**Unit-2 java**

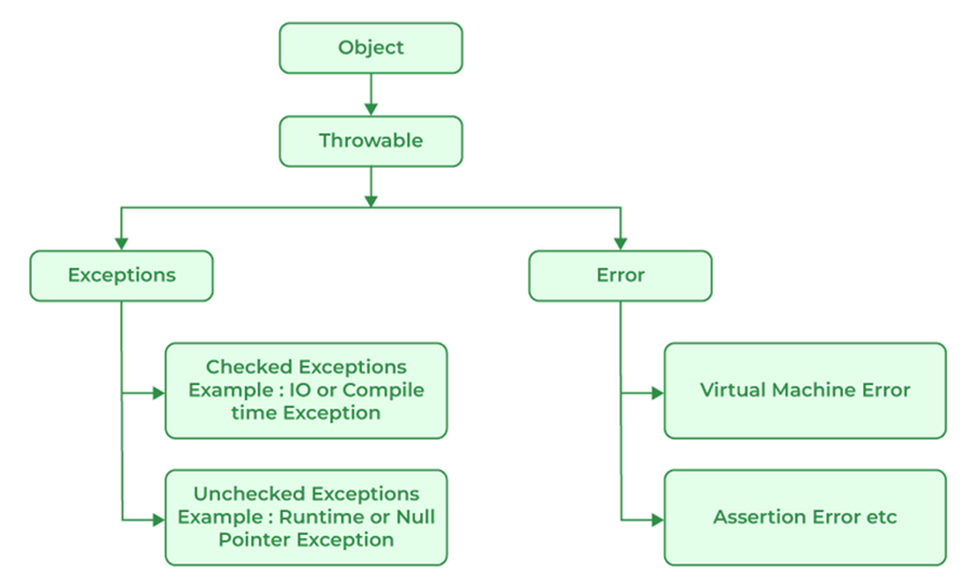
**Exception:** An exception is an unwanted or unexpected event that disrupts the normal flow of a program's execution. It can occur for various reasons, such as invalid input, division by zero, or trying to access a resource that is not available. Exceptions are objects that encapsulate information about the error, including its type and the state of the program when the error occurred.

In Java, exceptions are used to handle errors and exceptional conditions that occur during the execution of a program. The idea behind exceptions is to provide a mechanism for dealing with runtime errors in a structured and controlled way, rather than letting them propagate uncontrollably and potentially crashing the program.

**Errors:** In Java, errors are similar to exceptions but are used to indicate serious problems that are typically beyond the control of the programmer. Examples include out-of-memory errors or errors related to the Java Virtual Machine (JVM). Errors are usually not caught or handled by application code, as they typically indicate a catastrophic failure from which the program cannot recover.

**Exception Hierarchy :**

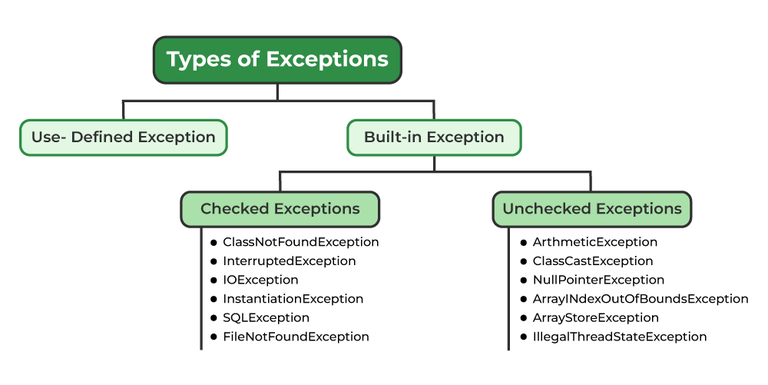
All exception and error types are subclasses of the class Throwable, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



**Types of Exceptions:** In Java, exceptions are categorized into two main types.

**1.Checked Exceptions:** Checked exceptions are exceptions that must be either caught or declared in the method signature using the throws clause. Examples include IOException, SQLException, etc. These exceptions are typically recoverable and are often related to external factors such as file I/O or network operations.

