# Unit-3

## Exception

An **Exception** is a run time error, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

## Exception Handling in Java

- The Exception Handling in Java is one of the powerful feature to handle the runtime errors so that normal flow of the application can be maintained.
- Exception Handling is used to convert system error message into user friendly error message.

## Advantage of Exception Handling

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

**Let's take a scenario:**

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

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

## Types of Exception

There are two types of exceptions

1.Checked Exception

2.Un-Checked Exception

## 1.Checked Exception

Checked Exceptions are the exception which checked at compile-time. These exceptions are directly sub-class of java.lang.Exception class.

**The following are a few built-in classes used to handle checked exceptions in java.**

- IOException
- FileNotFoundException
- ClassNotFoundException

## 2.Un-Checked Exception

Un-Checked Exceptions are the exception both identifies or raised at run time. These exceptions are directly sub-class of java.lang.RuntimeException class.
**Note:** In real time application mostly we can handle un-checked exception.

**The following are a few built-in classes used to handle unchecked exceptions in java:**

- **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
- **ArrayIndexOutOfBounds Exception:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
- **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
- **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
- **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative than the size of the string.

## Exception Models in Java

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java suports are as follows.

- **Termination Model**
- **Resumptive Model**

## 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.

## 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 util the result is satisfactory.

## Uncaught Exceptions(with out using try&catch)

**Example without Exception Handling**

```
class ExceptionDemo
{
    public static void main(String[] args)
    {
        int a=30, b=0;
        int c=a/b;
        System.out.println("Denominator should  not be zero");
    }
}
```

**Explanation:**

- Abnormally terminate program and give a message like below,

  **Exception in thread "main" java.lang.ArithmeticException: / by zero**
  **at ExceptionDemo.main(ExceptionDemo.java:7)**
- This error message is not understandable by user so we convert this error message into user friendly error message, like "denominator should not be zero".

## How to Handle the Exception

**Use Five keywords for Handling the Exception**

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

## 1.try block

- The try block contains set of statements where an exception can occur.
- In other words try block always contains problematic statements.
- A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

### Syntax

```
try
{
   //statements that may cause an exception
}
```

## 2.catch block

- A catch block is where we handle the exceptions, this block must follow the try block.
- A single try block can have multiple catch blocks associated with it. We can catch different exceptions in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception executes.
- For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

### Syntax of try-catch in java

```
try
{
     // statements causes problem at run time
}
catch(type of exception-1 object-1)
{
```

```
        // statements provides user friendly error message
    }
    catch(type of exception-2 object-2)
    {
        // statements provides user friendly error message
    }
    finally
    {
        // statements which will execute compulsory
    }
```

**Example(try&catch)**

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
            int a=30, b=0;
            try
            {
                int c=a/b;
            }
            catch (ArithmeticException e)
            {
                System.out.println("Denominator should not be zero");
            }
        }
}
```

**Output:**

Denominator should not be zero

## Multiple Catch Blocks

We can write multiple catch blocks for generating multiple user friendly error messages to make our application strong.

**Example**

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

```
        {
            int a=30, b=0;
                try
                {
                        int c=a/b;
                        System.out.println("Result: "+c);
                }
                catch(NullPointerException e)
                {
                       System.out.println("Enter valid number");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Denominator not be zero");
                }

        }
    }
```

## Nested try Statements

The try block within a try block is known as nested try block in java.

### 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

```
    try
    {
        statement 1;
        statement 2;
        try
        {
            statement 1;
            statement 2;
        }
        catch(Exception e)
        {
```

```
                ……
            }
        }
        catch(Exception e)
        {
            …….
        }
```

## Example

```
class NestedTry
{
        public static void main(String args[])
        {
                try
                {
                  try
                  {
                        int a[]=new int[5];
                        a[5]=4;
                  }
                  catch(ArrayIndexOutOfBoundsException e)
                  {
                     System.out.println(e);
                  }
                  }
            catch(Exception e)
            {
               System.out.println("handeled");
            }
            System.out.println("normal flow..");
        }
}
```

## 3.throw

- 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.

**Syntax**

throw Instance

**Example**

```
throw new ArithmeticException("/ by zero");
```

**Example**

```
class ThrowExcep
{
        static void fun()
        {
                try
                {
                     throw new NullPointerException("demo");
                }
                catch(NullPointerException e)
                {
                      System.out.println("Caught inside fun().");
                      throw e; // rethrowing the exception
                }
        }
        public static void main(String args[])
        {
                  try
                  {
                       fun();
                  }
                  catch(NullPointerException e)
                  {
                       System.out.println("Caught in main.");
                  }
          }
}
```

**Output:** Caught inside fun().

Caught in main.

## 4.throws

throws is a keyword in java language which is used to throw the exception which is raised in the called method to it's calling method throws keyword always followed by method signature.

### Syntax

