UNIT-4

KNOW MULTI THREADED PROGRAMMING AND EXCEPTION HANDLING

4.1 Explain The Thread Model Of Java

Thread Model of java:

A program is divided into two or more sub programs, which can be implemented in parallel. Each of this sub program is called a thread. Thus a thread is similar to a program that has a single flow of control. It has a beginning, a body and executes commands sequentially.

A Unique property of java is its support for multithreading i.e. java enables us to use multiple flows of control in developing programs. Each of control may be thought of as a separate tiny program known as thread, which runs in parallel to others. A program that contains multiple flows of control is known as multithreaded program.

Multithreading is a specialized form of multitasking. Multitasking threads required less overhead than multitasking processes. Processes are heavy weight task that required their own address separate spaces. On the other hand **threads are light weight** and **share the same address space.**

The advantage of multithreading is, when a thread blocks in a java program only the single thread is blocked pauses. All other threads continue to run.

Threads exist in several states:

- A thread can be **running** that the processor has given its time to the thread for its execution.
- A running thread can be **suspended**, which temporarily suspends it's activity.
- A suspended thread can be **resumed**, allowing it to pick up where it left off.
- A thread can be **blocked**, when waiting for a resource.
- At any time a thread can be **terminated**, which halts it's; execution immediately.
- Once terminated a thread cannot be resumed.

Differences between Multithreading and Multitasking

SNO	Multithreading	Multitasking
1	It is a programming concept in which a program or a process is divided into two or more sub programs or threads that are executed at the same time in parallel.	It is an operating system concept in which multiple tasks are performed simultaneously
2	It support execution of multiple parts of a single program simultaneously	It support execution of multiple programs simultaneously
3	The processor has to switch between different parts or threads of a program	The processor has to switch between different programs or processes
4	It is cost –effective in case of context switching	It is expensive in case of context switching
5	Light-weight process	Heavy weight process

4.2 Explain Thread Priorities

Every thread in java is assigned a priority value, which affects the order in which it is scheduled for running. Generally, highest priority thread is running before the lowest priority thread. The threads of the same priority are given equal treatment by the java scheduler and, therefore, execute on a FCFS basis.

Java permits us to set the priority of a thread using the **setPriority()** method as follows:

Threadname.setPriority(int number);

The **int number** is an integer value to which the thread's priority is set. The **Thread** class defines several priority constants:

1) **Thread. MIN_PRIORITY—1**
2) **Thread . MAX_PRIORITY—10**
3) **Thread. NORM_PRIORITY—5**

The DEFAULT PRIORITY is **Thread .NORM_PRIORITY**

whenever multiple threads are ready for execution, the java system chooses the highest priority threads and executes it , for a thread, of lower priority to given control, one of the following things should happen:

1) It stops running at the end of run().
2) It is made to sleep using sleep().
3) It is told to wait using wait().

Example1:

```
class Priorities extends Thread
{
        public void run( )
        {
        System.out.println("thread is running…"+Thread.currentThread().getName());
        System.out.println("thread priority is "+Thread.currentThread().getPriority());
        }
}
Class PriorityDemo
{
public static void main(String args[])
{
       Priorities ob=new Priorities();
       Thread t1=new Thread(ob);
       Thread t2=new Thread(ob);
       t1.setName("T1");
       t2.setName("T2");
       t1.setPriority(2);
       t2.setPriority(8);
       t1.start();
       t2.start();
}
}
```

```
C:\Users\SIVA\Desktop\java>java PriorityDemo
thread is running?T2
thread priority is8
thread is running?T1
thread priority is2
```

