Unit-IV

Multithreading

Multitasking

- Executing multiple tasks at a time is called Multi tasking.

(Or)

- Ability to execute more than one task at the same time is known as multitasking.

- There are two types of Multi tasking
  - Process based multitasking (Multiprocessing)
  - Thread based multitasking(Multithreading)

Process based multitasking

- In process based multitasking two or more processes and programs can be run concurrently.

- In process based multitasking a process or a program is the smallest unit.

Example:

- We can listen to music and browse internet at the same time. The processes in this example are the music player and browser.

Thread based multitasking

- In thread based multitasking two or more threads can be run concurrently.
- In thread based multitasking a thread is the smallest unit.

Example:

- While you are typing, multiple threads are used to display your document, asynchronously check the spelling and grammar of your document, generate a PDF version of the document.

Thread definitions

- A thread is a independent flow of execution in a program.

  or

- A thread is a part of the program.

  or

- Thread is a tiny program. It is sometimes called as light-weight process.

  or

- Thread class is defined in **java.lang** package

Multithreading v/s Multiprocessing

S.No	Multithreading	Multiprocessing
1	Thread is a fundamental unit of multithreading.	Process/Program is a fundamental unit of multiprocessing.
2	Multiple parts of single program gets executed in multithreading environment.	Multiple programs get executed in multiprocessing environment.
3	During multithreading the processor switches between multiple threads in the program.	During multiprocessing the processor switches between multiple programs/processes.
4	Cost effective because CPU can be shared among multiple threads at a time.	Expensive, because when a process uses CPU other process has to wait.
5	Highly efficient	Less efficient.
6	Develops efficient application programs	Develops efficient OS programs.

Defining Instantiating, Starting the Thread

We can define instantiate and starting a thread by using the following 2-ways.

- By extending **Thread** Class.
- By implementing **Runnable** interface.

By extending Thread Class

- We can create a thread by creating a child class to the **Thread** class.
- And we should override a method i.e, **run().**
- We can define a thread job inside **run().**
- By calling Thread class **start()** method we can start execution of a thread

Example:

```
class Mythread extends Thread
{
     @Override
     public void run()
    {

         for (int i=0;i<10;i++)
         {
              System.out.println("Thread is executing");
         }
     }
}
```

Thread Scheduler:

- If multiple threads are there then which thread will get chance first for execution will be decided by “**Thread Scheduler**”.

- Thread Scheduler is the post of JVM.

- The behavior of thread scheduler is vendor dependent and hence we can’t expect exact O/P for the program.

Difference between t.start() & t.run()

- In the case of t.start() a new thread will be created and which is responsible for the execution of run().

- But in the case of t.run() no new thread will be created and run() method will be executed just like a normal method by the main thread.

Importance of Thread Class start() method

- After Creating thread object compulsory we should perform registration with in the Thread scheduler.
- This will take care by start() of Thread class, So that the programmers has to concentrate on only job.
- With out executing Thread class start() method there is no chance of start a new Thread in java.

start()
{
- Register our thread with in the thread scheduler.
- Invoke run() method.

}

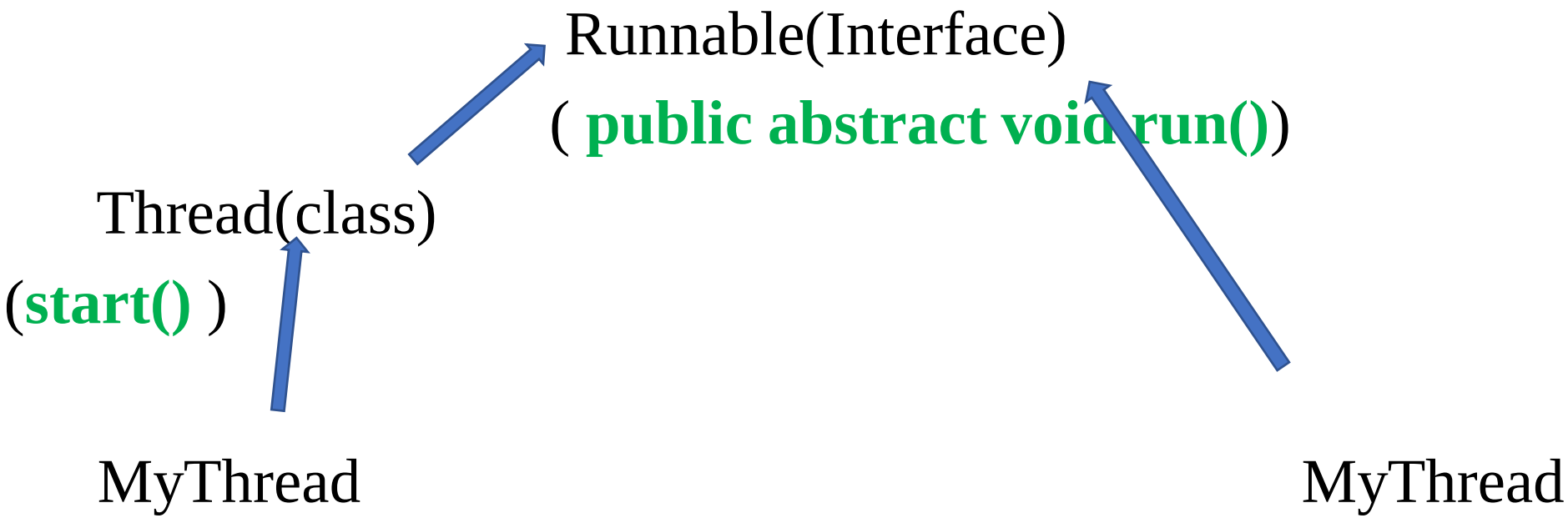
By Implementing Runnable Interface

- A Thread can be created by extending Thread class also. But Java allows only one class to extend, it wont allow multiple inheritance.

- So it is always better to create a thread by implementing Runnable interface. Java allows you to implement multiple interfaces at a time.

