~~Unit-3~~ Unit-3

Concepts of Exception handling :-

An exception means a run time error that a problem occurred during program's execution. Exception handling means to handle the exceptions by the programmer to recover the computer from computer hanging due to exceptions.

In Java, exception handling is done through five keywords - try, catch, throw, throws & finally. To handle exceptions place the code that may generate an exception in a 'try' block. When exception is generated, the control leaves the try block & searches for the matching 'catch' block to handle the exception. Each 'catch' block specifies the type of exception it can handle.

eg: ArithmeticException, ArrayIndexOutOfBoundsException

If the exception generated in try block matches with one of the catch block, then the code of that catch block is executed. If there is no exception in try block, then all the catch blocks are not executed and then control goes to the statement after the last catch block. After last catch block there is a 'finally' block. It is optional. If it present, then the code present in it is compulsory executed, even though an exception is generated & not.

The general form of exception handling is -

```
try
{
        // block of code that may generate an exception
}
catch (ExceptionType obj1)
{
        // statements used to handle exception
}
```

```
catch ( Exception Type    obj2)
{
        // statements used for exception handling
}
 :
finally
{
        // statements to be executed compulsory
}
```

Some situations that may raise exceptions are -

1. attempting to open a non existing file
2. the class file to be loaded is missing
3. when we are dividing a number with zero
4. when we are accessing invalid index array element.

There are 2 types are of error -

1. Compile time errors :-

Compile time errors are the syntax errors that occur at the compilation of the program. These errors are detected & displayed by the compiler. They are also called bugs. The process of finding bugs is called debugging. When compile time error occurs, the programmer can correct them & then recompile program to get class file.

eg : missing paranthesis & semicolon
missing opening or closing quotations
use of variables without declarations
misspelling of identifiers & keywords
missing opening or closing braces.

2. Run time errors :-

Runtime errors are the errors that occur at runtime of program and so that the execution of program is terminated. The runtime errors are called exceptions.

There are 3 types of runtime errors. They are -

a. input errors - these errors occur if the user provides unexpected inputs to the program.
eg: if a program wants an integer, but the user provides string.

b. System errors - these errors occur due to unreliable system software.

c. logical errors - these errors occur if the program is logically incorrect.
eg:, These errors generates incorrect ~~ooo~~ result & program is terminated.
eg: in a program addition of two numbers requires '+' operator, but if user uses the '-' operator, then he can get incorrect result.

Benefits of exception handling :-

1. the try - catch blocks used in exception handling helps in separating the working & functional code from the error handling code.

2. it provides a clear path for the error to propagate. i.e when a called method cannot manage a situation, it throws an exception and asks the calling method to deal with that situation

3. it implements powerful coding.

Termination, Resumptive models :-

Termination model :- in this model, the programmer will have to explicitly invoke the same method in which the error has occurred and then it was transferred to catch bock, so that the error can be handled.

Resumptive model :- In this model, the programmer will not have to explicitly invoke the same method in which the error has occurred & then it was transferred to catch ~~bot~~ block, so that the error can be handled.

Exception hierarchy :-

In this hierarchy, 'Object' is the root class & has a subclass "Throwable". It has two subclasses 'Exception' class & 'Error' class. 'Exception' class has two subclasses ie Checked Exception & Runtime Exception.

$$\begin{tikzcd}
 & \text{Object} \arrow[d] & \\
 & \text{Throwable} \arrow[dl] \arrow[dr] & \\
\text{Exception} \arrow[d] \arrow[dr] & & \text{Error} \\
\text{Checked Exception} & \text{Runtime exception.} &
\end{tikzcd}$$

1. Checked Exceptions :-

Checked exceptions are the exceptions thrown by a method if it is not able to handle by itself. The different checked exceptions are given below -

a. ClassNotFoundException - this exception is generated when the requested class does not exist or if the class name is invalid.

b. IllegalAccessException - it is generated when the requested class cannot be accessed.

c. InstantiationException - it is generated when we are trying to create an object of an abstract class.

d. InterruptedException - it is generated when one thread interrupts another thread

e. NoSuchFieldException - it is generated when we are trying to access a field, but it was not exist.

f. NoSuchMethodException - it is generated when we are trying to call a method, but it was not exists.

