Unit-3

Exception Handling:

1.Concepts of Exception handling:-

Dictionary Meaning: Exception is an abnormal condition.

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

Definition:-

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

Run Time Error:-

Run Time errors occur or we can say, are detected during the execution of the program. Sometimes these are discovered when the user enters an invalid data or data which is not relevant. Runtime errors occur when a program does not contain any syntax errors but asks the computer to do something that the computer is unable to reliably do. During compilation, the compiler has no technique to detect these kinds of errors. It is the JVM (Java Virtual Machine) that detects it while the program is running.

Ex:-

```
class TestRunTimeError

{

   public static void main (String args[])

   {

      int a=10;

      int b=0;
```

```
        int c;
        System.out.println("Dividing of two numbers is:"+a+" "+b);
        c=a/b;//raised run time exception
        System.out.println(" Program Ending");
    }
}
```

Out Put:-

Compile Time:-

```
Javac TestRunTimeError.java
```

Run Time:-

```
Java TestRunTimeError
Dividing of two numbers is:10 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

To handle the error during the run time we can use Exception handling to put our error code inside the try block and catch the error inside the catch block.

Example Program2:-

```
class TestRunTimeError
{
    public static void main (String args[])
    {
        int a=10;
        int b=0;
        int c;
        try{

        System.out.println("Dividing of two numbers is:"+a+" "+b);
        c=a/b;
        System.out.println(c);
        }
        catch(Exception e)
        {
            System.out.println(e);
            System.out.println("Number is can't divide by zero");
        }
        System.out.println("Program Ending");
    }
}
```

Out put:-

Compile Time:-

```
Javac TestRunTimeError.java
```

Run Time:-

```
Java TestRunTimeError

Dividing of two numbers is:10 0
java.lang.ArithmeticException: / by zero
Number is can't divide by zero
Program Ending
```

2.Benefits of Exception Handling:-

- It is used to handle runtime errors
- It is used to display message to user when an exception occurs
- It can handle any kind of exception
- You can even execute some code when an exception occurs
- Separating Error-Handling code from "regular" business logic code
- Propagating errors up the call stack
- Grouping and differentiating error types

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 need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

3.Exception Hierarchy:-

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

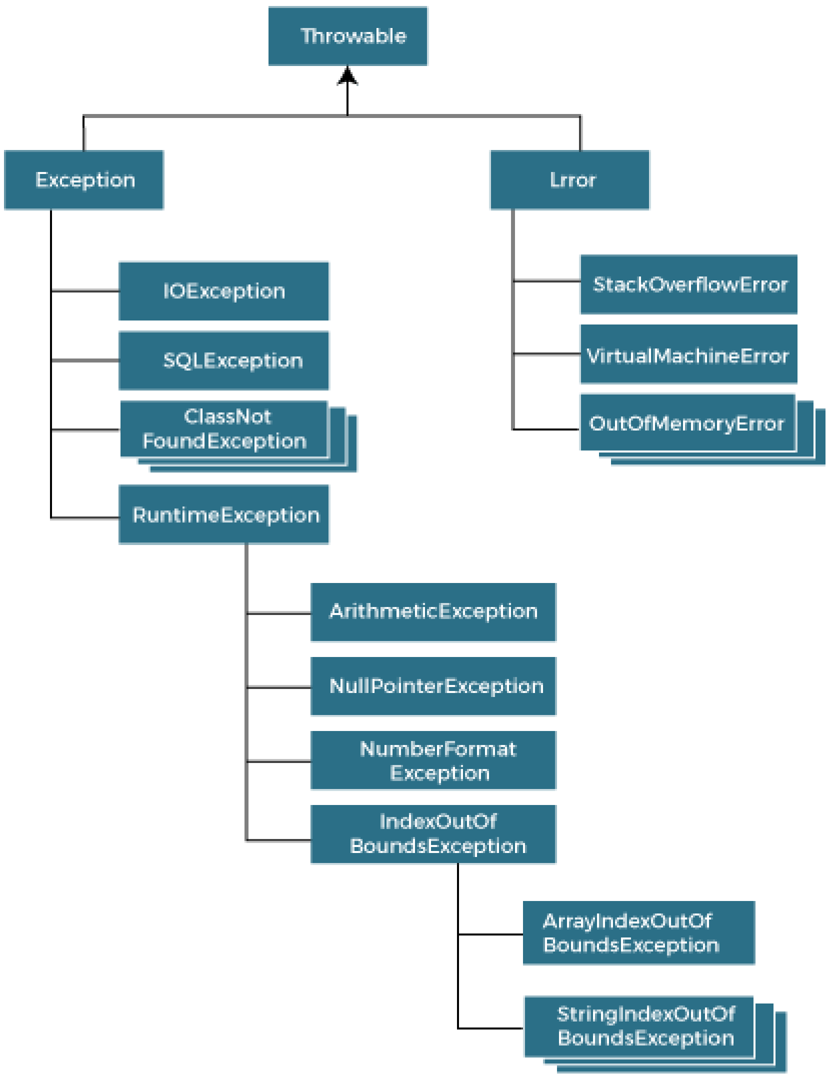


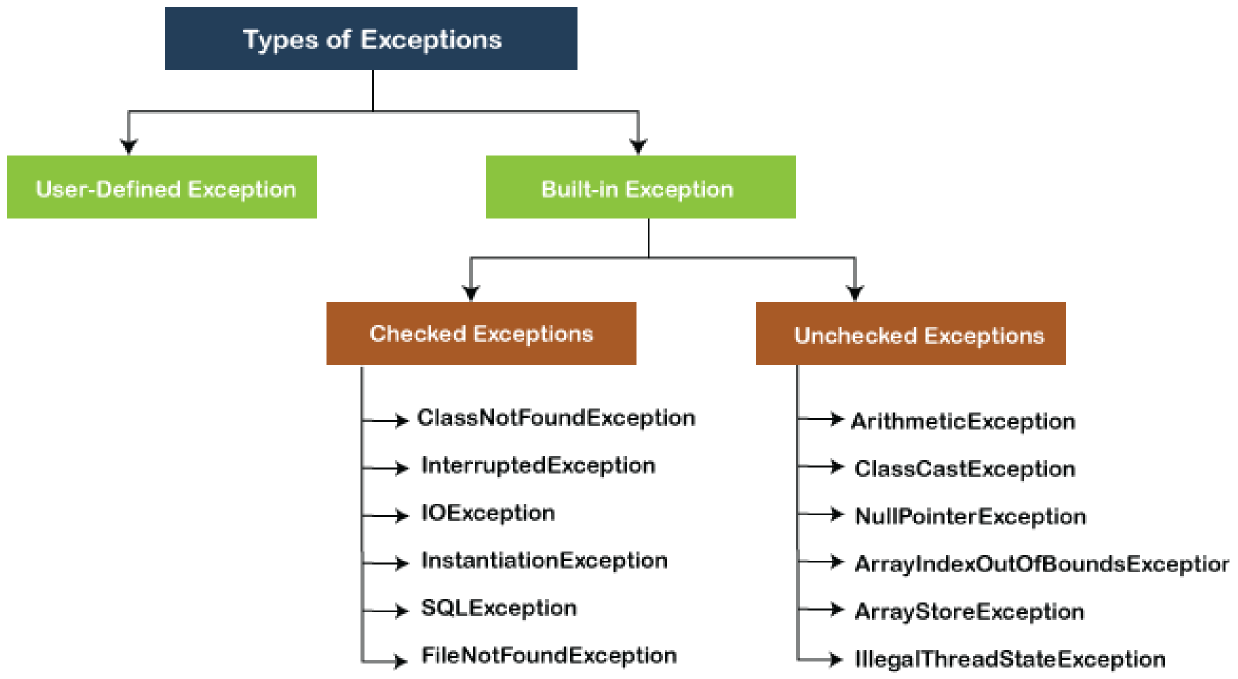
Fig. Exception Hierarchy

- ## Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

Built in exception:-

1. Checked Exception//both checked and unchecked exceptions comes under built-in exceptions
2. Unchecked Exception
3. Error



1.Checked Exception:-

Checked Exceptions are checked by the java Compiler so they are called compile Time Exceptions. Checked Exceptions occur at compile time. These Types of Exceptions can be handled at the time of compilation. Must be handled in a try-and-catch block, or be thrown by the invoking method

Ex:-Raised Compile Time Exception