Example 2:

```
  class A extends Thread
    {
        public void run()
        {
```

```
            System.out.println("thread is started");
            for(int i=1;i<=5;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<=5;j++)
            {
                System.out.println("from thread B:j="+j");
            }
            System.out.println("exist from B");
    }
}
class C extends Thread
{
    public void run()
    {
            System.out.println("thread C started ");
            for (int k=1;k<=5;k++)
            {
                System.out.println("from thread C: K ="+K);
            }
            System.out.println("exist from C");
    }
}
class ThreadPriority
{
    public static void main(String args[])
    {
            A threadA=new A();
            B threadB=new B();
            C threadC=new C();
            threadC.setPriority (Thread.MAX_PRIORITY);
            threadB.setPriority(threadA.getPriority()+1);
            threadA.setPriority(Thread.MIN_PRIORITY);
            System.out.println("start thread A");
            threadA.start();
            System.out.println("start thread B");
            threadB.start();
```

```
            System.out.println("startthreadC");
            threadC.start();
            System.out.println("End of main thread");
        }
    }
```

4.3 Explain The Concept Of Synchronization

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. It is like a room with only one door. A person who entered the room locks it. The second person who wants to enter the room should wait until the first person comes 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)

We can synchronize our code in either of two ways.

Using Synchronized keyword:

We can synchronize an entire method by using synchronized keyword.

When we declare a method as Synchronized, Java creates a monitor and hands it over to the thread that calls the method first time. As long as the thread holds the monitor no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock.

```
Synchronized void display()
{
------          // code here is synchronized
----
}
```

Sample program:

```
class A
{
    synchronized void multi(int i)
    {
        for(int n=1;n<=5;n++)
        {
            System.out.println(Thread.currentThread().getName()+":"+(i*n));
        }
    }
}
class Example extends Thread
{
    A ob=new A();
    public void run()
    {
      ob.multi(100);
```

```
    }
}
class ThreadSynch
{
  public static void main(String args[])
   {
        Example ob=new Example();
        Thread t1=new Thread(ob);
        Thread t2=new Thread(ob);
        t1.setName("T1");
        t2.setName("T2");
        t1.start();
        t2.start();
    }
}
```

```
C:\Users\SIVA\Desktop\java>javac ThreadSynch.java

C:\Users\SIVA\Desktop\java>java ThreadSynch
T1:100
T1:200
T1:300
T1:400
T1:500
T2:100
T2:200
T2:300
T2:400
T2:500
```

Using synchronized Block:

Here we can embed a group of statements of the object (inside run()) within a synchronized block. Here object represents the object to be locked or synchronized.

```
Synchronized(object)
{
statements to be synchronized;
}
```

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

Example

```
import java.util.*;
class Reservation implements Runnable
{
          int avail=10;
          int wanted;
          Scanner s=new Scanner(System.in);
           public void run()
           {
              synchronized(this)
              {
                  System.out.println("Available berths are="+avail);
```

```
                System.out.println("Enter how many berths you want");
                wanted=s.nextInt();
                if(avail>=wanted)
                {
                  String name=Thread.currentThread().getName();
                  System.out.println(wanted+"berths are reserved for"+name);
                  avail=avail-wanted;
                 }
               else
                System.out.println("sorry berths are not available");
            }
        System.out.println("thank you and happy journey");
         }
}
class ThreadSyncDemo
{
 public static void main(String args[])
 {
  Reservation obj=new Reservation();
  Thread t1=new Thread(obj);
  Thread t2=new Thread(obj);
  t1.setName("passenger1");
  t2.setName("passenger2");
 t1.start();
 t2.start();
 }
}
```

```
C:\Users\SIVA\Desktop\java>javac ThreadSyncDemo.java

C:\Users\SIVA\Desktop\java>java ThreadSyncDemo
Available berths are=10
Enter how many berths you want
2
2berths are reserved forpassenger1
thank you and happy journey
Available berths are=8
Enter how many berths you want
4
4berths are reserved forpassenger2
thank you and happy journey
```

4.4 Create thread using Thread Class

CREATING THREADS: Creating threads in java is simple. In most general we can create a thread by instating a object of type '**Thread**'. which is accomplished into two ways:

1) By extending thread class itself.

2) By Implementing Runnable interface.

