

## UNIT-3

### Exception Handling

The **exception handling in java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

#### What is exception

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

#### Exception Handling Fundamentals :

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. They form an interrelated subsystem in which the use of one implies the use of another. Throughout the course of this chapter, each keyword is examined in detail. However, it is useful at the outset to have a general understanding of the role each plays in exception handling. Briefly, here is how they work.

Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is **thrown**. Your code can catch this exception using **catch** and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system.

#### Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

## Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

### Difference between checked and unchecked exceptions :

- 1) Checked Exception:** The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.
- 2) Unchecked Exception:** The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
- 3) Error:** Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Termination Model :

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

## Resumptive Model :

The alternative of termination model is resumptive model. In resumptive model, the exception handler is expected to do something to stable the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that want resumption like behavior. We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

## Uncaught Exceptions in Java :

In java, assume that, if we do not handle the exceptions in a program. In this case, when an exception occurs in a particular function, then Java prints a exception message with the help of uncaught exception handler.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allow us to handle the exception use the keywords like try, catch, finally, throw, and throws.

When an uncaught exception occurs, the JVM calls a special private method known dispatchUncaughtException( ), on the Thread class in which the exception occurs and terminates the thread.

The Division by zero exception is one of the example for uncaught exceptions. Look at the following code.

### Example:

```
import java.util.Scanner;

public class UncaughtExceptionExample {

    public static void main(String[] args) {

        Scanner read = new Scanner(System.in);
        System.out.println("Enter the a and b values: ");
        int a = read.nextInt();
        int b = read.nextInt();
        int c = a / b;
```

```
        System.out.println(a + "/" + b + " = " + c);  
  
    }  
}
```

In the above example code, we are not using try and catch blocks, but when the value of b is zero the division by zero exception occurs and it is caught by the default exception handler.

## **Java Catch Multiple Exceptions**

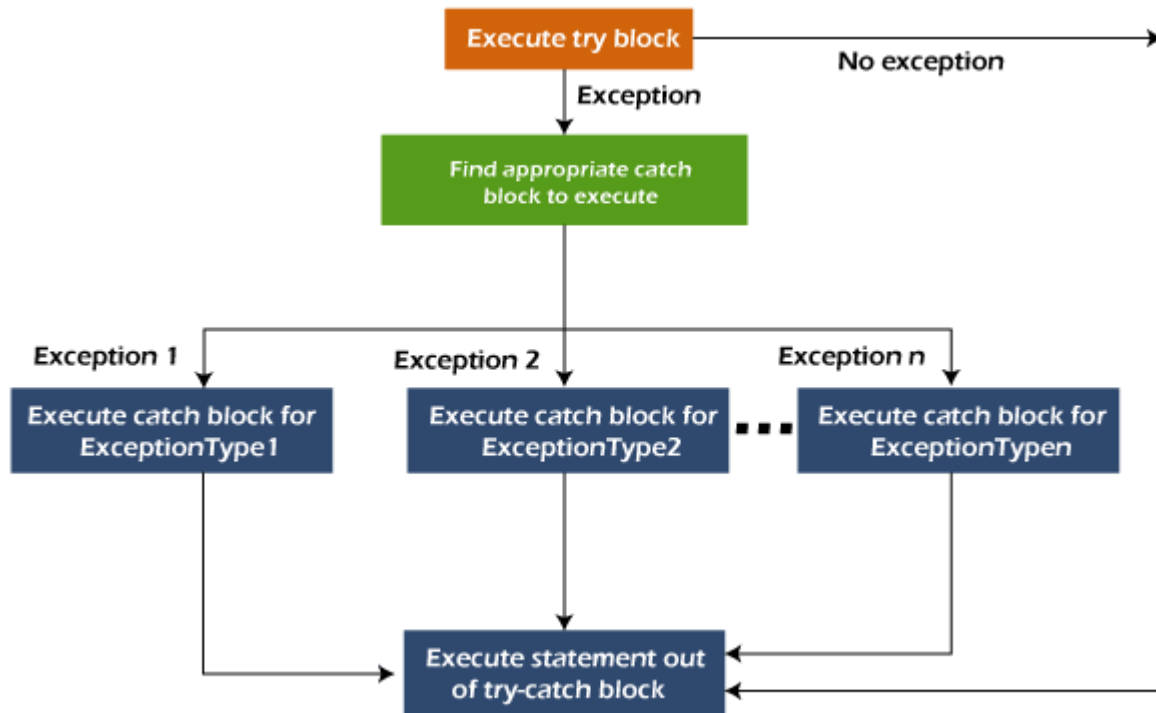
### **Java Multi-catch block**

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

#### **Points to remember**

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

#### **Flowchart of Multi-catch Block :**



### Example 1

Let's see a simple example of java multi-catch block.