```
Import java.io.FileReader;
class TestRunTimeError
{
    public static void main (String args[])
    {
FileReader file=new FileReader("somefile.txt");
//raised compile time exception FileNotFoundException
}
}
```

These Types of Exceptions can be handled at the time of compilation. Must be handled in a try-and-catch block, or be thrown by the invoking method

Ex2:-Solved Using Try and Catch Block

```
import java.io.FileReader;
import java.io.FileNotFoundException;
```

```
class TestCheckedException
{
    public static void main (String args[]) throws FileNotFoundException
    {
        try{
FileReader file=new FileReader("somefile.txt");
        }
        catch(Exception e){
            System.out.println(e);
        }
}
}
```

OutPut:-

```
java.io.FileNotFoundException: somefile.txt (The system cannot find the file specified)
```

2.Unchecked Exception:-

Unchecked Exceptions are not checked by the compiler.These are called runtime exceptions.Unchecked exceptions occur at runtime.These types of exceptions cannot be catched or handled at the time of compilation,because they get generated by the mistakes in the program.

Ex:-

```
class TestRunTimeError
{
    public static void main (String args[])
    {
        System.out.println(10/0); //raised run time exception
        System.out.println(" Program Ending");
    }
}
```

OutPut:-

Compile Time:-

```
Javac TestRunTimeError.java
```

Run Time:-//Unchecked by compile time and raised by run time exception is called unchecked exception

Java TestRunTimeError

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

- Handle this unchecked exception using Exception handling keyword try and catch block.

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.

```
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.

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

3) **A scenario where NumberFormatException occurs**

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

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

4) **A scenario where ArrayIndexOutOfBoundsException occurs**

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

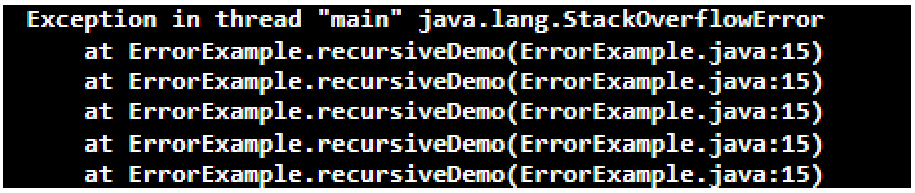
```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

3.Error:-

Errors are problems that mainly occur due to the lack of system resources. It cannot be caught or handled. It indicates a serious problem. It occurs at run time. These are always unchecked. An example of errors is **OutOfMemoryError, LinkageError, AssertionError**, etc. are the subclasses of the Error class.

Example program:-

```
public class ErrorExample
{
public static void main(String args[])
{
//method calling
recursiveDemo(10);
}
public static void recursiveDemo(int i)
{
while(i!=0)
{
//increments the variable i by 1
i=i+1;
//recursive called method
recursiveDemo(i);
}
}
}
```



