UNIT III - EXCEPTION HANDLING AND MULTITHREADING

Exception Handling basics - Multiple catch clauses - Nested try Statements - Java's Built-in Exceptions - User defined Exception. Multithreaded Programming : Java Thread Model - Creating a Thread and Multiple Threads - Priorities - Synchronization - Inter Thread Communication - Suspending - Resuming and Stopping Threads - Multithreading Wrappers - Auto boxing.

1. EXCEPTION HANDLING BASICS :

Exception Handling is a mechanism in which the statements that are likely to cause an exception are enclosed within 'try' block. As soon as exception occurs it is handled using 'catch' block.

Thus exception handling is a mechanism ~~which~~ that prevents the program from crashing when some unusual situation occurs.

Types of Exception :

1. Checked Exception - Need to be handled explicitly by the code itself either by using 'try-catch' block or by using 'throws'.

2. Unchecked Exception - Need not be handled explicitly. The JVM handles it.

Keywords used in Exception Handling:

1. try
2. catch
3. finally
4. throw
5. throws

try - catch block:

→ The statements that are likely to cause an exception are enclosed within a try block.

→ The catch block is responsible for handling the exception thrown by the try block.

Syntax:

```
try
{
    //code
}
catch(exception)
{
    //code
}
```

finally block:

It specifies the code that must be executed even though exception may or may not occur.

Syntax:

```
finally
{
    // clean up code
}
```

Example:

```
Public class Example
{
    Public static void main (String[] args)
    {
        try
        {
            int a = 5/0;
        }
        catch (Exception e)
        {
            System.out.println ("Can't divide by zero");
        }
        finally
        {
            System.out.println ("Final block executed");
        }
    }
}
```

Output:

Can't divide by zero
Final block executed.

Program Explanation:

On occurance of 'divide by zero' exception in the try block, the control goes to the catch block and the exception gets caught and the statement will be printed as output. The finally block always execute. to free the resources.

throw:

→ 'throw' keyword is used for throwing the exception.

→ It is normally used within a method.

→ we cannot throw multiple exceptions using throw.

Syntax:

throw new Throwable's subclass

Example:

```
Public class Example
{
    Static void checkAge (int age)
    {
        if (age < 18)
        {
            throw new ArithmeticException ("cant vote");
        }
        else
        {
            System.out.println ("can vote");
        }
    }
    Public static void main (String[] args)
    {
        checkAge (15);
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException:
                                        cant vote
        at Example.checkAge (Example.java:4)
        at Example.main (Example.java:12)
```

throws:

→ When a method wants to throw an exception then keyword 'throws' is used.

→ 'Throws' is used with method signature.

→ We can declare multiple exceptions using 'throws'.

Syntax:

```
methodname(Parameters) throws exceptionlist
{
    // method code
}
```

Example:

```
Public class Example
{
  Static void checkAge (int age) throws ArithmeticException
  {
      if (age < 18)
      {
          throw new ArithmeticException ("can't vote");
      }
      else
      {
          System.out.println (" can vote");
      }
  }
  public Static void main (string[] args)
  {
      checkAge (15);
  }
}
```

Program Explanation:

Throw an exception if age is below 18 (Print "can't vote"). If age is 18 or older, Print "can vote".

The 'throws' keyword indicates what exception type may be thrown by a method.

---

2. MULTIPLE CATCH CLAUSES:

→ It is not possible for the try block to throw a single exception always.

→ Different exceptions may be raised by a single try block statements.

→ Depending upon the type of exception thrown, it must be caught.

→ To handle such situation, multiple catch blocks may exist for the single try block statements

Syntax:

```
try
{
    // Exception occurs
}
catch (Exception_type e)
{
    // Exception is handled here
}
catch (Exception_type e)
{
    //Exception is handled here
}
```

Example:

```
~~Public~~ class Example
{
   Public static void main (string args[])
   {
      try
      {
         int a[] = new int[5];
         a[5] = 30/0;
         System.out.println (a[10]);
      }
      catch (ArithmeticException e)
      {
         system.out.println ("Arithmetic Exception occurs");
      }
      catch (ArrayIndexOutofBoundException e)
      {
         system.out.println ("AIoutofBound Exception occurs");
      }
      catch (Exception e)
      {
         system.out.println ("Parent Exception occurs");
      }
      System.out.println ("rest of the code");
   }
}
```

Output:

Arithmetic Exception occurs

rest of the code.

