

UNIT-IV

Multi-Threaded programming and Exception handling

4.1:-Explain the thread model of java

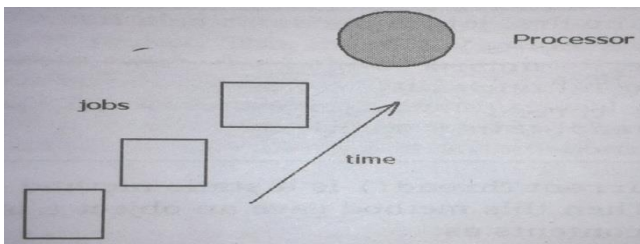
A thread represents a separate path of execution of a group of [statements](#). In a java program, if we write a group of statements then these statements are executed by JVM one by one, this execution is called a thread because JVM uses a thread to execute these [statements](#). This thread is used by JVM to execute the program statements

The way the statements are execute is of 2-types

1) Single tasking

2) Multi tasking

→**Single tasking**: - In Single tasking; **only one task is given to the processor at a time**. This means we are wasting a lot of processor time & microprocessor has to sit idle without any job for a long time. This is the drawback in single tasking.

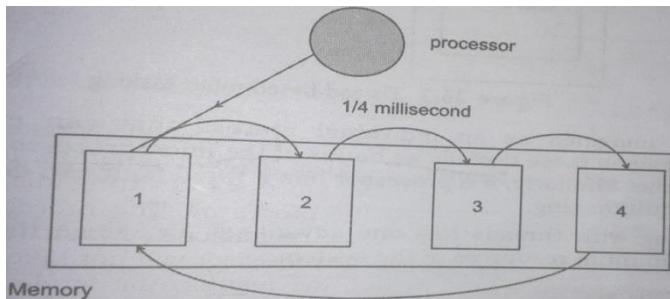


→**Multi tasking**: - To use the processor's time in an option way, we can give it **several jobs at a time**. This is called multi tasking. Multi tasking is at 2-types

a) process-based multi tasking

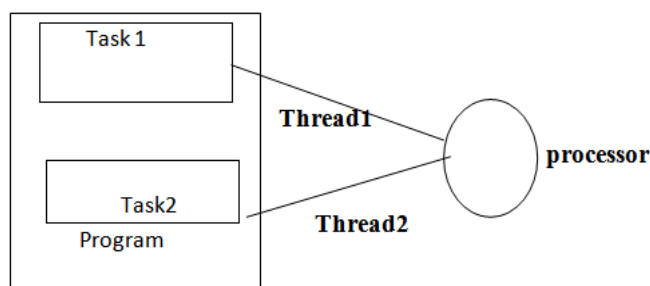
b) Thread- based multi tasking

→**Process-based multi tasking**: - In process-based multi tasking, **several programs are executed at a time, by the micro processor**.



Generally, we have only one processor in our computer system. [one](#) processor has to execute several task means that it has to execute them in a round-robin method. Strictly speaking, this is not multi tasking. The processor is quickly executing the task one by one so-quickly that we feel all the jobs are executed by the processor at a time.

→**Thread-based multi tasking**:- In Thread-based multi tasking **several parts of the same program** is executed at a time, by the Microprocessor.



Here we have a program, in this program, there are 2-parts. These 2-parts may represent 2-separate blocks of code. Each part may perform a separate [task](#). The processor should execute the 2- parts (tasks) simultaneously, so the processor uses 2- separate threads to execute these 2- tasks we can imagine these threads as **threads as hands of the microprocessor**.

Advantage of using threads to achieve multi tasking:-

- ✓ Since threads are light weight processes **they use minimum resources of the system**.

Why threads are called light-weight?

- ✓ Threads are light-weight **because they utilize minimum resources of the system**. This means they **take less memory & less processor time**.

Define a thread & write the uses of Threads:-

Thread:- Threads are light-weighted processes. A thread represents a separate path of execution of a group of statements.

Uses of threads:- Thread can be used for multiple purposes.

- 1) Threads are **mainly used in server-side programs to serve the needs of multiple clients on a network (or) internet**.
- 2) Threads are also **used to create games & animation**. Animation means moving the objects from one place to another. In many games, generally we have to perform more than one task simultaneously. There threads will be of invaluable help.

4.4 Create thread using Thread Class

We know that in every java program, there is a main thread available already. Apart from this main thread, we can also create our own thread in a program; the following steps should be used. Java defines 2-ways to create thread.

- 1) **We can implement the Runnable interface.**
- 2) **We can extend the 'Thread' class itself.**

Steps to create threads:

We can make **our class runnable as thread** by extending the class java.lang.Thread. This gives us access to all the thread methods directly. It includes the following steps:

1. Create the **MyClass** that **extending the Thread class**.
2. **Implement the run() method** that is responsible for executing the sequence of code that the thread will execute.
3. **Create an object to MyClass** so that the run () method is available for execution.
4. Finally **Run the Thread by the start()** method to initiate the thread execution

Now, the thread will start execution the object of MyClass. In that object, run() method is found, hence it will execute the statements inside that run() method.

```
//creating a thread and running it by extending the Thread class
class MyThread extends Thread //step1
{
    public void run( ) //step2
    {
        for(int i=1;i<=100;i++)
        {
            System.out.println(i);
        }
    }
}
```

```

class ThreadDemo
{
    public static void main(String args[])
    {
        MyThread obj=new MyThread();//step3
        obj.start(); //step4
    }
}

```

4.5 Create Thread using Runnable interface

By Implementing Runnable interface:

Easiest way to create a thread is to create class that implements “Runnable” interface which abstracts a unit of executable code. Then constructs a thread or any object that implements the Runnable.