```
Exception in thread "main" java.lang.StackOverflowError
    at ErrorExample.recursiveDemo(ErrorExample.java:15)
    at ErrorExample.recursiveDemo(ErrorExample.java:15)
    at ErrorExample.recursiveDemo(ErrorExample.java:15)
    at ErrorExample.recursiveDemo(ErrorExample.java:15)
    at ErrorExample.recursiveDemo(ErrorExample.java:15)
```

Difference Between Checked and Unchecked Exception

S. No	Checked Exception	Unchecked Exception
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	The code gives a compilation error in the case when a method throws a checked exception. The compiler is not able to handle the exception on its own.	The code compiles without any error because the exceptions escape the notice of the compiler. These exceptions are the results of user-created errors in programming logic.
4.	These exceptions mostly occur when the probability of failure is too high.	These exceptions occur mostly due to programming mistakes.

5.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
6.	These exceptions are propagated using the throws keyword.	These are automatically propagated.
7.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.

Difference Between Exception and Error

In Java, Error, and Exception both are subclasses of the Java Throwable class that belongs to java.lang package.

Basis of Comparison	Exception	Error
Recoverable/ Irrecoverable	Exception can be recovered by using the try-catch block.	An error cannot be recovered.
Type	It can be classified into two categories i.e. checked and unchecked.	All errors in Java are unchecked.
Occurrence	It occurs at compile time or run time.	It occurs at run time.
Package	It belongs to java.lang.Exception package.	It belongs to java.lang.Error package.
Known or unknown	Only checked exceptions are known to the compiler.	Errors will not be known to the compiler.
Causes	It is mainly caused by the application itself.	It is mostly caused by the environment in which the application is running.
Example	**Checked Exceptions:** SQLException,IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundException, NullPointerException, ArithmaticException	Java.lang.StackOverflowError, java.lang.OutOfMemoryError

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
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 necessary 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 specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

4.Usage of Try and catch :-

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 in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

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.

Java try block must be followed by either catch or finally block.

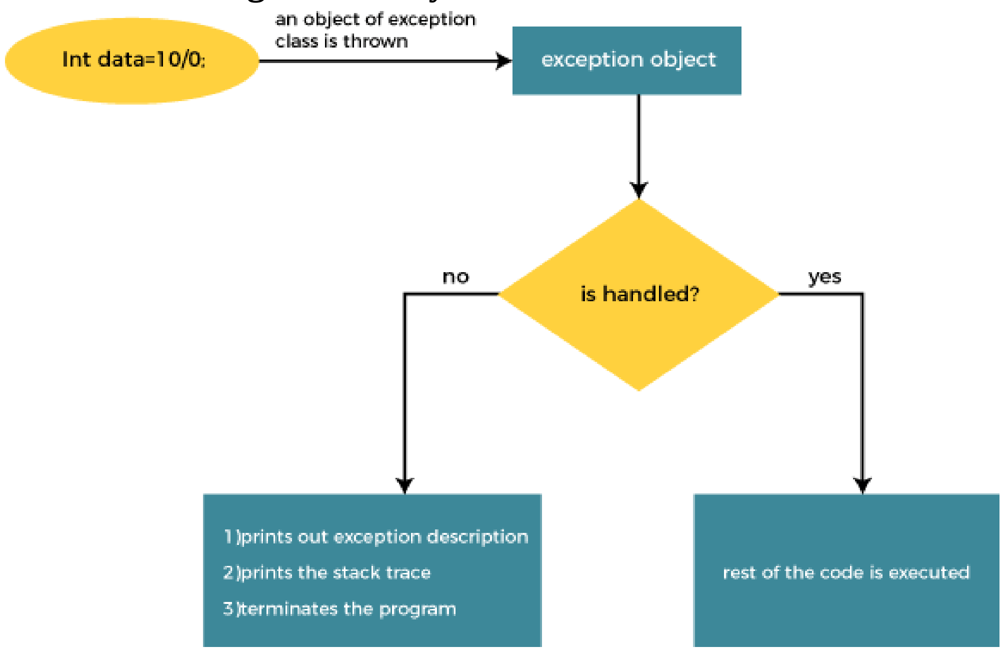
Syntax of Java try-catch

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception_class_Name ref){}

Syntax of try-finally block

1. **try**{
2. //code that may throw an exception
3. }**finally**{}

Internal Working of Java try-catch block


Int data=10/0;
an object of exception class is thrown
exception object
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Example program:-

public class TryCatchExample2 {

```
  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
      // if exception occurs, the remaining statement will not exceute
            System.out.println("try block code");

    }
      //handling the exception
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }

}
```

Output:

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

rest of the code
```

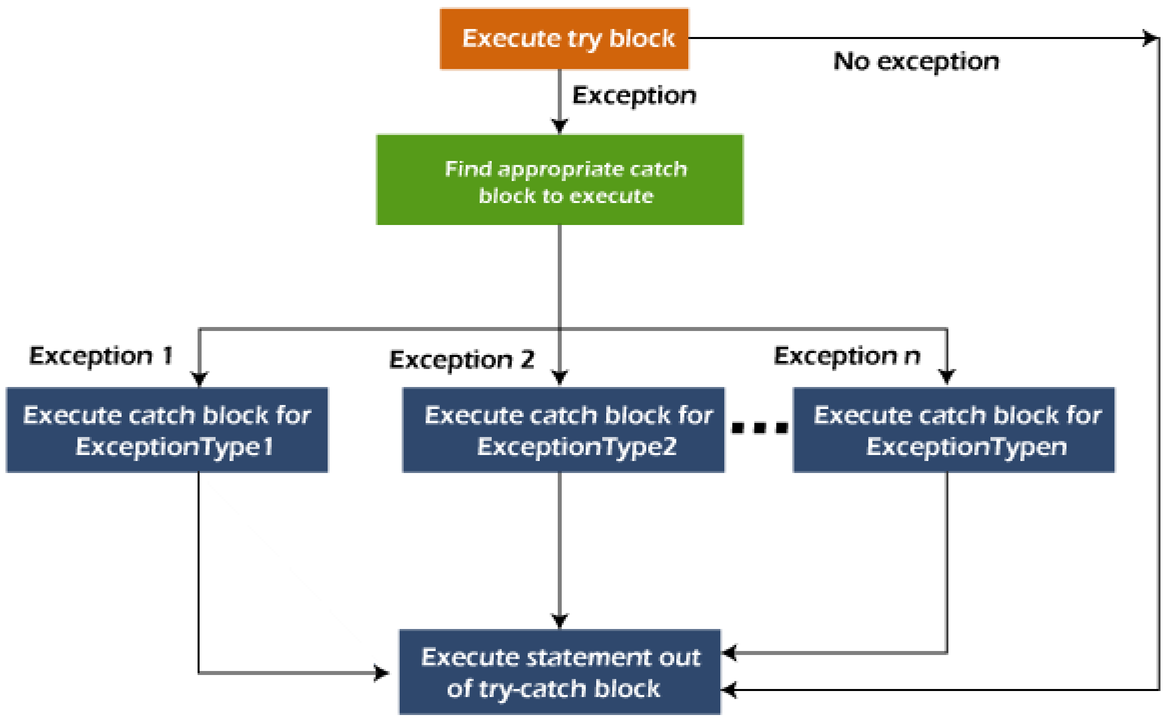
- ## Multiple Catch Block:-

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.

Flowchart of Multi-catch Block


Execute try block
No exception
Exception
Find appropriate catch block to execute
Exception 1
Exception 2
Exception n
Execute catch block for ExceptionType1
Execute catch block for ExceptionType2
Execute catch block for ExceptionTypen
Execute statement out of try-catch block

Example Program for Multiple Catch Block:-

```
public class MultipleCatchBlock1 {

  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("ArrayIndexOutOfBounds Exception occurs");
         }
       catch(Exception e)
         {
          System.out.println("Parent Exception occurs");
```

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

Arithmetic Exception occurs

Output:

```
Arithmetic Exception occurs
rest of the code
```

- **Multiple try Block:-**

One program you can write multiple try blocks and You cannot have multiple try blocks with a single catch block. Each try block must be followed by catch or finally. Still if you try to have single catch block for multiple try blocks a compile time error is generated.

