UNIT-V

Multi-threading:

Introduction:

JAVA language provides built-in support for multithreaded programming.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution. Thus multithreading is a specialized form of multitasking.

Generally there are two types of multitasking.

1) Process-based multitasking

2) Thread-based multitasking

Process-based multitasking:

- Process means a program that is under execution.
- In process-based multitasking, we are running two or more programs simultaneously.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
- Processes are heavyweight tasks that require their own separate address spaces.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.

Thread-based multitasking:

- It means a single program can perform two or more tasks simultaneously.
- For example, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Threads are light weight tasks, because they share the same address space and cooperatively share the same heavyweight process.
- Inter thread communication is inexpensive.
- Context switching from one thread to the next is lower in cost.

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable.**

The **Thread** class defines several methods that help manage threads.
Some of the methods are

Method	Meaning
getName()	Obtain a thread's name
getPriority()	Obtain a thread's priority
isAlive()	Determine if a thread is still running
join()	Wait for a thread to terminate
run()	Entry point for the thread
sleep()	Suspend a thread for a period of time
start()	Start a thread by calling its run method
setPriority()	Set the priority of a thread

Life cycle of a Thread (or) Thread States:

In the life cycle of a thread, thread exists in 5 states. They are

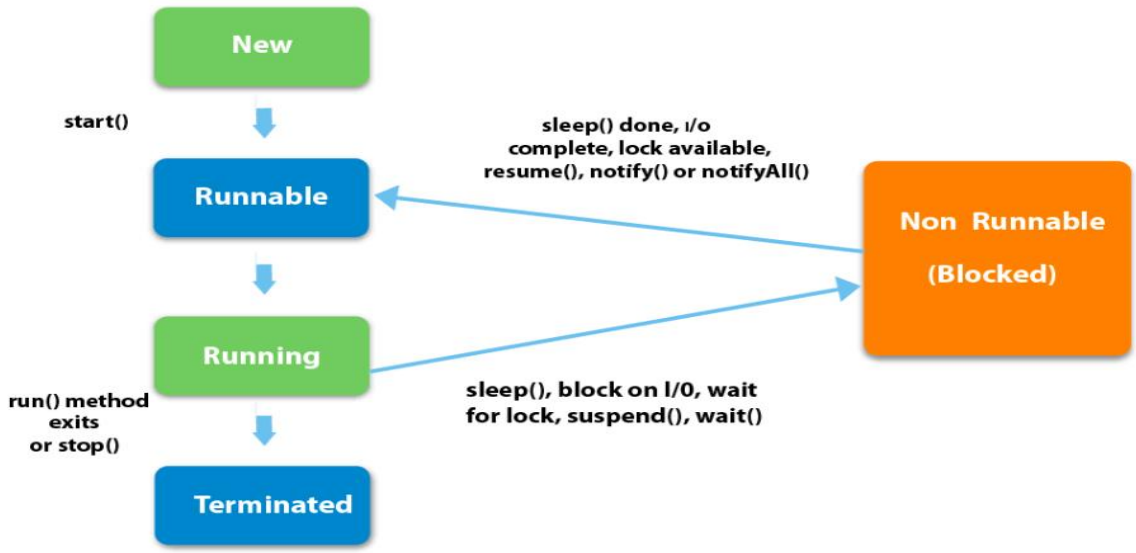
1) New

2) Runnable

3) Running

4) Blocked/Wait

5) Dead (or) Terminated



1) **New State:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.
2) **Runnable State:** When the start method is invoked, the thread is moved from New state to Runnable state. In Runnable state, the thread is ready to run and waiting for CPU. It is the duty of thread scheduler to assign CPU to the thread.
3) **Running State:** When the thread gets CPU, the thread is moved from Runnable state to Running state. In Running state, the thread starts its execution and performs a particular task.
4) **Blocked/Wait State:** When the thread is running state, if a thread is waiting for some I/O resources or sleep() is called on thread object then the thread is moved from running state to Blocked/wait state. After the sleeping time is over or resources available, thread is moving from Blocked/wait state to Runnable state.
5) **Dead (or) Terminated State:** Once the thread finishes its execution, the thread is moved to Dead state.

Main Thread:

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins.
The main thread is important for two reasons:

- It is the thread from which other "child" threads will be created.
- It must be the last thread to finish execution because it performs various shutdown actions.

