero
y = 1

Note that program did not stop at the point of exceptional condition.It catches the error condition, prints the error message, and continues the execution, as if nothing has happened.

If we run same program without try catch block we will not gate the y value in output. It displays the following message and stops without executing further statements.

Exception in thread "main"
java.lang.ArithmeticException: / by zero
at Thrw_Excp.TC_Demo.main(TC_Demo.java:10)

Here we write ArithmaticException in catch block because it caused by math errors such as divide by zero.

Now how to display description of an exception ?

You can display the description of thrown object by using it in a println() statement by simply passing the exception as an argument. For example;

```
catch (ArithmeticException e)
{
        system.out.pritnln("Exception:" +e);
}
```

Multiple catch blocks :

It is possible to have multiple catch blocks in our program.

EX :

```
public class MultiCatch
{
   public static void main(String[] args)
   {
      int a [] = {5,10};   int b=5;
      try       {
         int x = a[2] / b - a[1];
      }
      catch(ArithmeticException e)       {
         System.out.println("Divide by zero");
      }
      catch(ArrayIndexOutOfBoundsException e)       {
         System.out.println("Array index error");
      }
      catch(ArrayStoreException e)       {
         System.out.println("Wrong data type");
      }
      int y = a[1]/a[0];
      System.out.println("y = " + y);
   }          }
```

Output :

Array index error

y = 2

Note that array element a[2] does not exist. Therefore the index 2 is outside the array boundry.

When exception in try block is generated, the java treats the multiple catch statements like cases in switch statement.

The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

When you are using multiple catch blocks, it is important to remember that exception subclasses must come before any of their superclasses.

This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Thus, a subclass will never be reached if it comes after its superclass. And it will result into syntax error.

// Catching super exception before sub

EX :

```
class etion3
{
   public static void main(String args[])
   {
      int num1 = 100;
      int num2 = 50;
      int num3 = 50;
      int result1;

      try
      {
         result1 = num1/(num2-num3);
         System.out.println("Result1 = " + result1);
      }

      catch (Exception e)
      {
         System.out.println("This is mistake. ");
      }
      catch(ArithmeticException g)
      {
         System.out.println("Division by zero");

      }
   }
}
```

Important

Output :

If you try to compile this program, you will receive an error message because the exception has already been caught in first catch block.

Since ArithmeticException is a subclass of Exception, the first catch block will handle all exception based errors,

including ArithmeticException. This means that the second catch statement will never execute.

To fix the problem, revere the order of the catch statement.

Nested try statements :

The try statement can be nested.

That is, a try statement can be inside a block of another try.

Each time a try statement is entered, its corresponding catch block has to entered.

The catch statements are operated from corresponding statement blocks defined by try.

EX :

```
public class NestedTry
{
    public static void main(String args[])
    {
       int num1 = 100;
       int num2 = 50;
       int num3 = 50;
       int result1;
       try     {
          result1 = num1/(num2-num3);
          System.out.println("Result1 = " + result1);
          try   {
             result1 = num1/(num2-num3);
             System.out.println("Result1 = " + result1);
          }
          catch(ArithmeticException e)
          {
             System.out.println("This is inner catch");
          }
       }
       catch(ArithmeticException g)
       {
          System.out.println("This is outer catch");
       }
    }
}
```

Output :

This is outer catch

Finally

Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.

We can put finally block after the try block or after the last catch block.

The finally block is executed in all circumstances. Even if a try block completes without problems, the finally block executes.

EX :

```
public class Finally_Demo
{
    public static void main(String args[])
    {
       int num1 = 100;
       int num2 = 50;
       int num3 = 50;
       int result1;

       try
       {
          result1 = num1/(num2-num3);
          System.out.println("Result1 = " + result1);
       }
       catch(ArithmeticException g)
       {
          System.out.println("Division by zero");

       }
       finally
       {
          System.out.println("This is final");
       }
    }
}
```

Output :

Division by zero
This is final

Throw

We saw that an exception was generated by the JVM when certain run-time problems occurred.
It is also possible for our program to explicitly generate an exception.