2. Runtime exceptions :-

They also called unchecked exceptions. The different runtime exceptions are given below -

a. ArithmeticException - It is generated when an exceptional arithmetic condition has occurred.

eg : dividing a number with zero

b. ArrayStoreException - It is generated when we are storing a value in the array, but its data type is different from array type.

c. IllegalArgumentException - it is generated when an illegal argument is passed to a method.

d. IndexOutOfBoundsException - It is generated when an array object detects an out of range index.

e. NegativeArraySizeException - it is generated when an array of negative size is created.

f. NullPointerException - it is generated when an object is accessed using a null p object reference.

3. Error class :- they are -

a. ClassFormatError - it is generated when the definition of a class holds error.

b. AbstractMethodError - it is generated when abstract method is invoked.

c. IllegalAccessError - it is generated when a class is trying to access a variable or a method, but it is not having permission to access.

d. InstantiationError - it is generated when we try to create an object of abstract class.

e. NoSuchFieldError - it is generated when a variable is referenced, but it is not defined in the current class definition.

f. NoSuchMethodError - it is generated when a method is referenced, but it is not definded in the current class definition.

g. NoClassDefFoundError - it is generated when there is a problem in finding the class definition.

h. StackOverflowError - it is generated when there is a stack overflow.

i. OutOfMemoryError - it is generated when there is out of memory.

usage of 'try' :-

using try block, the run time errors can be handled & prevented. The block of code which may generate an exception will be placed in the try block. This code is then monitored to find out if any exceptions are present. Each try block is associated with a catch block, which is responsible for handling the exception.

Syntax is -

```
        try
        {
            // block of code that may generate an exception
        }
```

program :-

```
    class one
    {
      public static void main (String args[])
      {
          int a=10, b=0; c ;
          try
          {
             c = a/b;
          }
          catch (ArithmeticException e)
          {
             System.out.println ("you are dividing a number with zero");
          }
          System.out.println ("value of c =" +c);
      }
    }
```

Nested try :-

ie one try block can be present in other try block. whenever the control goes to the try block its content is pushed on stack. If inner try block doesn't have its matching catch for an exception, the stack is not popped & the catch block for next try block are inspected for a match. This process goes on until the exception is caught or all the nested try blocks are finished. If no catch block handles the exception then it is handled by Java runtime system.

Syntax is :-

```
try         // outer try
{
    // block of code that may generate an exception
    try     // inner try
    {
        // block of code that may generate an exception
    }
    catch (ExceptionType obj1)
    {
        // block of code to handle exception
    }
}
catch (ExceptionType obj1)
{
    // block of code to handle exception
}
```

program :-

```
class one
{
  public static void main (String args[])
  {
    int a, b, c;
    try
    {
      a = Integer.parseInt (args[0]);
      b = Integer.parseInt (args[1]);
      try
      {
         c = a/b;
         System.out.println ("value of c =" +c);
      }
```

```
            catch (ArithmeticException   e)
            {
              System.out.println ("second  number  should  not  be  zero");
            }
        }
        catch ( NumberFormatException    e)
        {
            System.out.println ("arguments  passed  should be  valid  numbers");
        }
      }
   }
```

usage of 'catch' :-

A catch block can ~~b~~ have a block of code which is used to handle the exception of a particular type.

Syntax :-

```
    catch ( ExceptionType      obj)
    {
        // code   used   to  handle   exception
    }
```

The catch block must immediately follow the try block.

program :-

Same program of 'try' block.

There are 2 cases -

1.

```
                      try
                      {
  if any        ┌          // code
  exception     │     }
  occurs        │     catch
                │     {
                └→        // code
  after         ┌     }
  handling      │     finally
  exception     │     {
                └→        // code
                      }
```

if any statement generates an exception in try block, then it must be handled by the corresponding catch block. After handling exception, then the code present in finally block is executed.

2.

```
              try
              {
          ┌──      // code
          │   }
  if      │   catch
  no      │   {
exception │      // code
          │   }
          │   finally
          │   {
          └──>   // code
              }
```

If a statement is not generated an exception in try block, then the catch block is not executed and then the code present in finally block is executed.