- By implementing Runnable interface, you need to provide implementation for run() method.

Relation between Runnable interface and Thread class



- To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method.

- Implementing Runnable interface does not create a Thread object, it only defines an entry point for threads in your object. It allows you to pass the object to the Thread(Runnable implementation) constructor.

Example:

```
class MyRunnableThread implements Runnable
{
    @Override
    public void run() {
        System.out.println("Runnable thread job");
    }
}

public class ThreadDemo
{
    public static void main(String[] args)
    {
        MyRunnableThread mythread=new MyRunnableThread();
        Thread t=new Thread(mythread);
        t.start();
    }
}
```

Thread lifecycle states

- A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java.
  - New
  - Runnable
  - Running
  - Non-Runnable (Blocked)
  - Terminated

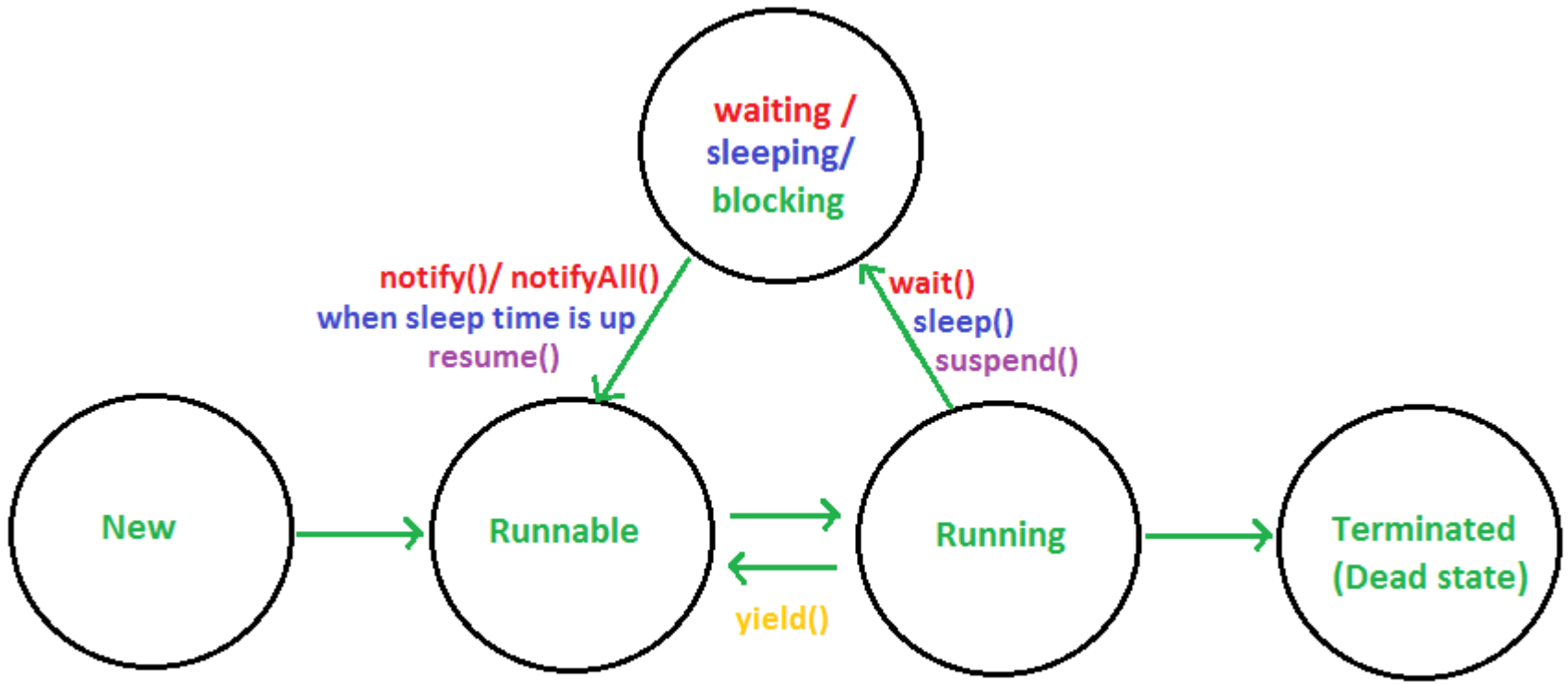


Fig. THREAD STATES

- **New:**

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

- **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.

- **Running:**

The thread is in running state if the thread scheduler has selected it.

- **Non-Runnable (Blocked):**

This is the state when the thread is still alive, but is currently not eligible to run.

- **Terminated:**

A thread is in terminated or dead state when its run() method exits.

Thread priorities

- Every Thread in java has some property.
- It may be default priority provided by the JVM or customized priority provided by the programmer.

- The valid range of thread priorities is 1 – 10. Where 1 is lowest priority and 10 is highest priority.

- The default priority of **main thread is 5**. The priority of child thread is inherited from the parent.

- Thread Scheduler will use priorities while allocating processor the thread which is having highest priority will get chance first and the thread which is having low priority.

- If two threads having the **same priority** then we can’t expect exact execution order it depends upon Thread Scheduler.

- The thread which is having **low priority** has to wait until completion of high priority threads.

Three constant values for the thread priority.

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5
- MAX_PRIORITY = 10

- Thread class defines the following methods to get and set priority of a Thread.
  - public final int getPriority()
  - public final void setPriority(int priority)

- Here ‘priority’ indicates a number which is in the allowed range of 1 – 10.

- Otherwise we will get Runtime exception saying “IllegalArgumentException”.

  - Ex: t.setPriority(11); // IllegalArgumentException

Thread class methods

sleep(long millis)

public static void sleep(long millis) throws InterruptedException

- Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
- The thread does not lose ownership of any monitors.