This can be done with a throw statement. Its form is as follows:

Throw object;

Inside a catch block, you can throw the same exception object that was provided as an argument.

This can be done with the following syntax:

```
catch(ExceptionType object)
{
        throw object;
}
```

Alternatively, you may create and throw a new exception object as follows:

```
Throw new ExceptionType(args);
```

Here, exceptionType is the type of the exception object and args is the optional argument list for its constructor.

When a throw statement is encountered, a search for a matching catch block begins and if found it is executed.

EX :

```
class Throw_Demo
{
   public static void a()
   {
      try
     {
         System.out.println("Before b");
         b();
      }
```

```
 catch(ArrayIndexOutOfBoundsException j) //manually thrown object catched here
 {
        System.out.println("J : " + j)  ;
  }
}
public static void b()
   {
      int a=5,b=0;
      try
      { System.out.println("We r in b");
        System.out.println("********");

                int x = a/b;
      }
      catch(ArithmeticException e)
      {
        System.out.println("c : " + e);
        throw new ArrayIndexOutOfBoundsException("demo");   //throw from here
      }
   }
```

```
 public static void main(String args[])
  {

     try
     {
        System.out.println("Before a");
        a();
        System.out.println("********");
        System.out.println("After a");
     }
     catch(ArithmeticException e)
     {
        System.out.println("Main Program : " + e)
     }
  }
}
```

Output :

Before a
Before b
We r in b
****
c :
java.lang.ArithmeticExcepti
on: / by zero
J :
java.lang.ArrayIndexOutOf
BoundsException: demo
****

After a

Throwing our own object :

If we want to throw our own exception, we can do this by using the keyword throw as follow.

throw new Throwable_subclass;

Example : **throw new ArithmaticException( );**
throw new NumberFormatException( );

EX :

```
import java.lang.Exception;
class MyException extends Exception
{
   MyException(String message)
   { super(message);

   }
}
```

```
class TestMyException
{
   public static void main(String[] args)
   {
      int x = 5, y = 1000;
      try
      {
         float z = (float)x / (float)y;
         if(z < 0.01)
         {
            throw new MyException("Number is too small");
         }
      }
      catch(MyException e)
      {
         System.out.println("Caught MyException"
         System.out.println(e.getMessage());
      }
      finally
      {
         System.out.println("java2all.com");
      }
   }
}
```

So super(msg) is linked to e.getMessage() in layman terms.

Output :

Caught MyException
Number is too small
java2all.com

Here The object e which contains the error message "Number is too small" is caught by the catch block which then displays the message using **getMessage**( ) method.

NOTE:

Exception is a subclass of Throwable and therefore MyException is a subclass of Throwable class. An object of a class that extends Throwable can be thrown and caught.

Throws

If a method is capable of causing an exception that it does not handle,it must specify this behavior so that callers of the method can guard themselves against that exception.

You do this by including a throws clause in the method's declaration. Basically it is used for IOException.

A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error or RuntimeException**,or any of their subclasses.

All other exceptions that a method can throw must be declared in the throws clause.

If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

type method-name(parameter-list) throws exception-list
{
// body of method
}

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Throw is used to actually throw the exception, whereas throws is declarative statement for the method. They are not interchangeable.

EX :

```
class NewException extends Exception
{
   public String toS()
   {
      return "You are in NewException ";
   }
}
```

```
class customexception
{
   public static void main(String args[])
   {
      try
      {
         doWork(3);
         doWork(2);
         doWork(1);
         doWork(0);
      }
      catch (NewException e)
      {
         System.out.println("Exception : " + e.toS());
      }
   }
   static void doWork(int value) throws NewException
   {
      if (value == 0)
      {
         throw new NewException();
      }
      else
      {
         System.out.println("****No Problem.****");
      }
   }
}
```

Output :

****No Problem.****
****No Problem.****
****No Problem.****
Exception : You are in NewException