Unit II Part D

Exception Handling in Java

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is Exception in Java

Dictionary Meaning: Exception is an abnormal condition. In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling

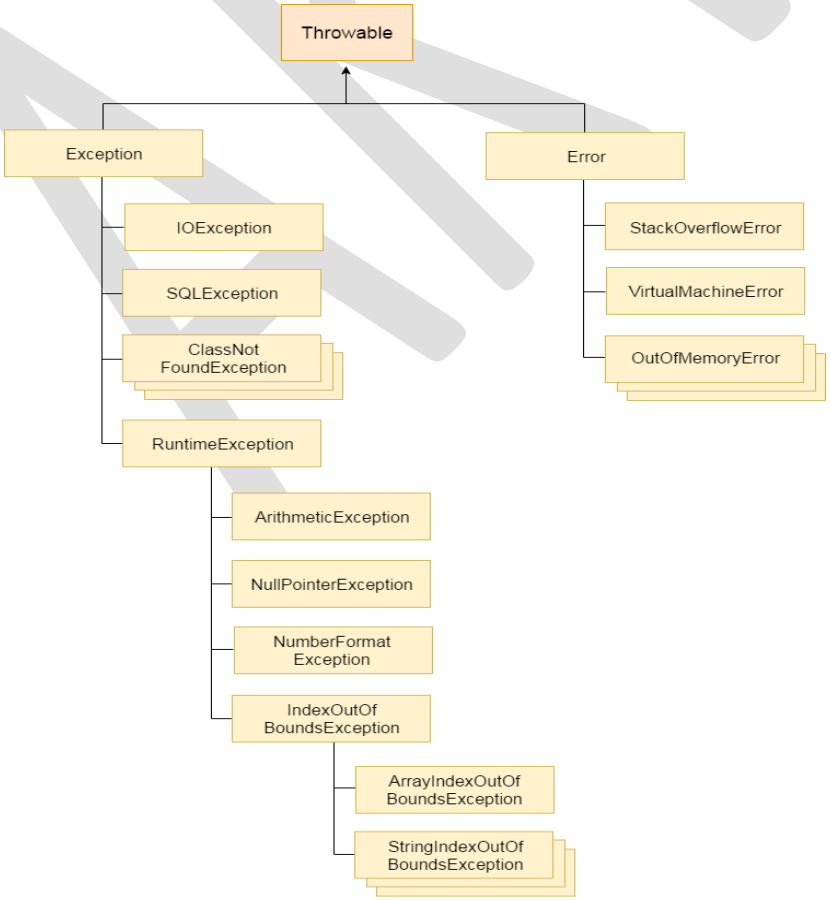
Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. e.g. Suppose there are 10 statements in the program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.

throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Exception Handling Example

```
public class JavaExceptionExample
{
        public static void main(String args[])
        {
                try
                {
                        //code that may raise exception
                        int data=100/0;
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e);
                }
                //rest code of the program
                System.out.println("rest of the code...");
        }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method. If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception. Java try block must be followed by either catch or finally block.

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception. The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Example 1

```
public class TryCatchExample1
{
	public static void main(String[] args)
	{
		int data=50/0; //may throw exception
		System.out.println("rest of the code");
	}
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

Solution by exception handling

Example 2

```
public class TryCatchExample2
{
	public static void main(String[] args)
	{
		try
		{
			int data=50/0; //may throw exception
		}
		// handling the exception
		catch(Exception e)
		{
```

```
                    // displaying the custom message
                    System.out.println("Can't divided by zero");
                }
        }
}
```

Output:

Can't divided by zero

Example 3

```
public class TryCatchExample3
{
        public static void main(String[] args)
        {
                try
                {
                        int arr[]= {1,3,5,7};
                        System.out.println(arr[10]); //may throw exception
                }
                // handling the array exception
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println(e);
                }
                System.out.println("rest of the code");
        }
}
```

Output:

java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code

Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example

```
public class MultiCatchBlock1
{
        public static void main(String[] args)
        {
                try
```

```
                {
                        int a[]=new int[5];
                        a[5]=30/0;
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Arithmetic Exception occurs");
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("ArrayIndexOutOfBoundsException
occurs");
                }
                catch(Exception e)
                {
                        System.out.println("Parent Exception occurs");
                }
                System.out.println("rest of the code");
        }
}
```

Output:

Arithmetic Exception occurs
rest of the code

Example

```
public class MultipleCatchBlock2
{
        public static void main(String[] args)
        {
                try
                {
                        String s=null;
                        System.out.println(s.length());
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Arithmetic Exception occurs");
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("ArrayIndexOutOfBoundsException
occurs");
                }
```

```
                catch(Exception e)
                {
                        System.out.println("Parent Exception occurs");
                }
                System.out.println("rest of the code");
        }
}
```

Output:

ArrayIndexOutOfBounds Exception occurs
rest of the code

Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
1.  ....
2.  try
3.  {
4.      statement 1;
5.      statement 2;
6.      try
7.      {
8.          statement 1;
9.          statement 2;
10.     }
11.     catch(Exception e)
12.     {
13.     }
14. }
15. catch(Exception e)
16. {
17. }
18. ....
```

Nested try example

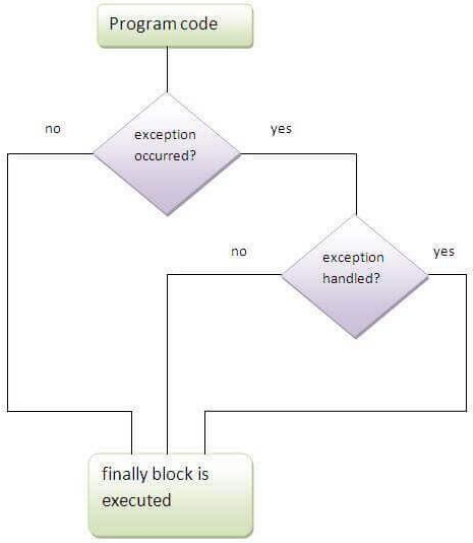
```
class NestedTry
{
        public static void main(String args[])
        {
                try
                {
                        try
                        {
```

```
                    System.out.println("going to divide");
                    int b =39/0;
                }
                catch(ArithmeticException e)
                {
                    System.out.println(e);
                }
                try
                {
                    int a[]=new int[5];
                    a[5]=4;
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                    System.out.println(e);
                }
                System.out.println("other statement");
            }
            catch(Exception e)
            {
                System.out.println("handeled");
            }
            System.out.println("normal flow..");
        }
}
```

finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
class TestFinallyBlock
{
	public static void main(String args[])
	{
		try
		{
			int data=25/5;
			System.out.println(data);
		}
		catch(NullPointerException e)
		{
			System.out.println(e);
		}
		finally
		{
			System.out.println("finally block is always executed");
		}
		System.out.println("rest of the code...");
	}
}
```

Output:

```
finally block is always executed
rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1
{
	public static void main(String args[])
	{
		try
		{
			int data=25/0;
			System.out.println(data);
		}
```

```
            catch(NullPointerException e)
            {
                System.out.println(e);
            }
            finally
            {
                System.out.println("finally block is always executed");
            }
            System.out.println("rest of the code...");
        }
}
```

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2
{
    public static void main(String args[])
    {
        try
        {
            int data=25/0;
            System.out.println(data);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output: Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

rest of the code...

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw

custom exception. We will see custom exceptions later. The syntax of java throw keyword is given below.

throw exception;

Example

throw new IOException("sorry device error);

```
public class TestThrow1
{
    static void validate(int age)
    {
        if(age<18)
            throw new ArithmeticException("not valid");
        else
            System.out.println("welcome to vote");
    }
     public static void main(String args[])
    {
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:not valid

throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

1. return_type method_name() throws exception_class_name{
2. //method code
3. }

throws example

```
import java.io.IOException;
class TestThrows
{
    void m()throws IOException
    {
        throw new IOException("device error");//checked exception
    }
     void n()throws IOException
```

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

Output:

exception handled
normal flow...

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Chained Exceptions in Java

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Constructors Of Throwable class Which support chained exceptions in java :

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of Throwable class Which support chained exceptions in java :

1. getCause() method :- This method returns actual cause of an exception.
2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

Example of using Chained Exception:

```
public class ChainedExceptionExample
{
        public static void main(String[] args)
        {
                try
                {
                        // Creating an exception
                        NumberFormatException ex = new
NumberFormatException("Exception");
                        // Setting a cause of the exception
                        ex.initCause(new NullPointerException("This is actual cause of the
exception"));
                         // Throwing an exception with cause.
                        throw ex;
                 }
                catch(NumberFormatException ex)
```

```
            {
                // displaying the exception
                 System.out.println(ex);
                // Getting the actual cause of the exception
                System.out.println(ex.getCause());
            }
        }
}
```

Output:

java.lang.NumberFormatException: Exception

java.lang.NullPointerException: This is actual cause of the exception