#### MultipleCatchBlock1.java:

```

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

```

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

```

## Java Nested try block :

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithmeticException** (division by zero).

## Why use nested try block :

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

*Syntax:*

```

....
//main try block
try
{
    statement 1;
    statement 2;
//try catch block within another try block
    try
    {
        statement 3;
        statement 4;
//try catch block within nested try block
        try
        {
            statement 5;
            statement 6;
        }
        catch(Exception e2)
        {
//exception message
        }

    }
    catch(Exception e1)

```

```

    {
//exception message
    }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....

```

## Java Nested try Example

### *Example 1*

Let's see an example where we place a try block within another try block for two different exceptions.

```

public class NestedTryBlock{
public static void main(String args[]){
//outer try block
try{
//inner try block 1
    try{
        System.out.println("going to divide by 0");
        int b =39/0;
    }
//catch block of inner try block 1
    catch(ArithmeticException e)
    {
        System.out.println(e);
    }

//inner try block 2
    try{
        int a[]=new int[5];

//assigning the value out of array bounds
        a[5]=4;
    }

//catch block of inner try block 2
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println(e);
    }
}
}

```

```

        System.out.println("other statement");
    }
    //catch block of outer try block
    catch(Exception e)
    {
        System.out.println("handled the exception (outer catch)");
    }

    System.out.println("normal flow..");
}
}

```

### Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..

```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

## Example 2

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

### NestedTryBlock.java

```

public class NestedTryBlock2 {

    public static void main(String args[])
    {

```



```

// outer (main) try block
try {

    //inner try block 1
    try {

        // inner try block 2
        try {
            int arr[] = { 1, 2, 3, 4 };

            //printing the array element out of its bounds
            System.out.println(arr[10]);
        }

        // to handles ArithmeticException
        catch (ArithmeticException e) {
            System.out.println("Arithmetic exception");
            System.out.println(" inner try block 2");
        }
    }

    // to handle ArithmeticException
    catch (ArithmeticException e) {
        System.out.println("Arithmetic exception");
        System.out.println("inner try block 1");
    }
}

// to handle ArrayIndexOutOfBoundsException
catch (ArrayIndexOutOfBoundsException e4) {
    System.out.print(e4);
    System.out.println(" outer (main) try block");
}
catch (Exception e5) {
    System.out.print("Exception");
    System.out.println(" handled in main try-block");
}
}
}

```

**Output:**

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer
(main) try block
```

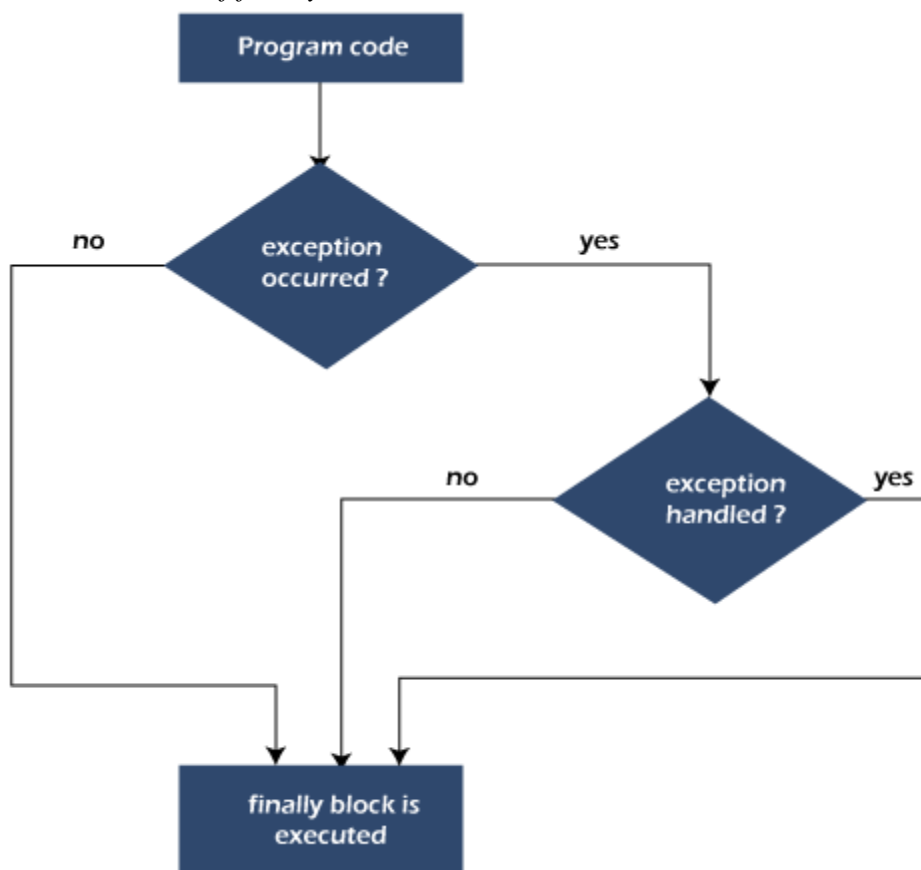
## Java finally block

**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.

*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.

#### Usage of Java finally

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

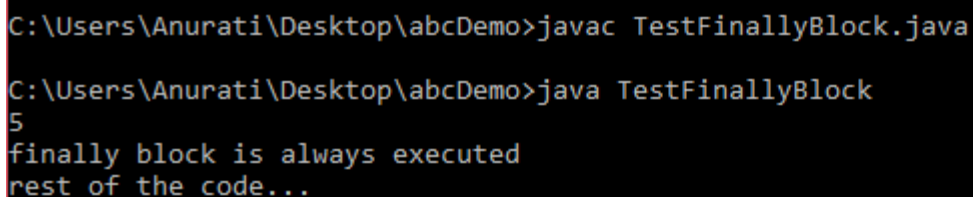
##### *Case 1: When an exception does not occur*

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

#### **TestFinallyBlock.java**

