

Java Exception Handling

Exception Handling is a mechanism to handle exception at runtime. Exception is a condition that occurs during program execution and lead to program termination abnormally. There can be several reasons that can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

Suppose we run a program to read data from a file and if the file is not available then the program will stop execution and terminate the program by reporting the exception message.

The problem with the exception is, it terminates the program and skip rest of the execution that means if a program have 100 lines of code and at line 10 an exception occur then program will terminate immediately by skipping execution of rest 90 lines of code.

To handle this problem, we use exception handling that avoid program termination and continue the execution by skipping exception code.

Java exception handling provides a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Uncaught Exceptions

Lets understand exception with an example. When we don't handle the exceptions, it leads to unexpected program termination. In this program, an `ArithmeticException` will be throw due to divide by zero.

```
class UncaughtException
{
    public static void main(String args[])
    {
        int a = 0;

        int b = 7/a;    // Divide by zero, will lead to exception
    }
}
```

Copy

This will lead to an exception at runtime, hence the Java run-time system will construct an exception and then throw it. As we don't have any mechanism for handling exception in the above program, hence the default handler (JVM) will handle the exception and will print the details of the exception on the terminal.

The diagram shows an exception message: **java.lang.ArithmeticException: / by zero**
at UncaughtException.main(UncaughtException.java:4)

Annotations with arrows pointing to parts of the message:

- name and description of Exception** points to `java.lang.ArithmeticException: / by zero`
- class name** points to `UncaughtException`
- file name** points to `UncaughtException.java`
- Stack Trace (line at which exception occurred)** points to `4`

Java Exception

A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an **exception handler**

How to Handle Exception

Java provides controls to handle exception in the program. These controls are listed below.

- Try : It is used to enclose the suspected code.
- Catch: It acts as exception handler.
- Finally: It is used to execute necessary code.
- Throw: It throws the exception explicitly.
- Throws: It informs for the possible exception.

We will discuss about all these in our next tutorials.

Types of Exceptions

In Java, exceptions broadly can be categories into checked exception, unchecked exception and error based on the nature of exception.

- **Checked Exception**

The exception that can be predicted by the JVM at the compile time for **example** : File that need to be opened is not found, SQLException etc. These type of exceptions must be checked at compile time.

- **Unchecked Exception**

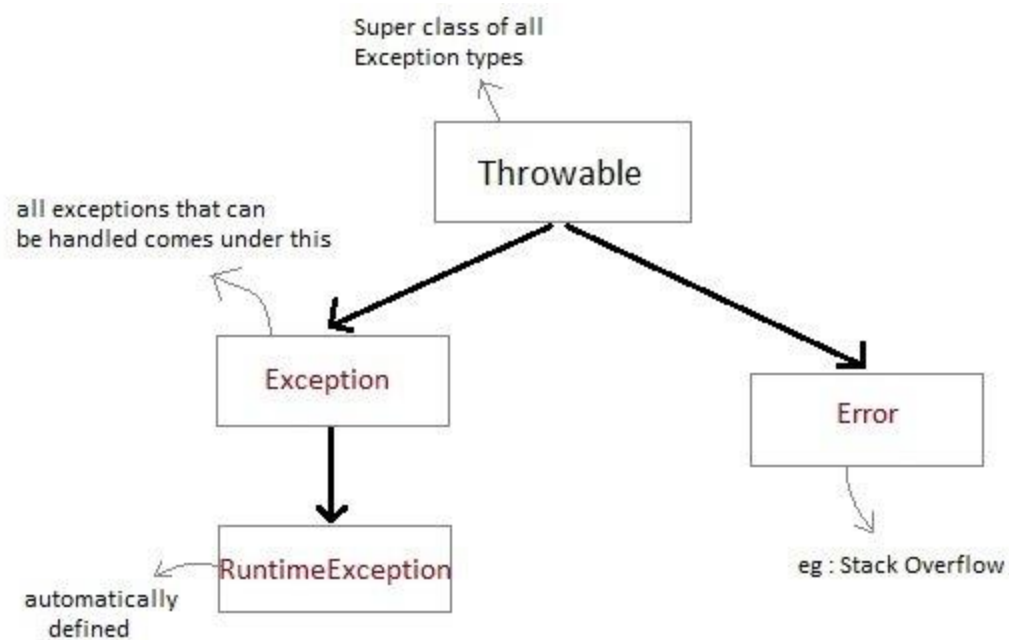
Unchecked exceptions are the class that extends RuntimeException class. Unchecked exception are ignored at compile time and checked at runtime. For **example** : ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

- **Error**

Errors are typically ignored in code because you can rarely do anything about an error. For **example**, if stack overflow occurs, an error will arise. This type of error cannot be handled in the code.