Parameters:

- millis - the length of time to sleep in milliseconds

Throws:

- **IllegalArgumentException :** If the value of millis is negative
- **InterruptedException :** If any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Example : sleep()

```
public class SleepDemo
{
public static void main(String[] args)
    {
        Child c=new Child();
        c.start();

    }
}
```

```
class Child extends Thread
{
    @Override
    public void run()

 {
        super.run();
        for (int i=0;i<10;i++)
        {
            System.out.println("Hello MLRIT
:"+i);
            try {
                sleep(1000);
            } catch (InterruptedException e)

{
                e.printStackTrace();
            }
        }
    }
}
```

Output:

Hello MLRIT :0

Hello MLRIT :1

…

Hello MLRIT :9

activeCount():

- This method is used to find out the number of threads in active state.

Synatx:

```
public static int activeCount();
```

- By default active count prints 2.Which means two threads will execute always i.e Main,Monitor

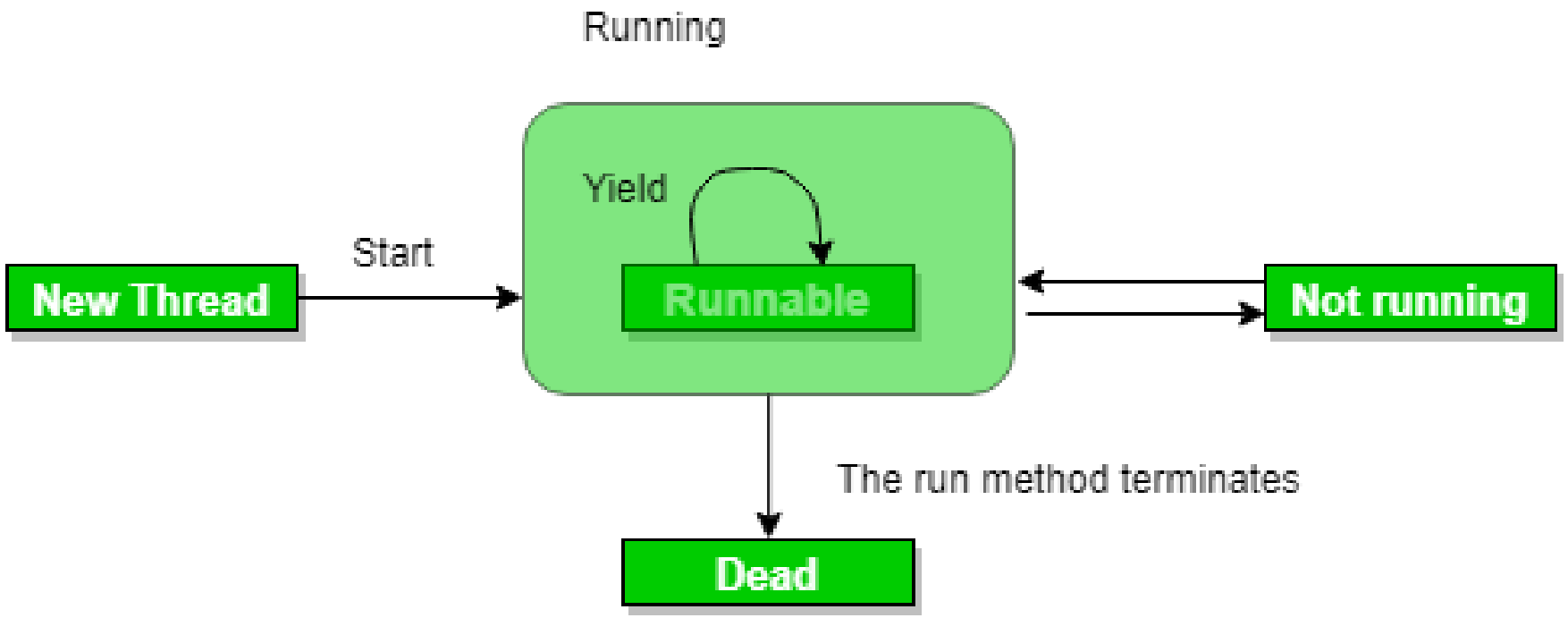
```
import java.util.Set;
public class ActiveThreadDemo
{
   public static void main(String[] args)
   {
      System.out.println(Thread.activeCount());
      Aa a=new Aa();
      a.setName("Ram");
      a.start();
      System.out.println("After starting A thread"+Thread.activeCount());
      B b=new B();
      b.setName("Bheem");
      b.start();
      System.out.println("After starting B thread"+Thread.activeCount());

      Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
      for ( Thread t : threadSet){
         if ( t.getThreadGroup() == Thread.currentThread().getThreadGroup()){
            System.out.println("Thread :"+t+":"+"state:"+t.getState());
         }
      }
   }
}
```

```
class Aa extends Thread
{
public void run()
   {
try {
         Thread.sleep(20000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
class B extends Thread
{
public void run() {
            try {
         Thread.sleep(20000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

yield():

- Suppose there are three threads t1, t2, and t3.
- Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state.
- The completion time for thread t1 is 5 hours and the completion time for t2 is 5 minutes.
- Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job.
- In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent the execution of a thread in between if something important is pending.


Running
Yield
Start
New Thread
Runnable
Not running
The run method terminates
Dead

- Whenever a thread calls java.lang.Thread.yield method gives hint to the thread scheduler that it is ready to pause its execution.

- The thread scheduler is free to ignore this hint.

- If any thread executes the yield method, the thread scheduler checks if there is any thread with the same or high priority as this thread.

- If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing.

- Once a thread has executed the yield method and there are many threads with the same priority is waiting for the processor, then we can’t specify which thread will get the execution chance first.

- The thread which executes the yield method will enter in the Runnable state from Running state.

- Once a thread pauses its execution, we can’t specify when it will get a chance again it depends on the thread scheduler.

- The underlying platform must provide support for preemptive scheduling if we are using the yield method.

join():

If a Thread wants to wait until completing some other thread then we should go for join() method.

- ❖public final void join() **throws** InterruptedExcetion
- ❖public final void join(long ms) **throws** InterruptedException

```java
public class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyThread t1=new MyThread();
        t1.setName("Sita");
        t1.start();
        t1.join();
        MyThread t2=new MyThread();
        t2.setName("Rama");
        t2.start();
        t2.join();
    }
}
class MyThread extends Thread

{
    @Override
    public void run() {
        for (int i=0;i<10;i++)
        {
            System.out.println(Thread.currentThread().getName()+" is executing..");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

Output:

Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Sita is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..
Rama is executing..

isAlive() :

used to check whether the thread is live or not.