**2.Unchecked Exceptions (Runtime Exceptions):** These are the exceptions that are not checked at compile. and exceptions that are not required to be caught or declared. They usually indicate programming errors or logical issues within the code. Examples include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.



**Handling Exceptions:**

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved.

Java Exception Handling is a mechanism to handle checked and unchecked Exception such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

**try:**

The **try** block is used to enclose the code that might throw an exception. It's followed by one or more **catch** blocks or a finally block.

**catch:**

The **catch** block is used to handle the exception thrown within the corresponding try block. It catches specific types of exceptions or a superclass of exceptions.

You can have multiple **catch** blocks to handle different types of exceptions.

**finally:**

The finally block is used to execute the code that needs to be executed whether an exception is thrown or not. It's typically used for cleanup tasks like closing resources (e.g., file streams, database connections).

**Structure:**

try {

// code that might throw an exception

}

catch (ExceptionType1 e1) {

// handle exception of type ExceptionType1

}

catch (ExceptionType2 e2) {

// handle exception of type ExceptionType2

}

finally {

// code that will always be executed

}

**Example :**

public class ExceptionHandling {

public static void main(String[] args) {

**try** {

int numerator = 10;

int denominator = 0;

int result = numerator / denominator;

// This line will not be executed if exception occurs

System.out.println("Result: " + result);

}

**catch (ArithmeticException e)** {

// Handling the ArithmeticException

System.out.println("Error: Division by zero");

}

**finally** {

// Cleanup tasks

System.out.println("Cleanup tasks executed.");

}

}

}

**Example-2:**

static void trys(){

        try{

            Scanner sc=new Scanner(System.in);

            System.out.println("enter number for divide");

            int d=sc.nextInt();

            int num=10;

            int r=num/d;

            int arr[]={1,2,4,4};

            System.out.println("enter a size");

            int size=sc.nextInt();

            arr[size]=5;

            String str=null;

            int length=str.length();

          }

          catch(ArithmeticException e){

           e.getMessage();

          }

          catch(ArrayIndexOutOfBoundsException e){

              e.getStackTrace();

          }

          catch(Exception e){

              System.out.println(e);

          }

          finally{

              System.out.println("it is finally block ");

           }

    }

**throw keyword :**

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

**Example:**

import java.util.Scanner;

public class imports {

            public static void main(String[] args) {

                try {

                    int[] arr = {1, 2, 3};

                    Scanner sc=new Scanner(System.in);

                    System.out.println("enter size of array");

                    int size=sc.nextInt();

                    if (size>3) {

                        throw new ArrayIndexOutOfBoundsException("Index out of bounds");

                    }

                    arr[size]=25;

                }

        catch (ArrayIndexOutOfBoundsException e) {

                   System.out.println("ArrayIndexOutOfBoundsException: " + e.getMessage());

                }

            }

        }

//output

enter size of array

4

ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 3

Java throws:

throws is a keyword in Java that is used in the signature of a method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Example:

import java.util.Scanner;

public class throws1 {

    static void f1() throws ArithmeticException{

       int arr[]={1,2,3};

       Scanner sc=new Scanner(System.in);

       System.out.println("enter size");

       int size=sc.nextInt();

       arr[size]=5;

}

    public static void main(String[] args) {

        f1();

}

}

**//output**

**Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 3**

**at throws1.f1(throws1.java:8)**

**at throws1.main(throws1.java:11)**

**In-built Exceptions:**

In-built exceptions are provided by the Java language and are part of the standard Java API. These exceptions cover a wide range of common error conditions that might occur during program execution.

Examples of in-built exceptions include NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, and many more.

These exceptions are already defined in the Java libraries and are used to handle various error situations in Java programs. You can directly use these exceptions in your code by catching or throwing them as needed.

**Example:**

public class Main{

public static void main(String[] args) {

try {

int result = 10 / 0;

}

catch (ArithmeticException e) {

System.out.println("Error: " + e.getMessage());

}}

}

**// output**

**Error: / by zero**

**User-Defined Exceptions:**

User-defined exceptions are created by the programmer to represent custom error conditions specific to their application or domain.

