Ch-4 Exceptional Handling and Multithreading Marks: 12

Error

- Errors are mistakes that can make program go wrong.
- Error may be logical or may be typing mistake.
- An error may produce an incorrect output or may terminate the execution of program abruptly or even may cause the system may crash.

Types of Errors: -

1. Compile time errors
2. Runtime errors

1. **Compile time errors**:

   All syntax errors will be detected and displayed by java compiler and therefore these errors are known

   as compile time errors. The most of common problems are:

   - Missing semicolon
   - Missing (or mismatch of) bracket in classes & methods
   - Misspelling of identifiers & keywords
   - Missing double quotes in string
   - Use of undeclared variables.
   - Bad references to objects.

2. **Runtime errors:**

   Sometimes a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. When such errors are encountered java typically generates an error message and aborts the program. The most common run-time errors are:

   - Dividing an integer by zero
   - Accessing an element that is out of bounds of an array
   - Trying to store value into an array of an incompatible class or type Passing parameter that is not in a valid range or value for method
   - Trying to illegally change status of thread
   - Attempting to use a negative size for an array

- Converting invalid string to a number
- Accessing character that is out of bound of a string

Exception

- An exception is an event, which occurs during the execution of a program, that stop the flow of the program's instructions and takes appropriate actions if handled.
- Exceptional handling mechanism provides a means to detect errors and throw exceptions, and then to catch exceptions by taking appropriate actions. Java Exception handles as follow
    - Find the problem (Hit the exception)
    - Inform that an error has occurred (throw the Exception)
    - Receive the error information (Catch the exception)
    - Take corrective action (Handle the Exception)

Exceptional handling in java by five keywords are as follows:

1. **try:** This block applies a monitor on the statements written inside it. If there exist any exception, the control is transferred to catch or finally block.

   Syntax:

   ```
   try
   {
    // block of code to monitor for errors
   }
   ```

2. **catch:** This block includes the actions to be taken if a particular exception occurs.

   Syntax:

   ```
   catch (ExceptionType1 exOb)
   {
    // exception handler for ExceptionType1
   }
   ```

3. **finally:** finally block includes the statements which are to be executed in any case, in case the exception is raised or not.

Syntax:

```
finally
{
 // block of code to be executed before try block ends
 }
```

4. **throw:** This keyword is generally used in case of user defined exception, to forcefully raise the exception and take the required action.

   Syntax:

   ```
   throw throwable instance;
   ```

5. **throws:** throws keyword can be used along with the method definition to name the list of exceptions which are likely to happen during the execution of that method. In that case, try … catch block is not necessary in the code.

   Syntax:

   ```
   Type method-name (parameter list) throws exception list
    {
    // body of method
   }
   e.g. public static void main (String a[]) throws IOException
   {
   ----
   }
   ```

Example:

```
 class DemoException
{
public static void main(String args[])
{
try
{
int b=8;
int c=b/0;
System.out.println("answer="+c);
}
catch(ArithmeticException e)
{
```

```
System.out.println(“Division by Zero”);
}
}
}
```

Example using throw and throws keyword

```
import java.lang.*;
class GreaterException1 extends Exception
{
GreaterException1(String msg) throws IOException
{
super(msg);
}
}
class GreaterException
{
public static void main(String a[])
{
try
{
int n1=12;
int n2=10;
if (n1>n2)
throw new GreaterException1("No.1 is greater");
else
System.out.println("No.2 is greater");
}
catch(GreaterException1 ge)
{
System.out.print(ge);
}
}
}
```

Example using finally keyword

```
import java.lang.*;
public class testfinally
{
public static void main(String a[]) throws IOException
{
System.out.println(""Start of execution)
int a, b, c;
try
{
a=Interger.parseInt(arg[0]);
a=Interger.parseInt(arg[1]);
c=a/b;
System.out.println("Answer is"+c);
}
catch(Exception e)
{
System.out.println(e)
}
finally
{
System.out.println("Finally will always execute");
}
System.out.println("Execution complete here");
}
}
```

Throw	**Throws**
Whenever we want to force an exception then we use **throw** keyword.	when we know that a particular exception may be thrown or to pass a possible exception then we use **throws** keyword.
"Throw" is used to handle user-defined exception.	JVM handles the exceptions which are specified by "throws".