Multiple catch statements :-

In some cases, the statements in a try block may generate more than one exception. But a single catch block cann't handle more than one exception. Because each catch block can handle only one type of exception. To handle these situations, we need to use more than one catch ~~bet~~ block, each is handling a different type of exception. So, a single try block can have any number of catch blocks.

program :-

```
class one
{
  public static void main (String args[])
  {
     int a=10, b=0, c;
     int d[] = {10,20,30,40};
     try
     {
        System.out.println ("value of c = " + a/b);
        System.out.println (" d[5]);
     }
    catch (Arithmetic Exception e)
    {
        System.out.println (" dividing a number with zero');
    }
```

```
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println ("size exceeds");
        }
        catch (IOException e)
        {
            System.out.println ("exception in I/O");
        }
    }
}
```

here in try block first statement generates an exception, then the corresponding catch block is executed to handle exception. So, the second statement in try block is never executed. To make it execute, then take some value other than zero to b. Then first statement print c value, but second statement generates an exception. Then it can be handled by second catch block.

usage of 'throw':-

A try block checks for an error & when an error occurs it throws the error & it is caught by the catch block and then appropriate action will be taken. Only the expressions thrown by the Java runtime system are being caught, but throw statement allows a program to throw an expression explicitly.

Syntax is -

throw Throwableobject;

↳ it is the object of 'Throwable' class.

This object can be created using 2 ways.

1. using 'new' operator.

eg: throw new IOException;

2. as a parameter into catch block.

eg:
```
catch( IOException e )
{
    =
    throw e ;
}
```

=

usage of 'throws' :-

In some cases the method may also cause exceptions. But some methods could not be handled by itself. Then the method should inform the method caller that it may throw exceptions of some type. A method can specify what type of exceptions it is going to throw by using the keyword 'throws'.

Syntax :-

```
type method-name (parameter-list) throws exceptiontype1, exceptiontype2, ...
{
    // body of method
}
```

Program :-

```
class one
{
    static void test() throws IOException
    {
        System.out.Println (" inside test () method ");
        throw new IOException (" exception in IO");
    }
    public static void main ( String args[] )
    {
        try
        {
            test();
        }
```

```
            catch (IOException    e)
            {
              System. out. println  (" caught :" + e) ;
            }
        }
    }
                  output :        inside  test ( )  method
                                  caught   :  IOException
```

=

usage of 'finally' :-

The statements present in 'finally' block are Compulsory executed, eventhough an exception is generated in try block or not.

Syntax :-

```
        try
        {
            // code
        }
        catch ( )
        {
          / code
        }
        finally
        {
            // code  to  be  execute  Compulsory
        }
```

program :-

```
     class  one
     {
       public  static  void  main  ( String  args[])
       {
         int  a=10, b=0, c;
         try
         {
            c = a/b ;
         }
```

```
catch (ArithmeticException  e)
{
    System.out.println (" dividing  a  number  with  zero");
}
finally
{
    System.out.println ("execution  of  program  ends here");
 }
}
}
```

=

Built-in exceptions :-

write about 'exception hierarchy'.

=

Creating own exceptions :-

Java's built-in exceptions can handle most common errors, but ~~we~~ ~~are~~ in some cases, we may require to create our own exception types. For this we must define a subclass of exception. Subclasses don't require to implement anything. Exception class don't define any methods of its own. It inherits the methods provided by 'Throwable'. All exceptions which we have created, have the methods defined by 'Throwable' availabe to them.

program :-

```
class  MyException  extends  Exception
{
    public  MyException ()  {  }
    public  String  toString ()
    {
        return  " My own  exception type  error " ;
    }
}
```

```
public static void main(String args[])
{
  try
  {
     add(50,100);
     add(30,-20);
  }
  catch(MyException   e)
  {
     System.out.println("Caught :"+c);
  }
 }
}
```

output :

```
Sum = 150
Caught : My own exception type error
```

V Difference between ~~A~~ multithreading and multitasking :-

Multithreading :-

A program is divided into two or more subprograms and these subprograms can be executed at the same time. This process is called multithreading. The part of a program or subprogram is called a thread and each thread executes separately. Multithreading is a powerful programming tool that enables to write efficient programs by making maximum use of CPU, because CPU idle time can be reduced. Multithreads enables the programmers to do multiple things at a time. It can divide a large program into threads and execute them in parallel.

advantages :-

1. it reduces the idle time of CPU
2. it reduces the execution time
3. it increases the performance of computer
4. users can provide priority threads

Depending on the number of processors present in the system, the systems are divided into 2 types. They are -

1. Single processor system :-

It is having only one processor. Here multithreading is achieved by time slicing. ie the processor switches between different threads from time to time. Because of having single processor, the threads cannot run concurrently. The processor switches between threads so fast that it gives the illusion of simultaneous execution of threads to the user.

2. Multiprocessor system :-

It is having more than one processor. Here multithreading is achieved by multiprocessing, where each thread is executed independently with another thread at a time. In multiprocessor system, the multithreading is achieved automatically. Here operating system uses all the processors, so that any number of threads can be executed simultaneously.