It includes the following steps to create thread that implements Runnable interface:

1. Create the **MyClass** that **extending the Thread class**.
2. **Implement the run() method** that is responsible for executing the sequence of code that the thread will execute.
3. **Create an object to MyClass** so that the run () method is available for execution.
4. Now **create a thread & attach the thread to the object**.
5. Finally **Run the Thread by the start()** method to initiate the thread execution.

```

//creating a thread and running it by implementing the Runnable interface
class MyThread implements Runnable //step1
{
    public void run() //step2
    {
        for(int i=1;i<=100;i++)
        {
            System.out.println(i);
        }
    }
}
class ThreadDemo2
{
    public static void main(String args[])
    {
        MyThread obj=new MyThread();//step3
        Thread t1=new Thread(obj);//step4
        t1.start();//step5
    }
}

```

Which method is to choose?

It is best to simply implement Runnable interface.

4.6 Create Multiple Threads

We have been using only two threads: The main thread and the child thread. However, our program can create as many threads as it needs. It is implemented in the below program.

Example1:

```

class A extends Thread
{
    public void run() {

```

```

        System.out.println("thread is started");
        for(int i=1;i<=4;i++)
        {
            System.out.println("from thread A: i="+i);
        }
        System.out.println("exit from A");
    }
}
class B extends Thread
{
    public void run()
    {
        System.out.println("thread B started");
        for (int j=1;j<=4;j++)
        {
            System.out.println("from thread B:j="+j);
        }
        System.out.println("exist from B");
    }
}
class MultipleThreads
{
    public static void main(String args[])
    {
        A obj1 =new A();
        B obj2=new B();
        obj1.start();
        obj2.start();
    }
}

```

java program showing 2-threads working simultaneously:-

```

//2-threads performing 2 tasks at a time-Theatre example
import java.lang.*;
class MyThread implements Runnable
{
    String str; //to represent the task
    MyThread(String str)
    {
        this.str=str;
    }
    public void run()
    {
        for(int i=1;i<=10;i++)
        {
            System.out.println(str+"."+i);
        }
    }
}

```

```

        try
        {
            Thread.sleep(1000); //cease thread execution for 1000 msec
        }
        catch(InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }
}

class MultiThread
{
    public static void main(String args[])
    {
        MyThread obj1=new MyThread("cut the ticket");
        MyThread obj2=new MyThread("show the seat");
        Thread t1=new Thread(obj1);
        Thread t2=new Thread(obj2);
        t1.start();
        t2.start();
    }
}

```

```

C:\Windows\system32\cmd.exe
C:\Users\chaanaky's\Desktop\java programs>javac MultiThread.java
C:\Users\chaanaky's\Desktop\java programs>java MultiThread
cut the ticket:1
show the seat:1
cut the ticket:2
show the seat:2
cut the ticket:3
show the seat:3
cut the ticket:4
show the seat:4
cut the ticket:5
show the seat:5
cut the ticket:6
show the seat:6
cut the ticket:7
show the seat:7
cut the ticket:8
show the seat:8
cut the ticket:9
show the seat:9
cut the ticket:10
show the seat:10
C:\Users\chaanaky's\Desktop\java programs>

```

4.2:- EXPLAIN THREAD PRIORITIES:

When the threads are created & started a 'thread- scheduler' program in JVM will load them into memory & execute [them](#). [This](#) scheduler will allot more JVM time to those threads which are having higher priorities. The threads of the same priority are given equal treatment by the java scheduler and, therefore, execute on a FCFS basis.

The priority number of a thread will change from 1 to 10.

- The minimum priority i.e, **Thread. MIN_PRIORITY** of a thread is **1**.
- The maximum priority i.e, **Thread. MAX_PRIORITY** of a thread is **10**.
- The normal priority (or) default priority i.e, **Thread. NORM_PRIORITY** of a thread is **5**.

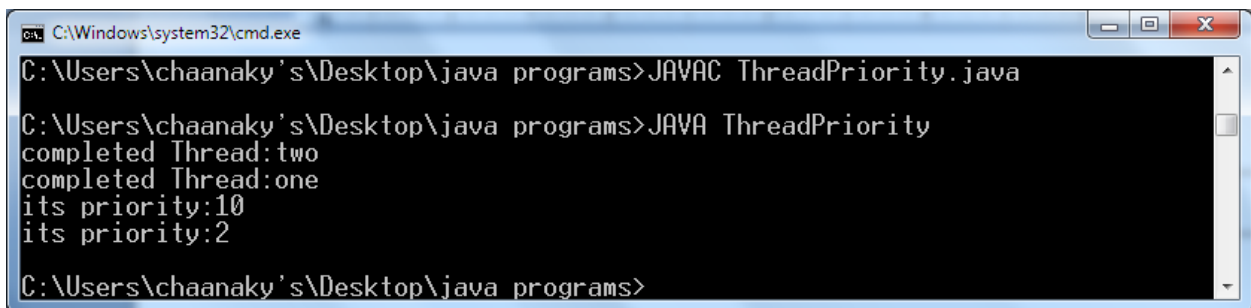
To set a thread's priority, use the `setPriority ()` method, which is a member of `Thread`. The general form is.

```
final void setPriority (int number);
```