Explanation:

Here, 30/0 throws arithmetic exception, so it is caught by the first catch block. After handling exception by the catch block, statements

after try-catch block has been executed.

3. NESTED TRY STATEMENTS:

→ When a try block is defined within another try. it is called nested try block.

→ When there are chances of occuring multiple exceptions of diffcient types by the same set of statemerits then such situation can be handled using the nested try statemerits.

Syntax:

```
try
{
    Statements;
    try
    {
        statements;
    }
    catch (Exception1 e1)
    {
        Statements;
    }
}
Catch (Exception2 e2)
{
    statements;
}
```

Inner try-catch

Outer try-catch

Example:

```
Class Example
{
    Public static void main(String args[])
    {
        try
        {
            int a[] = new int[10];
            System.out.println(a[12]);
            try
            {
                int res = 5/0;
            }
            catch(ArithmeticException e1)
            {
                System.out.println("Divide by 0 is ∞");
            }
        }
        catch(ArrayIndexOutofBoundException e2)
        {
            System.out.println("ArrayIndex Out of Bound");
        }
    }
}
```

Output:

Array Index Out of Bound

Explanation:

The outer try code tries to display the element at index 12 which gives rise to ArrayIndexOutofBoundException. Since the outer catch block handles this exception, we get the above output.

4. JAVA'S BUILT-IN EXCEPTIONS:

→ Exceptions that are already available in Java Libraries are referred to as built-in exception.

→ Java defines several exception classes inside the standard package java.lang.

→ It can be categorized into two broad Categories

1. Checked exceptions
2. Unchecked exceptions

→ Some of the common checked exceptions are classNotFoundException, IOException, FileNotFoundException etc,

→ Some of the common Unchecked exceptions are ArithmeticException, ArrayIndexOutofBoundException, NullPointerException etc.

Examples:

1. Arithmetic Exception:

It handles arithmetic exceptions such as 'divide by zero'.

```
Class Example
{
    Public Static void main (string args[])
    {
        try
        {
            int a = 30, b = 0;
            int c = a/b;
            System.out.println ("Result = " + c);
        }
```

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

Output:

Can't divide by zero.

2. ArrayIndexOutofBoundException:

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 Example
{
    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 (ArrayIndexOutofBoundException e)
        {
            System.out.println ("Array Index is out of bound");
        }
    }
}
```

Output:

Array Index is Out of bound.

3. FileNotFoundException:

This exception is raised when a file is not accessible or does not open.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class Example
{
    Public static void main (String args[])
    {
        try
        {
            File file = new File ("E://file.txt");
                                //This file does not exist
            FileReader fr = new FileReader(file);
        }
        Catch (FileNotFoundException e)
        {
            System.out.println ("File does not exist");
        }
    }
}
```

Output:

File does not exist.

4. IOException:

It is thrown when an input-output operation failed or interrupted.

```
import java.io.*;
class Example
{
    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.

5. class NotFound Exception:

This exception is raised when we try to access a class whose definition is not found.

6. Null Pointer Exception:

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

```
class Example
{
    Public static void main (String args[])
    {
        String a = null;
        System.out.Println(a.charAt(0));
    }
}
```

Output:

Exception in thread "main" java.lang.NullPointerException

---

5. **USER DEFINED EXCEPTION:**

[Creating own Exception or custom Exception]

→ In Java, we can create our own exception class and throw that exception using 'throw' keyword. These exceptions are known as user-defined or custom exceptions.

→ The Syntax for throwing out own exception is

```
throw new Throwable's subclass
```

→ Here the Throwable's subclass is actually a subclass derived from the Exception class.

Example:

```
throw new ArithmeticException();
```

↑ Throwable's subclass.

User-defined exception must extend Exception class.

Example Program:

```
import java.lang.Exception;
class MyException extends Exception
{
    MyException(String msg)
    {
        super(msg);
    }
}
class Example
{
    Public static void main(String args[])
    {
        int age = 15;
        try
        {
            if(age<21)
            throw new MyException("Age is less than condition");
        }
        catch(MyException e)
        {
            System.out.println("This is my Exception block");
            System.out.println(e.getMessage());
        }
    }
}
```

Output:

```
This is my Exception block.
Age is less than condition.
```

Explanation:

In the above code, the age value is 15 and in the try block - the 'if' condition throws the exception if the value is less than 21. As soon as the exception is thrown, the catch block gets executed. Hence as an output we get the message "This is my Exception block".

Then the control is transfered to the class MyException. The message is set and then printed by the catch block using the system.out.print statement by means of e.message.

6. MULTITHREADED PROGRAMMING:

→ In Java, we can write the programs that perform multitasking using the multithreading concept.

→ Thus Java allows to have multiple control flows in a single program by means of multithreading.

7. JAVA THREAD MODEL:

→ Thread is a tiny program running continuo-usly. It is sometimes called as 'Light-weight process'.

Thread	Process
* Thread is a light-weight process * Do not require separate address space for execution.	* Process is a heavy weight process. * Requires separate address space to execute

Thread Life cycle:

→ Thread life cycle specifies how a thread gets processed in the Java Program, by executing various methods.

→ During the life time of a thread, it can enter many states like new, ready (or) runnable, running, blocked (or) wait and dead (or) terminated state.

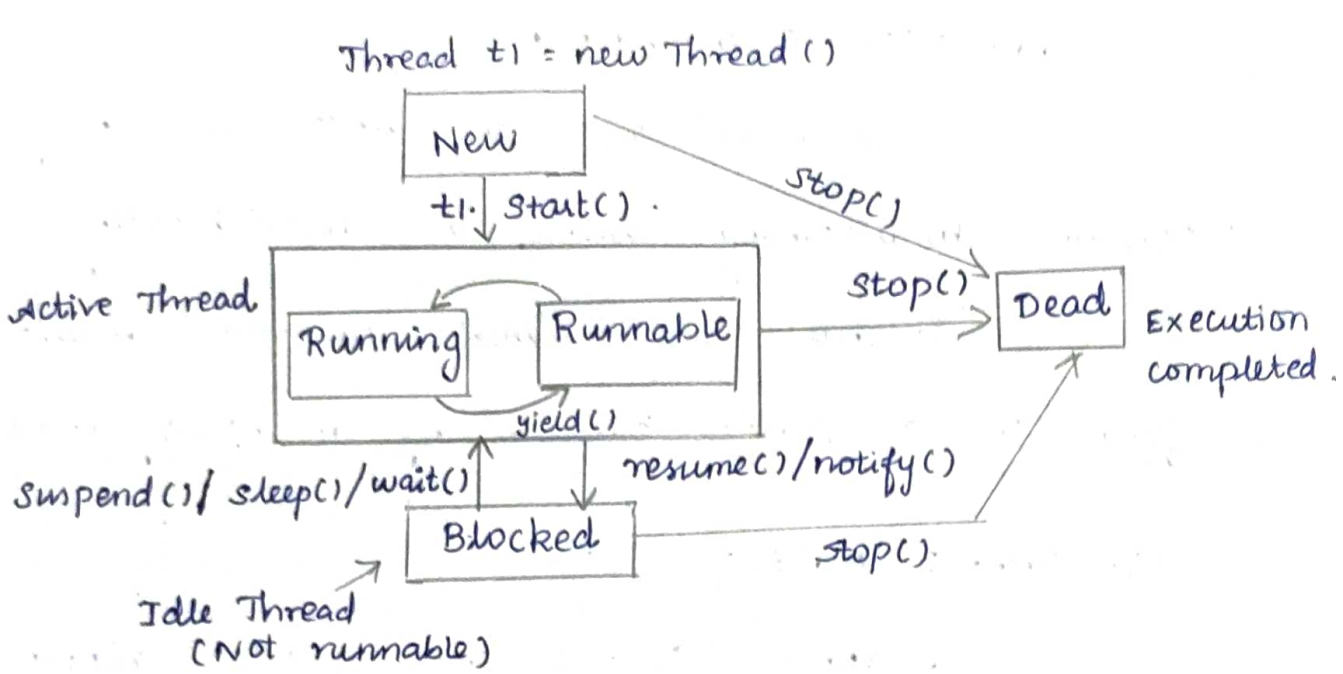


Fig: Thread Life cycle.

(i) New State:

→ When a thread object is created, then the thread is said to be in New state.

→ This state is also known as Born state.

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

(ii) Runnable state:

→ This is a state in which a thread is ready for execution and is waiting for the availability of Processor.

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

→ If all the threads that are waiting have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner.

→ The process of allocating time to threads is known as time slicing.

→ If a thread wants to give the control to another thread, it is done by using yeild() method.

(iii) Running state:

→ when the thread gets the CPU, it moves from the runnable to the running state.

→ The run() method of a thread called automatically by the start() method.

→ A thread can change its state to Runnable, Dead or Blocked from running state depending upon time slicing, thread completion of run() method or waiting for some resources.

(iv) Blocked State:

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

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

→ From blocked state, a thread can move back to runnable state when sleep time completed, waiting time completed, resume() method called etc.,

t1.resume();

(v) Dead State:

→ The dead state is also known as terminated state.

→ After succesful completion of the execution, the thread in runnable state enters the terminated state.

→ A thread can also be killed by using stop() method.

During thread life cycle, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time. All other threads live in some other states.

Program:

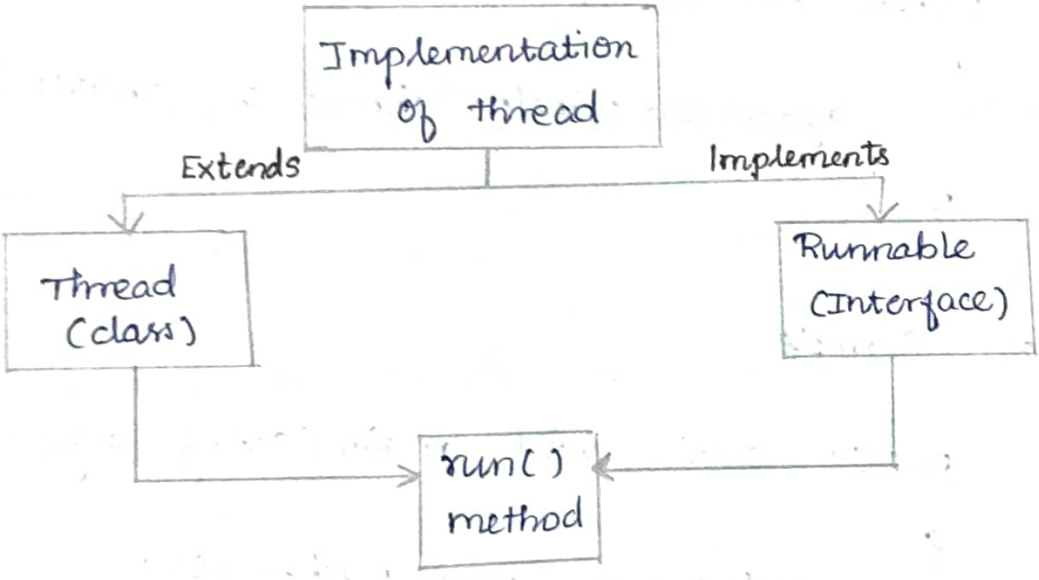
8. CREATING A THREAD AND MULTIPLE THREADS:

Creating a Thread:

→ When we run a java Program, the Program begins to execute its code starting from the main method. Therefore the JVM creates a thread to start executing the code present in main method. This thread is called as main thread.

→ In Java we can implement the thread programs using two ways

1. Using Thread class
2. Using Runnable interface.



→ run() method is the most important in threading Program. By using this, the thread's behaviour can be implemented.