```
class TestFinallyBlock {  
    public static void main(String args[]){  
        try{  
            //below code do not throw any exception  
            int data=25/5;  
            System.out.println(data);  
        }  
        //catch won't be executed  
        catch(NullPointerException e){  
            System.out.println(e);  
        }  
        //executed regardless of exception occurred or not  
        finally {  
            System.out.println("finally block is always executed");  
        }  
  
        System.out.println("rest of the code...");  
    }  
}
```

#### **Output:**



```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java  
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock  
5  
finally block is always executed  
rest of the code...
```

##### *Case 2: When an exception occurs but not handled by the catch block*

Let's see the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

### TestFinallyBlock1.java

```
public class TestFinallyBlock1 {
    public static void main(String args[]){

        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 occurred or not
        finally {
            System.out.println("finally block is always executed");
        }

        System.out.println("rest of the code...");
    }
}
1. }
```

### Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

*Case 3: When an exception occurs and is handled by the catch block*

### Example:

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

### TestFinallyBlock2.java

```

public class TestFinallyBlock2{
    public static void main(String args[]){

        try {

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

            //below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }

        //handles the Arithmetic Exception / Divide by zero exception
        catch(ArithmeticException e){
            System.out.println("Exception handled");
            System.out.println(e);
        }

        //executes regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }

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

```

### Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...

```

## Built-in Exceptions in Java with examples

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

### Examples of Built-in Exception:

1. **Arithmetic exception** : It is thrown when an exceptional condition has occurred in an arithmetic operation.

```
// Java program to demonstrate

// ArithmeticException

class ArithmeticException_Demo {

public static void main(String args[])

{

    try {

        int a = 30, b = 0;

        int c = a / b; // cannot divide by zero

        System.out.println("Result = " + c);

    }

    catch (ArithmeticException e) {

        System.out.println("Can't divide a number by 0");

    }

}

}
```

**Output:**

Can't divide a number by 0

**ArrayIndexOutOfBoundsException 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.

```
// Java program to demonstrate
```

```
// ArrayIndexOutOfBoundsException

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

**ClassNotFoundException :** This Exception is raised when we try to access a class whose definition is not found.

```
// Java program to illustrate the

// concept of ClassNotFoundException

class Bishal {

} class Geeks {
```

```

} class MyClass {

public static void main(String[] args)

{

    Object o = class.forName(args[0]).newInstance();

    System.out.println("Class created for" +
o.getClass().getName());

}

}

```

### **Output:**

ClassNotFoundException

**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

**IOException :** It is thrown when an input-output operation failed or interrupted

```

// Java program to illustrate IOException

import java.io.*;

class Geeks {

public static void main(String args[])

{

    FileInputStream f = null;

    f = new FileInputStream("abc.txt");

    int i;

```

```

while ((i = f.read()) != -1) {

    System.out.print((char)i);

}

f.close();

}

}

```

**Output:**

error: unreported exception IOException; must be caught or declared to be thrown

**InterruptedException** : It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

```

// Java Program to illustrate

// InterruptedException

class Geeks {

public static void main(String args[])

{

    Thread t = new Thread();

    t.sleep(10000);

}

}

```

**Output:**

error: unreported exception InterruptedException; must be caught or declared to be thrown

**NoSuchMethodException** : t is thrown when accessing a method which is not found.