To get the current priority of a thread, use the `getPriority ()` method of `Thread` class.

```
final int getPriority();
```

```
// java program to understand the thread priorities. The thread with higher priority number // will
// complete its execution first.
class MyThread extends Thread
{
    int count=0;
    public void run()
    {
        for(int i=1;i<=200;i++)
        {
            count++;
        }
        System.out.println("completed Thread:"+Thread.currentThread().getName());
        System.out.println("its priority:"+Thread.currentThread().getPriority());
    }
}
class ThreadPriority
{
    public static void main(String x[])
    {
        MyThread obj=new MyThread();
        Thread t1=new Thread(obj,"one");
        Thread t2=new Thread(obj,"two");
        t1.setPriority(2);
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
}
```



```
C:\Windows\system32\cmd.exe
C:\Users\chaanaky's\Desktop\java programs>JAVAC ThreadPriority.java
C:\Users\chaanaky's\Desktop\java programs>JAVA ThreadPriority
completed Thread:two
completed Thread:one
its priority:10
its priority:2
C:\Users\chaanaky's\Desktop\java programs>
```

4.3:-EXPLAIN THE CONCEPT OF SYNCHRONIZATION:-

What is Thread synchronization?

When a thread is already acting on an object, preventing any other thread from acting on the same object is called 'thread synchronization'(or) 'thread safe'. The object on which the threads are synchronized is called **synchronized object**. Thread synchronization is recommended when multiple threads are used on the same object.

- Synchronized object is like a locked object, locked on a thread. It is a room with only one door. A person has entered the room & locked from it. The second person who wants to enter the room should wait till the first person came out.
- A thread also locks the object after entering it. Then the next thread can't enter it till the first thread comes out. This means the object is locked mutually on threads. So, this object is called '**mutex**'(mutually exclusive lock).

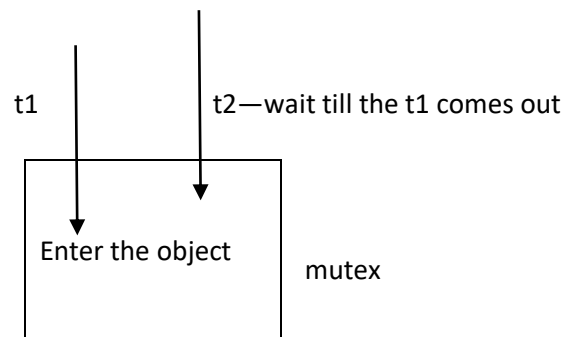


Fig: Thread synchronization

How can we synchronize the object? There are 2-ways of doing this.

1) Using synchronized block: -Here, **we can embed a group of statements of the object** (inside run () method) **within a synchronized block**, as shown here:

```
synchronized(object)
{
    //statements;
}
```

The statements inside the synchronized block are all available to only one thread at a time. They are not available to more than one thread simultaneously.

2) Using synchronized keyword:- we can **synchronize an entire method by using synchronized keyword**. For e g:- if we went to synchronize the code of display() method, then add the synchronized keyword before the method as show here.

```
synchronized void display()
{
    //statements;
}
```

Diff b/w synchronized block, synchronized keyword:

Synchronized block is useful to synchronize a block of statements. Synchronized keyword is useful to synchronize an entire method.

```
//java program to synchronize the threads acting on the same object.
class Reserve implements Runnable
{
    int available=1;
    int wanted;
    Reserve(int i)
    {
        wanted=i;
    }
}
```

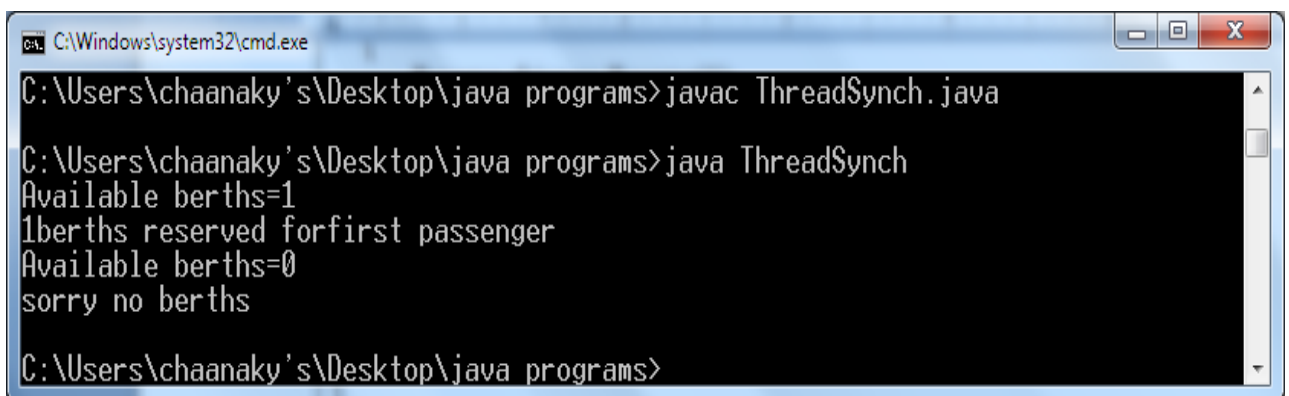
```

public void run()
{
    synchronized(this)
    {
        System.out.println("Available berths="+available);
        if(available>=wanted)
        {
            String name=Thread.currentThread().getName();
            System.out.println(wanted+"berths reserved for"+name);
            try
            {
                Thread.sleep(1500);
                available=available-wanted;
            }
            catch(InterruptedException ie)
            {
            }
        }
        else
            System.out.println("sorry no berths");
    }
}

}

class ThreadSynch
{
    public static void main(String x[])
    {
        Reserve obj=new Reserve(1);
        Thread t1=new Thread(obj);
        Thread t2=new Thread(obj);
        t1.setName("first passenger");
        t2.setName("second passenger");
        t1.start();
        t2.start();
    }
}