It can also pass a custom message to your exception handling module.	Point to note here is that the Java compiler very well knows about the exceptions thrown by some methods so it insists us to handle them.
Throw keyword can also be used to pass a custom message to the exception handling module i.e. the message which we want to be printed.	We can also use **throws clause** on the surrounding method instead of **try and catch exception handler.**
Throw is used to through exception system explicitly.	Throws is used for to throws exception means throws ioexception and servletexception and etc.
Throw is used to actually throw the exception.	Whereas *throws* is declarative for the method. They are not interchangeable.
It is used to generate an exception.	It is used to forward an exception.

Built-in Exception

- Java exceptions are the exceptions that are caused by run time error in the program. Some common exceptions in java are as follows:

Sr. No.	Built-in Exception	Explanation
1	ArithmeticException	It is caused by Math error such as divide by 0.
2	ArrayIndexOutOfBoundsException	It is caused when array index is out of bound
3	ArrayStoreException	It is caused when a program tries to store wrong type of data in an array.
4	FileNotFoundException	It is caused by an attempt to access a non-existent file.
5	IOException	It is caused by general IO failure such as inability to read from the file.

6	NullPointerException	It is caused by referencing a null object
7	NumberFormatException	It is caused when a conversion between between string and number fails.
8	OutOfMemoryException	It is caused when there is not enough memory to allocate a new object.
9	SecurityException	It is caused when an applet tries to perform an action not allowed by thebrowser security setting.
10	StackOverFlowException	It is caused when system runs out of space.
11	StringIndexOutOfBoundsException	It is caused when a program attempts to access non-existent character in a string.

Nested Try

The nested try is used to implement multiple try statements in a single block of main method.

Syntax: -

```
Try
{
----
----
Try
{
----
----
}
}
```

Chained Exception

- Whenever in a program first exception causes another exception to occur it is called a chained exception.

- Exception chaining is also known as nesting exception and it is technique for handling the exception which occurs one after the other that is most of the time given by an application in response to an exception by throwing another exception.
- Typically, the second exception caused by the first exception therefore chained exception helped programmer to know when one exception causes another.
- The methods and constructor in throwable that support chain exceptionare.

Q. Write a program to input name and age of person and throws user defined exception, if entered age is negative

```
import java.io.*;
class Negative extends Exception
{
Negative(String msg)
{
super(msg);
}
}
class Negativedemo
{
public static void main(String ar[])
{
int age=0;
String name;
BufferedReaderbr=new BufferedReader (new InputStreamReader(System.in));
System.out.println("enter age and name of person");
try
{
age=Integer.parseInt(br.readLine());
name=br.readLine();
{
if(age<0)
throw new Negative("age is negative");
```

```
else
System.out.println("age is positive");
}
}
catch(Negative n)
{
System.out.println(n);
}
catch(Exception e)
{
}
}
}
```

Write a program to accept password from user and throw 'Authentication failure' exception if password is incorrect.

```
import java.io.*;
class PasswordException extends Exception
{
PasswordException(String msg)
{
super(msg);
}
}
class PassCheck
{
public static void main(String args[])
{
BufferedReader bin=new BufferedReader(new
InputStreamReader(System.in));
try
{
System.out.println("Enter Password : ");
```

```
if(bin.readLine().equals("abc"))
{
System.out.println("Authenticated ");
}
else
{
throw new PasswordException("Authentication failure");
}
}
catch(PasswordException e)
{
System.out.println(e);
}
catch(IOException e)
{
System.out.println(e);
}
}
}
```

Write a program to throw a user defined exception "String Mismatch Exception" if two strings are not equal. (ignore case).

```
import java.io.*;
class MyException extends Exception
{
MyException(String msg)
{
super(msg);
}
}
class ExceptionTest
{
public static void main(String args[])
```

```
{
BufferedReader br=new BufferedReader (new
InputStreamReader(System.in));
String s1,s2;
try
{
System.out.println("Enter String one and String two ");
s1=br.readLine();
s2=br.readLine();
if(s1.equalsIgnoreCase(s2)) // any similar method which give correct result
{
System.out.println("String Matched");
}
else
{
throw new MyException("String Mismatch Exception");
}
}
catch(MyException e)
{
System.out.println(e);
}
catch(IOException e)
{
System.out.println(e);
}
}
}
```