```
Public void run()
{
    // Stmts for implementing thread
}
```

For invoking run(), thread object is needed. It can be obtained by creating & initiating a thread using the start() method.

(i) Extending Thread class.

Threads can be created using the Thread class. Using 'extend' keyword, your class ~~can~~ extends the Thread class.

Ex: class A extends Thread

Program:

```
class MyThread extends Thread
{
    Public void run()
    {
        System.out.println("Thread is created");
    }
}
class Example
{
    Public static void main(String args[])
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Output:

Thread is created

Explanation:

In the above program, we have created two class named MyThread and Example.

MyThread extends the Thread class. It has run() method which is called by t.start() in main method of class Example.

The thread gets created and executes by displaying the given message.

(ii) Implementing Runnable Interface:

Implementing thread Program using Runnable Interface is Preferable because

1. If a class extends a thread class, then it cannot extends any other classes which are needed
2. If Thread class is extended, then all its functionilities get inherited. This is expensive operation.

Program:

```
class MyThread implements Runnable
{
    Public void run()
    {
        System.out.println("Thread is created");
    }
}
class Example
{
    Public static void main(String args[])
    {
```

```
        MyThread Obj = new MyThread();
        Thread t = new Thread(obj);
        t.start();
    }
}
```

Output:

Thread is created.

Explanation:

→ While using the interface, it is necessary to use 'implements' keyword.