Example program for Multiple try Block:-

```
class MultipleTryCatchBlock{
  public static void main(String args[]){
   try{
    int a=10,b=20,c;
    c=30/0;
System.out.println(c);
   }

   catch(ArithmeticException e){System.out.println("task1 is completed");}

try{
    int a[]=new int[3];
    a[4]=10;
}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

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

Output:-

```
task1 is completed
task 2 completed
rest of the code...
```

- **Nested try Block:-**

A try block which contain inside another try block called nested try block.

```
Syntax:
....
//main try block
```

```
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e2)
{
//exception message
}
....
```

Example Program for nested try block:-

```
class NestedTry {

   // main method
   public static void main(String args[])
   {
      // Main try block
      try {

         // initializing array
         int a[] = { 1, 2, 3, 4, 5 };

         // trying to print element at index 5
         System.out.println(a[5]);

         // try-block2 inside another try block
         try {
                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
        }
    }
    // end of main method
}
```

Output:

```
ArrayIndexOutOfBoundsException
Element at such index does not exists
```

Whenever a try block does not have a catch block for a particular exception, then the catch blocks of parent try block are inspected for that exception, and if a match is found then that catch block is executed.

- **Nested catch Block:-**

An catch block which contain inside another catch block is called nested catch block.

Example Program:-

```
class NestedCatch {

    // main method
    public static void main(String args[])
    {
        // Main try block
        try {

            // initializing array
            int a[] = { 1, 2, 3, 4, 5 };

            // trying to print element at index 5
            System.out.println(a[5]);

            // try-block2 inside another try block
        }
        catch (ArrayIndexOutOfBoundsException e1) {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("Element at such index does not exists");
            try {
               int a[] = { 1, 2, 3, 4, 5 };
                // performing division by zero
                int x = a[2] / 0;
            }
            catch (ArithmeticException e2) {
                System.out.println("division by zero is not possible");
            }
        }
    }
    // end of main method
}
```

Output:-

```
ArrayIndexOutOfBoundsException
Element at such index does not exists
division by zero is not possible
```

5.Usage of Finally Block:-

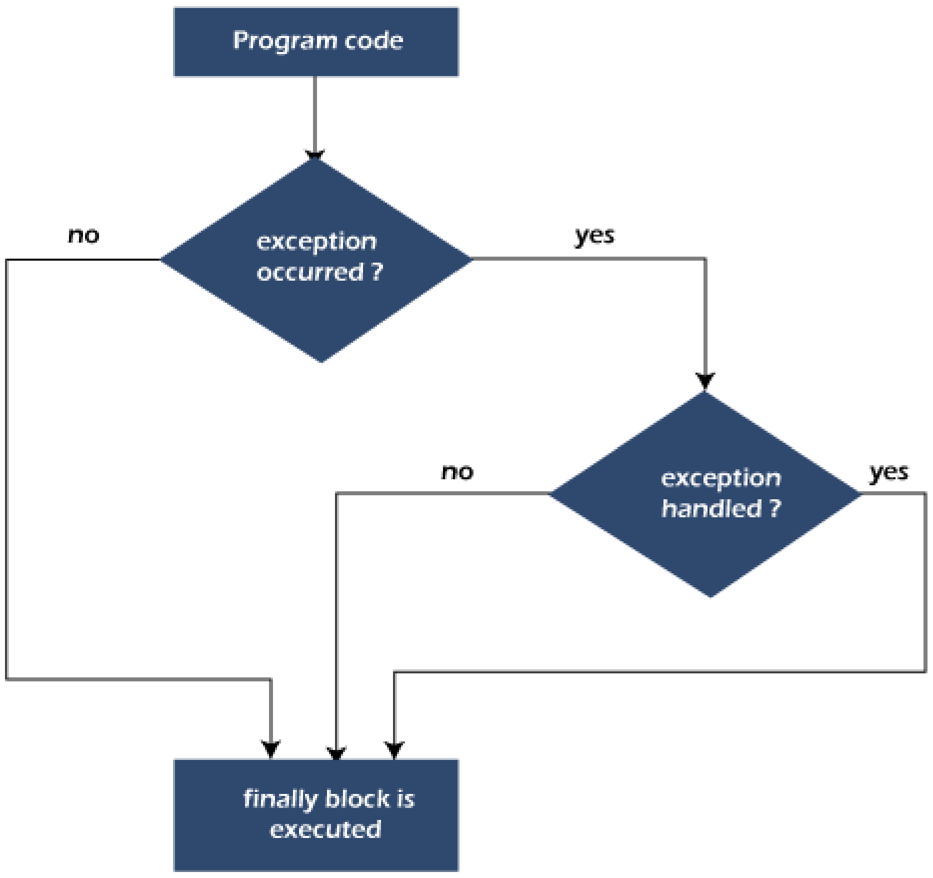
Defination:-Java finally block is a block used to execute important code such as closing the connection, etc. Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Syntax:-

```
try

{

}

finally

{

}
```

Flowchart of finally block



Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Finally block executed following cases

Case 1: When an exception does not occur Finally Block Executed

Example program1:-

```
class TestFinallyBlock {
  public static void main(String args[]){
//first final block without error
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
```

```
  catch(Exception e){
System.out.println(e);
}

//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed1");
}
  System.out.println("rest of phe code...");
```
//second final block with error

```
  try {
       int data=25/0;
       System.out.println(data);
      }

      catch(ArithmeticException e){
        System.out.println(e);
      }
  finally {
        System.out.println("finally block is always executed2");
      }
```
//third final block with error and wrong exception class

```
try {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception

       int data=25/0;
       System.out.println(data);
      }
      //cannot handle Arithmetic type exception
      //can only accept Null Pointer type exception
      catch(NullPointerException e){
        System.out.println(e);
      }

      //executes regardless of exception occured or not
      finally {
```

```
      System.out.println("finally block is always executed3");
    }

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

Output:-

```
5
finally block is always executed1
rest of phe code...
java.lang.ArithmeticException: / by zero
finally block is always executed2
Inside the try block
finally block is always executed3
    at Exception1.TestFinallyBlock.main(TestClass.java:212)
C:\Users\Admin\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned:
1
BUILD FAILED (total time: 0 seconds)
```

6.Usage of throw:-

Defination:-Throw keyword is used to throw the user defined or customize exception object to the JVM explicitly for that purpose we use throw keyword.
In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions(user defined exception) making the code recovery and debugging easier.