```
// Java Program to illustrate  
  
// NoSuchMethodException  
  
class Geeks {  
  
    public Geeks()  
  
    {  
  
        Class i;  
  
        try {  
  
            i = Class.forName("java.lang.String");  
  
            try {  
  
                Class[] p = new Class[5];  
  
            }  
  
            catch (SecurityException e) {  
  
                e.printStackTrace();  
  
            }  
  
            catch (NoSuchMethodException e) {  
  
                e.printStackTrace();  
  
            }  
  
        }  
  
        catch (ClassNotFoundException e) {  
  
            e.printStackTrace();  
  
        }  
  
    }  
  
}
```

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

{

    new Geeks();

}

}
```

**Output:**

error: exception NoSuchMethodException is never thrown  
in body of corresponding try statement

**NullPointerException** : This exception is raised when referring to the members of a null object. Null represents nothing

// Java program to demonstrate NullPointerException

```
class NullPointerException {

public static void main(String args[])

{

    try {

        String a = null; // null value

        System.out.println(a.charAt(0));

    }

    catch (NullPointerException e) {

        System.out.println("NullPointerException..");

    }

}
```

```
}
```

**Output:**

NullPointerException..

**NumberFormatException** : This exception is raised when a method could not convert a string into a numeric format.

```
// Java program to demonstrate  
  
// NumberFormatException  
  
class NumberFormat_Demo {  
  
    public static void main(String args[])  
  
    {  
  
        try {  
  
            // "akki" is not a number  
  
            int num = Integer.parseInt("akki");  
  
  
            System.out.println(num);  
  
        }  
  
        catch (NumberFormatException e) {  
  
            System.out.println("Number format exception");  
  
        }  
  
    }  
  
}
```

**Output:**

Number format exception

**StringIndexOutOfBoundsException** : It is thrown by String class methods to indicate that an index is either negative than the size of the string.

```
// Java program to demonstrate

// StringIndexOutOfBoundsException

class StringIndexOutOfBounds_Demo {

    public static void main(String args[])

    {

        try {

            String a = "This is like chipping "; // length is 22

            char c = a.charAt(24); // accessing 25th element

            System.out.println(c);

        }

        catch (StringIndexOutOfBoundsException e) {

            System.out.println("StringIndexOutOfBoundsException");

        }

    }

}
```

**Output:**  
StringIndexOutOfBoundsException

## Creating Own Exceptions in Java :

The Java programming language allow us to create our own exception classes which are basically subclasses built-in class **Exception**.

To create our own exception class simply create a class as a subclass of built-in Exception class.

We may create constructor in the user-defined exception class and pass a string to Exception class constructor using **super()**. We can use **getMessage()** method to access the string.

Let's look at the following Java code that illustrates the creation of user-defined exception.

### Example

```
import java.util.Scanner;

class NotEligibleException extends Exception{
    NotEligibleException(String msg){
        super(msg);
    }
}

class VoterList{
    int age;
    VoterList(int age){
        this.age = age;
    }

    void checkEligibility() {
        try {
            if(age < 18) {
                throw new NotEligibleException("Error: Not eligible for vote due to under age.");
            }
            System.out.println("Congrates! You are eligible for vote.");
        }
        catch(NotEligibleException nee) {
            System.out.println(nee.getMessage());
        }
    }

    public static void main(String args[]) {

        Scanner input = new Scanner(System.in);
        System.out.println("Enter your age in years: ");
        int age = input.nextInt();
        VoterList person = new VoterList(age);
    }
}
```

```

        person.checkEligibility();
    }
}

```

When we run this code, it produce the following output.

The screenshot shows the Eclipse IDE with the file `VoterList.java` open. The code defines a `NotEligibleException` and a `VoterList` class. The `main` method prompts the user for their age. The console output shows the user entered '16', and the program threw a `NotEligibleException` with the message "Error: Not eligible for vote due to under age."