Java Exception class Hierarchy

All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy.



- **Exception** class is for exceptional conditions that program should catch. This class is extended to create user specific exception classes.
- **RuntimeException** is a subclass of Exception. Exceptions under this class are automatically defined for programs.
- **Exceptions** of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

try and catch in Java

Try and **catch** both are Java **keywords** and used for **exception handling**. The try block is used to enclose the **suspected** code. Suspected code is a code that may raise an exception during program execution.

For example, if a code raise arithmetic exception due to divide by zero then we can wrap that code into the try block.

```
try{  
  
    int a = 10;  
  
    int b = 0  
  
    int c = a/b; // exception  
}
```

Copy

The **catch** block also known as **handler** is used to handle the exception. It handles the exception thrown by the code enclosed into the try block. Try block must provide a catch handler or a finally block. We will discuss about finally block in our next tutorials.

The catch block must be used after the try block only. We can also use multiple catch block with a single try block.

```
try{
```

```
int a = 10;

int b = 0

int c = a/b; // exception
}catch(ArithmeticException e){

    System.out.println(e);
}
```

Copy

Try Catch Syntax

To declare try catch block, a general syntax is given below.

```
try{

    // suspected code
}catch(ExceptionClass ec){}
```

Copy

Exception handling is done by transferring the execution of a program to an appropriate exception handler (catch block) when exception occurs.

Example: Handling Exception

Now lets understand the try and catch by a simple example in which we are dividing a number by zero. The code is enclosed into try block and a catch handler is provided to handle the exception.

```
class Excp

{

    public static void main(String args[])

    {

        int a,b,c;

        try

        {

            a = 0;

            b = 10;

            c = b/a;
```

```
        System.out.println("This line will not be executed");

    }

    catch(ArithmeticException e)

    {

        System.out.println("Divided by zero");

    }

    System.out.println("After exception is handled");

}

}
```

Copy

Divided by zero

After exception is handled

Explanation

An exception will thrown by this program as we are trying to divide a number by zero inside **try** block. The program control is transferred outside **try** block. Thus the line "*This line will not be executed*" is never parsed by the compiler. The exception thrown is handled in **catch** block. Once the exception is handled, the program control is continue with the next line in the program i.e after catch block. Thus the line "*After exception is handled*" is printed.

Multiple catch blocks

A try block can be followed by multiple catch blocks. It means we can have any number of catch blocks after a single try block. If an exception occurs in the guarded code(try block) the exception is passed to the first catch block in the list. If the exception type matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continue until the exception is caught or falls through all catches.

Multiple Catch Syntax

To declare the multiple catch handler, we can use the following syntax.

```
try

{

    // suspected code

}

catch(Exception1 e)

{
```

```
        // handler code

    }

    catch(Exception2 e)

    {

        // handler code

    }

}
```

Copy

Now lets see an example to implement the multiple catch block that are used to catch possible exception.

The multiple catch blocks are useful when we are not sure about the type of exception during program execution.

Examples for Multiple Catch blocks

In this example, we are trying to fetch integer value of an Integer object. But due to bad input, it throws number format exception.

```
class Demo{

    public static void main(String[] args) {

        try

        {

            Integer in = new Integer("abc");

            in.intValue();

        }

        catch (ArithmeticException e)

        {

            System.out.println("Arithmetic " + e);

        }

        catch (NumberFormatException e)

        {

            System.out.println("Number Format Exception " + e);

        }

    }

}
```

```
}  
  
}  
  
}
```

Copy

```
Number Format Exception java.lang.NumberFormatException: For input string: "abc"
```

In the above example, we used multiple catch blocks and based on the type of exception second catch block is executed.