```
returnType methodName(parameter) throws Exception_class....
{
.....
}
```

### Example

```
class ThrowsExecp
{
        static void fun() throws IllegalAccessException
        {
           System.out.println("Inside fun(). ");
           throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
                try
                {
                    fun();
                }
                catch(IllegalAccessException e)
                {
                    System.out.println("caught in main.");
                }
         }
}
```

**Output:**

```
Inside fun().
caught in main.
```

## 5.finally Block

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

### Example

```
class TestFinallyBlock
{
        public static void main(String args[])
       {
            try
            {
                int data=25/0;
                System.out.println(data);
            }
            catch(NullPointerException e)
            {
                System.out.println(e);
            }
           finally
           {
               System.out.println("finally block is always executed");
           }
           System.out.println("rest of the code...");
       }
  }
```

**Output:**

finally block is always executed

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

## Java's Built-in Exceptions:

Inside the standard package **java.lang**, Java defines several exception classes.

| Exception | Meaning |
|---|---|
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| EnumConstantNotPresentException | An attempt is made to use an undefined enumeration value. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| TypeNotPresentException | Type not found. |
| UnsupportedOperationException | An unsupported operation was encountered. |

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

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

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

## re-throwing exceptions

- Sometimes we may need to rethrow an exception in Java.
- If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.
- The rethrow expression causes the **originally thrown object to be rethrown.**

- Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred.
- Any catch blocks for the enclosing try block have an opportunity to catch the exception.

**Example**

```
class RethrowExcep
{
        static void fun()
        {
                try
                {
                      throw new NullPointerException("demo");
                }
                catch(NullPointerException e)
                {
                       System.out.println("Caught inside fun().");
                       throw e; // rethrowing the exception
                }
        }
        public static void main(String args[])
        {
                try
                {
                   fun();
                }
                catch(NullPointerException e)
                {
                   System.out.println("Caught in main.");
                }
        }
}
```

**Output:**

```
Caught inside fun().
Caught in main.
```

## Creating Own Exception(Custom Exception in Java)

If any exception is design by the user known as user defined or Custom Exception. Custom Exception is created by user.

### Rules to design user defined Exception

1. Create a package with valid user defined name.
2. Create any user defined class.
3. Make that user defined class as derived class of Exception or RuntimeException class.
4. Declare parametrized constructor with string variable.
5. call super class constructor by passing string variable within the derived class constructor.
6. Save the program with public class name.java

### Example

```
package nage;
public class InvalidAgeException extends Exception
{
    public InvalidAgeException (String s)
    {
        super(s);
    }
}
```

### A Class that uses above InvalidAgeException:

```
class CustomException
{
        static void validate(int age) throws InvalidAgeException
        {
            if(age<18)
             throw new InvalidAgeException("not valid");
            else
             System.out.println("welcome to vote");
         }
       public static void main(String args[])
       {
           try
           {
              validate(13);
```

```
            }
            catch(Exception m)
            {
                System.out.println("Exception occured: "+m);
            }
            System.out.println("rest of the code...");
        }
    }
```

**Output:**