- ❖public Boolean isAlive();

currentThread():

- ➢This method is used to represent current thread class object.
  - ❖public static thread currentThread();

isAlive() :

```
public class ThreadDemo
{
    public static void main(String[] args) throws InterruptedException {
        MyThread2 t=new MyThread2();
        System.out.println("t1 Thread state:"+t.isAlive());
        t.start();
        System.out.println("t1 Thread state:"+t.isAlive());
        Thread.sleep(20000);
        System.out.println("t1 Thread state:"+t.isAlive());
    }
}
class MyThread2 extends Thread
{
    @Override
    public void run() {
        for (int i=0;i<10;i++)
        {
            System.out.println("Hello..");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

interrupted():

- A thread can interrupt another sleeping or waiting thread.
- For this Thread class defines interrupt() method.
  - public void interrupt();

Note:

- The interrupt() is working good whenever our thread enters into waiting state or sleeping state.
- The interrupted call will be wasted if our thread doesn’t enters into the waiting/sleeping state.

Synchronizing threads

- **Synchronized** modifier is the modifier applicable only for methods and blocks but not for classes and variables.

- If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.

- The main advantage of synchronized modifier is we can resolve data inconsistency problems.

- But the main disadvantage of synchronized modifier is it increases the waiting time of the Thread and effects performance of the system.

- Hence if there is no specific requirement it is never recommended to use.

- The main purpose of this modifier is to reduce the data inconsistence problems.

Example:

```
public class SyncDemo
{
    public static void main(String[]
args) {
        Whish w=new Whish();
        Mt t1=new Mt("Purushotham",w);
        Mt t2=new Mt("Naresh",w);
        t1.start();
        t2.start();
    }
}
class Whish
{
    public synchronized void
whish(String name) throws
InterruptedException {
        for (int i=0;i<10;i++)
        {
            System.out.println("Good
mrng.."+name);
            Thread.sleep(2000);
        }
    }
}
```

```
class Mt extends Thread
{
    Whish w;
    String name;
    Mt(String name,Whish w)
    {
        this.name=name;
        this.w=w;
    }

    @Override
    public void run() {
        super.run();
        try {
            w.whish(name);
        } catch (InterruptedException e)
{
            e.printStackTrace();
        }
    }
}
```

Inter Thread Communication

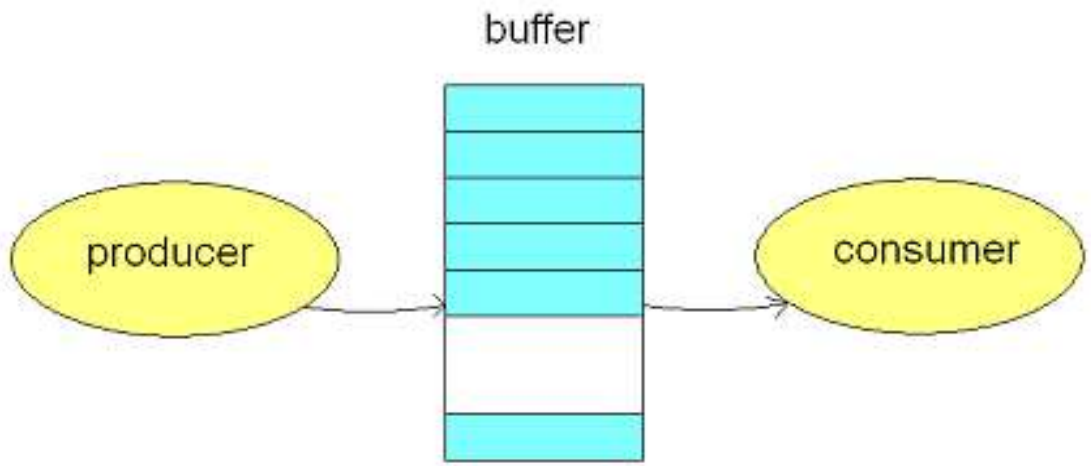
- Two threads can communicate with each other by using
  - wait(),
  - notify(),
  - notifyAll()

- These methods are available in **Object** class but not in Thread class. Because threads are calling these methods on any object.

- We should call these methods only from synchronized area other wise we will get runtime exception saying **IllegalMonitorStateException.**

Producer Consumer Problem

- The producer-consumer problem is a classic example of a multi-process synchronization problem.
- There are two processes, a producer and a consumer, that share a common buffer with a limited size.
- The producer “produces” data and stores it in the buffer, and the consumer “consumes” the data, removing it from the buffer.
- Having two processes that run in parallel, we need to make sure that the producer will not put new data in the buffer when the buffer is full.
- Consumer won’t try to remove data from the buffer if the buffer is empty.


buffer
producer
consumer

Producer Consumer Program:

```
import java.util.LinkedList;
public class ProducerConsumer
{
    public static void main(String[] args) throws InterruptedException {
        PC p=new PC();
        Thread t1=new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    p.produce();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

            }
        });
        Thread t2=new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    p.consumer();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();

}
```

```
class PC
{
    LinkedList<Integer> linkedList=new LinkedList<Integer>();
    int capacity=1;

    public synchronized   void produce() throws InterruptedException
    {
        int value=0;

       while (true)
       {
               while (linkedList.size()==capacity)
               {
                   wait();
               }
               System.out.println("Producer produced item-"+value);
               linkedList.add(value++);
               notify();
               Thread.sleep(3000);
       }
    }
    public synchronized void consumer() throws InterruptedException
    {
        while (true)
        {
                while (linkedList.size()==0) {
                    wait();
                }
                int val=linkedList.removeFirst();
                System.out.println("Consumer consumed item-"+val);
                notify();
                Thread.sleep(3000);
        }