In order to obtain the information about the currently running thread, the following method is used.

```
static Thread currentThread()
```

The above method returns a reference to the thread in which it is called.
Once we have a reference to the main thread, we can control it just like any other thread.

Example:

```
class CurrentThreadDemo
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try
```

```
        {
            for(int n = 5; n > 0; n--)
            {
                System.out.println(n);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted");
        }
    }
}
```

Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when **t** is used as an argument to **println( )**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs.

Creating threads:

In JAVA language, to create threads there are 2 ways.

1) By implementing Runnable interface
2) By extending Thread class

Creating threads by implementing Runnable interface:

To create threads by implementing Runnable interface, the following steps are used.

1. Create a class that implements Runnable interface
2. We have to instantiate Thread object by using the following constructor.
   Thread(Runnable obj, String threadname)
3. Once the Thread object is created, we can start it by calling start() method, which executes a call to run() method.
4. Implementation class must implement a single method called run() which is declared as follows
   public void run()

Example:

```
//Program to create thread by implementing Runnable interface
class NewThread implements Runnable
{
    Thread t;
    NewThread()
    {
       t=new Thread(this,"Demo thread"):
       System.out.println("Child Thread: "+t);
       t.start();
     }
    public void run()
    {
      try
      {
          for(int i=5;i>0;i--)
          {
            System.out.println("Child Thread: "+i);
            Thread.sleep(500);
           }
       }
      catch(InterruptedException ie)
      {
            System.out.println("Child Thread is interrupted");
       }
      System.out.println("Child thread is exiting");
     }
}
class ThreadDemo
{
   public static void main(String args[])
   {
     NewThread nt=new NewThread();
     try
     {
                for(int i=5;i>0;i--)
          {
            System.out.println("Main Thread: "+i);
            Thread.sleep(1000);
           }
       }
      catch(InterruptedException ie)
      {
            System.out.println("Main Thread is interrupted");
       }
      System.out.println("Main thread is exiting");
    }
}
```

Creating threads by extending Thread class:

To create a thread by extending Thread class, the following steps are used.

1. Create a class that extends Thread class.
2. Within that class, thread object is created by calling super class constructor.
3. Once the Thread object is created, we can start it by calling start() method, which executes a call to run() method.
4. Sub class must implement a method called run() which is declared as follows
   public void run()

Example:

```
//Program to create a thread by extending Thread class
 class NewThread extends Thread
{
    NewThread()
    {
       super("Demo thread"):
       System.out.println("Child Thread: "+this);
       start();
    }
    public void run()
    {
      try
      {
          for(int i=5;i>0;i--)
          {
            System.out.println("Child Thread: "+i);
            Thread.sleep(500);
          }
      }
      catch(InterruptedException ie)
      {
            System.out.println("Child Thread is interrupted");
      }
      System.out.println("Child thread is exiting");
    }
}
class ThreadDemo
{
   public static void main(String args[])
   {
     NewThread nt=new NewThread();
     try
     {
                for(int i=5;i>0;i--)
          {
            System.out.println("Main Thread: "+i);
```

```
            Thread.sleep(1000);
          }
        }
        catch(InterruptedException ie)
        {
            System.out.println("Main Thread is interrupted");
        }
        System.out.println("Main thread is exiting");
    }
}
```

Creating multiple threads:

```
 class NewThread implements Runnable
{
  Thread t;
  String name;
  NewThread(String threadname)
  {
    name=threadname;
    t=new Thread(this,name);
    System.out.println("New thread: "+t);
    t.start();
  }
  public void run()
  {
   try
   {
     for(int i=5;i>0;i--)
     {
      System.out.println(name+" : "+i);
      Thread.sleep(500);
     }
   }
   catch(InterruptedException e)
   {
     System.out.println(e);
   }
  System.out.println(name+ "is exiting");
}
}
class ThreadDemo
{
  public static void main(String args[])
  {
    NewThread nt1=new NewThread("one");
    NewThread nt2=new NewThread("two");
```

```
   NewThread nt3=new NewThread("three");
   try
   {
    for(int i=5;i>0;i--)
    {
      System.out.println("main thread: "+i);
      Thread.sleep(1000);
     }
   }
   catch(InterruptedException e)
   {
    System.out.println(e);
    }
  System.out.println("main thread exiting");
 }
}
```

