JAVA

Unit-5

Dr. Kiran Bala Dubey

Assistant Professor

Department of Computer Science

Govt. N. P. G. College of Science,

Raipur

UNIT-V

Multithreaded Programming : Java spread model, creating threads, and thread priorities, synchronization. Suspending resuming and stopping threads. **Input/Output:** Basic Streams, Byte and Character Stream, predefined streams, reading and writing from console and files. Using standard Java Packages (lang,util,io), **JDBC:** Setting the JDBC connectivity with backend database.

Multithreaded Programming

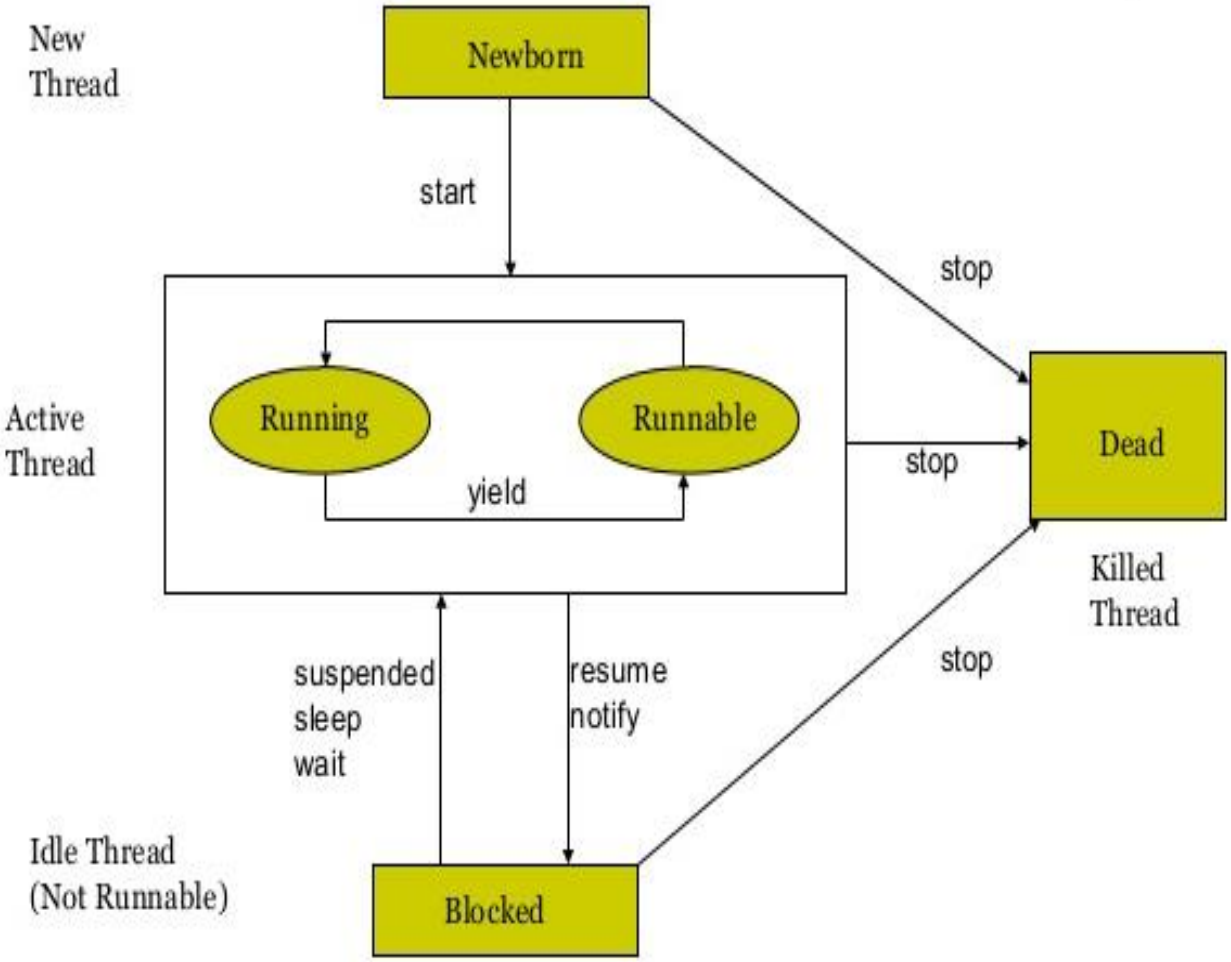
Multithreading is a Java feature that allows concurrent execution of two or more parts of a program and each part can handle a different task at the same time for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms -

1. Extending the Thread class
2. Implementing the Runnable Interface

Advantages of Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
2) You can perform many operations together, so it saves time.
3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.