Write a program to throw a user defined exception as 'Invalid Age', if age entered by the user is less than eighteen. Also mention any two common java exceptions and their cause.

```
import java.lang.Exception;
```

```
import java.io.*;
class myException extends Exception
{
myException(String msg)
{
super(msg);
}
}
class agetest
{
public static void main(String args[])
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
try
{
System.out.println("enter the age : ");
int n=Integer.parseInt(br.readLine());
if(n < 18 )
throw new myException("Invalid Age");
else
System.out.println("Valid age");
}
catch(myException e)
{
System.out.println(e.getMessage());
}
catch(IOException ie)
{
}
}
}
```

Define an exception called "No match Exception" that is thrown when a string is not equal to "MSBTE". Write program.(8 Marks)

```
import java.io.*;
class NoMatchException extends Exception
{
NoMatchException(String s)
{
super(s);
}
}
class test1
{
public static void main(String args[]) throws IOException
{
BufferedReader br= new BufferedReader(new InputStreamReader(System.in) );
System.out.println("Enter a word:");
String str= br.readLine();
try
{
if (str.compareTo("MSBTE")!=0)   // can be done with equals()
throw new NoMatchException("Strings are not equal");
else
System.out.println("Strings are equal");
}
catch(NoMatchException e)
{
System.out.println(e.getMessage());
}
}
}
```

Write a program to input name and balance of customer and thow an user defined exception if balance less than 1500.(4 Marks)

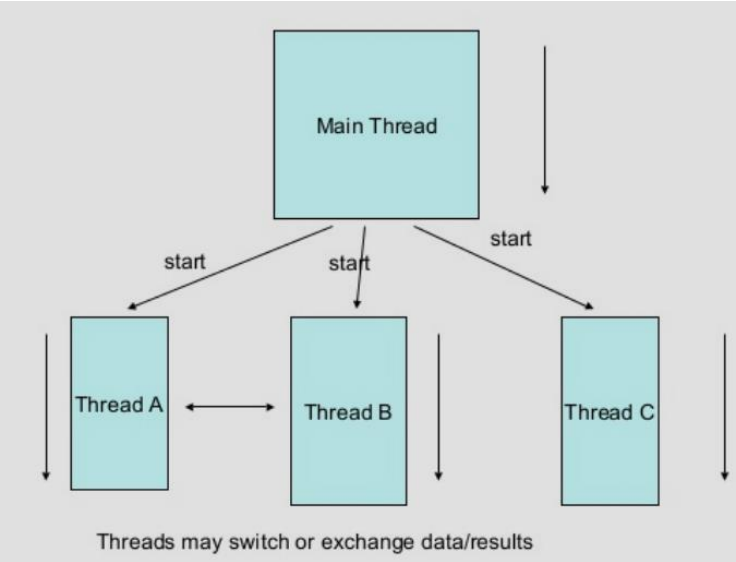
```
import java.io.*;
```

```
class MyException extends Exception
{
MyException(String str)
{
super(str);
}
}
class AccountDetails
{
public static void main(String a[])
{
try
{
BufferedReaderbr = new BufferedReader(new InputStreamReader(System.in));
String name;
int balance;
System.out.println("Enter name");
name = br.readLine();
System.out.println("Enter balance");
balance = Integer.parseInt(br.readLine());
try
{
if(balance<1500)
{
throw new MyException("Balance is less");
}
else
{
System.out.println("Everything alright");
}
}
catch(MyException me)
{
System.out.println("Exception caught"+me);
```

```
}
}
catch(Exception e)
{
System.out.println("Exception caught"+e);
}
}
}
```

Multithreading Programming

- It is technique that allows a program or process to execute many task concurrently.
- It allows process to run its task in parallel mode on single processor system.
- In multithreading concept several multiple light weight processes run in a single process by single processor.
- For e.g. When work with word processor you can perform many different task such as printing, spell check and so on. Multithreading software treats each process as a separate program.