Write a program that creates three threads in which first thread displays "Good morning" for every one second, second thread displays "Hello" for every two seconds and third thread displays "Welcome" for every three seconds.

```
class NewThread implements Runnable
{
 Thread t;
 String name;
 NewThread(String threadname)
 {
   name=threadname;
   t=new Thread(this,name);
   t.start();
 }
 public void run()
 {
  try
  {
    for(int i=5;i>0;i--)
    {
     if(t.getName().equals("one"))
     {
          System.out.println("Good morning");
         Thread.sleep(1000);
     }
     if(t.getName().equals("two"))
    {
       System.out.println("Hello");
       Thread.sleep(2000);
     }
    if(t.getName().equals("three"))
```

```
   {
      System.out.println("Welcome");
      Thread.sleep(3000);
     }
     }
  }
  catch(InterruptedException e)
  {
   System.out.println(e);
  }
 }
}
class Week9a
{
 public static void main(String args[])
 {
  NewThread nt1=new NewThread("one");
  NewThread nt2=new NewThread("two");
  NewThread nt3=new NewThread("three");
  }
}
```

The above program can also be written by extending Thread class.

```
class NewThread extends Thread
{
 NewThread(String threadname)
 {
  Super(threadname);
  start();
 }
 public void run()
 {
  try
  {
   for(int i=5;i>0;i--)
   {
    if(getName().equals("one"))
    {
         System.out.println("Good morning");
        Thread.sleep(1000);
    }
    if(getName().equals("two"))
   {
      System.out.println("Hello");
      Thread.sleep(2000);
    }
   if(getName().equals("three"))
```

```
   {
      System.out.println("Welcome");
      Thread.sleep(3000);
     }
     }
  }
  catch(InterruptedException e)
  {
   System.out.println(e);
  }
 }
}
class Week9a
{
 public static void main(String args[])
 {
   NewThread nt1=new NewThread("one");
   NewThread nt2=new NewThread("two");
   NewThread nt3=new NewThread("three");
   }
}
```

Write a program that illustrates the use of isAlive and join methods.

```
//Program to illustrate the use of isAlive and join methods
class NewThread implements Runnable {
String name;
Thread t;
NewThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
public void run()
 {
try
{
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
```

```
}
}
class Week9b
{
public static void main(String args[])
{
NewThread nt1 = new NewThread("One");
NewThread nt2 = new NewThread("Two");
NewThread nt3 = new NewThread("Three");

System.out.println("Thread One is alive: "+ nt1.t.isAlive());
System.out.println("Thread Two is alive: "+ nt2.t.isAlive());
System.out.println("Thread Three is alive: "+ nt3.t.isAlive());
try
{
System.out.println("Waiting for threads to finish.");
nt1.t.join();
nt2.t.join();
nt3.t.join();
}
catch (InterruptedException e)
{
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ nt1.t.isAlive());
System.out.println("Thread Two is alive: "+ nt2.t.isAlive());
System.out.println("Thread Three is alive: "+ nt3.t.isAlive());
}
}
```

Thread Priorities:

Every thread is having one priority value.

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

In general, always high priority thread will get CPU first.

Thread class defines 3 constants.

final static int MIN_PRIORITY=1

final static int NORM_PRIORITY=5

final static int MAX_PRIORITY=10

To assign the priority to a thread, the following method is used

void setPriority(int level)

To obtain the priority of current thread, the following method is used.

int getPriority()

Example:

```
//Program to illustrate the use of thread priorities
class SampleThread extends Thread
{
    public void run()
   {
         System.out.println("Inside SampleThread");
         System.out.println("Current Thread: " +Thread.currentThread().getName());
    }
}
class ThreadPriorityDemo
 {
    public static void main(String[] args)
   {
         SampleThread st1 = new SampleThread();
         SampleThread st2 = new SampleThread();
         st1.setName("first");
         st2.setName("second");
         st1.setPriority(4);
         st2.setPriority(Thread.MAX_PRIORITY);
         st1.start();
         st2.start();
      }
}
```

Daemon Threads:

Daemon threads are low priority threads which are running in background of a program.

Daemon threads are service provider which provides services to user threads.