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

- **throw new** exception_class("error message");

Let's see the example of throw IOException.

- **throw new** IOException("sorry device error");

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

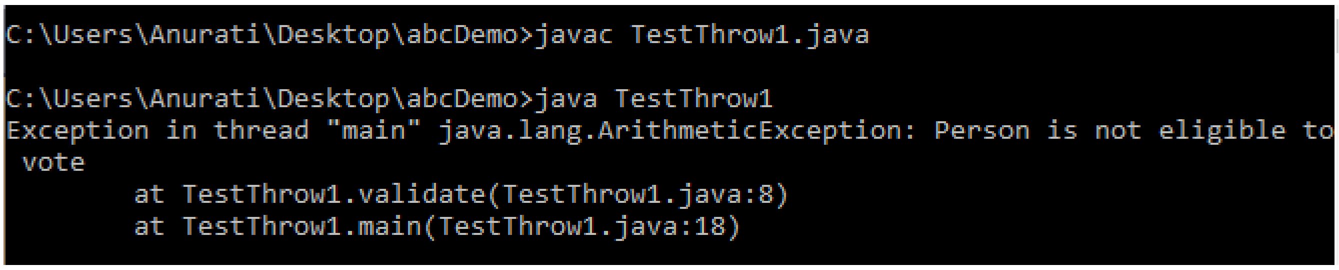
Example 1: Throwing Unchecked Exception

In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
    //main method
    public static void main(String args[]){
        //calling the function
        validate(13);
        System.out.println("rest of the code...");
  }
}
```

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

7.Usage of throws:-

Defination:-throws keyword is used when we doesn’t want to handle the exception and try to send the exception to the JVM(JVM or other method)

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 the normal flow of the program 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 checking the code before it being used.

Syntax of Java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

Q).Which exception should be declared?

Ans: Checked exception only, because:

- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Example program:-

```
public class TestClass {

  public static void display() throws InterruptedException
  {
    for(int i=0;i<=10;i++)
    {
      System.out.println(i);
      Thread.sleep(1000);
    }
  }
  public static void main(String args[]) throws InterruptedException
  {

  display();
    try
    {
      }
      finally{
           System.out.println("Successfully Executed");
       }
   }
}
```

Output:-

