Chapter 6: Exception Handling

In this chapter we will discuss exceptions in the object-oriented programming. We will learn how to handle exceptions using the try-catch construct, how to pass them to the calling methods and how to throw standard or our own exceptions using the throw construct. We will give various examples for using exceptions. We will look at the types of exceptions.

Exception is an abnormal condition or run time error that arises in the program during execution. When such a condition arises in the program, an appropriate code is written so that it can be handled.

6.1 Exception handling mechanism

i. try … catch block
ii. multiple catch
iii. Nested try… catch
iv. finally block
v. throws
vi. throw

i. try … catch block

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
}
```

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 follow the try is checked.

The below example is for division of two number if the value of n2 is zero then program will generated arithmetic exception and the output will display "You cannot divide a number by zero".

Example 1:

```
class pb1
{
    public static void main(String args[])
    {
```

```
                int n1,n2;
                n1=5;
                n2=0;
                try
                {
                        System.out.println("Div result="+n1/n2);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("You cannot divide a number by zero");
                }
        }
}
```

Output:
You cannot divide a number by zero

ii. Multiple catch

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 2:

```
class pb1
{
	public static void main(String args[])
	{
		try
		{
			int n1,n2;
			n1=Integer.parseInt(args[0]);
			n2=Integer.parseInt(args[1]);
			System.out.println("Div result="+n1/n2);
		}
		catch(ArithmeticException e)
		{
			System.out.println("You cannot divide a number by zero");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("Please Enter two numbers");
		}
		catch(NumberFormatException e)
		{
			System.out.println("NumberFormatException  generated");
		}
	}
}
```

Output 1:

Please Enter two numbers

Output 2:

NumberFormatException generated

Output:3:

You cannot divide a number by zero

iii. Nested try… catch

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

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.

Syntax:

```
....
try
{
   statement 1;
   statement 2;
```

```
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
....
```

Example 3:

```
class pb1
{
        public static void main(String args[])
        {
                try
                {
                        int n1,n2;
                        n1=Integer.parseInt(args[0]);
                        n2=Integer.parseInt(args[1]);
                        try
                        {
                                System.out.println("Div result="+n1/n2);
                        }
                        catch(ArithmeticException e)
                        {
                        System.out.println("You cannot divide a number by zero");
                        }
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Please Enter two numbers");
                }
        }
}
```

Output 1:

You cannot divide a number by zero

Output 2:

Please Enter two numbers

iv. finally block

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.
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:

```
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 4:

```
class TestFinally
{
    public static void main(String args[])
    {
        try
        {
            int data=25/0;
            System.out.println(data);
        }
        catch(ArithmeticException e)
        {
        System.out.println("You cannot divide a number by zero");
        }
        finally
        {
            System.out.println("finally block is always executed");
```

```
            }
        System.out.println("rest of the code...");
        }
}
```

Output:

```
You cannot divide a number by zero
finally block is always executed
rest of the code...
```

v. throws

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.

Example 5:

```
class test
{
        void add()throws NumberFormatException , ArithmeticException
        {
                int a,b;
                a=5;
                b=0;
                System.out.println("div="+ a/b );
        }
}
class pb2
{
        public static void main(String args[])
        {
                try{
                        test t1=new test();
                        t1.add();
                  }
                catch(NumberFormatException e)
                {
                        System.out.println("Incorrect number format is entered");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("you cannot divide number by 0");
                }

        }
}
```

Output:

```
you cannot divide number by 0
```

vi. throw

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

By using throw keyword in java you cannot throw more than one exception but using throws you can declare multiple exceptions.