In order to make the thread as daemon thread, the following method is used.

void setDaemon(boolean val)

To check whether the thread is Daemon thread or not, the following method is used.

boolean isDaemon()

Example:

```
//Program to illustrate the use of Daemon threads
class NewThread extends Thread

{

NewThread(String name)
{

super(name);

}
public void run()
 {
if(Thread.currentThread().isDaemon())
  {
System.out.println(getName() + " is Daemon thread");
   }
else
 {
     System.out.println(getName() + " is User thread");
 }
}
}
class Week9c
{
public static void main(String[] args)
{
  NewThread nt1 = new NewThread("first thread");

  NewThread nt2 = new NewThread("second thread");

  NewThread nt3 = new NewThread("third thread");
   nt1.setDaemon(true);
   nt1.start();
   nt2.start();
```

```
nt3.setDaemon(true);
 nt3.start();
 }
}
```

Inter thread communication:

Inter thread communication means providing communication between 2 or more threads.

Consider producer consumer problem. In this, producer produces the data and consumer consumes the data. Here until the producer produces data, consumer has to wait and until the consumer consumes the data, producer has to wait. By doing this, more CPU cycles are wasted.

To avoid this, inter thread communication is provided by using the following methods

wait( )-- tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )** or **notifyAll( )**.

notify( )-- wakes up a thread that called **wait( )** on the same object.

notifyAll( )-- wakes up all the threads.

Example:

```
//Program that illustrates producer consumer problem
class Q
{
 int n;
 synchronized void get()
 {
     System.out.println("Got: "+n);
  }
 synchronized void put(int n)
 {
     this.n=n;
     System.out.println("Put: "+n);
 }
}
class Producer implements Runnable
```

```
{
 Thread t;
  Q q;
 Producer(Q q)
 {
  this.q=q;
  t=new Thread(this,"Producer");
  t.start();
 }
 public void run()
 {
  int i=1;
  while(i<=5)
  {
   q.put(i++);
  }
 }
}
class Consumer implements Runnable
{
 Thread t;

 Q q;
 Consumer(Q q)
 {
  this.q=q;
  t=new Thread(this,"Consumer");
  t.start();
 }
 public void run()
 {
  int i=1;
```

```
   while(i<=5)
   {
    q.get();
    i++;
    }
 }
}
class PC
{
  public static void main(String args[])
  {
    Q q=new Q();
    Producer p1=new Producer(q);
    Consumer c1=new Consumer(q);
   }
}
```

In the above program, two threads such as producer and consumer are running simultaneously and there is no communication between two threads. So we will get erroneous output as follows

Put:1
Got:1
Put:2
Put:3
Put:4
Got:2
Got:3
Put:5
Got:4
Got:5

To get the correct output, we have to use wait and notify methods.

```
class Q
{
 int n;
  boolean b=false;
 synchronized void get()
 {
  while(!b)
  {
   try
   {
    wait();
   }
   catch(InterruptedException e)
   {
    System.out.println(e);
   }
  }
  System.out.println("GOt: "+n);
  b=false;
  notify();
 }
 synchronized void put(int n)
 {
     while(b)
  {
   try
   {
    wait();
   }
   catch(InterruptedException e)
   {
```

```
        System.out.println(e);
        }
      }
        this.n=n;
        System.out.println("Put: "+n);
        b=true;
        notify();
  }
}
class Producer implements Runnable
{
  Thread t;
    Q q;
  Producer(Q q)
  {
    this.q=q;
    t=new Thread(this,"Producer");
    t.start();
  }
  public void run()
  {
    int i=1;
    while(i<=5)
    {
      q.put(i++);
    }
  }
}
class Consumer implements Runnable
{
  Thread t;
```

```
 Q q;
 Consumer(Q q)
 {
  this.q=q;
  t=new Thread(this,"Consumer");
  t.start();
 }
 public void run()
 {
  int i=1;
  while(i<=5)
  {
   q.get();
   i++;
   }
 }
}
class PC
{
 public static void main(String args[])
 {
  Q q=new Q();
  Producer p1=new Producer(q);
  Consumer c1=new Consumer(q);
 }
}
```