## MULTI THREADING

### Process-Based and Thread-Based Multitasking

A **multitasking operating system** is an operating system that gives you the perception of 2 or more tasks/jobs/processes running simultaneously. It does this by dividing system resources amongst these tasks/jobs/processes and switching between the tasks/jobs/processes while they are executing over and over again. Usually, the CPU processes only one task at a time, but the switching is so fast that it looks like the CPU is executing multiple processes simultaneously. They can support either **preemptive** multitasking, where the OS doles out time to applications (virtually all modern OSes), or **cooperative** multitasking, where the OS waits for the program to give back control (Windows 3.x, Mac OS 9 and earlier), leading to hangs and crashes. Also known as **Timesharing**, multitasking is a logical extension of multiprogramming.

**Multitasking Programming is of Two Types:**

#### 1. Process-based Multitasking



## 2. Thread-based Multitasking

S. No.	Process-Based Multitasking	Thread-Based Multitasking
1.	In process-based multitasking, two or more processes and programs can be run concurrently.	In thread-based multitasking, two or more threads can be run concurrently.
2.	In process-based multitasking, a process or a program is the smallest unit.	In thread-based multitasking, a thread is the smallest unit.
3.	The program is a bigger unit.	Thread is a smaller unit.
4.	Process-based multitasking requires more overhead.	Thread-based multitasking requires less overhead.
5.	The process requires its own address space.	Threads share the same address space.
6.	The process to Process communication is expensive.	Thread to Thread communication is not expensive.
7.	Here, it is unable to gain access over the idle time of the CPU.	It allows taking gain access over idle time taken by the CPU.
8.	It is a comparatively heavyweight.	It is comparatively lightweight.
9.	It has a faster data rate multi-tasking.	It has a faster data rate multi-tasking.
10.	<b>Example:</b> We can listen to music and browse the internet at the same time. The processes in this example are the music player and browser.	<b>Example:</b> Using a browser we can navigate through the webpage and at the same time download a file. In this example, navigation is one thread, and downloading is another thread. Also in a word-processing application like MS Word, we can type text in one thread, and spell checker checks for mistakes in another thread.

## Java Thread Model :

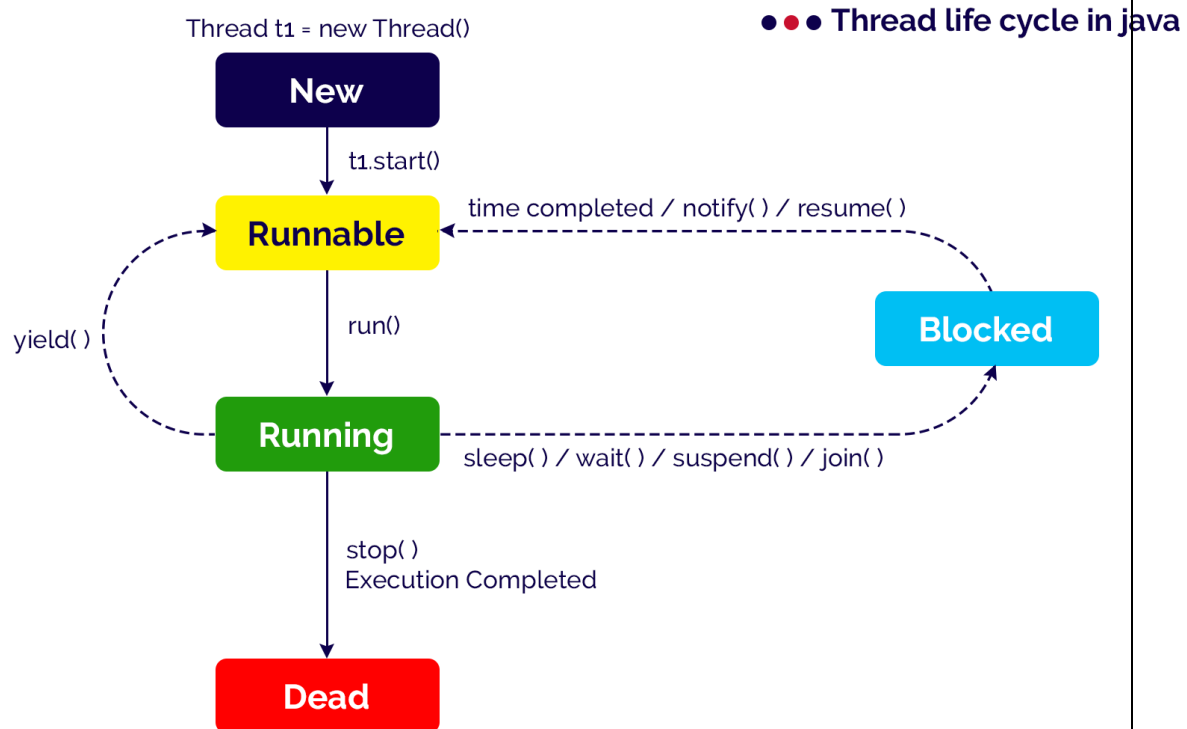
java programming language allows us to create a program that contains one or more parts that can run simultaneously at the same time. This type of program is known as a multithreading program. Each part of this program is called a thread. Every thread defines a separate path of execution in java. A thread is explained in different ways, and a few of them are as specified below.