These exceptions extend either the Exception class or one of its subclasses.

By creating custom exceptions, programmers can define their own exception types with specific error messages .

User-defined exceptions are useful for representing business logic errors, validation failures, or other exceptional conditions that are specific to your application domain**.**

**Example-1:**

class CustomException extends Exception {

    CustomException(String message) {

        super(message);

        System.out.println("1# " + message);

    }

}

public class inputstream {

    // Method that may throw the custom exception

    static void validateInput(int value) throws CustomException {

        if (value < 0) {

            throw new CustomException("2 Input cannot be negative");

        }

        // If input is valid, proceed with processing

        System.out.println("Input is valid: " + value);

    }

    public static void main(String[] args) {

        try {

            validateInput(-5);

        } catch (CustomException e) {

            System.out.println("CustomException caught: " + e.getMessage());

        }

    }

}

//output

1# 2 Input cannot be negative

CustomException caught: 2 Input cannot be negative

**Example-2:**

import java.util.Scanner;

class costom extends Exception{

     costom(String str){

          super(str);

     }

}

public class try\_catch {

    static void custom(){

      try {

           int i=2;

           Scanner sc=new Scanner(System.in);

           System.out.println("enter j");

           int j=sc.nextInt();

           if (j==0) {

            throw new costom("it is custopmexception got");

           }

           int r=i/j;

           System.out.println("result is "+r);

         }

         catch (costom e) {

            System.out.println(e);

         }

 }

public static void main(String[] args) throws Exception{

        //trys();

        custom();

    }

}

**//output:**

**enter j**

**0**

**costom: it is custopmexception got**

**example-3**

import java.util.Scanner;

class user\_exception extends Exception{

   user\_exception(String str){

          super(str);

     }

}

public class throws1 {

    static void f1(){

        try{

       int arr[]={1,2,3};

       Scanner sc=new Scanner(System.in);

       System.out.println("enter size");

       int size=sc.nextInt();

       if (size>3) {

        throw new user\_exception("out of bounds exception ");

       }

       arr[size]=5;

        }

        catch(user\_exception e){

          System.out.println(e.getMessage());

        }

}

    public static void main(String[] args) {

        f1();

}

}

**//output**

enter size

4

out of bounds exception

**input/output (I/O) operations :**

In Java, you can perform input/output (I/O) operations using byte streams and character streams. Byte streams are used to handle raw binary data, while character streams are used to handle text data. Additionally, you can read from and write to files using Java's I/O classes.

**Byte Streams and Character Streams:**

Byte Streams and Character Streams are two different types of streams used for input and output operations in Java. They handle data differently, especially regarding character encoding and decoding.

| **Feature** | **Byte Streams** | **Character Streams** |
| --- | --- | --- |
| Data Handling | Handle raw binary data byte by byte. | Handle character data. |
| Encoding/Decoding | Do not perform character encoding/decoding. | Perform character encoding/decoding. |
| Data Type | Dealing with bytes. | Dealing with characters. |
| Suitable For | Binary data such as images, audio, or non-text files. | Text-based data such as text files, CSV, or XML files. |
| Example Classes | **InputStream**, **OutputStream**, **FileInputStream**, **FileOutputStream**. | **Reader**, **Writer**, **FileReader**, **FileWriter**. |
| Usage | Low-level I/O operations. | Reading/writing text-based data. |
| Handling Encoding | Does not handle character encoding. | Automatically handle character encoding/decoding. |
| Reading/Writing | Reads/writes bytes directly. | Reads/writes characters with automatic encoding/decoding. |
| Complexity | Simpler to use as they deal with raw bytes. | Slightly more complex due to character encoding/decoding. |
| Efficiency | Efficient for binary data operations. | Efficient for text-based data operations. |
| Examples | Reading/writing image files, binary files. | Reading/writing text files, CSV files. |

Byte Streams:

They are suitable for reading and writing data that does not require any character encoding or decoding, such as image files or binary data.