```
Exception occured: InvalidAgeException:not valid
rest of the code...
```

## Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously. The aim of multithreading is to achieve the concurrent execution.

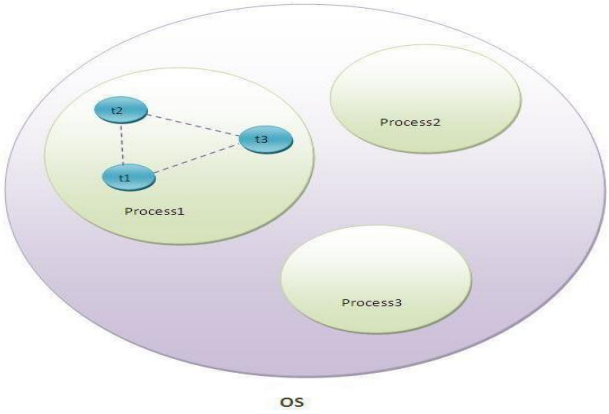
## Differences Between Multiprocessing and Multithreading

| Multiprocessing | Multithreading |
|---|---|
| Each process have its own address in memory i.e. each process allocates separate memory area. | Threads share the same address space. |
| Process is heavyweight. | Thread is lightweight. |
| Cost of communication between the process is high. | Cost of communication between the thread is low. |
| Process-based multitasking is totally controlled by the operating system. | thread-based multitasking can be controlled by the programmer to some extent in a program. |

## What is Thread?

A thread is a light weight subprocess, a smallest unit of processing.

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




As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

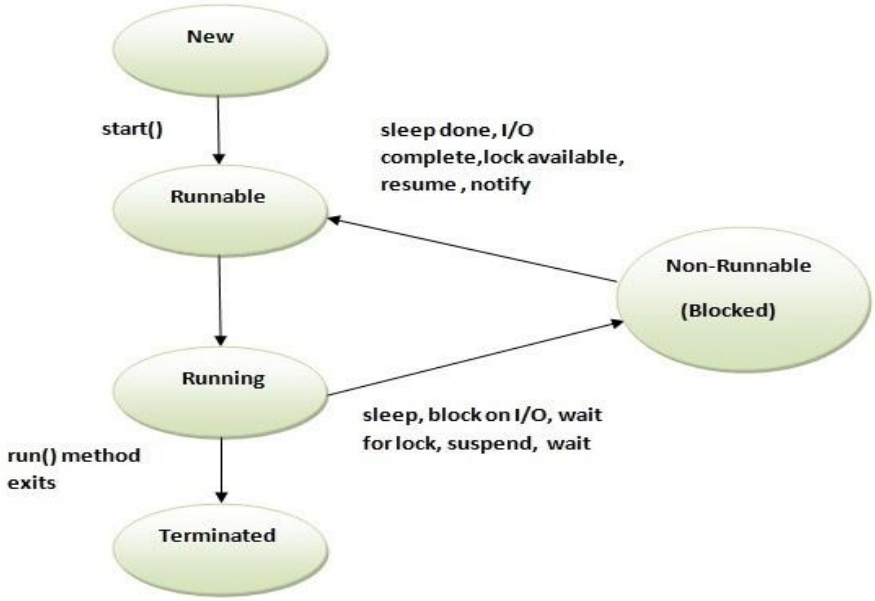
**Note:**At a time only one thread is executed.

## Java Thread Model (Life Cycle of a Thread)

A thread can be in one of the five states in the thread. The life cycle of the thread is controlled by JVM.

**The thread states are as follows:**

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



### New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

### Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### Running

The thread is in running state if the thread scheduler has selected it.

### Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### Terminated

A thread is in terminated or dead state when its run() method exits.

## Creating Threads

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### 1.Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface

**Commonly used Constructors of Thread class:**

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

**Commonly used methods of Thread class:**

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.

3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void suspend():** is used to suspend the thread(depricated).
15. **public void resume():** is used to resume the suspended thread(depricated).
16. **public void stop():** is used to stop the thread(depricated).

## Example: By extending Thread class

```
class Multi extends Thread
{
        public void run()
        {
              System.out.println("thread is running...");
        }
        public static void main(String args[])
        {
              Multi t1=new Multi();
              t1.start();
         }
}
```

**Output:** thread is running...

## 2.Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

**public void run():** is used to perform action for a thread

## Starting a thread:

**start() method** of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run() method will run.

## Example: By implementing the Runnable interface

```
class Multi3 implements Runnable
{
        public void run()
        {
            System.out.println("thread is running...");
        }
        public static void main(String args[])
        {
            Multi3 m1=new Multi3();
            Thread t1 =new Thread(m1);
            t1.start();
         }
}
```

**Output**:

```
thread is running...
```

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

## Thread Priorities

- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular 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 defiend 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:**

```
class MultiThread extends Thread
{
    public void run()
    {
      System.out.println("running thread name is:"+Thread.currentThread().getName());
      System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }
    public static void main(String args[])
    {
        MultiThread m1=new MultiThread ();
        MultiThread m2=new MultiThread ();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
   }
}
```

**Output:**

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

## Synchronizing Threads

## Synchronization

Synchronization is the capability of control the access of multiple threads to any shared resource. Synchronization is better in case we want only one thread can access the shared resource at a time.

## Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

## Thread Synchronization

There are two types of thread synchronizations.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
2. Cooperation (Inter-thread communication in java)

### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by two ways in java:

1. by synchronized method
2. by synchronized block

## Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent.

### Example:

```
class Table
{
      void printTable(int n)
      {
            //method not synchronized
             for(int i=1;i<=5;i++)
             {
                  System.out.println(n*i);
                  try
                  {
                        Thread.sleep(400);
                  }
                 catch(Exception e)
                 {
                        System.out.println(e);
```

```
            }
        }
    }
}
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}
class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
}
class TestSynchronization
{
    public static void main(String args[])
    {
        Table obj = new Table();    //only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
```