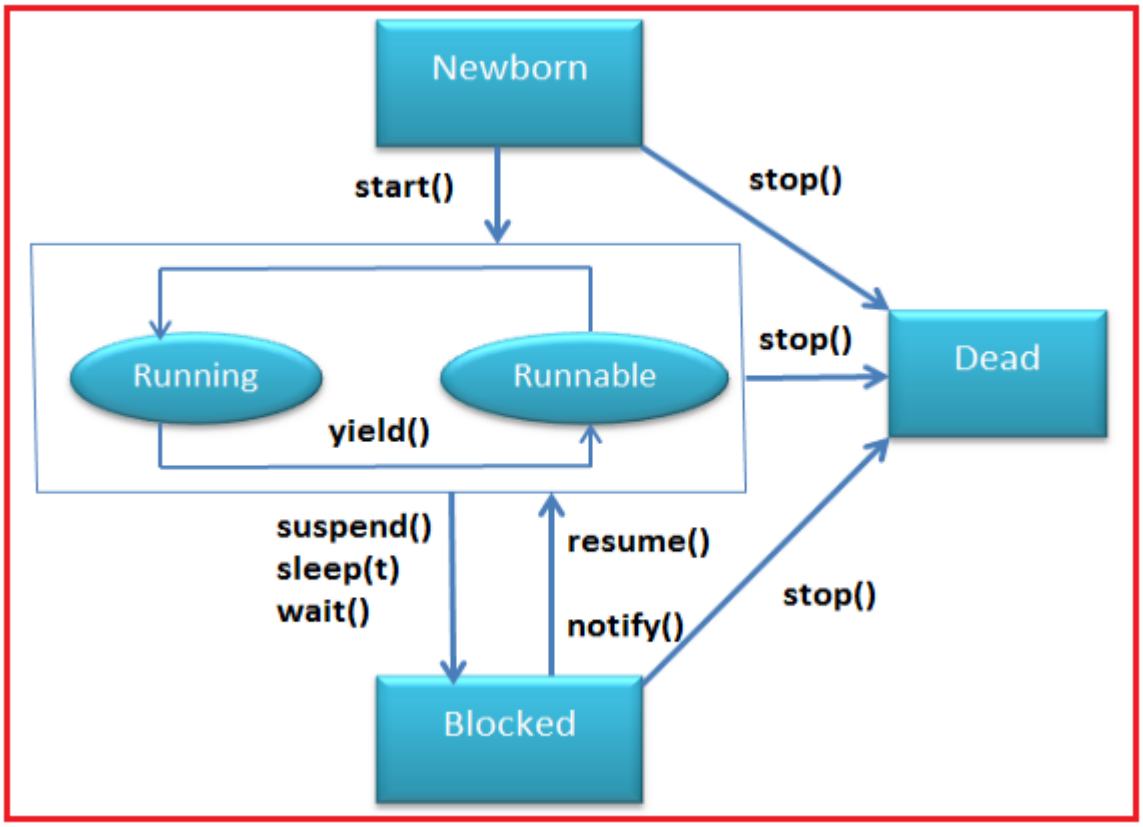


Thread

- Thread is a smallest unit of executable code or a single task is also called as thread.
- Each tread has its own local variable, program counter and lifetime.

- A thread is similar to program that has a single flow of control.
- It has beginning, body and end executes command sequentially.

Life Cycle of Thread



Thread Life Cycle Thread has five different states throughout its life.

1) Newborn State

When a thread object is created it is said to be in a newborn state. When the thread is in a new born state it is not scheduled running from this state it can be scheduled for running by start() or killed by stop(). If put in a queue it moves to runnable state.

2) Runnable State

It means that thread is ready for execution and is waiting for the availability of the processor i.e. the thread has joined the queue and is waiting for execution. If all threads have equal priority then they are given time slots for execution in round robin fashion. The thread that relinquishes control joins the queue at the end and again waits for its turn. A thread can relinquish the control to another before its turn comes by yield().

3) Running State

It means that the processor has given its time to the thread for execution. The thread runs until it relinquishes control on its own or it is pre-empted by a higher priority thread.

4) Blocked State

A thread can be temporarily suspended or blocked from entering into the runnable and running state by using either of the following thread method.

- suspend() : Thread can be suspended by this method. It can be rescheduled by resume().
- wait(): If a thread requires to wait until some event occurs, it can be done using wait method and can be scheduled to run again by notify().
- sleep(): We can put a thread to sleep for a specified time period using sleep(time) where time is in ms. It reenters the runnable state as soon as period has elapsed /over.

5) Dead State

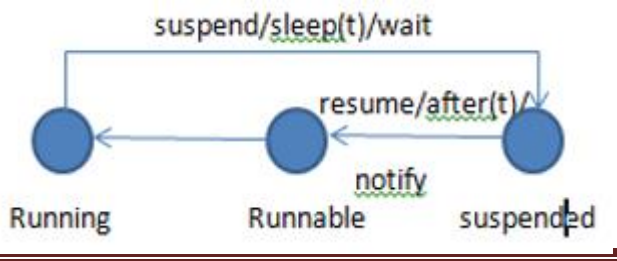
Whenever we want to stop a thread form running further we can call its stop(). The stop() causes the thread to move to a dead state. A thread will also move to dead state automatically when it reaches to end of the method. The stop method may be used when the premature death is required

Thread should be in any one state of above and it can be move from one state to another by different methods and ways.