Byte Streams deal directly with bytes and do not perform any character encoding or decoding.

Byte Streams are used for low-level I/O operations where binary data needs to be transferred accurately.

**1.Reading from Byte Streams:**

**=>**Create an instance of the InputStream subclass, such as FileInputStream, specifying the file path or other data source.

**=>**Use the read() method to read bytes from the stream. This method typically returns the next byte of data as an integer value (0 to 255) or -1 if the end of the stream is reached.

import java.io.FileInputStream;

import java.io.IOException;

class inputstream {

    public static void main(String[] args) {

        try {

            FileInputStream inputStream = new FileInputStream("input.txt");

            int data;

            while (true) {

                data = inputStream.read();

                if (data == -1) {

                    break;

                }

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

            }

            inputStream.close();

        } catch (IOException e) {

            System.out.println( e.getMessage());

        }

    }

}

**2.Writing to Byte Streams:**

=>Create an instance of the OutputStream subclass, such as FileOutputStream, specifying the file path or destination.

=> Use the write() method to write bytes to the stream. This method accepts a byte array or a single byte as input.

byte[] data = {65, 66, 67};

outputStream.write(data);

import java.io.\*;

class outputstream {

    public static void main(String[] args) {

        try {

            FileOutputStream ob = new FileOutputStream("output.txt");

            String content="hello world";

            byte content1[]=content.getBytes();

            ob.write(content1);

        } catch (IOException e) {

            System.out.println( e.getMessage());

        }

    }

}

Character Streams:

They are used for reading and writing text-based data, such as plain text files.

Character Streams automatically handle character encoding and decoding, converting characters to bytes when writing and bytes to characters when reading.

Character Streams are more convenient for dealing with text-based data as they handle encoding issues transparently.

**1.Reading from Character Streams:**

=>Create an instance of the Reader subclass, such as FileReader, specifying the file path or other data source.

=>Use the read() method to read characters from the stream. This method typically returns the next character of data as an integer value (0 to 65535) or -1 if the end of the stream is reached.

Example:

import java.io.\*;

class inputstream {

    public static void main(String[] args) {

        try {

     FileReader reader = new FileReader("input.txt");

            int data;

            while (true) {

                 data = reader.read();

                  if (data==-1){

                      break;

                  }

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

            }

            reader.close();

        } catch (IOException e) {

            System.out.println( e.getMessage());

        }

    }

}

**2.Writing to Character Streams:**

=>Create an instance of the Writer subclass, such as FileWriter, specifying the file path or destination.

=>Use the write() method to write characters to the stream. This method accepts a string, a character array, or a single character as input.

Example:

import java.io.\*;

class outputstream {

    public static void main(String[] args) {

        try {

            FileWriter writer = new FileWriter("output.txt");

            String data = "Hello, World!";

                   writer.write(data);

                   writer.close();

        } catch (IOException e) {

            System.out.println( e.getMessage());

        }

    }

}

**Multithreading :**

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

**Thread:**

In Java, the Thread class represents a thread of execution. You can create a thread by extending the Thread class or implementing the Runnable interface. The Thread class provides methods to perform various operations on threads like starting, stopping, suspending, and resuming.

Example:

class xyz1 extends Thread{

     @Override

     public void run() {

        for (int i = 0; i <=50; i++) {

            System.out.println("ajay");

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

                System.out.println(e);

            }

        }

    }

}

class xyz2 extends Thread{

     @Override

     public void run(){

        for (int i = 0; i <=50; i++) {

            System.out.println("vijay");

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

                System.out.println(e);

            }

        }

    }

}

public class threads\_12 {

    public static void main(String[] args) {

        xyz1 ob1=new xyz1();

        xyz2 ob2=new xyz2();

        ob1.start();

        ob2.start();

    }

}

//output

vijay

ajay

vijay

ajay

vijay

ajay

vijay

ajay

vijay

.

.