**A thread is a light weight process.**

A thread may also be defined as follows.

**A thread is a subpart of a process that can run individually.**

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



Let's look at each phase in detail.

### New

When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

### Example

```
Thread t1 = new Thread();
```

### **Runnable / Ready**

When a thread calls `start( )` method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

#### **Example**

```
t1.start( );
```

### **Running**

When a thread calls `run( )` method, then the thread is said to be Running. The `run( )` method of a thread called automatically by the `start( )` method.

### **Blocked / Waiting**

A thread in the Running state may move into the blocked state due to various reasons like `sleep( )` method called, `wait( )` method called, `suspend( )` method called, and `join( )` method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, `notify( )` or `notifyAll( )` method called, `resume( )` method called, etc.

#### **Example**

```
Thread.sleep(1000);  
wait(1000);  
wait();  
suspend();  
notify();  
notifyAll();  
resume();
```

### **Dead / Terminated**

A thread in the Running state may move into the dead state due to either its execution completed or the `stop( )` method called. The dead state is also known as the terminated state.

## Java Program to Create a Thread :

Thread can be referred to as a lightweight process. Thread uses fewer resources to create and exist in the process; thread shares process resources. The main thread of Java is the thread that is started when the program starts. The slave thread is created as a result of the main thread. This is the last thread to complete execution.

A thread can programmatically be created by:

1. Implementing the java.lang.Runnable interface.
2. Extending the java.lang.Thread class.

You can create threads by implementing the runnable interface and overriding the run() method. Then, you can create a thread object and call the start() method.

### Thread Class:

The Thread class provides constructors and methods for creating and operating on threads. The thread extends the Object and implements the Runnable interface.

```
// start a newly created thread.
```

```
// Thread moves from new state to runnable state
```

```
// When it gets a chance, executes the target run() method
```

```
public void start()
```

### Runnable interface:

Any class with instances that are intended to be executed by a thread should implement the Runnable interface. The Runnable interface has only one method, which is called run().

```
// Thread action is performed
```

```
public void run()
```

### Benefits of creating threads :

- When compared to processes, Java Threads are more lightweight; it takes less time and resources to create a thread.
- Threads share the data and code of their parent process.
- Thread communication is simpler than process communication.
- Context switching between threads is usually cheaper than switching between processes.

### Calling run() instead of start()

The common mistake is starting a thread using run() instead of start() method.

```
Thread myThread = new Thread(MyRunnable());
```

```
myThread.run(); //should be start();
```

The run() method is not called by the thread you created. Instead, it is called by the thread that created the **myThread**.

### Example 1: By using Thread Class

- Java

```

import java.io.*;

class GFG extends Thread {

    public void run()

    {

        System.out.print("Welcome to GeeksforGeeks.");

    }

    public static void main(String[] args)

    {

        GFG g = new GFG(); // creating thread

        g.start(); // starting thread

    }

}

```

### **Output**

Welcome to GeeksforGeeks.

### **Example 2: By implementing Runnable interface**

- Java

```

import java.io.*;

class GFG implements Runnable {

    public static void main(String args[])

    {

        // create an object of Runnable target

        GFG gfg = new GFG();
    }
}

```

```

// pass the runnable reference to Thread

Thread t = new Thread(gfg, "gfg");


// start the thread

t.start();


// get the name of the thread

System.out.println(t.getName());

}

@Override public void run()

{

    System.out.println("Inside run method");

}

}

```

### **Output**

gfg

Inside run method

## **Priority of a Thread (Thread Priority)**

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

## Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

**public final int getPriority():** The `java.lang.Thread.getPriority()` method returns the priority of the given thread.

**public final void setPriority(int newPriority):** The `java.lang.Thread.setPriority()` method updates or assigns the priority of the thread to `newPriority`. The method throws `IllegalArgumentException` if the value `newPriority` goes out of the range, which is 1 (minimum) to 10 (maximum).

38M

730

HTML Tutorial

3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (`NORM_PRIORITY`). The value of `MIN_PRIORITY` is 1 and the value of `MAX_PRIORITY` is 10.