Q) With proper syntax and example explain following thread methods:

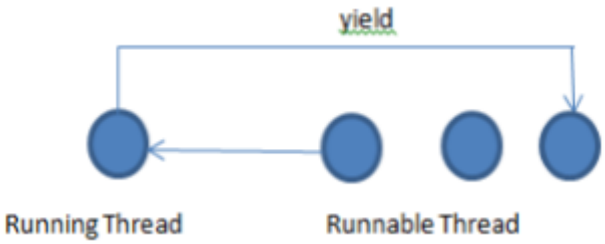
With proper syntax and example explain following thread methods:

1) suspend() - syntax : public void suspend() This method puts a thread in suspended state i.e blocked and can be resumed using resume() method.



2) resume() syntax : public void resume() This method resumes a thread which was suspended using suspend() method.The thread enter in active state i.e Runnable state.

3) yield() syntax : public static void yield() The yield() method causes the currently executing thread object to temporarily pause and move to runnable state from running state and allow other threads to execute.



4) wait() and notify() syntax : public final void wait() This method causes the current thread to wait until some event occurs and another thread invokes the notify() method or the notifyAll() method for this object.

5) stop() syntax: void stop() Used to kill the thread. It stops thread.

6)sleep() syntax: public static void sleep(long millis) throws InterruptedException We can put a thread to sleep for a specified time period using sleep(time) where time is in ms. It reenters the runnable state as soon as period has elapsed /over.

Eg.

```
class sus extends Thread implements Runnable
{
static Thread th;
float rad,r;
 public sus()
{
 th= new Thread();
 th.start();
 }
```

```
 public void op()
 {
 System.out.println("\nThis is OP");
 if(rad==0)
{
 System.out.println("Waiting for input radius");
 Try
 {
 wait();
 }
 catch(Exception ex)
{

}
}
 }
public void ip()
{
 System.out.println("\nThis is IP");
r=7;
rad= r;
 System.out.println(rad);
 System.out.println("Area = "+3.14*rad*rad);
notify();
}
public static void main(String arp[])
{
 Try
{
sus s1 = new sus();
 System.out.println("\nReady to go");
 Thread.sleep(2000);
 System.out.println("\nI am resuming");
 th.suspend();
```

```
Thread.sleep(2000);
th.resume();
 System.out.println("\nI am resumed once again");
s1.op();
s1.ip();
s1.op();
 }
 catch(Exception e)
 {
}
 }
 }
```

Creating Threads

There are two ways to create in java:

1. By extending thread class
    - User specified thread class is created by extending the class 'Thread' and overriding its run() method.
    - For creating a thread a class has to extends the thread class that is java.lang.Thread
    - Syntax: -

      Declare class as extending thread

      class Mythread extends Thread

      {

      _____

      }

Develop a program to create two threads such that one thread will print odd no. and another thread will print even no. between 1-20

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

```
for(int i=1;i<=20;i=i+2)
{
System.out.println("ODD="+i);
try
{
sleep(1000);
}
catch(Exception e)
{
System.out.println("Error");
}
}
}
}
class even extends Thread
{
public void run()
{
for(int i=0;i<=20;i=i+2)
{
System.out.println("EVEN="+i);
try
{
sleep(1000);
}
catch(Exception e)
{
System.out.println("Error");
}
}
}
}
class oddeven2
{
```

```
public static void main(String arg[])
{
odd o=new odd();
even e=new even();
o.start();
e.start();
}
}
```

Develop a program to create three threads such that one thread will print odd no. and another thread will print even no. and third thread will print all no. between 1-10 and set Priority.

```
class odd extends Thread
{
public void run()
{
for(int i=1;i<=10;i=i+2)
{
System.out.println("ODD="+i);
try
{
sleep(1000);
}
catch(Exception e)
{
System.out.println("Error");
}
}
}
}
class even extends Thread
{
public void run()
{
```

```
for(int i=0;i<=10;i=i+2)
{
System.out.println("EVEN="+i);
try
{
sleep(1000);
}
catch(Exception e)
{
System.out.println("Error");
}
}
}
}
class allnumbers extends Thread
{
public void run()
{
for(int i=0;i<=10;i++)
{
System.out.println("All numbers="+i);
try
{
sleep(1000);
}
catch(Exception e)
{
System.out.println("Error");
}
}
}
}
class oddeven1
{
```

```
public static void main(String arg[])
{
odd o=new odd();
even e=new even();
allnumbers a=new allnumbers();
o.setPriority(6);
e.setPriority(4);
a.setPriority(8);
o.start();
e.start();
a.start();
}
}
```