}}
```

Daemon Threads

- Daemon thread in Java is a low-priority thread that performs background operations such as garbage collection, finalizer, Action Listeners, Signal dispatches, etc.

- Daemon thread in Java is also a service provider thread that helps the user thread.

- Its life is at the mercy of user threads i.e, when all user threads expire, JVM immediately terminates this thread.

- Usually daemon thread are running with low priority but based on our requirement we can increase their priority also.

- We can check whether the given thread is daemon or not by using the following thread class thread.
  - **public boolean isDaemon();**

- we can change the daemon nature of a thread by using setDaemon() method of thread class.
  - **public void setDaemon(Boolean b);**

Note:

IllegalThreadStateException:

- If you call the setDaemon() method after the thread has started, it will throw an exception.

Daemon thread example

```
class MyThread extends Thread
{

}
public class Daemon
 {
    public static void main(String[] args)
    {
        MyThread t1=new MyThread();
        System.out.println(t1.isDaemon());
        t1.setDaemon(true);
        System.out.println(t1.isDaemon());
    }
}
```

Output:

fasle

true

JDBC
(Java Database Connectivity)

Storage areas or options

Storage areas or options

- As the part of our Applications, we required to store our data like customers information, Billing Information, Calls Information etc..
- To store this Data, we required Storage Areas. There are 2 types of Storage Areas.
  - Temporary Storage Areas
  - Permanent Storage Areas

Temporary Storage Areas:

These are the Memory Areas where Data will be stored temporarily.

Ex: All JVM Memory Areas (like Heap Area, Method Area, Stack Area etc).
Once JVM shutdown all these Memory Areas will be cleared automatically.

Permanent Storage Areas:

- Also known as Persistent Storage Areas.
- Here we can store Data permanently.
  - Ex: File Systems, Databases, Data warehouses, Big Data Technologies

File Systems:

- File Systems can be stored unstructured data.
- File Systems can be provided by Local operating System.
- File Systems are best suitable to store very less Amount of Information.

Limitations:

- We cannot store huge amount of data.
- There is no Query Language support and hence operations will become very complex.
- There is no security for the data.
- There is no mechanism to prevent duplicate data. Hence there may be a chance of data inconsistency problems.