```
0
1
2
3
4
5
6
7
8
9
10
Successfully Executed
```

8.Java Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Exception Propagation Example

TestExceptionPropagation1.java

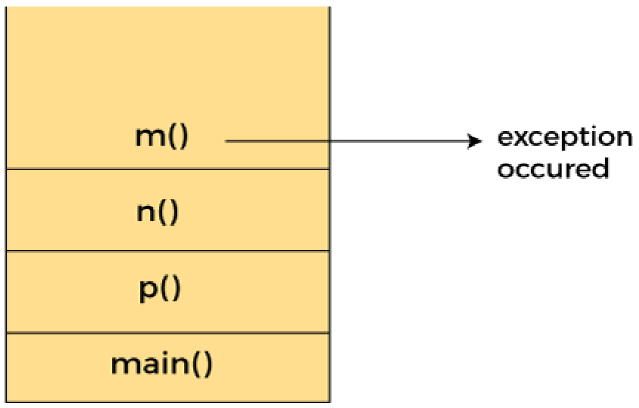
```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
       normal flow...
```

In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.

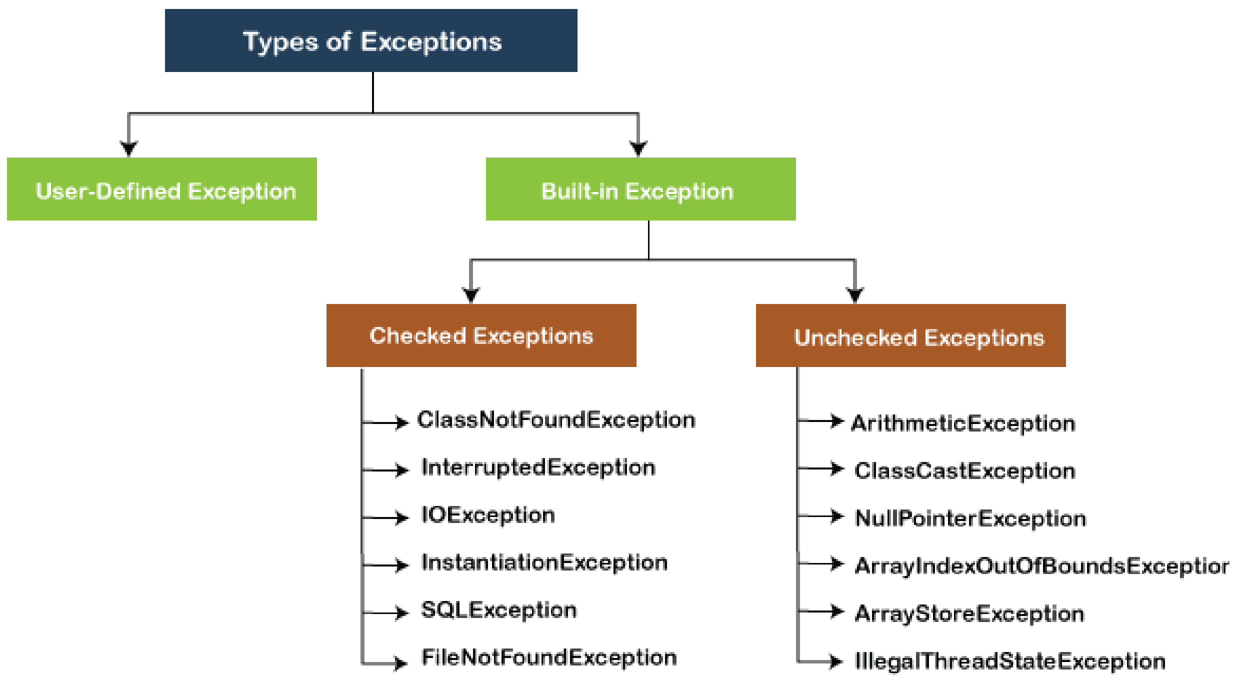


Call Stack

9.Built in Exceptions:-

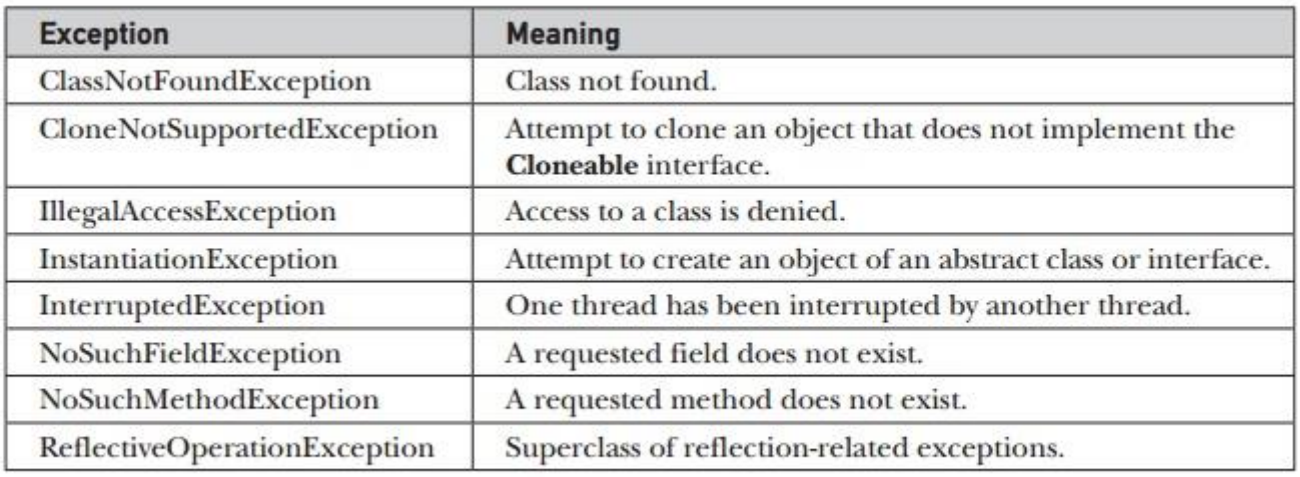
Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.



Checked Exception

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.



Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java's Checked Exceptions Defined in **java.lang**

Example program:-

1. **FileNotFoundException :** This Exception is raised when a file is not accessible or does not open.

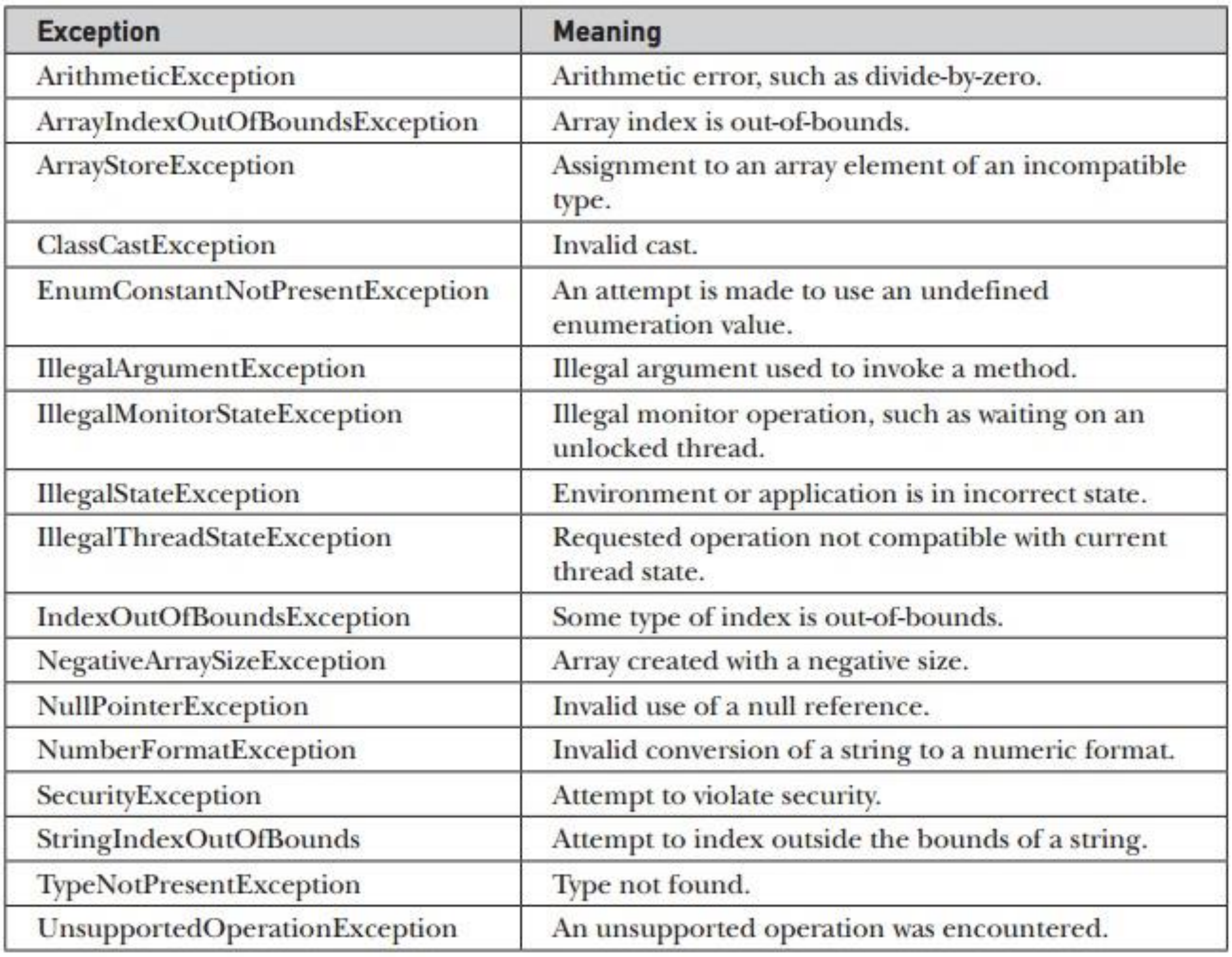
```
// Java program to demonstrate
// FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo {

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

        // Following file does not exist
        File file = new File("E:// file.txt");

        FileReader fr = new FileReader(file);
     }
     catch (FileNotFoundException e) {
        System.out.println("File does not exist");
     }
  }
}
```

Output:

```
File does not exist
```

Unchecked Exceptions

The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.



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.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
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.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Example Program:-

1. **ArrayIndexOutOfBounds Exception :** It is 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.