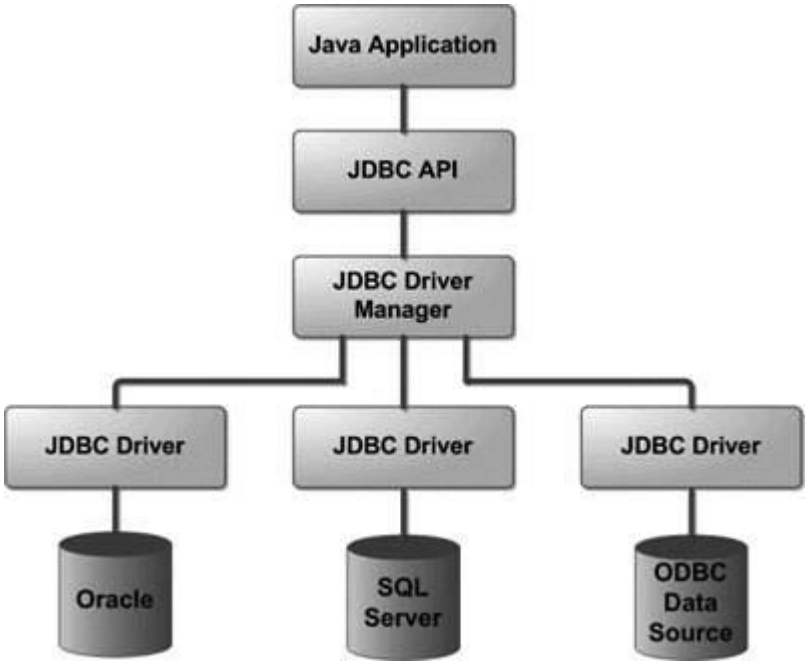
Java Database Connectivity (JDBC) is an **Application Programming Interface (API)** used to connect Java application with Database.

JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server.

JDBC can also be defined as the platform-independent interface between a relational database and Java programming. It allows java program to execute SQL statement and retrieve result from database.

The JDBC API consists of classes and methods that are used to perform various operations like: connect, read, write and store data in the database.

JDBC Architecture:



1. **Application:** Application in JDBC is a Java applet or a Servlet that communicates with a data source.

2. JDBC API: JDBC API provides classes, methods, and interfaces that allow Java programs to execute SQL statements and retrieve results from the database. Some important classes and interfaces defined in JDBC API are as follows:

- DriverManager
- Driver
- Connection
- Statement
- PreparedStatement

- CallableStatement
- ResultSet
- SQL data

3. Driver Manager: The Driver Manager plays an important role in the JDBC architecture. The Driver manager uses some database-specific drivers that effectively connect enterprise applications to databases.

4. JDBC drivers: JDBC drivers help us to communicate with a data source through JDBC. We need a JDBC driver that can intelligently interact with the respective data source.

Types of JDBC Drivers:

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
This is now discouraged because of thin driver.

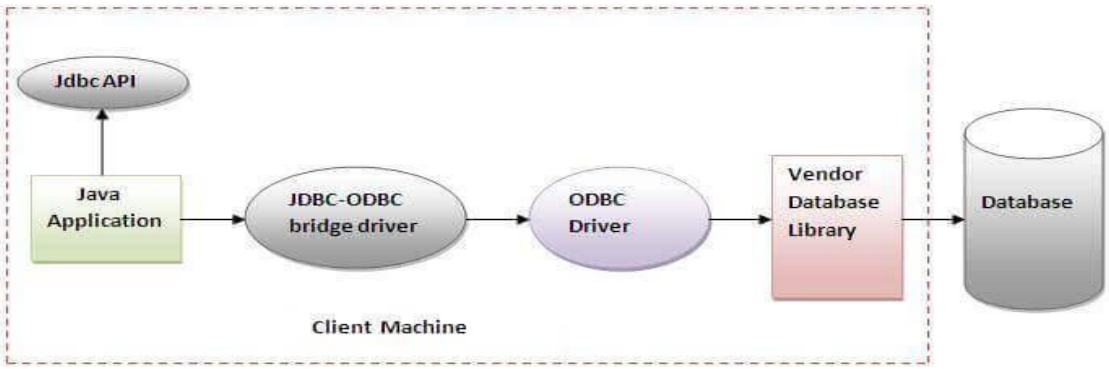


Figure- JDBC-ODBC Bridge Driver

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Native-API driver:

The Native API driver uses the client-side libraries of the database.
The driver converts JDBC method calls into native calls of the database API.
It is not written entirely in java.