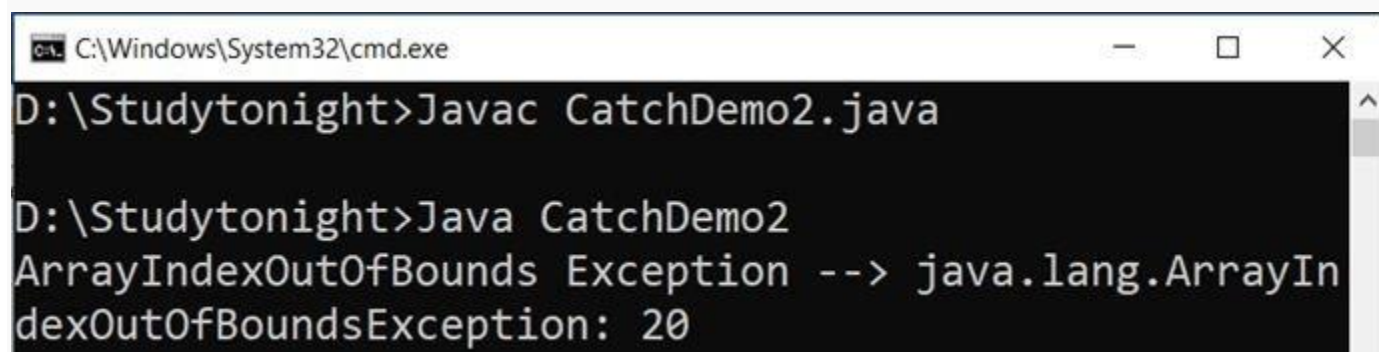
Example: Multiple Exception

Lets understand the use of multiple catch handler by one more example, here we are using three catch handlers in catch the exception.

```
public class CatchDemo2  
{  
  
    public static void main(String[] args)  
  
    {  
  
        try  
  
        {  
  
            int a[]=new int[10];  
  
            System.out.println(a[20]);  
  
        }  
  
        catch(ArithmeticException e)  
  
        {  
  
            System.out.println("Arithmetic Exception --> "+e);  
  
        }  
  
        catch(ArrayIndexOutOfBoundsException e)  
  
        {  
  
            System.out.println("ArrayIndexOutOfBounds Exception --> "+e);  
  
        }  
  
        catch(Exception e)  
  
        {  
  
            System.out.println(e);  
  
        }  
  
    }  
  
}
```

```
}  
  
}  
  
}
```

Copy



```
C:\Windows\System32\cmd.exe  
D:\Studytonight>Javac CatchDemo2.java  
  
D:\Studytonight>Java CatchDemo2  
ArrayIndexOutOfBoundsException --> java.lang.ArrayIn  
dexOutOfBoundsException: 20
```

At a time, only one exception is processed and only one respective catch block is executed.

Example for Unreachable Catch block

While using multiple **catch** statements, it is important to remember that sub classes of class Exception inside **catch** must come before any of their super classes otherwise it will lead to compile time error. This is because in Java, if any code is unreachable, then it gives compile time error.

```
class Excep  
  
{  
  
    public static void main(String[] args)  
  
    {  
  
        try  
  
        {  
  
            int arr[]={1,2};  
  
            arr[2]=3/0;  
  
        }  
  
        catch(Exception e)    //This block handles all Exception  
  
        {  
  
            System.out.println("Generic exception");  
  
        }  
  
        catch(ArrayIndexOutOfBoundsException e)    //This block is  
unreachable  
  
        {  
  
            System.out.println("array index out of bound exception");  
  
        }  
  
    }  
  
}
```



```
}  
  
}  
  
}
```

Copy

Generic exception

Nested try statement

try statement can be **nested** inside another block of **try**. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner **try** block does not have a **catch** handler for a particular exception then the outer **try catch block** is checked for match.

```
class Excep  
  
{  
  
    public static void main(String[] args)  
  
    {  
  
        try  
  
        {  
  
            int arr[]={5,0,1,2};  
  
            try  
  
            {  
  
                int x = arr[3]/arr[1];  
  
            }  
  
            catch(ArithmeticException ae)  
  
            {  
  
                System.out.println("divide by zero");  
  
            }  
  
            arr[4]=3;  
  
        }  
  
        catch(ArrayIndexOutOfBoundsException e)  
  
        {  
  
            System.out.println("array index out of bound exception");  
  
        }  
  
    }  
  
}
```

```
}  
  
}  
  
}
```

Copy

```
divide by zero  
array index out of bound exception
```

Important points to Remember

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
2. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
3. While using multiple catch block, always make sure that sub-classes of Exception class comes before any of their super classes. Else you will get compile time error.
4. In nested try catch, the inner try block uses its own catch block as well as catch block of the outer try, if required.
5. Only the object of Throwable class or its subclasses can be thrown.