New
Thread
Newborn
start
stop
Active
Thread
Running
Runnable
yield
stop
Dead
Killed
Thread
suspended
sleep
wait
resume
notify
stop
Idle Thread
(Not Runnable)
Blocked

1) **New -** The thread is in new state if you create an instance of Thread class. but before the invocation of start() method.
2) **Runnable -** The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3) **Running -** The thread is in running state if the thread scheduler has selected it.
4) **Waiting/Blocked -** This is the state when the thread is still alive, but is currently not eligible to run.
5) **Dead/Terminated -** A thread is in terminated or dead state when its run() method exits.

Commonly used methods of Thread class

- **run():** is used to perform action for a thread.
- **start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **getPriority():** returns the priority of the thread.
- **setPriority(int priority):** changes the priority of the thread.
- **yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **suspend():** is used to suspend the thread.
- **resume():** is used to resume the suspended thread.
- **stop():** is used to stop the thread.

Thread Priorities

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.

```
Class A extends Thread
{
  public void run()
{
    System.out.println("Thread A
     started");
    for(int i=1; i<=4;i++)
 { System.out.println("\t From thread A :
    i = " +i); }
System.out.println("Exit from A ");
}
}
Class B extends Thread
{
public void run()
{
    System.out.println("Thread B
started");
     for(int j=1; j<=4;j++)
 {System.out.println("\t From thread B : j "
+j; }
System.out.println("Exit from B ");
}
}
Class C extends Thread
{
  public void run()
{
    System.out.println("Thread C
started");
     for(int k=1; k<=4;k++)
{System.out.println("\t From thread C : k = "
+k; }
System.out.println("Exit from C ");
}
}
```

```
Class Therad1
{
  public static void main(String args[])
  {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
     c1.setPriority(Thread.MAX_PRIORITY);
     b1.setPriority(a1.getPriority()+1);
     a1.setPriority(Thread.MIN_PRIORITY);
     System.out.println("Start  thread A");
     a1.start();
System.out.println("Start  thread B");
     b1.start();
System.out.println("Start  thread  C");
     c1.start();
System.out.println("End  of main thread");
  }
}
```

```
Class A extends Thread
{
  public void run()
{
    System.out.println("Thread A
     started");
    for(int i=1; i<=4;i++)
 {if(i==1) yeild();
System.out.println("\t From thread A : i
    = " +i); }
System.out.println("Exit from A ");
}
}
Class B extends Thread
{
public void run()
{
    System.out.println("Thread B
started");
     for(int j=1; j<=4;j++)
 {System.out.println("\t From thread B : j "
+j);
If(j==3) stop();
}
System.out.println("Exit from B ");
}
}
Class C extends Thread
{
  public void run()
{
    System.out.println("Thread C
started");
```

```
for(int k=1; k<=4;k++)
{System.out.println("\t From thread C : k = "
    +k);
If(k==1)
Try(sleep(1000))} catch(Exception e) {}
System.out.println("Exit from C ");
} }
Class Therad1
{   public static void main(String args[])
 {  A a1 = new A();
   B b1 = new B();
   C c1 = new C();
    System.out.println("Start thread A");
    a1.start();
System.out.println("Start thread B");
    b1.start();
System.out.println("Start thread  C");
    c1.start();
System.out.println("End of main
thread");
  }
}
```

```
Class X implements Runnable
 {
 public void run()
{
    System.out.println(“Thread X started”);
     for(int i=1; i<=4;i++)
 { System.out.println(“\t From thread X : i = “ +i); }
System.out.println(“Exit from X “);
} }
Class Therad2
{
  public static void main(String args[])
  {
    X x1 = new X();
    Thread t1= new Thread(x1);
    t1.start();
   System.out.println(“End of main thread”);
  }   }
```

Synchronization

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- It allows only one thread to access the shared resource at a time.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.
- Java provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks.

Input/Output

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Example –

```
System.out.println("simple message");
System.err.println("error message");
```

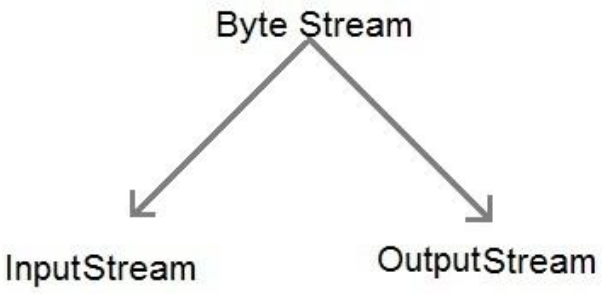
```
int i=System.in.read();          //returns ASCII code of 1st character
System.out.println((char)i);     //will print the character
```

- Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.

- Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,
- **Byte Stream :** It provides a convenient means for handling input and output of byte.
- **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Byte Stream Classes

- Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

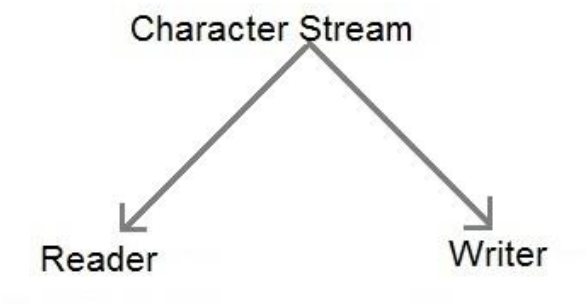
- These classes define several key methods. Two most important are-

read() : reads byte of data.

write() : Writes byte of data

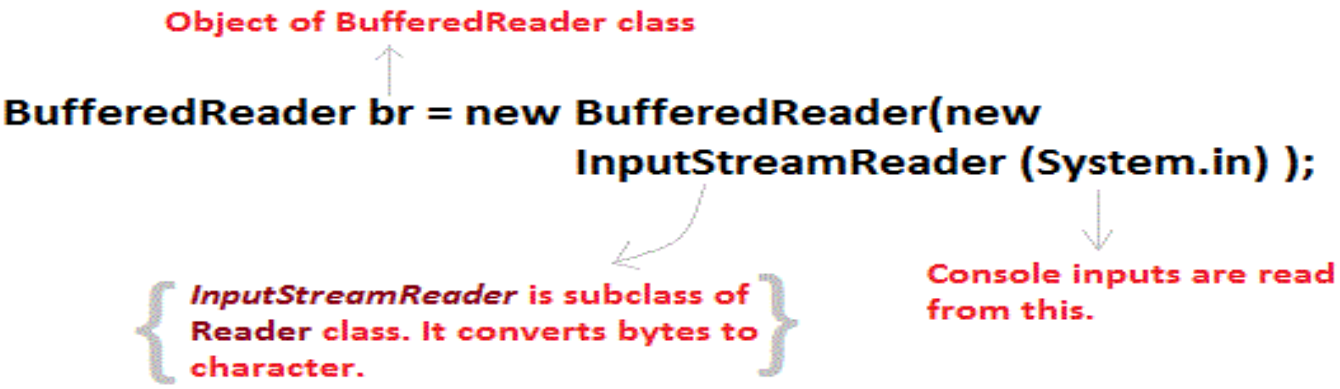
Character Stream Classes

- Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



These two abstract classes have several concrete classes that handle unicode character.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output


Object of BufferedReader class
BufferedReader br = new BufferedReader(new
InputStreamReader (System.in));
InputStreamReader is subclass of Reader class. It converts bytes to character.
Console inputs are read from this.