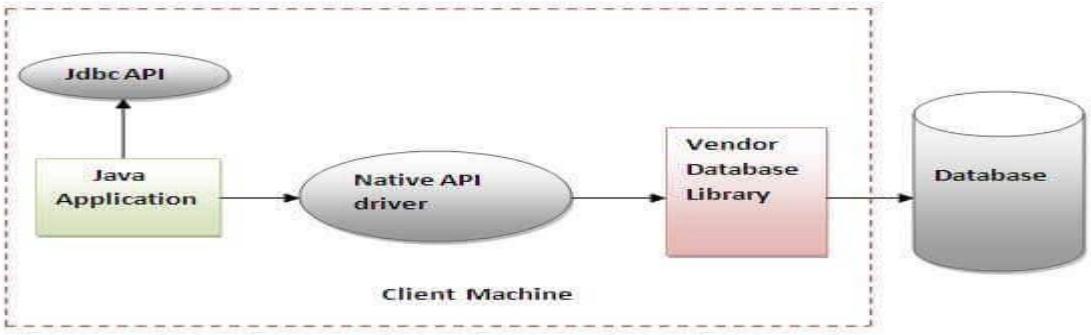


Figure- Native API Driver

Advantage:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

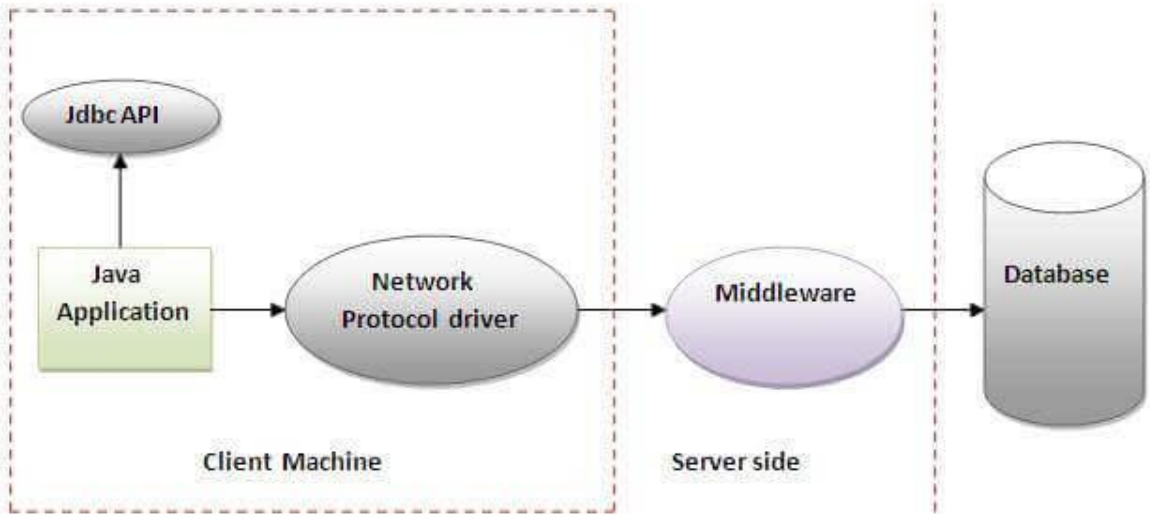


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin driver:

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

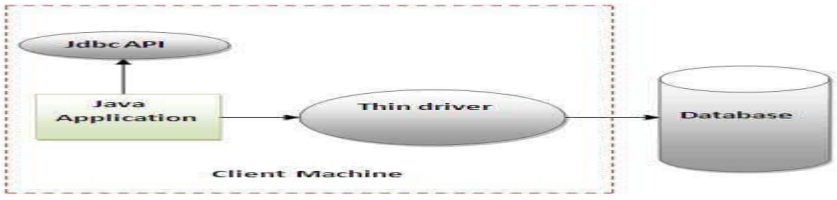


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

Steps to connect JAVA program to a database:

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

1. Register the Driver
2. Create a Connection
3. Create SQL Statement
4. Execute SQL Statement
5. Closing the connection

Register the Driver:

It is first and essential part to create JDBC connection. JDBC API provides a method Class.forName() which is used to load the driver class explicitly.

The Driver Class for oracle database is **oracle.jdbc.driver.OracleDriver** and

Class.forName("oracle.jdbc.driver.OracleDriver") method is used to load the driver class for Oracle database.

Create a Connection:

After registering and loading the driver in step1, now we will create a connection using getConnection() method of DriverManager class. This method has several overloaded methods that can be used based on the requirement. Basically it require the database name, username and password to establish connection.

getConnection(String url)

getConnection(String url, String username, String password)

getConnection(String url, Properties info)

The Connection URL for Oracle is


```
Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","username","password"
);
```

Create SQL Statement:

In this step we will create statement object using createStatement() method. It is used to execute the sql queries and defined in Connection class. Syntax of the method is given below.