```



```

C:\Windows\system32\cmd.exe
C:\Users\chaanaky's\Desktop\java programs>javac ThreadSynch.java
C:\Users\chaanaky's\Desktop\java programs>java ThreadSynch
Available berths=1
1berths reserved forfirst passenger
Available berths=0
sorry no berths
C:\Users\chaanaky's\Desktop\java programs>

```


4.9: EXPLAIN THREAD DEAD LOCK :

- Thread dead lock: deadlock occurs when 2-threads have a circular dependency on a pair of synchronized objects.
- That is, when a thread has locked on object & waiting for another object to be released by another thread, and the other thread is also waiting for the first thread to release the first object, both the threads will continue waiting forever.

```
import java.io.*;
class BookTicket extends Thread
{
    Object train,comp;
    BookTicket(Object train,Object comp)
    {
        this.train=train;
        this.comp=comp;
    }
    public void run()
    {
        synchronized(train)
        {
            System.out.println("bookticket is locked by train");
            try
            {
                Thread.sleep(150);
            }
            catch(InterruptedException e)
            {
            }
            System.out.println("bookticket is waiting to lock on compartment");
            synchronized(comp)
            {
                System.out.println("bookticket locked on compartment");
            }
        }
    }
}
class CancelTicket extends Thread
{
    Object train,comp;
    CancelTicket(Object comp ,Object train )
    {
        this.comp=comp;
        this.train=train;
    }
    public void run()
    {
        synchronized(comp)
        {
            System.out.println("cancel ticket locked on compartment");
        }
    }
}
```

```

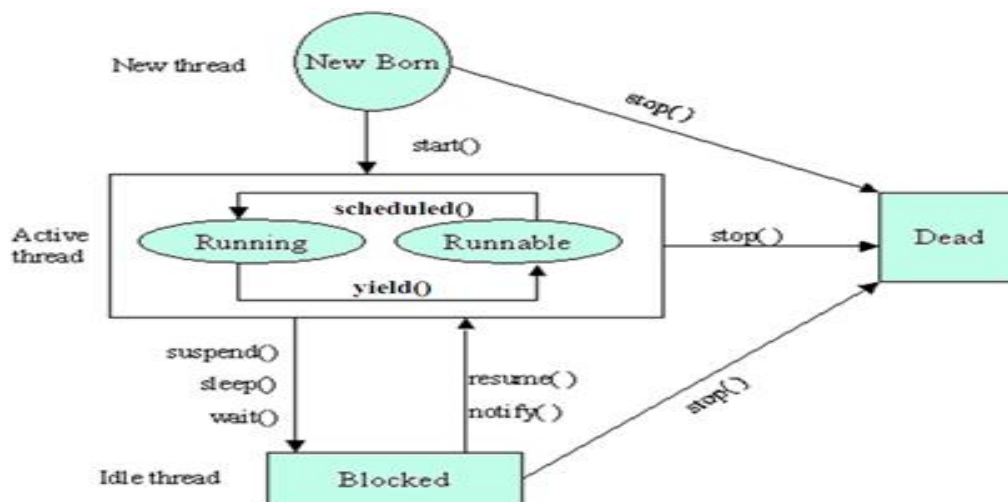
        try
        {
            Thread.sleep(200);
        }
        catch(InterruptedException e)
        {
        }
        System.out.println("cancel ticket waiting for train ticket object to lock");
        synchronized(train)
        {
            System.out.println("cancel ticket locked on train");
        }
    }
}
}
class ThreadDeadlock
{
    public static void main(String arg[])
    {
        Object train = new Object();
        Object comp = new Object();
        BookTicket obj1 = new BookTicket(train,comp);
        CancelTicket obj2 = new CancelTicket(comp,train);
        Thread t1=new Thread(obj1);
        Thread t2=new Thread(obj2);
        t1.start();
        t2.start();
    }
}

```

Extra topic:

Explain the Thread Life Cycle:

A thread can be in any of the five following states:



1. Newborn State: When a thread object is created a new thread is born and said to be in Newborn state.

2. Runnable State: If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion.

3. Running State: It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occur.

a. Thread give up its control on its own and it can happen in the following situations.

- A thread gets suspended using **suspend ()** method which can only be revived with **resume()** method.
- A thread is made to sleep for a specified period of time using **sleep(time)** method, where time is in milliseconds.
- A thread is made to wait for some event to occur using **wait ()** method. In this case a thread can be scheduled to run again using **notify()** method.

b. A thread is pre-empted by a higher priority thread.

4. Blocked State: If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.

5. Dead State: A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.

4.8 Explain Inter Thread Communication

In some cases, 2 or more threads should communicate with each other. For example consider the **classic queuing problem**, where one thread is producing some data and another is consuming it.

Here, **polling is usually implemented** by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.

In polling the producer has to wait until the consumer is finished before it generates more data. The consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling wasting more CPU cycles waiting for the consumer to finish, and so on.

In this way, the producer and consumer can communicate with each other. But this is not an efficient way of communication. **To avoid polling, java includes an elegant inter-process communication mechanism via wait(),notify(),notifyAll() methods.**

1. **obj.notify():**- this method releases an object and sends a notification to a waiting thread that the object is available.
2. **obj.notifyAll():**- this method is useful to send notifications to all waiting threads that the object is available.
3. **obj.wait():**- this method makes a thread wait for the object till it receives a notification from notify() or notifyAll() methods.