Java **try** with Resource Statement

Try with resource is a new feature of Java that was introduced in Java 7 and further improved in Java 9. This feature add another way to exception handling with resources management. It is also referred as **automatic resource management**. It close resources automatically by using AutoCloseable interface..

Resource can be any like: file, connection etc and we don't need to explicitly close these, JVM will do this automatically.

Suppose, we run a JDBC program to connect to the database then we have to create a connection and close it at the end of task as well. But in case of try-with-resource we don't need to close the connection, JVM will do this automatically by using AutoCloseable interface.

Try with Resource Syntax

```
try(resource-specification(there can be more than one resource))  
{  
  
    //use the resource  
  
}  
  
catch()  
  
{  
  
    // handler code  
  
}
```

```
}
```

Copy

This **try statement** contains a **parenthesis** in which one or more resources is declared. Any object that implements `java.lang.AutoCloseable` or `java.io.Closeable`, can be passed as a parameter to **try statement**. A resource is an object that is used in program and must be closed after the program is finished. The **try-with-resources statement** ensures that each resource is closed at the end of the statement of the try block. We do not have to explicitly close the resources.

Example without using `try` with Resource Statement

Lets see the scenario where we are not using try-with-resource block whereas we are using normal try block that's why we need to close the file reference explicitly.

```
import java.io.*;

class Demo
{
    public static void main(String[] args)
    {
        try {
            String str;

            //opening file in read mode using BufferedReader stream
            BufferedReader br = new BufferedReader(new
FileReader("d:\\myfile.txt"));

            while((str=br.readLine())!=null)
            {
                System.out.println(str);
            }

            br.close();        //closing BufferedReader stream
        }

        catch(IOException ie)
        {
            System.out.println("I/O Exception "+ie);
        }
    }
}
```

Copy

```
I/O Exception java.io.FileNotFoundException: d:\myfile.txt (No such file or directory)
```

Example **try** with Resource Statement

Here, we are using try-with-resource to open the file and see we did not use close method to close the file connection.

```
import java.io.*;

class Demo

{

    public static void main(String[] args)

    {

        try(BufferedReader br = new BufferedReader(new
FileReader("d:\\myfile.txt")))

        {

            String str;

            while((str = br.readLine()) != null)

            {

                System.out.println(str);

            }

        }

        catch(IOException ie)

        {

            System.out.println("I/O Exception "+ie);

        }

    }

}
```

Copy

NOTE: In the above example, we do not need to explicitly call **close()** method to close **BufferedReader stream**.

Try with resource – Java 9

In Java 7, try-with-resource was introduced and in which resource was created inside the try block. It was the limitation with Java 7 that a connection object created outside can not be refer inside the try-with-resource.

In Java 9 this limitation was removed so that now we can create object outside the try-with-resource and then refer inside it without getting any error.

Example

```
import java.io.*;

class Demo

{

    public static void main(String[] args) throws
FileNotFoundException

    {

        BufferedReader br = new BufferedReader(new
FileReader("d:\\myfile.txt"));

        try(br)    // resource is declared outside the try

        {

            String str;

            while((str = br.readLine()) != null)

            {

                System.out.println(str);

            }

        }

        catch(IOException ie)

        {

            System.out.println("I/O Exception "+ie);

        }

    }

}
```

Copy

Points to Remember

1. A resource is an object in a program that must be closed after the program has finished.
2. Any object that implements `java.lang.AutoCloseable` or `java.io.Closeable` can be passed as a parameter to try statement.

3. All the resources declared in the try-with-resources statement will be closed automatically when the try block exits. There is no need to close it explicitly.
4. We can write more than one resources in the try statement.
5. In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

Java throw, throws and finally Keyword

Throw, throws and finally are the keywords in Java that are used in exception handling. The throw keyword is used to throw an exception and throws is used to declare the list of possible exceptions with the method signature. Whereas finally block is used to execute essential code, specially to release the occupied resources.

Now lets discuss each in details with the examples.