```
class ArrayIndexOutOfBound_Demo {
public static void main(String args[])
    {
        try {
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of
            // size 5
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index is Out Of Bounds");
        }
    }
}
```

Output:

```
Array Index is Out Of Bounds
```

Build-in Exceptions example programs:-

1. **Class Not Found Exception:** ClassNotFoundException exception is caused when the Java Virtual Machine is unable to find the required class. This type of exception is generally thrown by Class.forName(), ClassLoader.findSystemClass() or ClassLoader.loadClass() functions. Since this is a checked exception, we need to use the throws keyword to specify that this method may cause the class not found exception, otherwise our code will not compile. Example code causing such an exception:

Code:

```
public class classNotFound
{
   static String classname = "missingClass";
   public static void main(String args[]) throws ClassNotFoundException
   {
      Class.forName(classname);
   }
}
```

Output:

```
java.lang.ClassNotFoundException: missingClass
```

Explanation: The Class.forName() function returns the object of the class or interface whose name is passed in the parameter as a string. Now, if there is no class with the given name then this will cause the ClassNotFoundException, and it will terminate the execution of our code.

2. **Interrupted Exception:** Threads are used in Java to improve the efficiency of the code by allowing it to do multiple things together at the same time. Threads are extended from the predefined Thread class in Java. It provides various functions like sleep which allows us to temporarily stop the execution of the thread (for the specified number of milliseconds). InterruptedException is thrown when a thread is sleeping, waiting, or occupied and it is disturbed.

Let us look at an example code throwing such an exception:

Code:

```
class interruptException extends Thread
{
   public void run()
   {
      try
      {
         // code causing the exception
         Thread.sleep(1000);
      }
      /* we need to use try catch, because with run
       * method we can not use throws */
      catch (InterruptedException e)
      {
         System.err.println(e);
      }
   }
   public static void main(String args[])
   {
      interruptException obj = new interruptException();
      obj.start();
      obj.interrupt();
   }
}
```

Output:

```
java.lang.InterruptedException: sleep interrupted
```

Explanation: Here we have first made an object of the class and then used the threads to start the execution. While the thread is occupied with the task of sleep, we interrupt it using the interrupt function. This causes the InterruptedException since the thread is disturbed.

3. **IO Exception** IOException is one of the most commonly handled exceptions. It is thrown when there is some sort of discrepancy in Input or Output. Using throws suppresses it and not using throws will give a compile-time error. Let us look at the code for the same:

Code:

```
import java.io.*;
import java.util.*;
public class ioexception
{
   public static void main(String args[])
   {
      String dir = "readThisFile.txt";
      Scanner sc = new Scanner(new File(dir));
   }
}
```

Compile Time Exception:

The possible exception of type java.io.FileNotFoundException needs to be handled

Explanation: Since opening the file can cause IOExceptions, the compiler gives a compile-time exception and the code is not compiled.

4. **Instantiation Exception** An object is called an instance of a class. Now, a class can not be instantiated if it is an abstract class, an interface, or a void class. InstantiationException is thrown when we try to create an instance of a class using the newInstance method, but that class can not be instantiated. A simple scenario where a class can not be instantiated is when the class which we want to instantiate is an abstract class. Let us see an example where we use the newInstance method to throw InstantiationException. This exception is also thrown at the compile time.

Code:

```
import java.util.*;
public class instexception
{
   public static void main(String args[])
   {
      Class cls = Class.forName("instexception");
      instexception obj = (instexception)cls.newInstance();
   }
}
```

Compile Time Exception

The possible exception of type java.lang.InstantiationException needs to be handled.

Explanation: Here we are using the forName function (discussed in *ClassNotFoundException*) to create an instance (object) of the class. While doing so, the newInstance function might throw *InstantiationException*, which is why the compiler is giving compile time error.

5. **SQL Exception** *SQLException* is thrown if there is an error in database access or other database errors. To access the Database, we use various functions like Connection, DriverManager, and getConnection. Let us say that we want to access a database at some URL, now if that URL is not accessible to the code, then it will throw the *SQLException*. Let us see the code for the same. We will use throws to handle the *SQLException*.

Code:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class sqlexception
{
   public static void main() throws SQLException
   {
      Connection conn = DriverManager.getConnection("Database_URL");
   }
}
```

Output:

```
java.sql.SQLException: No suitable driver found for Database_URL
at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:708)
at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:253)
at Main.main(Main.java:9)
```

Explanation: Here we are using the getConnection method to access a database, but since the URL is not accessible, the code is throwing an *SQLException*.

6. **FileNotFoundException** FileNotFoundException is thrown when we try to access a file in a directory, and the file is not found. Let us see an example of a code throwing this error:

Code:

```
import java.io.*;
import java.util.*;
class filenotfound
{
   public static void main(String args[]) throws FileNotFoundException
   {
      String dir = "missingfile.txt";
      Scanner sc = new Scanner(new File(dir));
   }
}
```

Output:

```
java.io.FileNotFoundException: missingfile.txt (No such file or directory)
```

Explanation: Here we are trying to access a file using the File class. But the file is not present in our system. We have used throws to suppress the compile time error. Now, this code throws the *FileNotFoundException* while executing since the file is not found.

Unchecked Exceptions An *Unchecked Exception* is an exception that can only be thrown at the run time (during the execution of the code). These are also called Runtime Exceptions. It includes logical errors, bugs, or improper use of functions. We do not have to declare unchecked exceptions using the throws keyword. A code with unchecked exceptions compiles correctly. Arithmetic Exception is a perfect example of unchecked exceptions. A code that divides a number by 0 compiles without throwing any problems, but it throws an Arithmetic Exception upon execution. Let us discuss some unchecked exceptions in detail:

7. **Arithmetic Exceptions** An *ArithmeticException* is thrown when there is wrong arithmetic or mathematical operation done by the code while executing. Divide by 0 is the most common type of wrong mathematical operation. Let us see the code for the same:

Code:

```
class arithmeticException
{
   public static void main()
   {
      int a = 10, b = 0;
      int c = a / b;
   }
}
```

Output:

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

Explanation: Since we are trying to divide 10 by 0, we are causing a mathematical error. This is predefined in the ArithmeticException of Exception class in Java. Hence the code is throwing the exception.