By extending thread class itself:

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:

- ✓ Declare the class as extending the Thread class.
- ✓ Implement the **run()** method that is responsible for executing the sequence of code that the thread will execute.

- ✓ Create a Thread object and **call the start()** method to initiate the thread execution.

Example:

```
class Sample extends Thread
{
        public void run()
        {
        System.out.println("thread is running");
        }
}
class ThreadDemo1
{
     public static void main(String args[])
     {
       Sample ob=new Sample();
       ob.start();
     }
}
```

```
C:\Users\SIVA\Desktop\java>javac ThreadDemo1.java

C:\Users\SIVA\Desktop\java>java ThreadDemo1
thread is running
```

Note:To create a new thread, our program will either "extends" Thread class (or) implementing the Runnable interface.

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.**

1.Declare the thread class as implementing the runnable interface

2.Implements the run () method

3.Create a thread by defining an object that is instantiated from this "runnable " class as the target of the thread

4.Call the threads start () method to run the thread

Example:

```
class Sample implements Runnable
{
        public void run()
        {
        System.out.println("thread is running");
        }
}
class RunnableDemo
{
     public static void main(String args[])
     {
       Sample ob=new Sample();
```

```
            Thread t=new Thread(ob);
            ob.start(ob);
        }
    }
```

```
C:\Users\SIVA\Desktop\java>javac ThreadDemo2.java

C:\Users\SIVA\Desktop\java>java ThreadDemo2
thread is running
```

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.

- ✓ **We can create multiple threads to perform single task.**
- ✓ **We can create multiple threads to perform multiple tasks.**

Example program multiple threads to perform single task:

```
class SingleTask extends Thread
{
    public void run()
    {
    System.out.println("thread running");
    }
}
class MultiDemo
{
     public static void main(String args[])
     {
     SingleTask t1=new SingleTask();
     SingleTask t2=new SingleTask();
     t1.start();
     t2.start();
     }
}
```

```
C:\Users\SIVA\Desktop\java>javac MultiDemo.java

C:\Users\SIVA\Desktop\java>java MultiDemo
thread running
thread running
```

Example program on multiple threads to perform multiple tasks.

```
class A extends Thread
{
    public void run()
    {
     for(int i=1;i<=3;i++)
        System.out.println("thread of A is running..");
    }
}
```

```
class  B extends Thread
{
     public void run()
     {
      for(int i=1;i<=3;i++)
        System.out.println("thread of B is running..");
     }
}
class C extends Thread
{
     public void run()
     {
      for(int i=1;i<=3;i++)
       System.out.println("thread of c is running..");
     }
}
class MultiDemo1
{
   public static void main(String args[])
   {
      A t1=new A();
      B t2=new B();
      C t3=new C();
      t1.start();
      t2.start();
      t3.start();
   }
}
```

```
C:\Users\SIVA\Desktop\java>javac MultiDemo1.java

C:\Users\SIVA\Desktop\java>java MultiDemo1
thread of A is running..
thread of A is running..
thread of A is running..
thread of c is running..
thread of c is running..
thread of c is running..
thread of B is running..
thread of B is running..
thread of B is running..
```

4.7 Understand Alive (), Join (), Suspend (), Resume () methods

The thread class has several methods. Here we will discuss about the following thread methods isAlive(), join(), suspend(), resume().

isAlive (): This method tests whether the thread is alive or not.

public final Boolean isAlive();

The isAlive() method returns true, if the thread upon which it is called is still running, otherwise it returns false. isAlive() method is occasionally useful.

join(): This method used to wait for a thread to terminate. The calling thread will simply wait until the thread it is joining with either completes the task or is not alive. The general form is

final void join() throws InterruptedException;

Suspend(): This method is used to temporarily stop the execution of the thread. When the code encounters suspend() method, it saves all the states and resources of the thread. Such a suspended thread can be restarted by another thread calling the resume() method. This method can lead to deadlocks and therefore it should be avoid.The general form of suspend() method is

final void suspend();