Multi tasking :-

E In multithreading each thread is executed separately independent of other threads. Thus multithreading is a special form of multitasking.

There are 2 types of multitasking. They are-

1. process - based multitasking :

A process is a program which is in the execution. process-based multitasking allows two or more processes can run concurrently.

2. thread - based multitasking :

Thread - based multitasking allows two or more tasks of a single program can run simultaneously.

Creating threads :-

Threads are implemented in the form of objects that contain a method called run(). This method contains the body of a thread ie the behaviour of a thread can be implemented. we can start the execution of a thread by using start() method.

The syntax of run() method is -

```
public void run()
{
        // statements for implementing thread
}
```

A new thread can be created in 2 ways.

1. by creating a thread class
2. by implementing 'Runnable' interface.

1. by extending the 'Thread' class :-

It has 3 steps. They are -

a. declare the class as extending 'Thread' class -

eg: the 'Thread' ~~Can~~ class can be extended as -

```
class thread1 extends Thread
{
        ≡
}
```

now we have a new type of thread 'thred1'

b. implement the thread using run() method -

we can implement the run() method used for executing the statements that the thread can execute.

Syntax is -

```
public void run()
{
        ≡
}
```

c. start the execution of a thread using Start() method -

we can initiate the execution of a thread using start() method. For this, create an object of thread class and then call the Start() method using this object.

```
eg: thread1    t1 = new thread1( );
               t1 . Start( );
```

then 'thred1' can start its execution.

2. by implementing 'Runnable' interface :-

The 'Runnable' interface declares the run() method that is required for implementing threads in our programs. To do this the steps are -

a. declare the class as implementing 'Runnable' interface

b. implement the run() method

c. Create the thread by defining an object that is instantiated from this 'runnable' class

d. to initiate execution of a thread call the Start() method.

program :-

```
class x implements Runnable
{
  public void run ()
  {
    for(int i=1 ; i<=10; i++)
    {
       System.out.Println ("thread x :" + i);
    }
  }
}

Class one
{
  public static void ~~vol~~ main (String args[])
  {
        X  x = new X();
        Thread t1 = new Thread(x);
        t1.start();
      }
   }
```

program :-

```
   class A extends Thread
   {
      public void run()
      {
        for (int i=1; i<=5; i++)
        {
          System.out.println("from thread A :"+i);
        }
      }
   }

   class B extends Thread
   {
     public void run()
     {
       for (int j=1; j<=5; j++)
       {
         System.out.println("from thread B :"+j);
       }
     }
   }

   class C extends Thread
   {
     public void run()
     {
       for (int k=1; k<=5; k++)
       {
         System.out.println("from thread C :"+k);
       }
     }
   }
```

```
class one
{
    public static void main (String args[])
    {
        A   a = new  A();
        B   b = new  B();
        C   c = new  C();
            a. Start();
            b. Start();
            c. Start();
    }
}
```

Stopping and blocking a thread :-

Stopping a thread :-

we can stop the execution of a thread by using Stop() method.

eg : a. Stop();