- To overcome the above problems of File Systems, we should recommended to use Databases.

Databases:

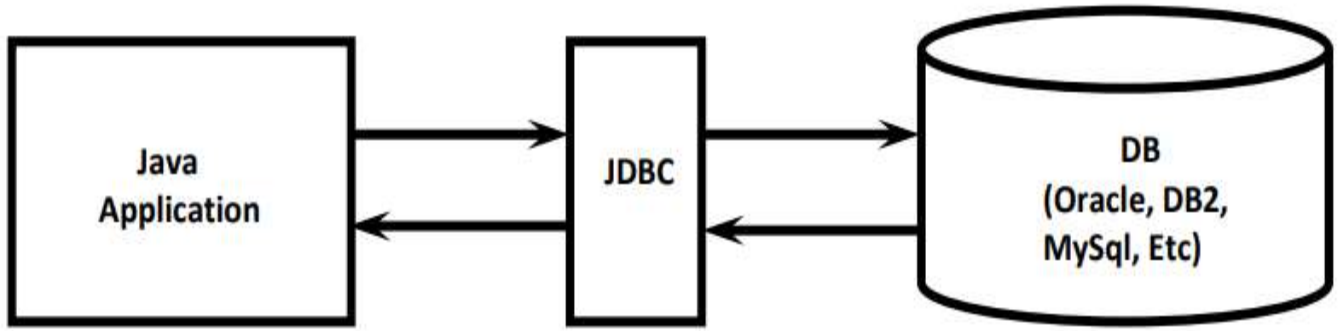
- We can store huge amount of data in the Databases.
- Query language support is available for every Database and hence we can perform Database operations very easily.
- To access data present in the Database, compulsory username and pwd must be required. Hence data is secured.
- Inside Database data will be stored in the form of Tables. While developing database table schemas, Database admin follow various normalization techniques and can implement various constraints like unique key constrains, primary key constraints etc which prevent data duplication.
- Hence there is no chance of Data Inconsistency Problems.

Limitations of Databases:

- Database cannot hold very huge amount of information like terabytes of Data.
- Database can provide support only for structured data (Tabular Data OR Relational Data) and cannot provide support for semi structured data (like XML Files) and Unstructured Data (like Video Files, Audio Files, Images etc)
- To overcome these problems we should go for more advanced storage areas like Big Data Technologies, Data warehouses etc..

JDBC

- JDBC is a Technology, which can be used to communicate with Database from Java Application.



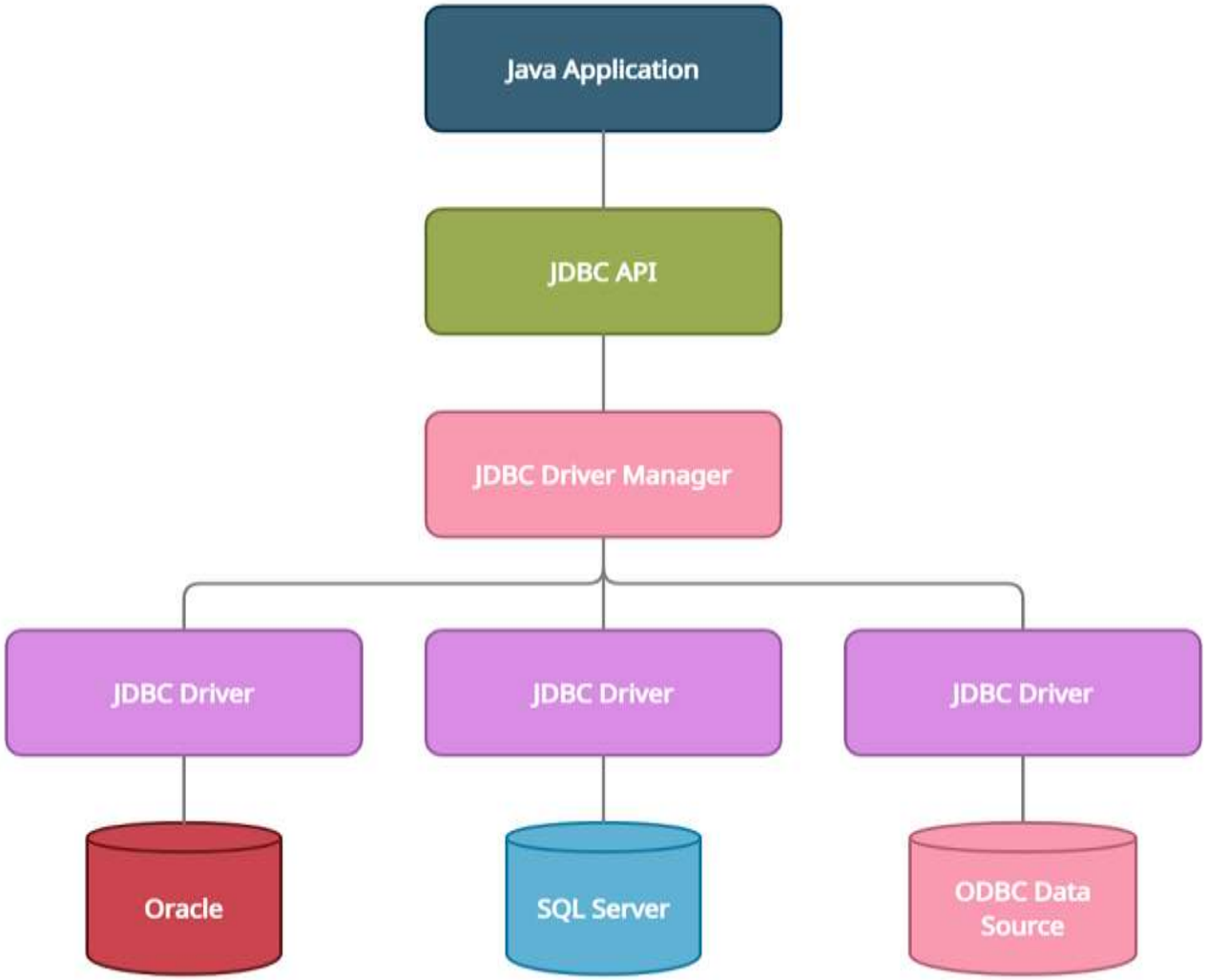
- JDBC is the Part of Java Standard Edition.
- JDBC is a Specification defined by Java Vendor (Sun Micro Systems) and implemented by Database Vendors.
- Database Vendor provided Implementation is called "Driver Software".

JDBC Features:

- JDBC API is standard API. We can communicate with any Database without rewriting our Application i.e. it is Database independent API.
- JDBC Drivers are developed in Java and hence JDBC Concept is applicable for any Platform. i.e. JDBC is platform independent technology.
- By using JDBC API, we can perform basic CRUD operations very easily.
- We can also perform complex operations (like Inner Joins, Outer Joins, calling Stored Procedures etc) very easily by using JDBC API.
- JDBC API supported by large number of vendors and they developed multiple Products based on JDBC API.

JDBC Architecture

JDBC Architecture



JDBC Architecture

- **Application:** It is a java applet or a servlet that communicates with a data source.
- **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:
- **DriverManager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.
- **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

DriverManager:

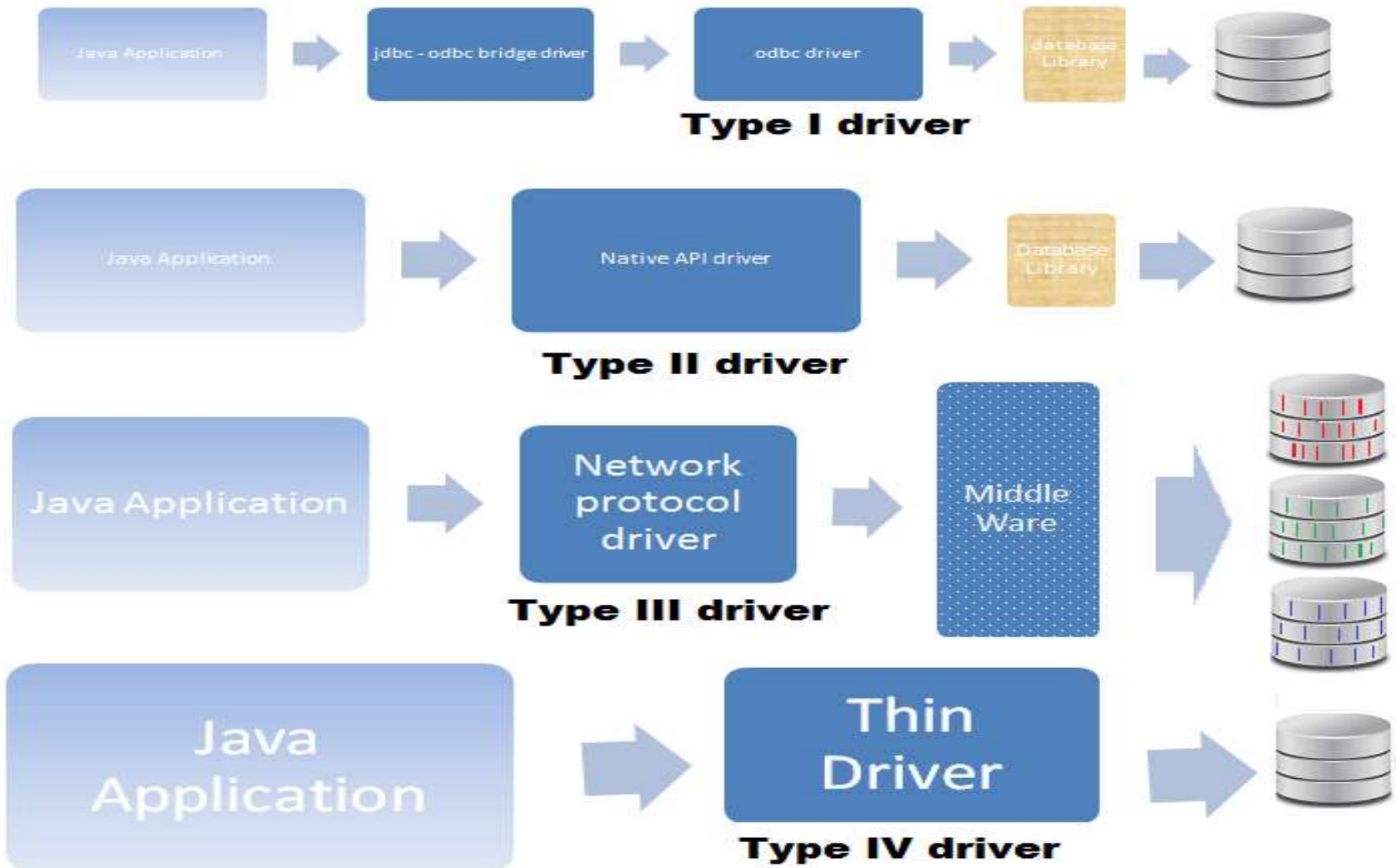
- It is the Key Component in JDBC Architecture.
- DriverManager is a Java Class present in **java.sql p**ackage.
- It is responsible to manage all Database drivers available in our system.
- DriverManager is responsible to register and unregister Database Drivers.
  - DriverManager.registerDriver(Driver);
  - DriverManager.unregisterDriver(Driver);
- DriverManager is responsible to establish connection to the Database with the help of Driver Software.
  - Connection con = DriverManager.getConnection (jdbcurl, username, pwd);

JDBC API

- JDBC API provides several Classes and Interfaces.
- Programmer can use these Classes and Interfaces to communicate with the Database.
- Driver Software Vendor can use JDBC API while developing Driver Software.
- JDBC API defines 2 Packages
  - java.sql Package:
  - javax.sql Package:

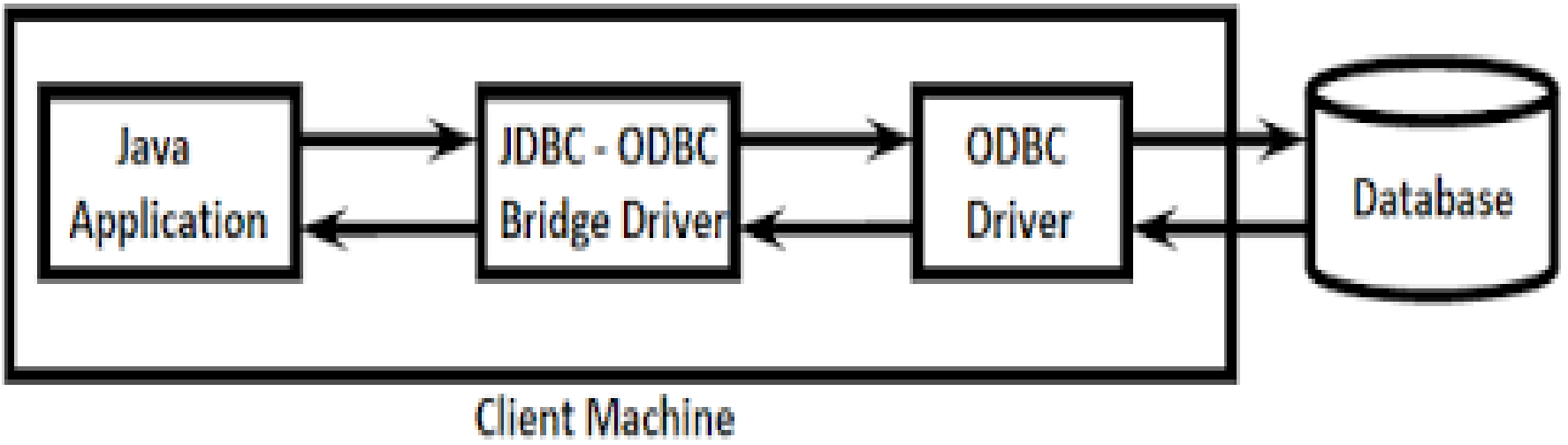
Types of drivers

- While communicating with Database, we have to convert Java calls into Database specific calls and Database specific calls into Java calls. For this driver software is required.
- There are many drivers are available. But based on functionality all drivers are divided into 4 Types.
  - Type-1 Driver (JDBC-ODBC Bridge Driver OR Bridge Driver)
  - Type-2 Driver (Native API-Partly Java Driver OR Native Driver)
  - Type-3 Driver (All Java Net Protocol Driver OR Network Protocol Driver OR Middleware Driver)
  - Type-4 Driver (All Java Native Protocol Driver OR Pure Java Driver OR Thin Driver)



Types of JDBC drivers

Type-1 Driver



Type-1 Driver:

- Type-1 driver provided by Sun Micro Systems as the part of JDK. But this Support is available until 1.7 version only.
- Internally this driver will take support of ODBC Driver to communicate with Database.
- Type-1 Driver converts JDBC Calls (Java Calls) into ODBC Calls and ODBC Driver converts ODBC calls into Database specific Calls.
- Hence Type-1 Driver acts as Bridge between JDBC and ODBC.

Advantages :

- It is very easy to use and maintain.
- We are not required to install any separate Software because it is available as the Part of JDK.
- Type-1 Driver won't communicates directly with the Database. Hence it is Database Independent Driver. Because of this migrating from one Database to another Database will become very easy.

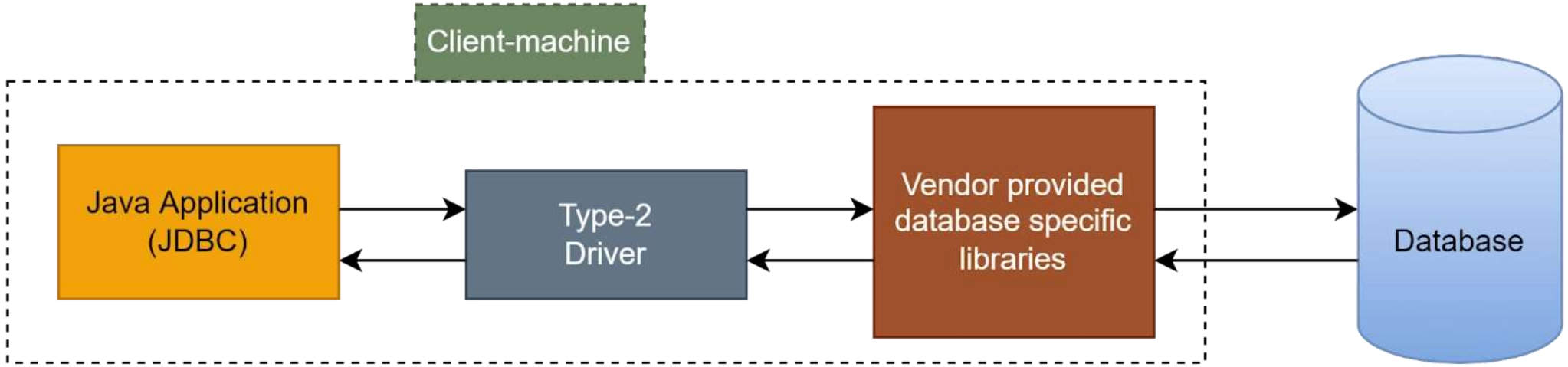