Resume(): This method resumes the execution of a suspended thread. After executing this method, if the suspended thread has a higher priority than the running thread, the running thread method will be preempted; otherwise this resumed thread will wait in queue for its turn to run. Like suspend () method the resume() method can lead to deadlocks and therefore it should be avoided. The general form of resume() is **final void resume();**

Example program for join() and isAlive():

```
class TestJoinMethod2 extends Thread
{
 public void run()
{
  for(int i=1;i<=5;i++)
{
   try
{
    Thread.sleep(500);
   }
catch(Exception e)
{
System.out.println(e);
}
  System.out.println(Thread.currentThread().getName()+" :"+i);
  }
 }

}
class TestJoin123
{
public static void main(String args[])
{
 TestJoinMethod2 t1=new TestJoinMethod2();
 TestJoinMethod2 t2=new TestJoinMethod2();
 TestJoinMethod2 t3=new TestJoinMethod2();
 t1.start();
System.out.println("Thread one"+t1.isAlive());
System.out.println("Thread two"+t2.isAlive());
 try
{
  t1.join();
 }
```

```
catch(Exception e)
{
System.out.println(e);
}
t2.start();
t3.start();
System.out.println("Thread one"+t1.isAlive());
System.out.println("Thread two"+t2.isAlive());
System.out.println("Thread three"+t3.isAlive());
}
}
```

Example program for suspend() and resume()

```
class MyThread  extends Thread
{
        public void run()
        {
                try
                {
                        for(int i=1;i<=5;i++)
                        {
                                Thread.sleep(1000);
                        }
                }
                catch(InterruptedException  e)
                {
                        System.out.println(Thread.currentThread().getName+"interrupted");
                }
                System.out.println(Thread.currentThread().getName+":"+i);
        }
}
class ThreadResume
{
        public static void main(String  args[])
        {
        MyThread   t1=new MyThread ();
        MyThread   t2=new MyThread ()
                t1.start();
                t2.start();
                try
                {
                        Thread.sleep(1000);
                        t1.suspend();
                        System.out.println("thread one is suspended");
                        Thread.sleep(1000);
                        t1.resume();
                        System.out.println("thread one  is resumed");
                        t2.suspend();
```

```
                System.out.println("thread two is suspended");
                Thread.sleep(1000);
                t2.resume();
                System.out.println("thread two is resumed");
            }
            catch(InterruptedException e)
            {
                System.out.println("main thread interrupted");
            }
        }
}
```

4.8 Inter Thread Communication

Inter thread communication can be defined as the exchange of messages between two or more threads. The transfer of messages takes place before or after the change of state of a thread. For example, an active thread may notify to another suspended thread just before switching to the suspend state. Java implements inter-thread communication with the help of the following three methods:

notify(): This method releases an object and sends a notification to a waiting Thread that the object is available**.** The object class declaration of notify() method is shown below:

final void notify();

notifyAll(): This method releases an object and sends a notification to all waiting Thread that the object is available.The execution of these threads happens as per priority. The object class declaration of notify all() method is shown below:

final void notifyAll();

wait(): This method makes a Thread wait for the object till it receives a notification from notify() or notifyAll() methods. One can also specify the time for which thread has to wait. The desired waiting time period is specified as an argument to the wait () method. The object class declaration of wait () method is shown below

final void wait();

Since, the methods are declared final they cannot be overridden. All the three methods throw "**INTERRUPTEDEXCEPTION**"

Example program for interthread communication

```
class Customer
{
    int amount=10000;
    synchronized void withdraw(int amount)
    {
        System.out.println("going to withdraw...");
        if(this.amount<amount)
        {
        System.out.println("Less balance; waiting for deposit...");
        try
        {
        wait();
        }
```

```
          catch(Exception e)
          {
          }
        }
          this.amount  -=  amount;
          System.out.println("withdraw completed...");
  }
 synchronized void deposit(int amount)
 {
      System.out.println("going to deposit...");
      this.amount += amount;
      System.out.println("deposit completed... ");
      notify();
  }
}
class Test
{
         public static void main(String args[])
         {
              final Customer c=new Customer();
              new Thread()
              {
              public void run()
              {
                c.withdraw(15000);
              }
              }.start();
              new Thread()
              {
              public void run()
              {
                  c.deposit(10000);
              }
              }.start();

         }
}
```

4.9 Explain Dead Lock

DeadLock occurs when two Threads have a circular dependency on a pair of synchronized objects.