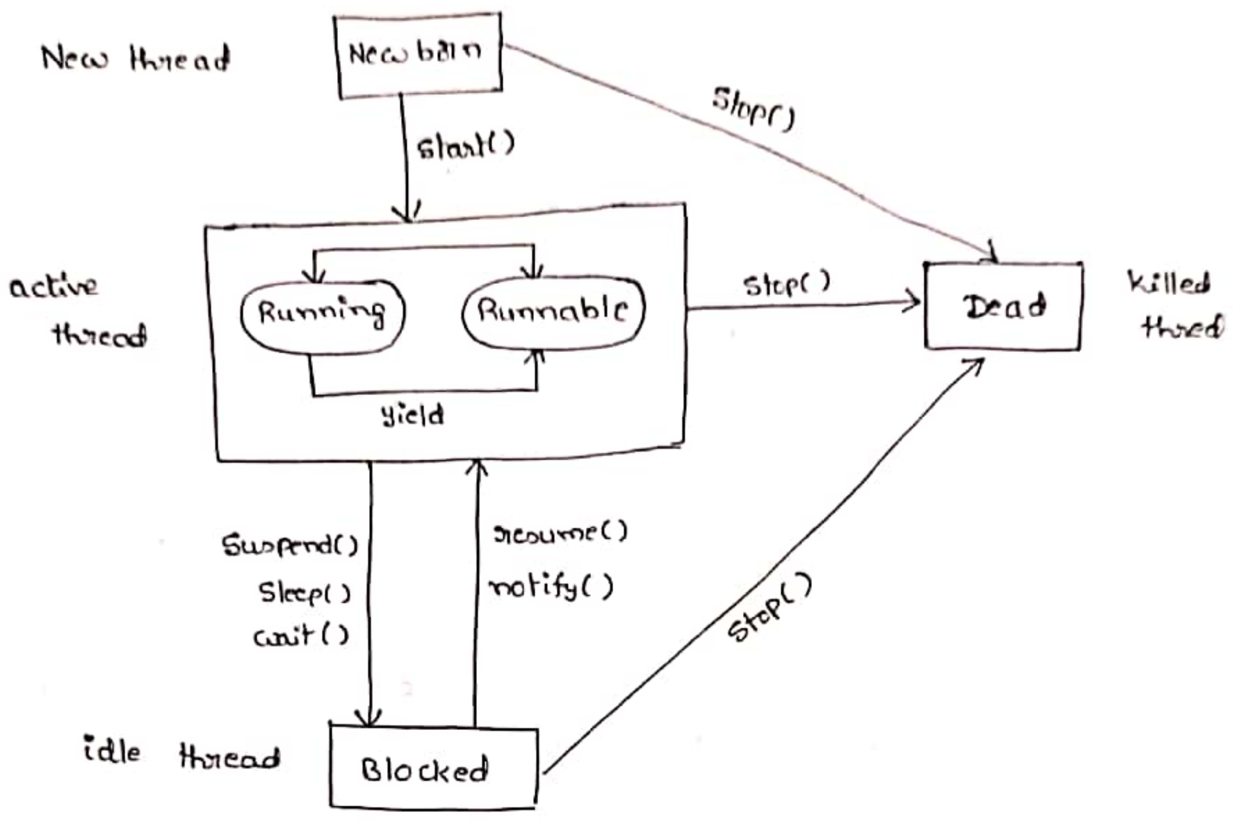
Blocking a thread :-

Blocking a thread means we can stop the execution of a thread temporarily. But Stopping a thread means permanently the execution of a thread is stopped. There are three methods to block a thread. They are -

1. Sleep() - if we use this method the thread is blocked for a specified time. Here ~~to restart~~ the execution of thread can be restarted after the completion of the given time.
2. Suspend() - if we use this method the thread is blocked until further order occurs. Here the execution of thread can be restarted using suspend() method.
3. wait() - if we use this method the thread is blocked until some condition occurs. Here the execution of thread can be restarted using notify() method.

Life cycle of a thread :-

it is also called state transition diagram of a thread.



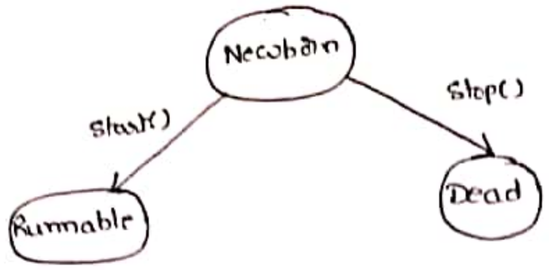
There are 5 states in a life cycle of a thread.

1. New born state
2. Runnable "
3. Running "
4. Blocked "
5. Dead "

1. New born State :-

When we create a thread object, then the thread is born and it is said to be in newborn state. At this state, we cando only one of the following things with it -

- we can schedule it for running using Start() method
- we can kill it using Stop() method.



2. Runnable State :-

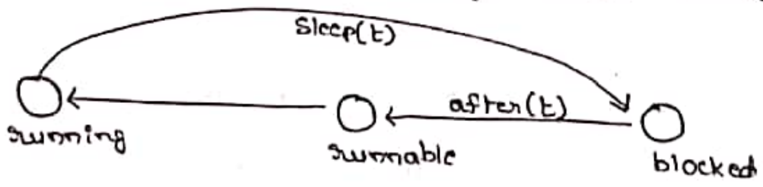
When the thread is ready to execute and it is waiting for the availability of the processor, then the thread is said to be in runnable state. ~~when~~ If there are any threads which are ready to execute, then they are joined in the form of a queue. If all threads have equal priority, then they have given time slots for execution in first come first serve manner. After completion of time slot of a thread, then it will be placed at the end of queue and CPU will be given to the next thread and so on.

If we want a thread to give up control of CPU to another thread of equal priority, before its turn comes, then we use yield() method.