8. **Class Cast Exception** Type casting is changing the type from one type to another. Casting a class is changing its type. *ClassCastException* is thrown by JVM when we try to cast a class from one type to another, and it violates some rules.

Code:

```
class Animal

{

 }

class Dog extends Animal

{

   public static void main(String args[])

   {

     Animal a=new Animal();

     Dog d=(Dog)a;

   }

}
```

Output:

```
ja Exception in thread "main" java.lang.ClassCastException: test.Animal cannot be cast to test.Dog
```

9. **Null Pointer Exception** *NullPointerException* is thrown by the JVM when we try to access a pointer that is pointing to Null (or Nothing). Pointing to Null means that no memory is allocated to that specific object. Let us say we define a string as Null, and then try to access it, then JVM will throw the *NullPointerException*. Let us look at the code for the same:

Code:

```
class nullpointerexception
{
   public static void main()
   {
      String s = null;
      System.out.println(s.length());
   }
}
```

Output:

```
java.lang.NullPointerException
```

Explanation: Here we have defined the s as null. Now, since the length method does not support null strings, we are getting the *NullPointerException*.

10. **Array Index Out of Bounds Exception** *ArrayIndexOutOfBounds* is one of the most common unchecked exceptions. It is thrown when we try to access an array index that does not exist. Let us say that we have an array of size 10, and we try to access the 15th element. Then JVM will throw an *ArrayIndexOutOfBounds* exception. Let us look at the code for the same:

Code:

```
class arrayindexoutofbounds
{
   public static void main()
   {
      int arr[] = {1,2,3,4,5,6,7,8,9,10};
      System.out.println(arr[15]);
   }
}
```

Output

```
java.lang.ArrayIndexOutOfBoundsException: Index 15 out of bounds for length 10
```

Explanation: Here we have defined an integer array of size 10. We are trying to access the 15th element, but since 15 is out of bounds, the code is throwing the *ArrayIndexOutOfBoundsException*.

11. **Array Store Exception** *ArrayStoreException* is thrown by JVM when we try to store the wrong type of object in the array of objects. Let us say we have an object array of double and we try to store an integer. This will cause an exception at run time since there is a type mismatch. Let us look at the code for the same:

Code:

```
class arraystoreexception

{

   public static void main(String args[])

   {

      Object[] array = new Double[2];

      array[1] = 10;

   }
}
```

Output:

```
java.lang.ArrayStoreException: java.lang.Integer
```

Explanation: Here we have defined the array as of Double type. Then we are trying to store an Integer value in it which is causing the ArrayStoreException.

12. **Illegal Thread State Exception** *IllegalThreadStateException* is thrown by JVM when a thread is not in a proper state to perform the given operations. An example of the same would be to give commands to the thread while it is sleeping. Since we can get *InterruptException* as well, we will handle it using the throws keyword. Let us look at the code for the same:

Code:

```
class thread extends Thread

{

    public void run()

    {

        System.out.println("Welcome to CMRIT");

    }

}

class illegalthreadstateexception

{

    public static void main(String args[])

    {

        thread t = new thread();

        t.start();

        t.start();

    }

}
```

Output:

```
Welcome to CMRIT

Exception in thread "main" java.lang.IllegalThreadStateException
```

Explanation:

Firstly, we have started processing the thread using the start method. Then we asked it to sleep for *1000 ms*. Now while it was sleeping, we again used the start method. This has caused the *IllegalThreadStateException*.

13.NumberFormateException:-

```
public class UncheckedEx {
    public static void main(String args[])
    {
        try{
       String s="a";
       int a=Integer.parseInt(s);
      System.out.println(a);
        }
        catch(NumberFormatException o)
        {
            System.out.println(o);
        }
      System.out.println("rest of the code");
    }
}
```

O/P:-

```
java.lang.NumberFormatException: For input string: "a"
rest of the code
```

14.StringIndexOutOfBoundsException:-

```
public class UncheckedEx {
    public static void main(String args[])
    {
       String s="abc";
      System.out.println(s.charAt(6));
      System.out.println("rest of the code");
    }
}
```

O/P:-

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 6

at java.lang.String.charAt(String.java:646)

at test.UncheckedEx.main(UncheckedEx.java:77)
```

10.Creating own Exception sub classes:-

Custom Exception or User Defined Exception:-

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message.

Following are few of the reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions.
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

```
public class WrongFileNameException extends Exception {
   public WrongFileNameException(String errorMessage) {
   super(errorMessage);
   }
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

Example 1:

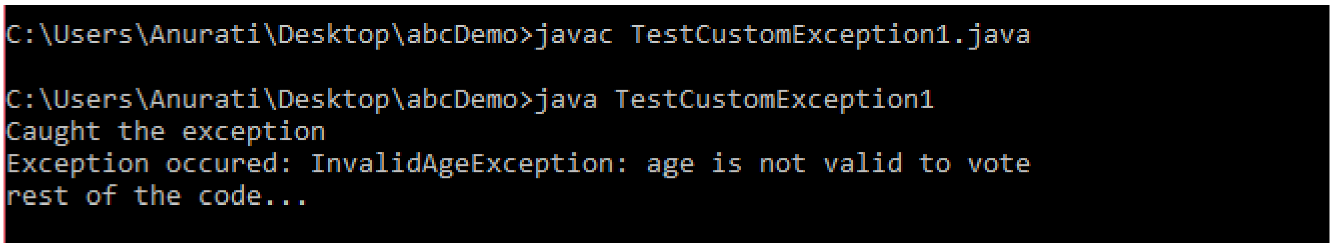
Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

TestCustomException1.java

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

   // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){

      // throw an object of user defined exception
      throw new InvalidAgeException("age is not valid to vote");
   }
     else {
      System.out.println("welcome to vote");
      }
    }

   // main method
   public static void main(String args[])
   {
      try
      {
         // calling the method
         validate(13);
      }
      catch (InvalidAgeException ex)
      {
         System.out.println("Caught the exception");

         // printing the message from InvalidAgeException object
         System.out.println("Exception occured: " + ex);
```

```
        }

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

Output:



```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Difference between throw and throws in Java

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.		Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.

5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Difference between final, finally and finalize

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.

3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.