Java Throw

The throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

Syntax :

```
throw ThrowableInstance
```

Copy

Creating Instance of Throwable class

We allows to use new operator to create an instance of class Throwable,

```
new NullPointerException("test");
```

Copy

This constructs an instance of NullPointerException with name test.

Example throw Exception

In this example, we are throwing Arithmetic exception explicitly by using the throw keyword that will be handle by catch block.

```
class Test
{
    static void avg()
    {
        try
        {
            throw new ArithmeticException("demo");
        }
    }
}
```

```
        catch(ArithmeticException e)

        {

            System.out.println("Exception caught");

        }

    }

    public static void main(String args[])

    {

        avg();

    }

}
```

Copy

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement and thus, the program prints the output "Exception caught".

Java **throws** Keyword

The throws keyword is used to declare the list of exception that a method may throw during execution of program. Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the **throws** keyword.

Syntax:

```
type method_name(parameter_list) throws exception_list

{

    // definition of method

}
```

Copy

Example throws Keyword

Here, we have a method that can throw Arithmetic exception so we mentioned that with the method declaration and catch that using the catch handler in the main method.

```
class Test

{

    static void check() throws ArithmeticException

}
```

```
{

    System.out.println("Inside check function");

    throw new ArithmeticException("demo");

}

public static void main(String args[])

{

    try

    {

        check();

    }

    catch(ArithmeticException e)

    {

        System.out.println("caught" + e);

    }

}

}
```

Copy

Inside check function

caughtjava.lang.ArithmeticException: demo

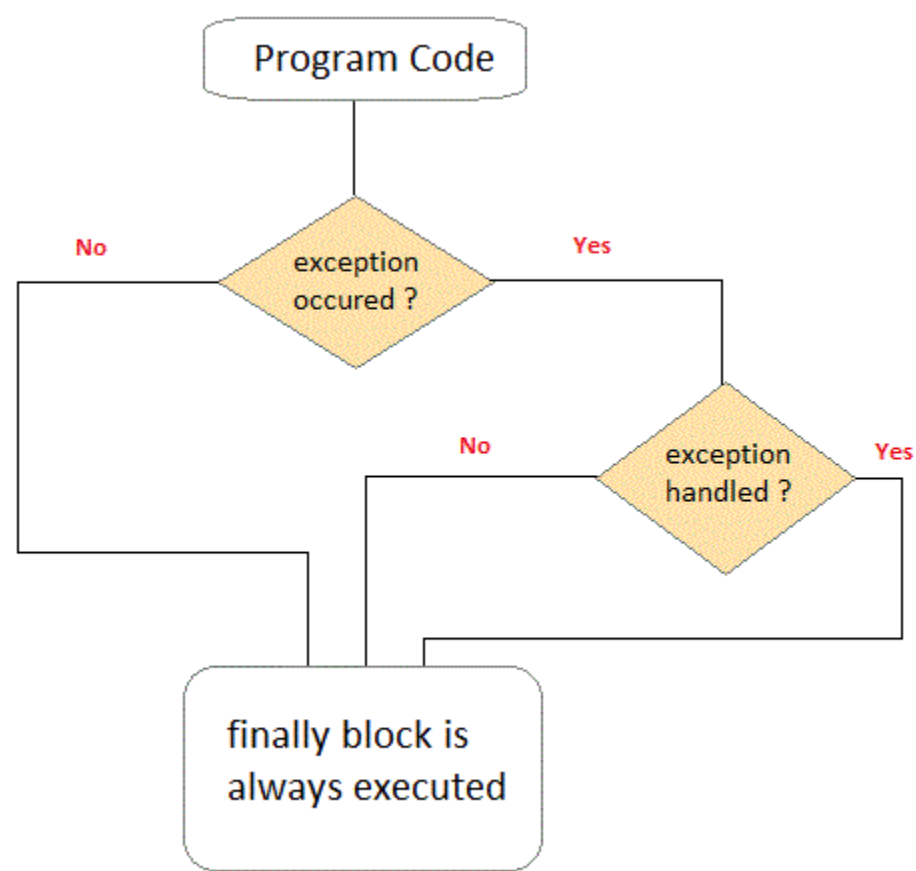
Difference between throw and throws

throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.

throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

finally clause

A finally keyword is used to create a block of code that follows a try block. A finally block of code is always executed whether an exception has occurred or not. Using a finally block, it lets you 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 catch block.