### Example of priority of a Thread:

**FileName:** `ThreadPriorityExample.java`

```
// Importing the required classes
```

```
import java.lang.*;
```

```
public class ThreadPriorityExample extends Thread  
{
```

```
// Method 1
```

```
// Whenever the start() method is called by a thread
```

```
// the run() method is invoked
```

```
public void run()
```

```
{
```

```
// the print statement
```

```
System.out.println("Inside the run() method");
```

```
}
```

```
// the main method
```

```
public static void main(String argsv[])
```

```
{
```

```
// Creating threads with the help of ThreadPriorityExample class
```

```

ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

// 1st Thread
// Displaying the priority of the thread
// using the getPriority() method
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread
// Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 3rd Thread
// // Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// Setting priorities of above threads by
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);

// 6
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 3
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 9
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

// Main thread

// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName()
);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// Priority of the main thread is 10 now

```



```
Thread.currentThread().setPriority(10);
```

```
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());  
}  
}
```

### Output:

```
Priority of the thread th1 is : 5  
Priority of the thread th2 is : 5  
Priority of the thread th2 is : 5  
Priority of the thread th1 is : 6  
Priority of the thread th2 is : 3  
Priority of the thread th3 is : 9  
Currently Executing The Thread : main  
Priority of the main thread is : 5  
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

**FileName:** ThreadPriorityExample1.java

```
// importing the java.lang package  
import java.lang.*;  
  
public class ThreadPriorityExample1 extends Thread  
{  
  
    // Method 1  
    // Whenever the start() method is called by a thread  
    // the run() method is invoked  
    public void run()  
    {  
        // the print statement  
        System.out.println("Inside the run() method");  
    }  
  
    // the main method  
    public static void main(String argsv[])  
    {  
  
        // Now, priority of the main thread is set to 7  
        Department of CSE
```

```

Thread.currentThread().setPriority(7);

// the current thread is retrieved
// using the currentThread() method

// displaying the main thread priority
// using the getPriority() method of the Thread class
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// creating a thread by creating an object of the class ThreadPriorityExample1
ThreadPriorityExample1 th1 = new ThreadPriorityExample1();

// th1 thread is the child of the main thread
// therefore, the th1 thread also gets the priority 7

// Displaying the priority of the current thread
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
}
}

```

### Output:

```

Priority of the main thread is : 7
Priority of the thread th1 is : 7

```

**Explanation:** If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

### *Example of IllegalArgumentException*

We know that if the value of the parameter *newPriority* of the method `getPriority()` goes out of the range (1 to 10), then we get the `IllegalArgumentException`. Let's observe the same with the help of an example.

**FileName:** `IllegalArgumentException.java`

```

// importing the java.lang package
import java.lang.*;

public class IllegalArgumentException extends Thread
{

// the main method
public static void main(String argsv[])
{

```

```
// Now, priority of the main thread is set to 17, which is greater than 10
Thread.currentThread().setPriority(17);

// The current thread is retrieved
// using the currentThread() method

// displaying the main thread priority
// using the getPriority() method of the Thread class
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}
}
```

When we execute the above program, we get the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentException
    at java.base/java.lang.Thread.setPriority(Thread.java:1141)
    at IllegalArgumentException.main(IllegalArgumentException.java:12)
```

## Synchronization in Java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point in time. Java provides a way of creating threads and synchronizing their tasks using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

```
// Only one thread can execute at a time.
// sync_object is a reference to an object
// whose lock associates with the monitor.
// The code is said to be synchronized on
// the monitor object
synchronized(sync_object)
{
    // Access shared variables and other
    // shared resources
}
```

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi-threading with synchronized.

- Java

```
// A Java program to demonstrate working of
```

```
// synchronized.
```

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

```
import java.util.*;
```

```
// A Class used to send a message
```

```
class Sender
```

```
{
```

```
    public void send(String msg)
```

```
    {
```

```
        System.out.println("Sending\t" + msg );
```

```
        try
```

```
        {
```

```
            Thread.sleep(1000);
```

```
        }
```

```
        catch (Exception e)
```

```
        {
```

```
            System.out.println("Thread interrupted.");
```

```
    }

    System.out.println("\n" + msg + "Sent");

}

}
```

// Class for send a message using Threads

class ThreadedSend extends Thread

```
{

    private String msg;

    Sender sender;

    // Receives a message object and a string
    // message to be sent

    ThreadedSend(String m, Sender obj)

    {

        msg = m;

        sender = obj;

    }

    public void run()

    {

        // Only one thread can send a message

        // at a time.
```

```

        synchronized(sender)

        {

            // synchronizing the send object

            sender.send(msg);

        }

    }

}

// Driver class

class SyncDemo

{

    public static void main(String args[])

    {

        Sender send = new Sender();

        ThreadedSend S1 =

            new ThreadedSend( " Hi " , send );

        ThreadedSend S2 =

            new ThreadedSend( " Bye " , send );

        // Start two threads of ThreadedSend type

        S1.start();

        S2.start();

```

```

        // wait for threads to end

        try

        {

            S1.join();

            S2.join();

        }

        catch(Exception e)

        {

            System.out.println("Interrupted");

        }

    }

}

```

### Output

Sending    Hi

Hi Sent

Sending    Bye

Bye Sent

The output is the same every time we run the program.