**Threads can be created by using two mechanisms**

1.Extending the Thread class

2.Implementing the Runnable Interface

**1.Thread creation by extending the Thread class**

We create a class that extends the java.lang.Thread class or Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

**Example-1**

// Extending Thread class

class MyThread extends Thread {

public void run() {

System.out.println("Thread "+Thread.currentThread().getId()+" is running");

}

}

public class Main {

public static void main(String[] args) {

for (int i = 0; i <=3; i++) {

MyThread thread1 = new MyThread();

thread1.start();

System.out.println(i);}

}

}

**// output**

0

1

2

3

Thread 10 is running

Thread 13 is running

Thread 12 is running

Thread 11 is running

**Example-2**

class Innerthreads1 extends Thread{

    @Override

  public void run(){

            for (int i = 0; i <=5; i++) {

               System.out.print(i+" ");

            }

  }

}

public class threads1 {

    public static void main(String[] args) {

        System.out.println("start");

        System.out.println(Thread.currentThread().getName());

        try {

            Thread.sleep(5000);

        } catch (Exception e) {

            System.out.println("something wrong");

        }

        System.out.println("end");

       Innerthreads1 ob=new Innerthreads1();

        ob.start();

    }

}

**2.Thread creation by implementing the Runnable Interface**

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

**Example-1:**

class MyRunnable implements Runnable {

    public void run() {

    System.out.println("Thread "+Thread.currentThread().getId()+" is running");

    }

}

public class threads {

    public static void main(String[] args) {

        for (int i = 0; i <=3; i++) {

            MyRunnable t=new MyRunnable();

            Thread thread = new Thread(t);

            thread.start();

            System.out.println(i);

            }}

}

//Output

0

1

2

3

Thread 23 is running

Thread 21 is running

Thread 22 is running

Thread 20 is running

Example-2

class MyThread implements Runnable {

    @Override

    public void run() {

        // task for thread...

        for (int i = 1; i <= 10; i++) {

            System.out.println("value of i is " + i);

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

            }

        }

    }}

class threads {

    public static void main(String[] args) {

        // create object of MyThread class

        MyThread t1 = new MyThread();

        Thread thr = new Thread(t1);

        thr.start();

    }

}

**Example-3:**

class xyz1 implements Runnable{

     @Override

     public void run() {

        for (int i = 0; i <=50; i++) {

            System.out.println("ajay");

        }

    }

}

class xyz2 implements Runnable{

     @Override

     public void run(){

        for (int i = 0; i <=50; i++) {

            System.out.println("vijay");

        }

    }

}

public class threads\_12 {

    public static void main(String[] args) {

        Runnable ob1=new xyz1();

        Runnable ob2=new xyz2();

        Thread T1=new Thread(ob1);

        Thread T2=new Thread(ob2);

        T1.start();

        T2.start();

  }

}

**Thread Life Cycle:**

**1.New:** A thread is in this state when it's created but has not yet started.

**2.Runnable:** The thread is ready to run and is waiting for the scheduler to pick it.

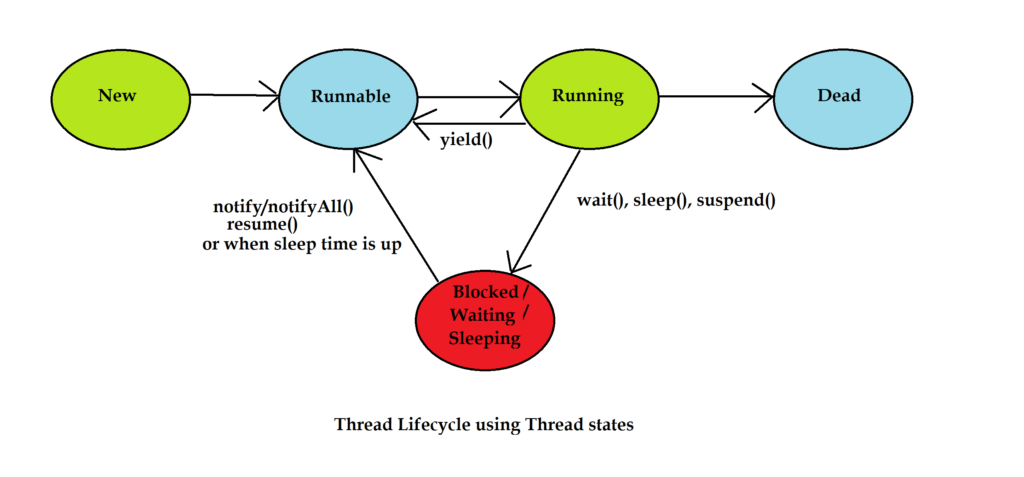
**3.Running:** The thread is executing.