Note: these methods are used inside the synchronized blocks

```
//this program shows how to use wait, notify methods and this is the most efficient way of thread
//communication
import java.io.*;
class Producer extends Thread
{
    StringBuffer sb;
    Producer()
    {
        sb=new StringBuffer();
    }
}
```

```

    }
    public void run()
    {
        synchronized(sb)
        {
            for(int i=0;i<10;i++)
            {
                try
                {
                    sb.append(i+":");
                    Thread.sleep(100);
                    System.out.println("appending");
                }
                catch(Exception e)
                {
                }
            }
            sb.notify();
        }
    }
}
class Consumer extends Thread
{
    Producer prob;
    Consumer(Producer prob)
    {
        this.prob=prob;
    }
    public void run()
    {
        synchronized(prob.sb)
        {
            try
            {
                prob.sb.wait();
            }
            catch(Exception e)
            {
            }
        }
        System.out.println(prob.sb);
    }
}
class Communication
{
    public static void main(String args[])
    {
        Producer obj1=new Producer();
        Consumer obj2=new Consumer(obj1);
        Thread t1=new Thread(obj1);
    }
}

```

```

        Thread t2=new Thread(obj2);
        t2.start();
        t1.start();
    }
}

```

```

C:\> Command Prompt
E:\267 java lab>javac Communication.java
E:\267 java lab>java Communication
appending
appending
appending
appending
appending
appending
appending
appending
appending
appending
0:1:2:3:4:5:6:7:8:9:

```

4.7: Describe `isAlive()`, `join()`, `suspend()` and `resume()` methods:

→ **`isAlive()`**: this method tests whether the thread is alive or not. Following is the declaration of `isAlive()` method :

```
public final boolean isAlive();
```

This method returns true if this thread is alive, otherwise returns false.

→ **`join()`**: this method is used to wait for a thread to terminate. Its name comes from the concept of the calling thread waiting until the specified thread joins it. The general form is:

```
final void join();
```

`join()` allows you to specify a maximum amount of time that we want to wait for the specified thread to terminate.

→ **`suspend()`**: this method is useful to pause the execution of a thread. The general form is:

```
final void suspend();
```

→ **`resume()`**: this method is used to restart the execution of a thread. The general form is:

```
final void resume ();
```

```

class MyThread extends Thread {
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
                System.out.println(Thread.currentThread().getName()+":"+i);
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {
            System.out.println(Thread.currentThread().getName()+"interrupted.");
        }
        System.out.println(Thread.currentThread().getName()+"exiting.");
    }
}

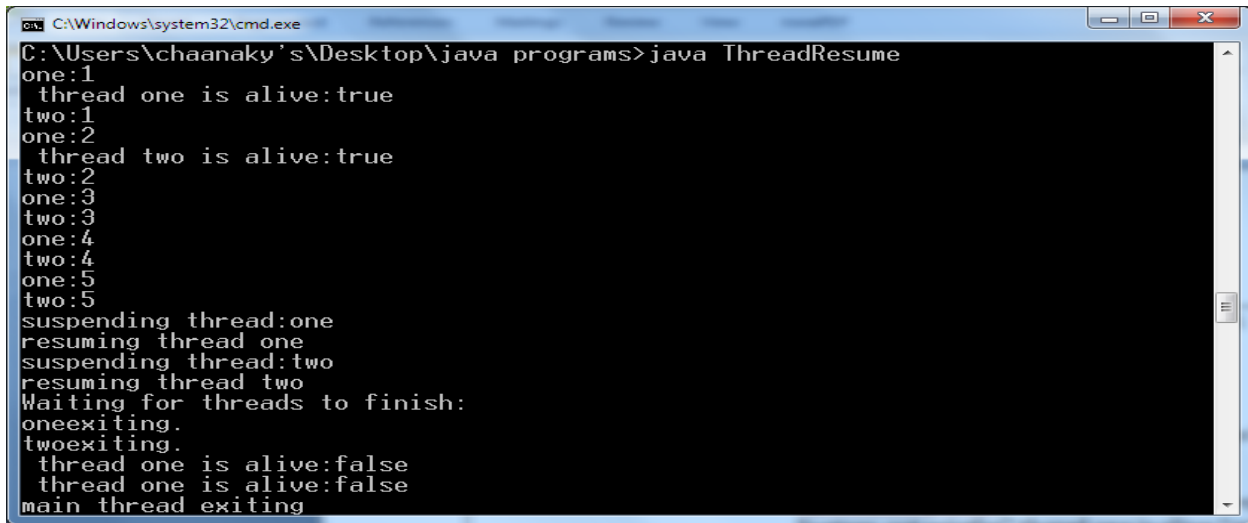
```

```

class ThreadResume
{
    public static void main(String x[])
    {
        MyThread obj1=new MyThread();
        MyThread obj2=new MyThread();
        Thread t1=new Thread(obj1,"one");
        Thread t2=new Thread(obj2,"two");
        t1.start();
        t2.start();
        System.out.println(" thread one is alive:"+t1.isAlive());
        System.out.println(" thread two is alive:"+t2.isAlive());
        try
        {
            t1.sleep(100);
            t1.suspend();
            System.out.println("suspending thread:"+t1.getName());
            t1.sleep(100);
            t1.resume();
            System.out.println("resuming thread one");
            t2.sleep(100);
            t2.suspend();
            System.out.println("suspending thread:"+t2.getName());
            t2.sleep(100);
            t2.resume();
            System.out.println("resuming thread two");
        }
        catch(InterruptedException ie)
        {
            System.out.println("main thread interrupted.");
        }

        try
        {
            System.out.println("Waiting for threads to finish:");
            t1.join();
            t2.join();
            System.out.println(" thread one is alive:"+t1.isAlive());
            System.out.println(" thread two is alive:"+t2.isAlive());
        }
        catch(InterruptedException ie)
        {
            System.out.println("main thread interrupted.");
        }
        System.out.println("main thread exiting");
    }
}