→ Inside main method

* create instance of class MyThread & pass this as a parameter to Thread class.

* Using the instance of class Thread, invoke the start() method. The start() method in-turn calls the run() in class MyThread. Then it executes and displays the given message.

CREATING MULTIPLE THREADS:

Multiple threads can be created both by extending Thread class and by implementing Runnable interface.

Program:

Extending Thread class

```
class A extends Thread
{
    Public void run()
    {
        for(int i=0;i<=5;i++)
        {
            System.out.println(i);
        }
    }
}
class B extends Thread
{
    Public void run()
    {
        for(int i=10;i>=5;i--)
        {
            System.out.println(i);
        }
    }
}
class Example
{
    Public static void main
                (string args[])
    {
        A t1=new A();
        B t2=new B();
        t1.start();
        t2.start();
    }
}
```

Output:
0
1
2
3
4
5
10
9
8
7
6
5

Implementing Runnable Interface

```
class A implements Runnable
{
    Public void run()
    {
        for(int i=0;i<=5;i++)
        {
            System.out.println(i);
        }
    }
}
class B implements Runnable
{
    Public void run()
    {
        for(i=10;i>=5;i--)
        {
            System.out.println(i);
        }
    }
}
class Example
{
    Public static void main(String
                        args[])
    {
        A obj1=new A();
        B obj2=new B();
        Thread t1=new Thread(obj1);
        Thread t2=new Thread(obj2);
        t1.start();
        t2.start();
    }
}
```

Output:
0
1
2
3
4
5
10
9
8
7
6
5

9. THREAD PRIORITIES:

→ Thread Priority is a number assigned to a thread that is used by Thread scheduler to decide which thread should be allowed to execute

→ Each thread have a priority. Priorities are represented by number between 1 and 10.

→ In most cases, 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.

→ Three constants defined in Thread class are

- ⊙ Public static int MIN_PRIORITY
- ⊙ Public static int NORM_PRIORITY
- ⊙ Public static int MAX_PRIORITY

→ Default priority of a thread is 5 (NORM_PRIORITY)

→ The value of MIN_PRIORITY is 1

→ The value of MAX_PRIORITY is 10.

Methods used:

→ getPriority() method returns priority of given thread.

→ setPriority(int newPriority) method changes the priority of thread to the value newPriority.

→ Void start() - to start the specified thread

→ Boolean isAlive() - Returns true if thread is started

and not died.

→ void run() - This method encapsulates the functionality of a thread.

Example: (using Thread class)

```
class A extends Thread
{
    Public void run()
    {
        for(int i=1; i<=3; i++)
        {
            System.out.println( "From thread A : i = " + i);
        }
        System.out.println("exit from A");
    }
}
class B extends Thread
{
    Public void run()
    {
        for(int j=1; j<=3; j++)
        {
            System.out.println("From thread B: j = " +j);
        }
        System.out.println("exit from B");
    }
}
class Main
{
    Public static void main(string args.[])
    {
        A ta = new A();
        ta.setPriority(Thread.MIN_PRIORITY+1);
```

```
        B tb = new B();
        tb.setPriority(6);
        ta.start();
        tb.start();
    }
}
```

Output:

```
From thread B: j = 1
From thread B: j = 2
From thread B: j = 3
exit from B
From thread A: i = 1
From thread A: i = 2
From thread A: i = 3
exit from A
```

Explanation:

→ Suppose there are two threads ta and tb with priorities 2 (MIN_PRIORITY+1) and 6, the thread tb will execute first based on maximum priority 6. After that thread ta will execute.

→ The above Program can also be implemented using interface.

~~Disadvantage~~:

→ Thread Priority does not guarantee execution order. They are highly OS specific.

10. SYNCHRONIZATION:

→ The process of ensuring one access at a time by one thread is called synchronization.

→ There are two ways to achieve the synchronization.

1. Using synchronized Methods
2. Using Synchronized Blocks (statements)

1. synchronized method:

→ If we declare any method as synchronized, it is known as synchronized method.

→ Synchronized method is used to lock an object for any shared resource.