In the above example, we choose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized**, producing the same result. Then we don't have to synchronize the Message object inside the run() method in ThreadedSend class.

```

// An alternate implementation to demonstrate
// that we can use synchronized with method also.

```

```

class Sender {
    public synchronized void send(String msg)
    {
        System.out.println("Sending\t" + msg);
        try {
            Thread.sleep(1000);
        }
        catch (Exception e) {
            System.out.println("Thread interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}

```

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods make this possible.

```

// One more alternate implementation to demonstrate
// that synchronized can be used with only a part of
// method

```

```

class Sender
{
    public void send(String msg)
    {
        synchronized(this)
        {
            System.out.println("Sending\t" + msg );
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {
                System.out.println("Thread interrupted.");
            }
        }
    }
}

```



```

        }
        System.out.println("\n" + msg + "Sent");
    }
}
}

```

## Inter-thread Communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

### *1) wait() method*

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

47.4M  
766

### Hello Java Program for Beginners

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

### *2) notify() method*

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

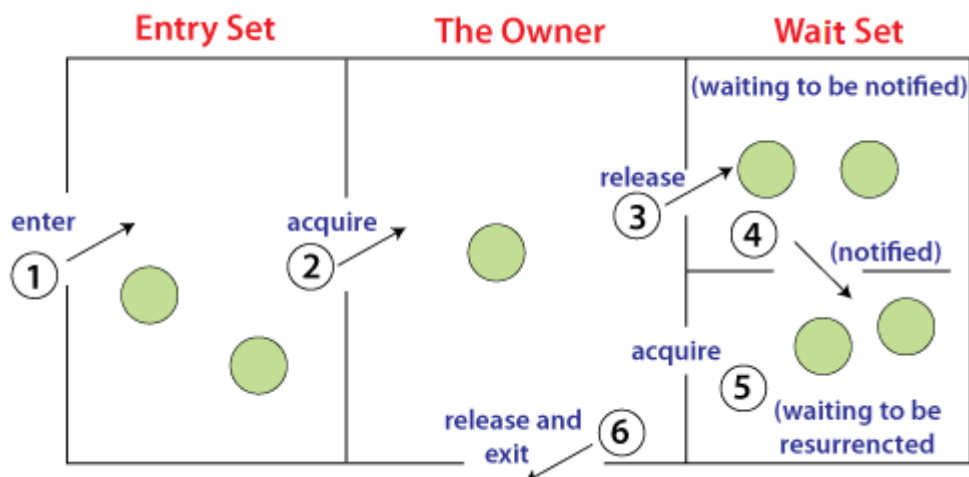
**Syntax:**

**public final void** notify()  
*3) notifyAll() method*

Wakes up all threads that are waiting on this object's monitor.

**Syntax:**

**public final void** notifyAll()  
Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

*Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?*

It is because they are related to lock and object has a lock.

*Difference between wait and sleep?*

Let's see the important differences between wait and sleep methods.

<b>wait()</b>	<b>sleep()</b>
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

*Example of Inter Thread Communication in Java*

Let's see the simple example of inter thread communication.

**Test.java**

```

class Customer{
    int amount=10000;

    synchronized void withdraw(int amount){
        System.out.println("going to withdraw...");

        if(this.amount<amount){
            System.out.println("Less balance; waiting for deposit...");
            try{ wait();}catch(Exception e){ }
        }
        this.amount-=amount;
        System.out.println("withdraw completed...");
    }

    synchronized void deposit(int amount){
        System.out.println("going to deposit...");
        this.amount+=amount;
        System.out.println("deposit completed... ");
        notify();
    }
}

class Test{
    public static void main(String args[]){
        final Customer c=new Customer();
        new Thread(){
            public void run(){c.withdraw(15000);}
        }.start();
        new Thread(){

```

```
public void run(){c.deposit(10000);}
}.start();
```

```
}}
```

**Output:**

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```