**4.Blocked:** The thread is waiting for a monitor lock to enter a synchronized block/method or waiting indefinitely for another thread to release a lock.

**5.Waiting:** The thread is waiting indefinitely for another thread to perform a particular action.

**6.Timed Waiting:** The thread is waiting for another thread to perform a particular action, but with a specified waiting time.

**7.Terminated:** The thread has completed its execution.



**thread priorities:**

In Java, threads can have priorities ranging from Thread.MIN\_PRIORITY (1) to Thread.MAX\_PRIORITY (10), with Thread.NORM\_PRIORITY (5) being the default priority. Thread priorities are used by the thread scheduler to determine the order in which threads are executed. However, the actual behavior might vary depending on the JVM implementation and the underlying operating system.

**Rules for thread priorties:**

1.Higher priority threads are given preference over lower priority threads.

2.If multiple threads of the same priority are competing for CPU time, the scheduler typically uses a round-robin algorithm.

3.Setting thread priorities should be done with caution, as it might affect the overall performance and responsiveness of the application.

4.Thread priorities are platform-dependent, and their behavior might vary across different operating systems.

**Example:**

class MyThread extends Thread {

    public void run() {

        System.out.println("Thread "+Thread.currentThread().getId()+ "running");

    }

}

public class threads {

    public static void main(String[] args) {

        int numThreads = 3;

        for (int i = 0; i <=numThreads; i++) {

            Thread thread = new MyThread();

           thread.setPriority(Thread.MAX\_PRIORITY); // Set thread priority

           int priority = thread.getPriority(); // Get thread priority

           System.out.println("Thread " + i + " priority: " + priority);

           thread.start();

        }

    }

}

**// output**

Thread 0 priority: 10

Thread 1 priority: 10

Thread 2 priority: 10

Thread 3 priority: 10

Thread 11running

Thread 13running

Thread 10running

Thread 12running

**Example-3:**

public class threads\_12 {

    public static void main(String[] args) {

        System.out.println("start");

        System.out.println(Thread.currentThread().getName());//main

        System.out.println(Thread.MIN\_PRIORITY);//1

        System.out.println(Thread.MAX\_PRIORITY);//10

        System.out.println(Thread.NORM\_PRIORITY);//5

        for (int i = 0; i <=10; i++) {

            System.out.println("h1");

            try {

                Thread.sleep(2000);

            } catch (InterruptedException e) {

                System.out.println(e.getMessage());

            }

            System.out.println("h2");

        }

        System.out.println("end");

  }

}

//output

start

main

1

10

5

h1

h2

h1

h2

end

**Example-3**

//creating our thread using Runnable Interface

class MyThread implements Runnable {

    public void run() {

        // task for thread...

        for (int i = 1; i <= 10; i++) {

            System.out.println("value of i is " + i);

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

            }

        }

    }

}

// Creating thread using Thread class

class MyAnotherThread extends Thread {

    public void run() {

        // task for thread

        for (int i = 10; i >= 1; i--) {

            System.out.println("another thread = " + i);

            try {

                Thread.sleep(2000);

            } catch (Exception e) {

            }

        }

    }

}

class threads {

    public static void main(String[] args) {

        // create object of MyThread class

        MyThread t1 = new MyThread();

        Thread thr = new Thread(t1);

        // object of AnotherThread

        MyAnotherThread t2 = new MyAnotherThread();

        thr.start();

        t2.start();

    }

}

**Thread synchronization :**

Thread synchronization mechanism can be achieved using Lock framework, which is present in java.util.concurrent package. Lock framework works like synchronized blocks except locks can be more sophisticated than Java’s synchronized blocks. Locks allow more flexible structuring of synchronized code. This new approach was introduced in Java 5 to tackle some problem of synchronization.