```



```
C:\Windows\system32\cmd.exe
C:\Users\chaanaky's\Desktop\java programs>java ThreadResume
one:1
thread one is alive:true
two:1
one:2
thread two is alive:true
two:2
one:3
two:3
one:4
two:4
one:5
two:5
suspending thread:one
resuming thread one
suspending thread:two
resuming thread two
Waiting for threads to finish:
oneexiting.
twoexiting.
thread one is alive:false
thread one is alive:false
main thread exiting
```

Extra topics in Threads:

Advantages Or Usage Of Threads?

With a thread package, a programmer can create several threads within a process. Threads execute concurrently, and within a multithreaded process, there are multiple points of execution at any time. Threads execute within a single address space. Multithreaded programming offers the following advantages:

Performance: Threads improve the performance (throughput, computational speed, responsiveness, or some combination) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors.

You cannot make any assumptions about the start or finish times of threads or the sequence in which they execute, unless they are explicitly synchronized.

Shared Resources: When you use multiple threads instead of separate processes multiple threads share a single address space, all open files, and other resources.

Potential Simplicity: Multiple threads may reduce the complexity of some applications that are inherently suited for threads.

Explain thread properties:

Thread class:As we know already, java provides two ways of creating threads.

- By extending thread class
- By implementing the runnable interface

Thread class is defined in java.lang package

The thread class provides

- Constructors
- Methods
- fields

Constructors of thread class:

Thread(): allocates a new thread object.

Thread(string name): creates a new thread object with a given name.

- **Thread(runnable target):**creates a thread object for a given a runnable object .
- **Thread (runnable thread target, string name):** creates a thread object for a given a runnable object .and give a name to the thread.

Methods of thread class:

- **currentThread ()**:Returns a reference to the currently executing thread object.
- **yield ()**:Causes the runtime to context switch from current thread to next available thread.
- **sleep (int n)**:Current thread sleeps for n milliseconds.
- **start ()**: Used to start a thread. Run () will be called
- **run ()**: It is the body of the running thread to be overridden.
- **stop ()**: Causes the thread to stop immediately.
- **setName(string name)**:Sets the name for the current thread.
- **getName()**:Returns the name of the current thread
- **activeCount()**:Returns the no of active threads in the current threads groups.
- **getThreadGroup()**:Returns the thread group to which this thread belongs.
- **getState()**:Returns the state of the thread
- **getPriority()**:obtains a thread 's priority.
- **isAlive()**:determine if a thread is still running.
- **join()**:wait for a thread to terminate.

Fields of the threads class:

These are the fields of the thread classes, and they are

- 1.Thread.MAX_PRIORITY
- 2.Thread.NORM_PRIORITY
- 3.Thread.MIN_PRIORITY

Exception Handling Concepts

4.10 Explain the Source Of Errors

SOURCE OF ERRORS: Errors are the wrongs that can make a program wrong. Errors may be classified into 2 categories:-

- i) Compile time errors
- ii) Run time errors
- iii) Logical Errors

COMPILE TIME ERRORS:

- These are **syntactical errors found in the code**, due to which a program fails to compile.
- Whenever the compiler displays errors, it will not create the .class file. It is necessary that we correct all the errors before we can successfully compile and run program.

The most common compile time errors are:

- Missing semicolons
- Missing (or) mismatch brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double codes in strings
- Use of undeclared variables
- Incompatible type in assignments (or) initialization
- Use of equal to in place of double equal to operator etc.

RUN TIME ERRORS: These errors **represent inefficiency of the computer system to execute a particular statement.**

Sometimes a program may compile successfully creating the .class file but may not run properly.

Run-time errors are not detected by the Java compiler. They are detected by the JVM, only at run-time.

Most common run time errors are:

- Dividing an integer by zero
- Accessing an element i.e. out of the bounds of an array
- Trying to store a value into an array of incompatible class or type
- Passing a parameter i.e. is not in a valid range (or) value for a method.
- Trying to illegally change the state of thread.
- Attempting to use a negative size for an array.
- Converting invalid string to a number
- Accessing a character i.e. out of bounds of string

LOGICAL ERRORS: These errors **explain flaws in the logic of the program.**

e.g.: The programmer might be using a wrong formula or the design of the program itself is wrong. Logical errors are not detected either by Java compiler or JVM. The programmer himself is responsible for them.

4.11 Write the Advantages Of Exceptions:

1. Separating error handling code from regular code.
2. It prevents the program for automatic termination.
3. Grouping error types and error differentiation.

4.12 Explain How To Deal With Exceptions:

EXCEPTION: An exception is a run time error in the program.

EXCEPTION HANDLING: When, the java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it i.e. informs an error has occurred.

If the exception object is not caught and handle properly, the interpreter will display an error message. If we want the program to continue with the execution of the program to continue with the remaining then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective action. This task is known as **exception handling.**

Java exception handling is managed by using the keywords. Those are

1. *Try*
2. *Catch*
3. *Throw*
4. *Throws*
5. *finally*

TRY AND CATCH KEYWORDS:

TRY BLOCK: The programmer should observe the statements in his program, where there may be a possibility of exceptions. Such statements should be written inside a try block.