That is, when a Thread has locked on object and 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.

Ex: Assume that the thread A must access method1 before it can release method2, but the thread B cannot release method1 until it gets hold of method2. Because these are mutually exclusive conditions a deadlock occurs. The code below illustrates this:

```
Thread A
Synchronized method2()
{
Synchronized method1()
{
-------------
-------------
}
}
Thread B
Synchronized method1()
{
Synchronized method2()
{
-------------
-------------
}
}
```

Example:

```
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(1000);
                        }
                        catch (InterruptedException e)
                        {
                        }
                        System.out.println("book ticket is waiting to lock on compartment");
                        Synchronized(comp)
                        {
                                System.out.println("bookticket locked on compartment");
                        }
```

```
                }
        }
}
class  CancelTicket extends Thread
{
        Object train,comp;
        CancelTicket(Object train, Object comp)
        {
                this.train=train;
                this.comp=comp;
        }
        public void run()
        {
                Synchronized(comp)
                {
                        System.out.println("cancelticket is locked by compartment");

                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e)
                        {
                        }
                        System.out.println("cancel ticket is waiting for train ticket object to
                lock");
                        Synchronized(train)
                        {
                                System.out.println("cancelticket locked on train");
                        }
                }
        }
}
class DeadLock
{
        Public static void main(String args[])
        {
                Object train=new Object();
                Object comp=new Object();
                BookTicket obj1=new BookTicket(train,comp);
                CancelTicket obj2=new CancelTicket(train,comp);
                Thread t1=new Thread(obj1);
                Thread t1=new Thread(obj2);
                t1.start();
                t2.start();
        }
}
```

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 inter face

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

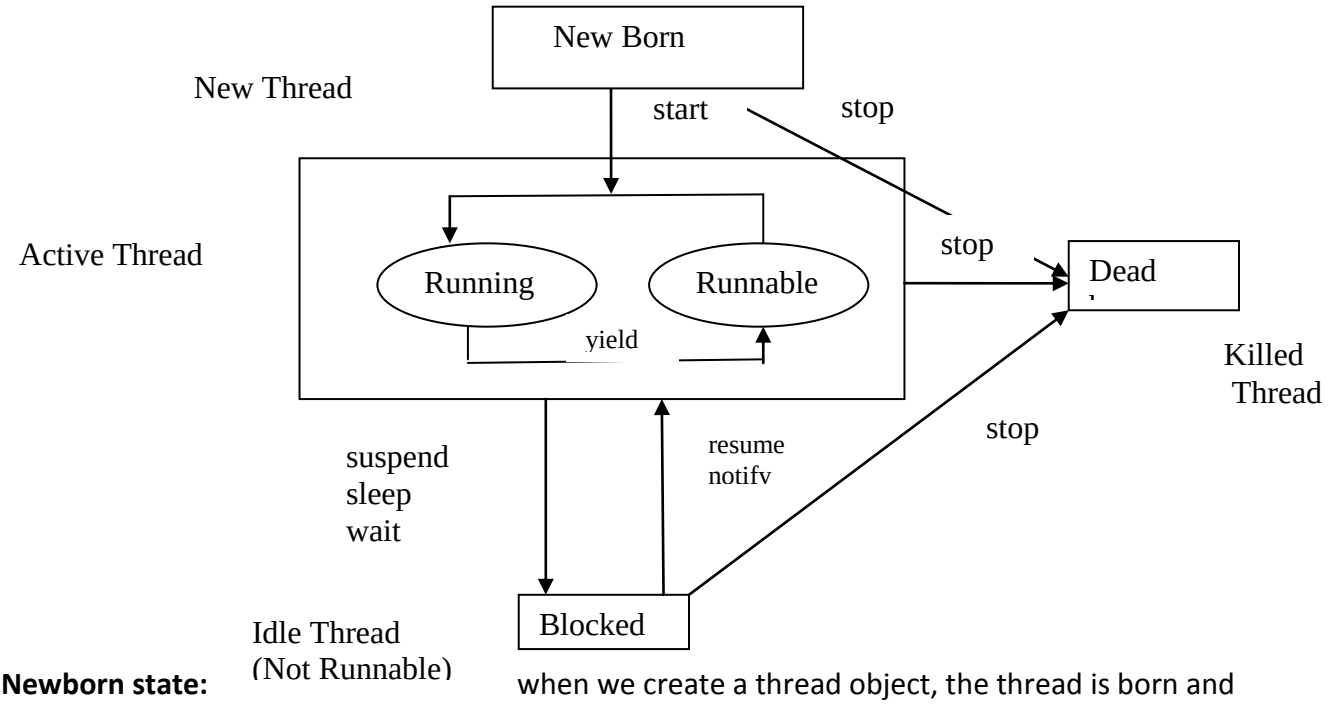
- 1.Thread.MAX_PPRIORITY
- 2.Thread.NORM_PRIORITY
- 3.Thread.MIN_PRIORITY