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Counter {

private int count;

private Lock lock = new ReentrantLock();

public void increment() {

lock.lock();

try {

count++;

} finally {

lock.unlock();

}

}

public int getCount() {

lock.lock();

try {

return count;

} finally {

lock.unlock();

}

}

}

public class Main {

public static void main(String[] args) {

Counter counter = new Counter();

// Create multiple threads that increment the counter

for (int i = 0; i < 5; i++) {

Thread thread = new Thread(() -> {

for (int j = 0; j < 1000; j++) {

counter.increment();

}

});

thread.start();

}

// Wait for all threads to finish

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

// Print the final count

System.out.println("Final Count: " + counter.getCount());

}

}

**// output**

**Final Count: 5000**

**Inter-thread communication:**

Inter-thread communication is essential for coordinating actions between threads and sharing data safely in a multi-threaded environment. Java provides several mechanisms

for inter-thread communication, including wait(), notify(), and notifyAll() methods, as well as higher-level constructs like BlockingQueue, Semaphore, and CountDownLatch.

**Example-1:**

class SharedResource {

private boolean available = false;

public synchronized void produce() {

// Wait until the resource is consumed

while (available) {

try {

wait(); // Release the lock and wait

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

// Produce the resource

available = true;

System.out.println("Resource produced");

// Notify waiting threads that the resource is available

notify();

}

public synchronized void consume() {

// Wait until the resource is produced

while (!available) {

try {

wait(); // Release the lock and wait

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

// Consume the resource

available = false;

System.out.println("Resource consumed");

// Notify waiting threads that the resource has been consumed

notify();

}

}

public class Main {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

Thread producer = new Thread(() -> {

for (int i = 0; i < 5; i++) {

resource.produce();

}

});

Thread consumer = new Thread(() -> {

for (int i = 0; i < 5; i++) {

resource.consume();

}

});

producer.start();

consumer.start();

}

}

**//output**

Resource produced

Resource consumed

Resource produced

Resource consumed

Resource produced

Resource consumed

Resource produced

Resource consumed

Resource produced

Resource consumed

**Example-2:**

class Company {

*int* n;

*boolean* available = false;

    // f=false: chance: producer

    // f=true: chance :consumer

    synchronized public *void* produce\_item(*int* *n*) throws Exception {

        if (available) {

            wait();

        }

        this.n = *n*;

        System.out.println("Produced : " + this.n);

        available = true;

        notify();

    }

    synchronized public *int* consume\_item() throws Exception {

        if (!available) {

            wait();

        }

        System.out.println("Consumed : " + this.n);

        available = false;

        notify();

        return this.n;

    }

}

class Producer extends Thread {

    Company c;

    Producer(Company *c*) {

        this.c = *c*;

    }

    @*Override*

    public *void* run() {

*int* i = 1;

        while (true) {

            try {

                this.c.produce\_item(i);

            } catch (Exception e) {

                e.printStackTrace();

            }

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

            }

            i++;

        }

    }

}

class Consumer extends Thread {

    Company c;

    Consumer(Company *c*) {

        this.c = *c*;

    }

@*Override*

    public *void* run() {

        while (true) {

            try {

                this.c.consume\_item();

            } catch (Exception e) {

                e.printStackTrace();

            }

            try {

                Thread.sleep(1000);

            } catch (Exception e) {

            }

        }

    }

}

public class threads12 {

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

        Company comp = new Company();

        Producer p = new Producer(comp);

        Consumer c = new Consumer(comp);

        p.start();

        c.start();

    }

}

//output

Produced : 1

Consumed : 1

Produced : 2

Consumed : 2

Produced : 3

Consumed : 3

Produced : 4

Consumed : 4

Produced : 5

Consumed : 5

.

.

.