Syntax: try

```
{  
    //statements  
}
```

When JVM understands that there is an exception, in the try block, then **it stores the exception details in an exception stack and then jumps into a catch block.**

CATCH BLOCK: Immediately following the try block includes catch where we should display the exception details to the user.

Syntax:

```
catch (ExceptionType variable)
{
    // stmts to be displayed when exception occurred
}
```

A try block and its catch statement form a unit. Remember that every try statement should be followed by at least one catch statement otherwise compile time error will occur.

FINALLY:

Finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. Cleanup operations code is written inside the 'finally' block.

The finally block will be executed whether or not an exception is thrown. The finally clause is optional.

Syntax:

```
finally
{
    //Block of statements which releases the resources;
}
```

EXAMPLE: The following program includes a try block and a catch block which process the Arithmetic Exception generated by the division by zero error.

```
class A
{
    public static void main (String args[])
    {
        int d=0;
        try
        {
            int a=42/d;
            System.out.println("not printed");
        }
        catch (ArithmeticException e )
        {
            System.out.println("division by zero not possible");
            System.out.println("after try/catch block");
        }
        finally
        {
            System.out.println(" I am finally block..");
        }
    }
}
```

OUTPUT:

```
division by zero not possible
after try/catch block
I am from finally block
```

THROW:

In general, we are catching exceptions that are thrown by java runtime system. **However, is possible for our program throw an exception explicitly, using the throw statement.**

The general form of throw is here

Syntax: throw throwableinstance;

Here, throwable instance must be an object of type Throwable class

Ex: in the following program, we are creating an object of NullPointerException class and throwing it out of try block as shown here:

Example: throw new NullPointerException("exception data");

In the above statement NullPointerException class is created and 'exception data' is stored into its object. Then it is thrown using throw statement now, we can catch it using catch block as,

```
catch(NullPointerException ne)
{
}
```

Example:

```
class Throwdemo
{
    static void demo()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch (NullPointerException ne)
        {
            System.out.println("caught inside demo");
        }
    }
    public static void main(String args [])
    {
        demo();
    }
}
```

OUTPUT: caught inside demo

THROWS:

A throws clause lists the types of exception that a method may throw. All exception that a method can throw must be declared in the throws clause.

The general form at a method declaration that includes a throws clause is

Syntax:

```
type method_name(parameters list) throws Exception list
{
    Body of method;
}
```

Here, Exception list is a comma separated list of the exceptions that a method can throws.

Example:

```
class Throwsdemo
{
    static void divide() throws ArithmeticException
    {
        int x=22,y=0,z;
        z=x/y;
    }

    public static void main(String args[])
    {
        try
        {
            divide();
        }
        catch(ArithmeticException e)
        {
            System.out.println("caught"+e);
        }
    }
}
```

OUTPUT: caught java.lang. ArithmeticException: / by zero

4.13 Explain The Concept Of Multi-Catch Statements Programs:

Multi catch statements:

Most of the times there is possibility of **more than one exception present in the program**. In this case, the programmer **should write multiple catch blocks to handle each one of them**.

In this program, there are 2 – exceptions ArithmeticException and ArrayIndexOutOfBoundsException. These exceptions are handled with the help of 2 catch blocks.

Example:

```
class edemo {
    public static void main (String [] args)
    {
        try
        {
            int n=args.length;
            System.out.println("n=" +n);
            int a=45/n;
            System.out.println ("DIVISION VALUE = "+a);
            int b[]={10,20,30};
            b[50]=100;
        }
    }
}
```

```

catch(ArithmeticException Ae)
{
    System.out.println(Ae);
    System.out.println ("DONT ENTER ZERO FOR DENOMINATOR...");
}
catch(ArrayIndexOutOfBoundsException A)
{
    System.out.println(A);
    System.out.println ("ARRAY INDEX IS OUT OF RANGE..");
}
finally
{
    System.out.println ("I AM FROM FINALLY...");
}
}
}

```

Note: even if there is scope of multiple exceptions, only one exception at a time will occur.

NESTED TRY STATEMENTS:

The try statement can be nested. That is, a *try* statement can be inside the block of another *try*. There could be situations where there is possibility of generation of multiple exceptions of different types with in a particular block of the program code. We can use nested try statements in such situations. The execution of the corresponding catch blocks of nested try statements is done using a stack.

Syntax:

```

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

```

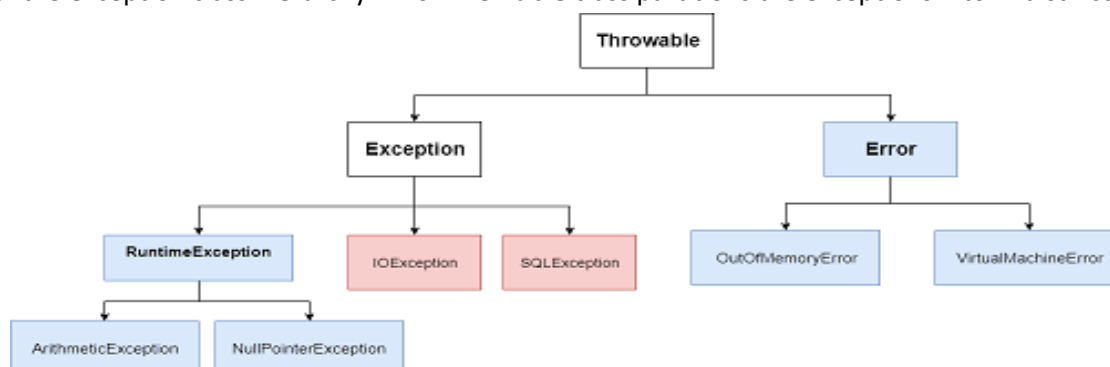
When nested try blocks are used, the inner try block is executed first. Any exception thrown in the inner try block is caught in the corresponding catch block. If a matching catch block is not found, then catch block of the outer try block are inspected until all nested try statements are exhausted. If no matching blocks are found, the Java Runtime Environment handles the execution.