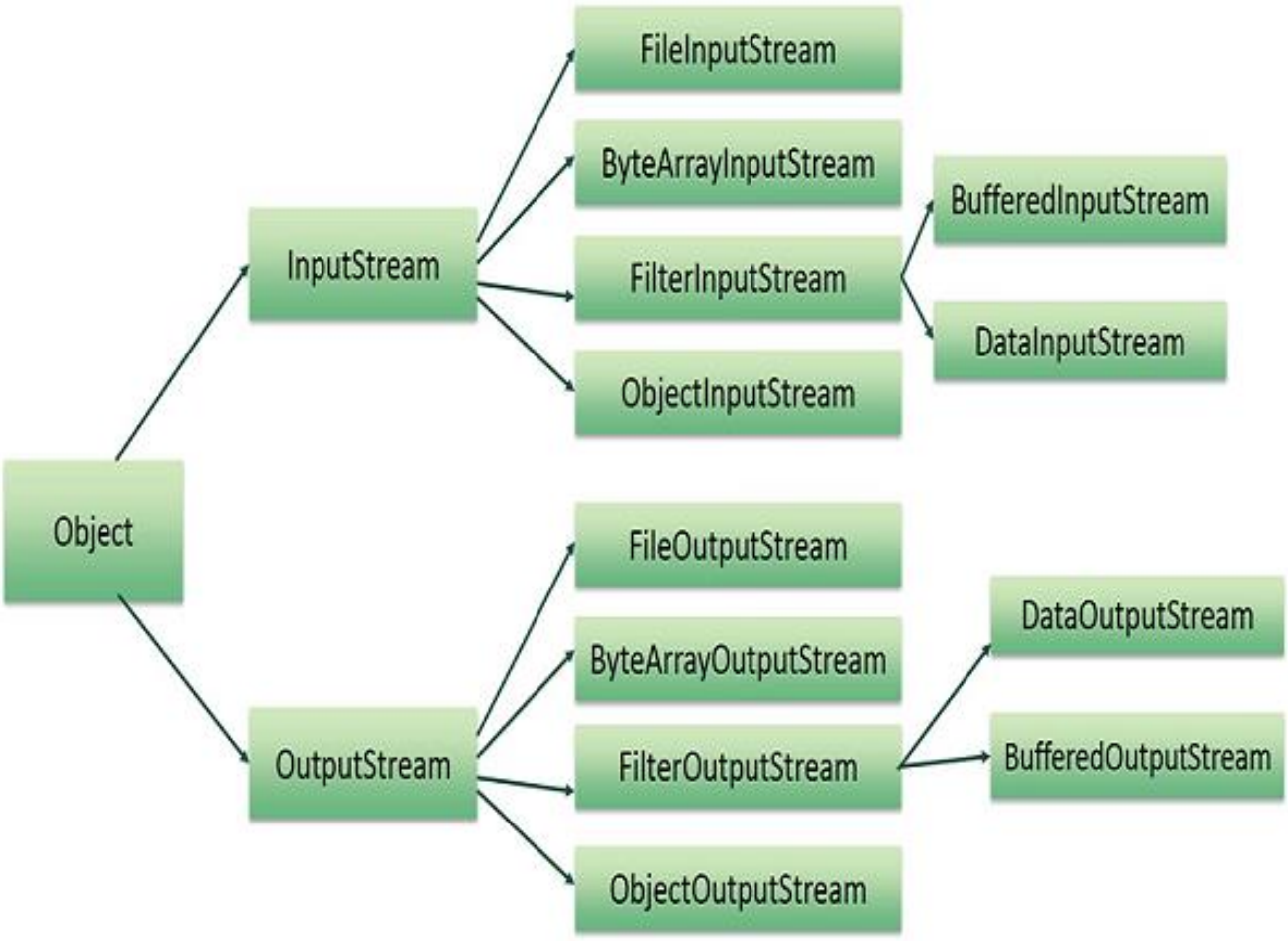
Byte Stream

```
import java.io.*;
public class CopyFile
{
 public static void main(String args[])
     throws IOException
{
   FileInputStream in = null;
   FileOutputStream out = null;
   try {
    in = new FileInputStream("input.txt");
    out = new FileOutputStream("output.txt");
     int c;
        while ((c = in.read()) != -1)
        { out.write(c); }
        }
finally {
if (in != null)
{ in.close(); }
if (out != null)
{ out.close(); }
}
}
}
```

Character Streams

```
import java.io.*;
public class CopyFile
{
    public static void main(String
     args[]) throws IOException
   {
     FileReader in = null;
     FileWriter out = null;
    try {
 in = new FileReader("input.txt");
 out = new FileWriter("output.txt");
      int c;
      while ((c = in.read()) != -1)
      {
         out.write(c);
     }
     }
     finally {
     if (in != null)
     { in.close(); }
     if (out != null)
     { out.close(); }
     }
     }
     }
```

Reading and Writing Files



```
// Reading & writing from consol and file
Import java.util.*;   //For using StringTokenizer class
Import java.io.*;
Class Inventory
{ static DataInputStream din=new DataInputStream(System.in);
 static StringTokenizer st;
Public static void main(String args[]) throws IOException
{
 DataOutputStream dos=new DataOutputStream(new
     FileOutputStream("invent.dat"));
//Reading from console
System.out.println("Enter code  - ");
St=new StringTokenizer(din.readLine());
Int code=Integer.parseInt(st.nextToken());

System.out.println("Enter qty of items - ");
St=new StringTokenizer(din.readLine());
Int item=Integer.parseInt(st.nextToken());
```

```
System.out.println("Enter cost - ");
St=new StringTokenizer(din.readLine());
Double cost=new
    Double.nextToken()).dobleValue;
// Writing to the file
Dos.writeInt(code);
Dos.writeInt(item);
Dos.writeDouble(cost);
Dos.close();
//Processing data from the file
DataInputStream dis=new DataInputStream
    (new FileInputStream("invent.dat"));
Int code1=dis.readInt();
Int item1=dis.readInt();
Double cost1=dis.readDouble();
Double tcost=item1*cost1;
Dis.close();
// writing to consol
System.out.println();
System.out.println("Code  - "+code1);
System.out.println("Item qty - "+item1);
System.out.println("Item cost - "+cost1);
System.out.println("Total cost - "+tcost);
}
}
```

Output-

```
Enter code –
1001
Enter qty of item –
193
Enter cost-
452
Code – 1001
Item  qty – 193
Item cost – 452.0
Total cost – 87236.0
```

JDBC

- JDBC is front-end tool for connecting to a server and is similar to ODBC.
- JDBC can connect only Java clients and it uses ODBC for the connectivity. It is called a low level API since any data manipulation, storage and retrieval has to be done by the program itself.
- JDBC is written entirely in Java and ODBC is a C interface.