Life cycle of thread:

During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of five states. It can move from one state to another via a variety of ways as shown in figure.



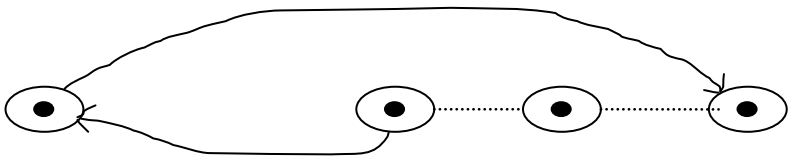
Newborn state: when we create a thread object, the thread is born and is said to be in newborn state. The thread is not at scheduled for running at this state we can do only one of the following things with it.

1. Schedule it for running using start() method.
2. Kill it using stop() method.

If scheduled, it moves to the runnable state.

Runnable state: The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. If all threads have equal priority then they are given time slots for execution in round robin fashion i.e. first come first serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as time – slicing.

yield



Running

Fig: Relinquishing control using yield() method

Running state: Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

1. It has been suspended using suspend() method. A suspended thread can be revived by using the resume() method. This approach is useful when we want to suspended a thread for some time due to certain reas suspend) not want to kill it.

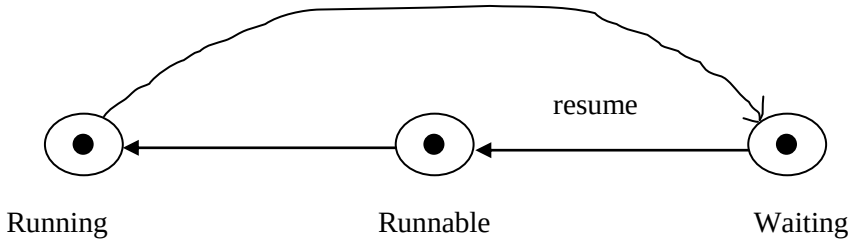
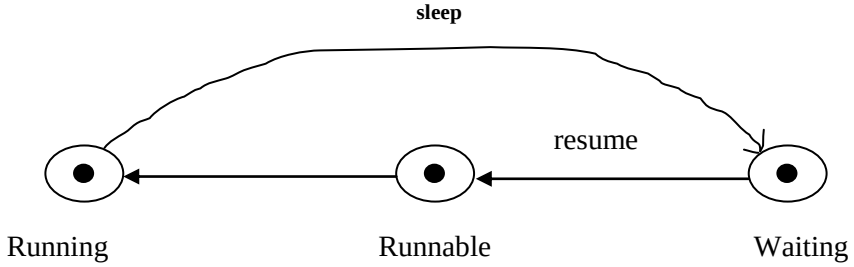


Fig: Relinquishing control using suspend() method

2. It has been made to sleep, we can put a thread to sleep for a specified time period using the method sleep(time) where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re –enters the runnable state as soon as this time period is elapsed.