→ When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example:

```
class Table
{
    Synchronized void PrintTable(int n)
    {
        System.out.println("TABLE" +n);
        for(int i=1; i<=5; i++)
        {
            System.out.println(n*i);
        }
    }
}
```

```
Public class Main
{
    Public static void main (String[] args)
    {
        Table t = new Table();
        Thread t1 = new Thread()
        {
            Public void run()
            {
                t.printTable(5);
            }
        };
        Thread t2 = new Thread()
        {
            Public void run()
            {
                t.printTable(10);
            }
        };
        t1.start();
        t2.start();
    }
}
```

Output

```
TABLE 5
5
10
15
20
25
TABLE 10
10
20
30
40
50
```

Explanation :

In the above program, we have created two threads t1 and t2. PrintTable() method is invoked with parameters 5 and 10 respectively. This method executes thread t1 at first and then t2 in synchronized manner.

2. Synchronized Block:

→ Synchronized block can be used to perform synchronization on any specific resource of the method.

→ Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

Example:

```
Class Table
{
    Void PrintTable(int n)
    {
        Synchronized(this)
        {
            System.out.println("TABLE" + n);
            for(int i=1; i<=5; i++)
            {
                system.out.println(n*i);
            }
        }
    }
}
Public class Main
{
    Public Static void main(string[] args)
    {
        Table t = new Table();
        Thread t1 = new Thread()
        {
            Public void run()
            {
                t.PrintTable(5);
            }
        };
```

← This is a synchronized block

```
        Thread t2 = new Thread()
        {
            public void run()
            {
                t.printTable(10);
            }
        };
        t1.start();
        t2.start();
    }
}
```

Output:

```
TABLE 5
5
10
15
20
25
TABLE 10
10
20
30
40
50
```

→ If we put all the codes of the method in the synchronized block, it will work same as synchronized method.

Advantages & Limitations:

* Synchronization is a good way to achieve mutual exclusion.
* All methods of a class need not be synchronized.
* we cannot synchronize the constructors.
* object has one lock. Thread can acquire more than one lock.
* Java synchoonization does not allow concurrent reads.
* Synchronized methods run very slowly and can degrade the performance. So synchronize the method only when it is absolutely necessary.

11. INTER - THREAD COMMUNICATION:

→ Two or more threads communicate with each other by exchanging the messages. This mechanism is called inter-thread communication.

→ There are three in-built methods that take part in inter-thread communication.

1. notify() - If a particular thread is in the sleep mode, then that thread can be resumed using the notify call.

2. notifyall() - This method resumes all the thread that are in suspended state.

3. wait() - The calling thread can be send into a sleep mode.

Example:

```
class myclass
{
    int val;
    boolean flag = false;
    synchronized int get()
    {
        if (!flag)
        try
        {
            wait();
        }
        catch (InterruptedException e)
        {
            System.out.println ("Interrupted Exception!");
        }
```

```
        System.out.println("consumer consuming :" + val);
        flag = false;
        notify();
        return val;
    }
    synchronized void Put(int val)
    {
        if(flag)
        try
        {
            wait();
        }
        catch(InterruptedException e)
        { }
        this.val = val;
        flag = true;
        System.out.println("Producer Producing:" + val);
        notify();
    }
}
class Producer extends Thread
{
    myclass t1;
    Producer(myclass t)
    {
        t1 = t;
    }
    Public void run()
    {
        for(int i=0; i<3; i++)
        {
            t1.put(i);
        }
    }
}
```

Producer thread is writing the numbers from 0 to 2

```
class consumer extends Thread
{
    myclass t2;
    consumer(myclass t)
    {
        t2 = t;
    }
    Public void run()
    {
        for(int i=0; i<3; i++)
        {
            t2.get();
        }
    }
}
class Interthread
{
    Public Static void main(String[] args)
    {
        myclass obj = new myclass();
        Producer P = new Producer(obj);
        consumer C = new consumer(obj);
        P.start();
        C.start();
    }
}
```

consumer thread is reading the numbers from 0 to 2

Output:

```
Producer producing: 0
Consumer consuming: 0
Producer Producing: 1
Consumer Consuming: 1
Producer Producing: 2
Consumer Consuming: 2
```

Explanation:

The above program has two threads producer and consumer.

• Inside get() → The wait() suspends the execution, by that time producer writes the value. when the data is ready, it notifies other thread.

→ When the consumer reads the data, execution inside get() is suspended. After the data is obtained, get() calls notify().

→ This tells the Producer can write new data in the queue.

• Inside put() → The wait() suspends the execution, by that time, the consumer removes the item from queue.

→ when execution resumes, the next item is put in the queue and notify() is called.

→ when notify() is issued, the consumer can remove the corresponding item for reading.