JDBC Architecture

1) JDBC Driver Manager
2) JDBC Driver
3) JDBC-ODBC Bridge
4) Application.

JDBC Driver Manager : work of the driver manager is to find out available drivers in the system and connect the application to the appropriate database, whenever a connection is requested. To help the driver manager identify different types of drivers, each driver should be registered with the driver manager.

JDBC Driver : work of the JDBC Driver is to accept the SQL calls from the application and convert them into native calls to the database.

JDBC-ODBC Bridge : SunSoft provides a special JDBC driver called JDBC-ODBC Bridge driver which can be used to connect to any existing database, that is ODBC enabled database.

Application : Application is a java program that needs the information to be modified in some database or wants to retrieve the information.

Database Connectivity

The java.sql package contains classes that help in connecting to a database, sending embedded statement to the database and processing query result.

The Connection Class – The object of Connection class represents a connection with a database. You may have several connection objects in an application that connect to one or more database.

Loading the JDBC-ODBC Bridge and establishing the connection –

To establish a connection with a database, you need to register the JDBC_ODBC driver by calling ***forName()*** method from the ***Class*** class***.***

GetConnection() Method – attempts to locate the driver that connects to the database represented by JDBC-URL passed to the getConnection() method from DriverManager class.

Querying a database – Once a connection with the database is established, you can query to database and process the result set.

JDBC provides the following three classes for sending SQL statements to a database.

1) The Statement Class
2) The PreparedStatement Class
3) The CallableStatement Class

The Statement Class - The object of this class is used to send simple queries to the database and also allows you to execute simple queries. It has three methods for the purpose of querying –

- The **executeQuery()** Method executes a simple select query and returns a single result set object.
- The **executeUpdate()** method executes the sql INSERT, UPDATE and DELETE statements.
- The **execute()** method executes an sql statements that may return multiple results**.**

The ResultSet Class- provides methods to access data from the table.

Executing a statement usually generates a ResultSet object. It maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row. The **next()** method moves the cursor to the next row. You can access data from the ResultSet rows by calling the **getxxx()** method. Where xxx is the data type of the parameter.

```
import java.sql.*;   // Table creation
import java.lang.*;
class KJdbc
{
public static void main(String s[])
{
 try {
Class.forName("oracle.jdbc.driver.OracleDriver");
 Connection con=DriverManager.getConnection ("jdbc:oracle:thin:@localhost:1521:xe","SYS as
      SYSDBA","MANAGER");

//Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"):
//Connection con= DriverManager.getConnection("jdbc:odbc:koracle","scott","tiger");

Statement st= con.createStatement();
St.execute("create  table kdiary(pnam varchar2(15), phone number (10))");
St.close();
Con.close();
}  catch(Exception e) { System.out.println(e); }
}
}
```

```
import java.sql.*;  // Data insertion
import java.lang.*;
class KJdbcIns
{  public static void main(String s[])
{
 try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
 Connection con=DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:xe","SYS as SYSDBA","MANAGER");

//Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”):
//Connection con= DriverManager.getConnection(“jdbc:odbc:koracle”,”scott”,”tiger”);
Statement st= con.createStatement();
St.executeUpdate(“insert into kdiary values (‘”+s[0]+”’,”+s[1]+”)”);
St.close();
Con.close();
}  catch(Exception e)  {  System.out.println(e);}
}
}
```

```
import java.sql.*; //Display record
import java.lang.*;
class KJdbc
{
 public static void main(String s[])
{
 try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
 Connection con=DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:xe","SYS
      as SYSDBA","MANAGER");


//Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"
      ):
//Connection con=
      DriverManager.getConnection("jdbc:odbc:kor
      acle","scott","tiger");
Statement st= con.createStatement();
ResultSet rst=st.executeQuery("select  * from
      kdiary");
While(rst.next())
{
System.out.println(rst.getString(1)+rst.
getString(2));
}

rst.close();
St.close();
Con.close();
}
catch(Exception e)
{
  System.out.println(e);
}
}
}
```