Example finally Block

In this example, we are using finally block along with try block. This program throws an exception and due to exception, program terminates its execution but see code written inside the finally block executed. It is because of nature of finally block that guarantees to execute the code.

```
Class ExceptionTest
{
    public static void main(String[] args)
    {
        int a[] = new int[2];
        System.out.println("out of try");
        try
        {
```

```

        System.out.println("Access invalid element"+ a[3]);

        /* the above statement will throw ArrayIndexOutOfBoundsException
*/

    }

    finally

    {

        System.out.println("finally is always executed.");

    }

}

}

```

Copy

Out of try

finally is always executed.

Exception in thread main java. Lang. exception array Index out of bound exception.

You can see in above example even if exception is thrown by the program, which is not handled by catch block, still finally block will get executed.

Example : Finally Block

finally block executes in all the scenario whether exception is caught or not. In previous example, we use finally where exception was not caught but here exception is caught and finally is used with handler.

```

class Demo

{

    public static void main(String[] args)

    {

        int a[] = new int[2];

        try

        {

            System.out.println("Access invalid element"+ a[3]);

            /* the above statement will throw ArrayIndexOutOfBoundsException
*/

        }

        catch(ArrayIndexOutOfBoundsException e) {

            System.out.println("Exception caught");

        }

    }

}

```

```
    }

    finally

    {

        System.out.println("finally is always executed.");

    }

}

}
```

Copy

Exception caught

finally is always executed.

User defined Exception subclass in Java

Java provides rich set of built-in exception classes like: `ArithmeticException`, `IOException`, `NullPointerException` etc. all are available in the `java.lang` package and used in exception handling. These exceptions are already set to trigger on pre-defined conditions such as when you divide a number by zero it triggers `ArithmeticException`.

Apart from these classes, Java allows us to create our own exception class to provide own exception implementation. These type of exceptions are called user-defined exceptions or **custom** exceptions.

You can create your own exception simply by extending java **Exception** class. You can define a constructor for your Exception (not compulsory) and you can override the `toString()` function to display your customized message on catch. Lets see an example.

Example: Custom Exception

In this example, we are creating an exception class `MyException` that extends the Java Exception class and

```
class MyException extends Exception

{

    private int ex;

    MyException(int a)

    {

        ex = a;

    }

    public String toString()

    {

        return "MyException[" + ex + "] is less than zero";

    }

}
```

```

    }
}

class Demo
{
    static void sum(int a,int b) throws MyException
    {
        if(a<0)
        {
            throw new MyException(a); //calling constructor of user-defined
exception class
        }

        else
        {
            System.out.println(a+b);
        }
    }

    public static void main(String[] args)
    {
        try
        {
            sum(-10, 10);
        }

        catch(MyException me)
        {
            System.out.println(me); //it calls the toString() method of user-
defined Exception
        }
    }
}

```

Copy

```
MyException[-10] is less than zero
```

Example: Custom Exception

Lets take one more example to understand the custom exception. Here we created a class ItemNotFound that extends the Exception class and helps to generate our own exception implementation.

```
class ItemNotFound extends Exception
{
    public ItemNotFound(String s) {
        super(s);
    }
}

class Demo
{
    static void find(int arr[], int item) throws ItemNotFound
    {
        boolean flag = false;
        for (int i = 0; i < arr.length; i++) {
            if(item == arr[i])
                flag = true;
        }
        if(!flag)
        {
            throw new ItemNotFound("Item Not Found"); //calling constructor
of user-defined exception class
        }
        else
        {

```

```
        System.out.println("Item Found");
    }

}

public static void main(String[] args)

{

    try

    {

        find(new int[]{12,25,45}, 10);

    }

    catch (ItemNotFound i)

    {

        System.out.println(i);

    }

}

}
```

Copy

ItemNotFound: Item Not Found

Points to Remember

1. Extend the Exception class to create your own exception class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

Java Method Overriding with Exception Handling

There are few things to remember when overriding a method with exception handling because method of super class may have exception declared. In this scenario, there can be possible two cases: either method of super class can declare exception or not.

If super class method declared exception then the method of sub class may declare same exception class, sub exception class or no exception but can not parent of exception class.

For example, if a method of super class declared ArithmeticException then method of subclass may declare ArithmeticException, its subclass or no exception but cannot declare its super/parent class like: Exception class.