Fig: Relinquishing control using sleep() method



It has been told to wait until some event occurs. This is done using the wait() method. The thread can be scheduled to run again using the notify() method.

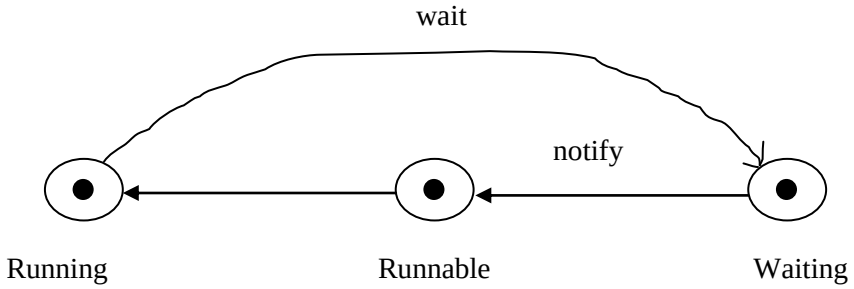


Fig: Relinquishing control using wait() method

Blocked state: A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is

suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.
Dead state: Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable"(blocked) condition.

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

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:

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:

In some cases more than one exception could be raised by a single piece of code. To handle this type of situation we can specify two or more catch classes, each catching a different type of exception. Then an exception is thrown each catch statement checked in order and one whose type matches that of the exception is executed. After one catch statement is executes, the other catch blocks are not checked.

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...");
}
}
}
```

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

Example:

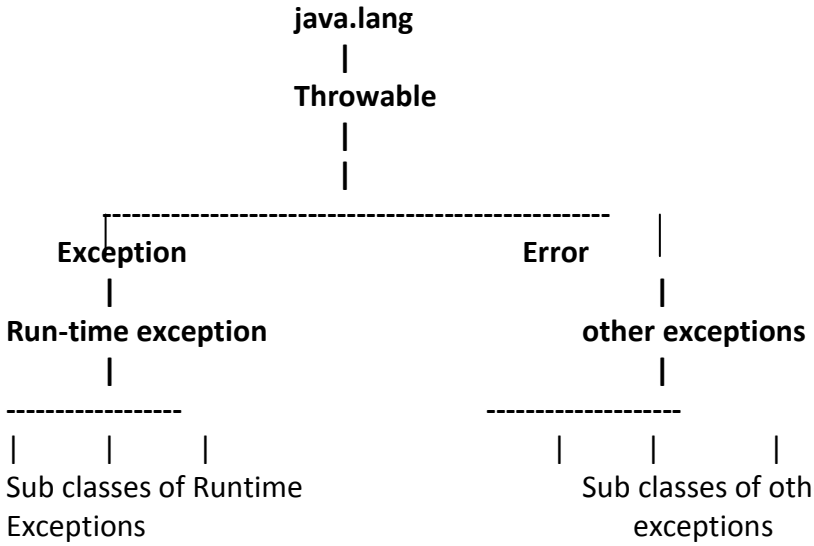
```
class Nestedtryex
{
        public static void main(String args[])
        {
                try
                {
                        String s1=args[0];
                        String s2=args[1];
                        int n1=Integer.parseInt(s1);
                        int n2=Integer.parseInt(s2);
                        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 –2distinct branches

1. **Exception**
2. **Error**



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.
ArrayIndexOutOfBoundException	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.

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.

```
MyException( )                    MyException(String str)
{                                 {
                  (or)                        super(str);
}                                 }
```

   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");
```