//The following example illustrates how to use nested try blocks:

```
class Nestedtryex
{
    public static void main(String args[])
    {
        try
        {
            String s1=args[0];
            String s2=args[1];
            int n1=Integer.parseInt(s1);
            int n2=Integer.parseInt(s2);
            try
            {
                int n3=n1/n2;
                System.out.println("division value:"+n3);
            }
            catch(ArithmeticException ae)
            {
                System.out.println ("DONT ENTER ZERO FOR DENOMINATOR...");
            }
        }
        catch(ArrayIndexOutOfBoundsException aioobe)
        {
            System.out.println ("PASS ONLY TWO ARGUMENTS...");
        }
        catch(NumberFormatException nfe)
        {
            System.out.println ("PASS ONLY INTEGER VALUES...");
        }
    }
}
```

4.14 Explain The Types Of Exceptions:

Hierarchy of Exception types: all exception types are sub classes of built in class **Throwable**. Thus, **Throwable** class is at the top of the exception class hierarchy. This **Throwable** class partitions the exceptions into –2 distinct branches:



Exception class: this class is **used for exceptional conditions that user programs should catch**. We can also create our own custom exception types below this class among those these is an important sub class called 'runtime exception '.

Exceptions of this type are **automatically handled** examples are **division by zero, invalid array indexing.....**

Error class: this class **defines exceptions that are not expected to caught under normal circumstances by our program**.

Exceptions of type error are used by the java run time system to indicate errors and can be caught by java run time itself.

Ex: stack overflow

➤ **Checked Exceptions:** Checked exceptions are checked at compile-time.

It is named as **checked exception** because these exceptions are **checked** at Compile time. Checked exceptions are extended from the **java.lang.Exception** class.

➤ **Unchecked Exceptions:** The exceptions that are checked at runtime by the JVM. Unchecked exceptions are extended from the **java.lang.RuntimeException** class.

Types of Exceptions: Following are the exceptions available in Java:

1. **Built-in Exceptions**

2. **User-defined Exceptions**

Built-in Exceptions: Built-in Exceptions are the **exceptions which are already available in Java**. These exceptions are suitable to explain certain error situations. Table 21.1 lists the important Built-in Exceptions.

Exception class	Meaning
ArithmeticException	Thrown when an exceptional condition has occurred in an arithmetic operation.
ArrayIndexOutOfBoundsException	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.
ClassNotFoundException	This exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	Raised when a file is not accessible or does not open.
IOException	Thrown when an input-output operation failed or interrupted.
InterruptedException	Thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
NoSuchFieldException	Thrown when a class does not contain the field for (or variable) specified.
NoSuchMethodException	Thrown when accessing a method which is not found.
NullPointerException	Raised when referring to the members of a null object, null represents nothing.
NumberFormatException	Raised when a method could not convert a string into a numeric format.
RuntimeException	This represents any exception which occurs during runtime.
StringIndexOutOfBoundsException	Thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

Example of Built-in Exception: Refer the 4.13 topic example program.

User-defined Exceptions: sometimes, the built-in exceptions in java are not able to describe a certain situations. In such situations, like built-in exceptions, the user can also create his own exceptions which are called “user-defined” exceptions. The following steps are followed in creation of user-defined exceptions.

1. The user should create an exception class that extends Exception class.

class MyException extends Exception

2. Create either default constructor or parameterized constructor to store exception details. In this constructor call super class i.e., Exception constructor from this & send the string there.

<pre>MyException() { } (or) { }</pre>	<pre>MyException(String str) { super(str); }</pre>
----------------------------------------------------------------------	--------------------------------------------------------------------

Default constructor does not store any exception details.

3. When the user wants to raise his own exception, he should create an object to his exception class & throw it using throw clause.

e.g.:

MyException me=new MyException(“Exception details”);

```
//User-defined Exceptions to throw whenever balance is below 1000/-  
class MyException extends Exception  
{  
    private static int accno[]={ 101,102,103,104,105};  
    private static String name[]={ "subramanyam","neeraja","nani","sai","mahesh"};  
    private static double bal[]={ 10000,15000,5600.50,999.50,4000.00};  
    MyException() { }  
    MyException(String str)  
    {  
        super(str);  
    }  
    public static void main(String agrs[])  
    {  
        try  
        {  
            System.out.println("ACCONT NO "+"CUSTOMER NAME "+"BALANCE");  
            for(int i=0;i<5;i++)  
            {  
                System.out.println(accno[i]+"\\t"+name[i]+"\\t"+bal[i]);  
                if(bal[i]<1000)  
                {  
                    MyException me=new MyException("balance is less than 1000 at :"+accno[i]);  
                    throw me;  
                }  
            }  
        }  
        catch(MyException me)  
        {  
            me.printStackTrace();  
        }  
    }  
}
```