In other scenario, if super class method does not declare any exception, then sub class overridden method cannot declare checked exception but it can declare unchecked exceptions. Lets see an example.

Method overriding in Subclass with Checked Exception

Checked exception is the exception which is expected or known to occur at compile time hence they must be handled at compile time.

```
import java.io.*;

class Super
{
    void show() {
        System.out.println("parent class");
    }
}

public class Sub extends Super
{
    void show() throws IOException //Compile time error
    {
        System.out.println("parent class");
    }

    public static void main(String[] args)
    {
        Super s=new Sub();

        s.show();
    }
}
```

Copy

In the example above, the method `show()` doesn't throw any exception when its declared/defined in the **Super** class, hence its overridden version in the class **Sub** also cannot throw any checked exception.

If we try to do so, we will get a **compile time error**.

Method overriding in Subclass with UnChecked Exception

Unchecked Exceptions are the exception which extend the class **RuntimeException** and are thrown as a result of some runtime error.

```
import java.io.*;

class Super
{
    void show() {
        System.out.println("parent class");
    }
}

class Sub extends Super
{
    void show() throws ArrayIndexOutOfBoundsException
    {
        System.out.println("child class");
    }

    public static void main(String[] args)
    {
        Super s = new Sub();

        s.show();
    }
}
```

Copy

child class

Because **ArrayIndexOutOfBoundsException** is an unchecked exception hence, overridden **show()** method can throw it.

More about Overriden Methods and Exceptions

If Super class method throws an exception, then Subclass overridden method can throw the same exception or no exception, but must not throw parent exception of the exception thrown by Super class method.

It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class).

Example of Subclass overridden method with same Exception

Method of a sub class can declare same exception as declared in the super class. See the below example.

```
import java.io.*;

class Super

{

    void show() throws Exception

        {   System.out.println("parent class");   }

}


public class Sub extends Super {

    void show() throws Exception           //Correct

        {   System.out.println("child class");   }

}


public static void main(String[] args)

{

    try {

        Super s=new Sub();

        s.show();

    }

    catch(Exception e){}

}

}
```

Copy

child class

Example of Subclass overridden method with no Exception

It is optional to declare the exception in sub class during overriding. If method of super class declared an exception then it is upto the subclass to declare exception or not. See the below example.

```
import java.io.*;

class Super

{

    void show() throws Exception

        {   System.out.println("parent class");   }

}

public class Sub extends Super {

    void show()                                //Correct

        {   System.out.println("child class");   }

}

public static void main(String[] args)

{

    try {

        Super s=new Sub();

        s.show();

    }

    catch(Exception e){}

}

}
```

Copy

child class

Example of Subclass overridden method with parent Exception

It is not allowed to declare parent class exception in the subclass method, we get compile time error if we try to compile that program. See the below example.

```
import java.io.*;
```

```

class Super

{

    void show() throws ArithmeticException

    {    System.out.println("parent class");    }

}


public class Sub extends Super {

    void show() throws Exception                //Compile time Error

    {    System.out.println("child class");    }


    public static void main(String[] args)

    {

        try {

            Super s=new Sub();

            s.show();

        }

        catch(Exception e){}

    }

}

```

Copy

Compile time error

Chained Exception in Java

Chained Exception was added to Java in JDK 1.4. This feature allows you 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.

Two new constructors and two new methods were added to **Throwable** class to support chained exception.

1. **Throwable**(Throwable cause)
2. **Throwable**(String str, Throwable cause)

In the first constructor, the paramter **cause** specifies the actual cause of exception. In the second form, it allows us to add an exception description in string form with the actual cause of exception.

getCause() and **initCause()** are the two methods added to **Throwable** class.

- **getCause()** method returns the actual cause associated with current exception.
- **initCause()** set an underlying cause(exception) with invoking exception.

Time for an Example!