Syntax

public Statement creates

atement() throws SQLException

Example to create a SQL statement

```
Statement s=con.createStatement();
```

Execute SQL Statement:

After creating statement, we have to execute SQL statements by using the following methods. These methods are provided by Statement Interface.

Syntax

public boolean execute(String sql) throws SQLException—This method is used to execute DDL commands.

public int executeUpdate(String sql) throws SQLException---This method is used to execute DML commands. This method returns an integer value which represents number of rows are effected or updated.

public ResultSet executeQuery(String query) throws SQLException---This method is used to execute SELECT command and retrieve the data from database.

Closing the connection:

This is final step which includes closing all the connection that we opened in our previous steps. After executing SQL statement you need to close the connection and release the session. The close() method of Connection interface is used to close the connection.

Syntax

public void close() throws SQLException

Example of closing a connection

con.close();

Write a program to check whether the connection with database is established or not.

```
import java.sql.*;
class JDBCDemo
{
   public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
      Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","
abc123456");
    if(con==null)
      System.out.println("Connection is not established");
   else
       System.out.println("Connection is established");
   }
}
```

Write a program to create a table in oracle data base.

```
import java.sql.*;
class JDBCDemo
{
   public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","
abc123456");
       Statement stmt=con.createStatement();
       String s1="create table student(id int,name varchar(20),marks int)";
      stmt.execute(s1);
     System.out.println("Table is created");
    con.close();
   }
}
```

Inserting records in oracle database table:

Write a program to insert a record in a table.

```
 import java.sql.*;
class InsertDemo
{
   public static void main(String args[]) throws Exception
```

```
  {
     Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","
abc123456");
      Statement stmt=con.createStatement();
      String s1="insert into student values(1,'abc',75)";
     int n=stmt.executeUpdate(s1);
    System.out.println(n+" row inserted");
    con.close();
  }
}
```

PreparedStatement:

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement.

Syntax:

public PreparedStatement prepareStatement(String query)**throws** SQLException

Write a program to insert multiple records in a table

```
import java.sql.*;
import java.util.*;
class InsertDemo
{
  public static void main(String args[]) throws Exception
  {
     Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
     PreparedStatement stmt=con.prepareStatement("insert into student values(?,?,?)");
    Scanner sc=new Scanner(System.in);
    for(int i=1;i<=5;i++)
```

```
   {
      System.out.println("enter id");
      int id=sc.nextInt();
       System.out.println("enter name");
      String name=sc.next();
      System.out.println("enter marks");
       int m=sc.nextInt();
       stmt.setInt(1,id);
       stmt.setString(2,name);
       stmt.setInt(3,m);
       stmt.executeUpdate();
    }
  con.close();
   }
}
```

Updating records in a table:

```
import java.sql.*;
class UpdateDemo
{
    public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
      Statement stmt=con.createStatement();
      String s1="update student set marks=100 where id=1";
      int n=stmt.executeUpdate(s1);
      System.out.println(n+" row is updated");
      con.close();
   }
}
```

```
import java.sql.*;
class UpdateDemo
{
   public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
     PreparedStatement stmt=con.prepareStatement("update student set marks=? where
id=?");
       stmt.setInt(1,100);
       stmt.setInt(2,2);
      int n=stmt.executeUpdate();
     System.out.println(n+" row is updated");
     con.close();
   }
}
```

Deleting a record in a table:

```
import java.sql.*;
class DeleteDemo
{
    public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
      Statement stmt=con.createStatement();
      String s1="delete from student where id=1"
      int n=stmt.executeUpdate(s1);
      System.out.println(n+" row is deleted");
     con.close();
   }}
```

ResultSet interface:

The result of the query after execution of database statement is returned as table of data according to rows and columns. This data is accessed using the **ResultSet** interface.

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

we can make this object to move forward and backward direction and object as updatable by using two constants in createStatement() method as follows.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_ISENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface:

) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10)public String getString(String olumnName):	is used to return the data of specified column name of the current row as String.

Example:

```
import java.sql.*;

class ResultDemo

{

    public static void main(String args[]) throws Exception

    {

       Class.forName(“oracle.jdbc.driver.OracleDriver”);

      Connection
con=DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:XE”,”SYSTEM”,”abc123456”);

      Statement
stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);

     String s1=”select * from student”;

    ResultSet rs=stmt.executeQuery(s1);

    while(rs.next())

    {

       System.out.println(rs.getInt(1)+” “+rs.getString(2)+” “+rs.getInt(3));

    }

   con.close();

   }

}
```