Write a program to create two threads; one to print numbers in original order and other to reverse order from 1 to 50.

```
class original extends Thread
{
public void run()
{
try
{
for(int i=1; i<=50;i++)
{
System.out.println("\t First Thread="+i);
Thread.sleep(300);
}
}
catch(Exception e)
{}
}
}
class reverse extends Thread
{
```

```
public void run()
{
try
{
for(int i=50; i>=1;i--)
{
System.out.println("\t Second Thread="+i);
Thread.sleep(300);
}
}
catch(Exception e)
{
}
 }
}
class orgrev
{
public static void main(String args[])
{
new original().start();
new reverse().start();
}
}
```

Q. Write a program to create two threads one to print odd numbers from 1 to 10 and other to print even numbers from 11 to 20

```
import java.lang.*;
class even extends Thread
{
public void run()
{
try
{
for(int i=0;i<=10;i=i+2)
```

```
{
System.out.println("\tEven thread="+i);
Thread.sleep(300);
}
}
catch(InterruptedException e)
{}
}
}

class odd extends Thread
{
public void run()
{
try
{
for(int i=1;i<=10;i=i+2)
{
System.out.println("\todd thread="+i);
Thread.sleep(300);
} }
catch(InterruptedException e)
{
}
}
}
class evenodd
{
public static void main(String args[])
{
new even().start();
new odd().start();
}
}
```

2. Creating thread using Runnable Interface

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

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

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

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

For E.g.

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

Write a thread program for implementing the "Runnable interface". (8 Marks) Or program to print even numbers from 1 to 20 using Runnable Interface

```
class mythread implements Runnable
{
public void run()
{
System.out.println("Even numbers from 1 to 20 : ");
for(int i= 1 ; i<=20; i++)
{
```

```
if(i%2==0)
System.out.print(i+ " ");
}
}
}
class test
{
public static void main(String args[])
{
mythread mt = new mythread();
Thread t1 = new Thread(mt);
t1.start();
}
}
```

What is synchronization? When do we use it? Explain synchronization of two threads.

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization. Synchronization used when we want to –
    1) prevent data corruption.
    2) prevent thread interference.
    3) Maintain consistency If multiple threads require an access to an Object.

Program based on synchronization:

```
class Callme
{
void call(String msg)
{
System.out.print("[" +msg);
try
{
Thread.sleep(1000);
```

```
}
catch(InterruptedException e)
{
System.out.println("Interrupted ");
}
System.out.print("]");
}
}
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public Caller(Callme targ,String s)
{
target=targ;
msg=s;
t=new Thread(this);
t.start();
}
public void run()
{
synchronized(target)
{
target.call(msg);
}
}
class Synch
{
public static void main(String args[])
{
Callme target=new Callme();
Caller ob1=new Caller(target,"Hello");
Caller ob2=new Caller(target,"Synchronized");
```

```
try
{
ob1.t.join();
ob2.t.join();
}
catch(InterruptedException e)
{
System.out.println("Interrupted ");
}
}
}
```

Thread Priority

- Threads in java are sub programs of main application program and share the same memory space. They are known as light weight threads.
- A java program requires at least one thread called as main thread. The main thread is actually the main method module which is designed to create and start other threads.
- A Thread is similar to a program that has a single flow of control. Every thread has a beginning, a body and an end.
- However, thread is not a program, but runs within a program.
- **Thread Priority:** In java each thread is assigned a priority which affects the order in which it is scheduled for running. Threads of same priority are given equal treatment by the java scheduler.
- The thread class defines several priority constants as: -

  MIN_PRIORITY =1

  NORM_PRIORITY = 5

  MAX_PRIORITY = 10
- Thread priorities can take value from 1-10.

  1) setPriority () This method is used to assign new priority to the thread

Syntax: Thread.setPriority (priority value);

2) getPriority() It obtain the priority of the thread and returns integer value.