Lets understand the chain exception with the help of an example, here, ArithmeticException was thrown by the program but the real cause of exception was IOException. We set the cause of exception using initCause() method.

```
import java.io.IOException;

public class ChainedException
{
    public static void divide(int a, int b)
    {
        if(b == 0)
        {
            ArithmeticException ae = new ArithmeticException("top layer");
            ae.initCause(new IOException("cause"));
            throw ae;
        }
        else
        {
            System.out.println(a/b);
        }
    }

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

```

        divide(5, 0);

    }

    catch(ArithmeticException ae) {

        System.out.println( "caught : " +ae);

        System.out.println("actual cause: "+ae.getCause());

    }

}

}

}

```

Copy

```

caught:java.lang.ArithmeticException: top layer
actual cause: java.io.IOException: cause

```

Example

lets see one more example to understand chain exception, here NumberFormatException was thrown but the actual cause of exception was a null pointer exception.

```

public class ChainedDemo1

{

    public static void main(String[] args)

    {

        try

        {

            NumberFormatException a = new NumberFormatException("====>
Exception");

            a.initCause(new NullPointerException("====> Actual cause of the
exception"));

            throw a;

        }

        catch(NumberFormatException a)

```

```
{

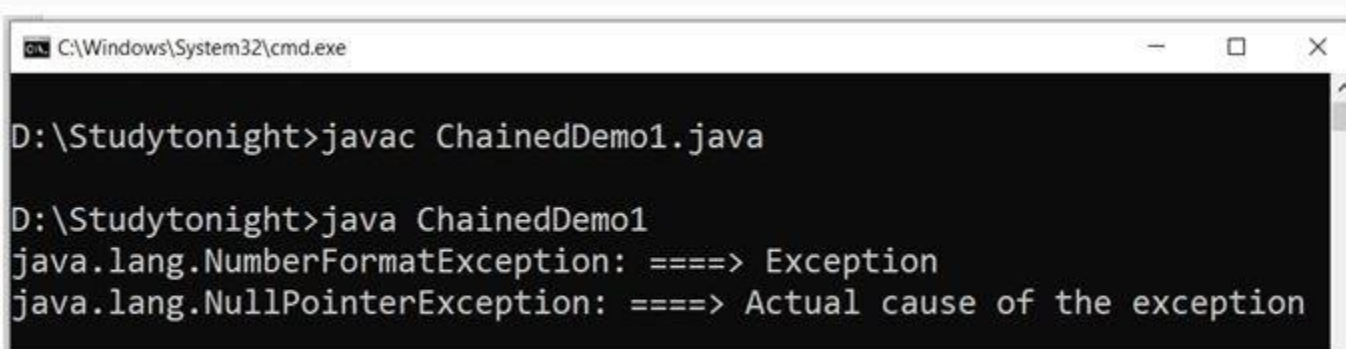
    System.out.println(a);

    System.out.println(a.getCause());

}

}
```

Copy



```
C:\Windows\System32\cmd.exe

D:\Studytonight>javac ChainedDemo1.java

D:\Studytonight>java ChainedDemo1
java.lang.NumberFormatException: ===> Exception
java.lang.NullPointerException: ===> Actual cause of the exception
```

Exception propagation

In Java, an exception is thrown from the top of the stack, if the exception is not caught it is put in the bottom of the stack, this process continues until it reaches to the bottom of the stack and caught. It is known as exception propagation. By default, an unchecked exception is forwarded in the called chain.

Example

In this example, an exception occurred in method a1 which is called by method a2 and a2 is called by method a3. Method a3() is enclosed in try block to provide the safe guard. We know exception will be thrown by method a1 but handled in method a3(). This is called exception propagation.

```
class ExpDemo1{

    void a1()

    {

        int data = 30 / 0;

    }

    void a2()

    {

        a1();

    }

}
```

```

    }

    void a3()

    {

        try {

            a2();

        }

        catch (Exception e)

        {

            System.out.println(e);

        }

    }


    public static void main(String args[])

    {

        ExpDemo1 obj1 = new ExpDemo1();

        obj1.a3();

    }

}

```

Copy

```
java.lang.ArithmeticException: / by zero
```

Example

Lets take another example, here program throw an IOException from method m1() which is called inside the n1(). Exception thrown by method m1() is handled by method n1().

```

import java.io.IOException;

class Demo{

    void m1() throws IOException

    {

        throw new IOException("device error");

    }

}

```

```
void n1() throws IOException

{

    m1();

}

void p1()

{

    try {

        n1();

    }

    catch (Exception e)

    {

        System.out.println("Exception handled");

    }

}

public static void main(String args[])

{

    Demo obj = new Demo();

    obj.p1();

}

}
```

Copy

- [← Prev](#)
- [Next →](#)