Example 6 Write an application that generates custom exception if first argument from command line argument is 0:

```
class Zero extends Exception
{
        Zero()
        {
                System.out.println("Zero Exception Generated");
        }
}
class pb3
{
        public static void main(String args[])
        {
                System.out.println("***Exception Demo***");
                try{
                if (Integer.parseInt(args[0])==0)
                throw new Zero();
                }
                catch(Zero z)
                {
                        System.out.println("Please give some another number");
                }
        }
}
```

Output:

```
***Exception Demo***
Zero Exception Generated
Please give some another number
```

Example 7 Write an application that generates custom exception if any of its command line arguments are negative:

```
class Negative extends Exception
{
```

```
        Negative()
        {
                System.out.println("Negative Exception generated");
        }
}

class pb4
{
        public static void main(String args[])
        {
                System.out.println("****/////Negative Exception Demo///////******");
         try{
                for(int i=0;i<args.length;i++)
                {
                                if(Integer.parseInt(args[i]) < 0)
                                {
                                        throw new Negative();
                                }
                }
           }
           catch(Negative n)
           {
                System.out.println("Any one or more of your command line args is negative");

           }

        }
}
```

Output 1:

****/////Negative Exception Demo///////******

Output 2:

****/////Negative Exception Demo///////******

Negative Exception generated

Any one or more of your command line args is negative

6.2 Difference between throw and throws

1) throws is use to declare an exception ,throw keyword is use to throw an exception explicitly .
2) If we see syntax wise than throw is follow by an instance variable and throws is follow by exception class name.

3) throw is use inside method body to invoke an exception and throws is used in method declaration(signature).
4) By using throw keyword in java you can not throw more than one exception but using throws you can declare multiple exceptions.
5) E.g

Throw

```
try {
        Throw new ArithmaticException();
}
catch(ArithmeticException e)
{
        System.out.println("You can't divide by zero");
}
```

Throws

```
void fun(int a,int b) throws ArithmeticException,NumerFormatException
{
        int c;
        try {
                c=a/b;
        }
        catch(ArithmeticException e)
        {
                System.out.println("You can't divide by zero");
        }
}
```

6.3 Difference between Checked and Unchecked Exceptions

Checked Exception:

- The exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
- Checked exceptions are checked at compile-time.
- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.

Unchecked Exception:

- The exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.
- 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.

6.4 Types of Access Modifiers

Java provides many access modifiers to access the variable, methods, and constructors.

There are 4 types of access variables in Java:

1. Private
2. Public
3. Default
4. Protected

#1) Private

If a variable is declared as private, then it can be accessed within the class. This variable won't be available outside the class. So, the outside members cannot access the private members.

Note: Classes and interfaces cannot be private.

#2) Public

Methods/variables with public access modifiers can be accessed by all the other classes in the project.

#3) Protected

If a variable is declared as protected, then it can be accessed within the same package classes and sub-class of any other packages.

Note: Protected access modifier cannot be used for class and interfaces.

#4) Default Access Modifier

If a variable/method is defined without any access modifier keyword, then that will have a default modifier access.

Access Modifiers	Visibility
Public	Visible to All classes.
Protected	Visible to classes with in the package and the subclasses of other package.
No Access Modifier (Default)	Visible to the classes with the package
private	Visible with in the class. It is not accessible outside the class.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

6.5 Rethrowing Exception

The Exception occurs in try block and it is caught in catch block.
The caught exception can be rethrown in the catch block. This re-thrown exception needs to be handled by another catch block somewhere in the program, otherwise the program gets terminated sudden.
Example:

```
class ex
{
   public static void main(String[] args)
   {
      try
      {
         methoddemo();
      }
      catch(NullPointerException ex)
      {
         System.out.println("NullPointerException Re-thrown in methodWithThrow() ");
      }
   }
   static void methoddemo()
   {
       try
      {
          String s = null;
          System.out.println(s.length());   //This statement throws NullPointerException
      }
      catch(NullPointerException ex)
      {
            System.out.println("NullPointerException is caught here");
            throw ex;     //Re-throwing NullPointerException
      }
   }
}
```

Output:

```
NullPointerException is caught here
NullPointerException Re-thrown in methodWithThrow()
```

6.5 Chained Exception

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

```
class ChainDemo
{
	public static void main(String args[])throws Exception
	{
		try
		{
			m1();
		}
		catch(Exception e)
		{
			System.out.println(e.getMessage());
		}
	}
	static void m1()throws Exception
	{
		try
		{
			m2();
		}
		catch(Exception e)
		{
			throw new Exception("M1 throws "+e.getMessage());
		}
	}
	static void m2()throws Exception
	{
		throw new Exception("M2 throws");
	}
}
```

Output:

```
M1 throws M2 throws
```