Syntax: int Thread.getPriority ();

Interprosses Communication

- Java provide benefits of avoiding thread pooling using inter-thread communication.
- The **wait()**, **notify()**, and **notifyAll()** methods of Object class are used for this purpose. These method are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context.

- **wait()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- **notify()** wakes up a thread that called wait() on same object.
- **notifyAll()** wakes up all the thread that called wait() on same object.

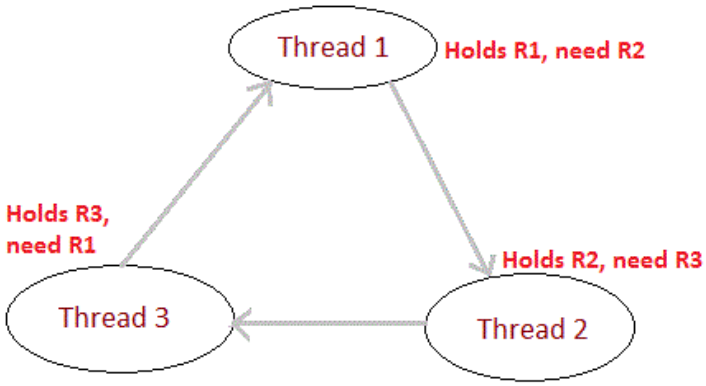
```
import java.util.Scanner;
public class threadexample
{
public static void main(String[] args)
throws InterruptedException
{
 final PC pc = new PC();

       // Create a thread object that calls pc.produce()
 Thread t1 = new Thread(new Runnable()
  {
          @Override
   public void run()
   {
    try
   {
    pc.produce();
    }
    catch(InterruptedException e)
    {
     e.printStackTrace();
    }
```

```
 }
 }
};

    // Create another thread object that calls
    // pc.consume()
Thread t2 = new Thread(new Runnable()
 {
       @Override
  public void run()
  {
  try
  {
            pc.consume();
          }
          catch(InterruptedException e)
          {
            e.printStackTrace();
          }
       }
    });

    // Start both threads
    t1.start();
    t2.start();

    // t1 finishes before t2
    t1.join();
    t2.join();
 }

 // PC (Produce Consumer) class with produce() and
 // consume() methods.
 public static class PC
 {
    // Prints a string and waits for consume()
    public void produce()throws InterruptedException
    {
       // synchronized block ensures only one thread
       // running at a time.
       synchronized(this)
       {
          System.out.println("producer thread running");

          // releases the lock on shared resource
          wait();

          // and waits till some other method invokes notify().
          System.out.println("Resumed");
       }
    }
```

```
        // Sleeps for some time and waits for a key press. After key
        // is pressed, it notifies produce().
        public void consume()throws InterruptedException
        {
            // this makes the produce thread to run first.
            Thread.sleep(1000);
            Scanner s = new Scanner(System.in);

            // synchronized block ensures only one thread
            // running at a time.
            synchronized(this)
            {
                System.out.println("Waiting for return key.");
                s.nextLine();
                System.out.println("Return key pressed");

                // notifies the produce thread that it
                // can wake up.
                notify();

                // Sleep
                Thread.sleep(2000);
            }
        }
    }
}
```

Deadlock

- Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources.
- In the diagram, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3.
- Hence none of them can finish and are stuck in a deadlock.




```
public class TestThread {

   public static Object Lock1 = new Object();

   public static Object Lock2 = new Object();


   public static void main(String args[]) {

      ThreadDemo1 T1 = new ThreadDemo1();

      ThreadDemo2 T2 = new ThreadDemo2();

      T1.start();

      T2.start();

   }

   private static class ThreadDemo1 extends Thread {

      public void run() {

         synchronized (Lock1) {

            System.out.println("Thread 1: Holding lock 1...");


            try { Thread.sleep(10); }

            catch (InterruptedException e) {}

            System.out.println("Thread 1: Waiting for lock 2...");


            synchronized (Lock2) {

               System.out.println("Thread 1: Holding lock 1 & 2...");

            }
```

```
            }
        }
    }
    private static class ThreadDemo2 extends Thread {
        public void run() {
            synchronized (Lock2) {
                System.out.println("Thread 2: Holding lock 2...");

                try { Thread.sleep(10); }
                catch (InterruptedException e) {}
                System.out.println("Thread 2: Waiting for lock 1...");

                synchronized (Lock1) {
                    System.out.println("Thread 2: Holding lock 1 & 2...");
                }
            }
        }
    }
}
```