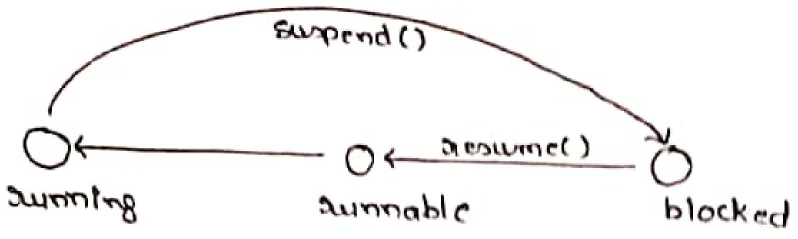
3. Running State :-

When a thread is in the execution, then it is said to be present in running state. There are 3 cases -

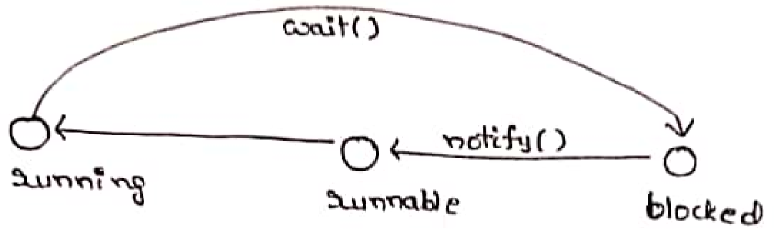
a. When a thread is in running state, then if we use sleep() method then the thread is blocked for some time. Then state changes from running to blocked state. After completion of time period, the state changes from blocked to runnable state.



b. When a thread is in running state, then if we use suspend() method then the thread is blocked. Then state changes from running to blocked state. After using the resume() method, the state changes from blocked to runnable state.



c. when a thread is in running state, then if we use wait() method then the thread is blocked. Here, the state changes from running to blocked state. After using the notify() method, the state changes from blocked to runnable state.



4. Blocked State :-

If we use sleep(), suspend() & wait() methods, then the execution of thread will be terminated temporarily. Then the state of the thread is said to be in blocked state.

5. Dead state :-

A running thread ends its life when it has completed executing its run() method. we can kill the thread using stop() method. After killing the thread, it is said to be in dead state.

priority threads :-

If all the threads are having equal priority then they are executed sequentially ie one by one. If we assign different priorities for the threads, then the ~~ex~~ order of execution of the threads is changed. In Java, we assign priorities for the threads using setpriority() method. Syntax is -

Thread object. setpriority (int number);

where 'number' is the integer priority number

The 'Thread' class defines different priority constants :

MIN-PRIORITY = 1

NORM-PRIORITY : 5

MAX-PRIORITY = 10

The default priority is NORM-PRIORITY

program :-

```
class A extends Thread
{ public void run()
  {
   for (int i=1 ; i<=5 ; i++)
   {
     System.out.println (" from thread A : "+ i);
   }
  }
}

class B extends Thread
{
 public void run()
 {
  for (int j=1 ; j<=5 ; j++)
  {
    System.out.println (" from thread B ; "+ j);
  }
 }
}

class C extends Thread
{
  public void run()
  {
   for (int k=1 ; k<=5 ; k++)
   {
     System.out.println ("from thread c : "+k) ;
   }
  }
}
```

```
class one
{
    public static void main (String args[])
    {
        A  a = new A();
        B  b = new B();
        C  c = new C();

           a. setPriority (Thread.MIN_PRIORITY);
           b. setPriority (Thread.NORM-PRIORITY);
           c. setPriority (Thread.MAX-PRIORITY);

           a. Start();
           b. Start();
           c. Start();
    }
}
```

Synchronized threads :-

when a thread wants to access a resource, but already it was allocated to another thread, then there will be a problem. Because the same resource is already allocated to another thread. In Java, we can solve this problem using a technique known as "synchronization". The keyword "Synchronized" is used to solve such problems. For example, the method that will read information from a file and a method that will update the same file may be declared as "Synchronized".

eg:

```
synchronized void update()
{
        // code here is synchronized
}
```

When we declare a method 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.

Due to some reason, the condition on which the waiting threads rely on to gain control does not happen. This result is known as deadlock. For example, 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.

eg:

```
thread A
        Synchronized method2()
        {
            Synchronized method1()
            {
                ⋮
            }
        }
thread B
        Synchronized method1()
        {
            Synchronized method2()
            {
                ⋮
            }
        }
```

=