```
    }
```

**Output:**

```
5
100
10
200
15
300
20
400
25
500
```

## Java synchronized method

- If you 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)
    {
        //method not synchronized
        for(int i=1;i<=5;i++)
        {
            System.out.println(n*i);
            try
            {
                Thread.sleep(400);
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
    }
```

```
 }
class MyThread1 extends Thread
{
          Table t;
          MyThread1(Table t)
          {
              this.t=t;
          }
          public void run()
          {
              t.printTable(5);
          }
   }
  class MyThread2 extends Thread
  {
          Table t;
          MyThread2(Table t)
          {
              this.t=t;
          }
          public void run()
          {
              t.printTable(100);
          }
 }
  class TestSynchronization
  {
        public static void main(String args[])
        {
              Table obj = new Table();    //only one object
              MyThread1 t1=new MyThread1(obj);
              MyThread2 t2=new MyThread2(obj);
              t1.start();
              t2.start();
        }
  }
```

**Output:**

```
5
10
15
20
25
100
200
300
400
500
```

## Synchronized Block in Java

- 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.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
- Points to remember for Synchronized block
- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

## Syntax to use synchronized block

```
synchronized (object reference expression)
{
    //code block
}
```

## Example of synchronized block

```
class Table
{
        void printTable(int n)
        {
                synchronized(this)
                { //synchronized block
                        for(int i=1;i<=5;i++)
                        {
```

```
                    System.out.println(n*i);
                    try
                    {
                        Thread.sleep(400);
                    }
                    catch(Exception e)
                    {
                        System.out.println(e);
                    }
                }
            }
        }//end of the method
}
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }

}
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
        {
            this.t=t;
        }
        public void run()
        {
            t.printTable(100);
        }
        public static void main(String args[])
```

```
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
}
```

**Output:**

```
5
10
15
20
25
100
200
300
400
500
```

## Interthread Communication

- Inter thread communication is the concept where two or more threads communicate to solve the problem of **polling**.
- In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
- That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
- The inter-thread communication allows the synchronized threads to communicate with each other.

**Java provides the following methods to achieve inter thread communication.**

| Method | Description |
|---|---|
| **void wait( )** | It makes the current thread to pause its execution until other thread in the same monitor calls notify( ) |
| **void notify( )** | It wakes up the thread that called wait( ) on the same object. |

| Method | Description |
|---|---|
| **void notifyAll()** | It wakes up all the threads that called wait( ) on the same object. |

## Let's look at an example problem of producer and consumer.

- The producer produces the item and the consumer consumes the same.
- But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced.
- So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same.
- Here we use the inter-thread communication to implement the producer and consumer problem.

**Example**

```
class ItemQueue
{
        int item;
        boolean valueSet = false;
        synchronized int getItem()
        {
                while (!valueSet)
                try
                {
                    wait();
                }
                catch (InterruptedException e)
                {
                    System.out.println("InterruptedException caught");
                }
                System.out.println("Consummed:" + item);
                valueSet = false;
                try
                {
                    Thread.sleep(1000);
```

```
                }
                catch (InterruptedException e)
                {
                        System.out.println("InterruptedException caught");
                }
                notify();
                return item;
        }
        synchronized void putItem(int item)
        {
                while (valueSet)
                try
                {
                        wait();
                }
                catch (InterruptedException e)
                {
                        System.out.println("InterruptedException caught");
                }
                this.item = item;
                valueSet = true;
                System.out.println("Produced: " + item);
                try
                {
                        Thread.sleep(1000);
                }
                catch (InterruptedException e)
                {
                        System.out.println("InterruptedException caught");
                }
                notify();
        }
}
class Producer implements Runnable
{
        ItemQueue itemQueue;
        Producer(ItemQueue itemQueue)
        {
```

```
                this.itemQueue = itemQueue;
                new Thread(this, "Producer").start();
        }
        public void run()
        {
                int i = 0;
                while(true)
                {
                        itemQueue.putItem(i++);
                }
        }
}
class Consumer implements Runnable
{
        ItemQueue itemQueue;
        Consumer(ItemQueue itemQueue)
        {
                this.itemQueue = itemQueue;
                new Thread(this, "Consumer").start();
        }
        public void run()
        {
                while(true)
                {
                        itemQueue.getItem();
                }
        }
}
class ProducerConsumer
{
        public static void main(String args[])
        {
                ItemQueue itemQueue = new ItemQueue();
                new Producer(itemQueue);
                new Consumer(itemQueue);
        }
}
```

**Output:**

Produced: 0
Consummed:0
Produced: 1
Consummed:1
Produced: 2
Consummed:2
Produced: 3
Consummed:3
Produced: 4
Consummed:4
Produced: 5
Consummed:5
Produced: 6
Consummed:6
Produced: 7
Consummed:7
Produced: 8
Consummed:8