Limitations:

- It is the slowest Driver among all JDBC Drivers (Snail Driver), because first it will convert JDBC Calls into ODBC Calls and ODBC Driver converts ODBC Calls into Database specific Calls.

- This Driver internally depends on ODBC Driver, which will work only on Windows Machines. Hence Type-1 Driver is Platform Dependent Driver.

- No Support from JDK 1.8 Version onwards.

Type-2 Driver



Type-2 Driver

- It is also known as Native API -partly Java Driver OR Native Driver.
- Type-2 Driver is exactly same as Type-1 Driver except that ODBC Driver is replaced with vendor specific Native Libraries.
- Type-2 Driver internally uses vendor specific native libraries to communicate with Database.
- Native libraries means the set of Functions written in Non-Java (Mostly C OR C++).
- We have to install Vendor provided Native Libraries on the Client Machine.
- Type-2 Driver converts JDBC Calls into Vendor specific Native Library Calls, which can be understandable directly by Database Engine.

Advantages:

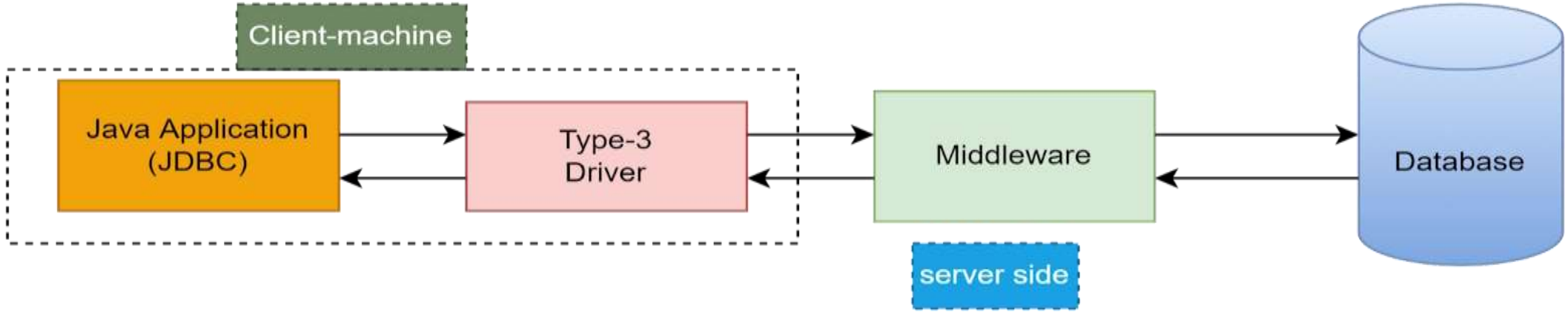
- When compared with Type-1 Driver Performance is High, because it required only one Level Conversion from JDBC to Native Library Calls.
- No need of arranging ODBC Drivers.
- When compared with Type-1 Driver, Portability is more because Type-1 Driver is applicable only for Windows Machines.

Limitations:

- Internally this Driver using Database specific Native Libraries and hence it is Database Dependent Driver. Because of this migrating from one Database to another Database will become Difficult.
- This Driver is Platform Dependent Driver.
- On the Client Machine compulsory we should install Database specific Native Libraries.
- There is no Guarantee for every Database Vendor will provide This Driver.
- (Oracle is providing Type-2 Driver but MySql won’t providing this Driver)

Type-3 Driver:

- Also known as All Java Net Protocol Driver OR Network Protocol Driver OR Middleware Driver.
- Type-3 Driver converts JDBC Calls into Middleware Server specific Calls. Middleware Server can convert Middleware Server specific Calls into Database specific Calls.
- Internally Middleware Server may use Type-1, 2 OR 4 Drivers to communicates with Database.



Advantages:

- This Driver won't communicate with Database directly and hence it is Database Independent Driver.
- This Driver is Platform Independent Driver.
- No need of ODBC Driver OR Vendor specific Native Libraries

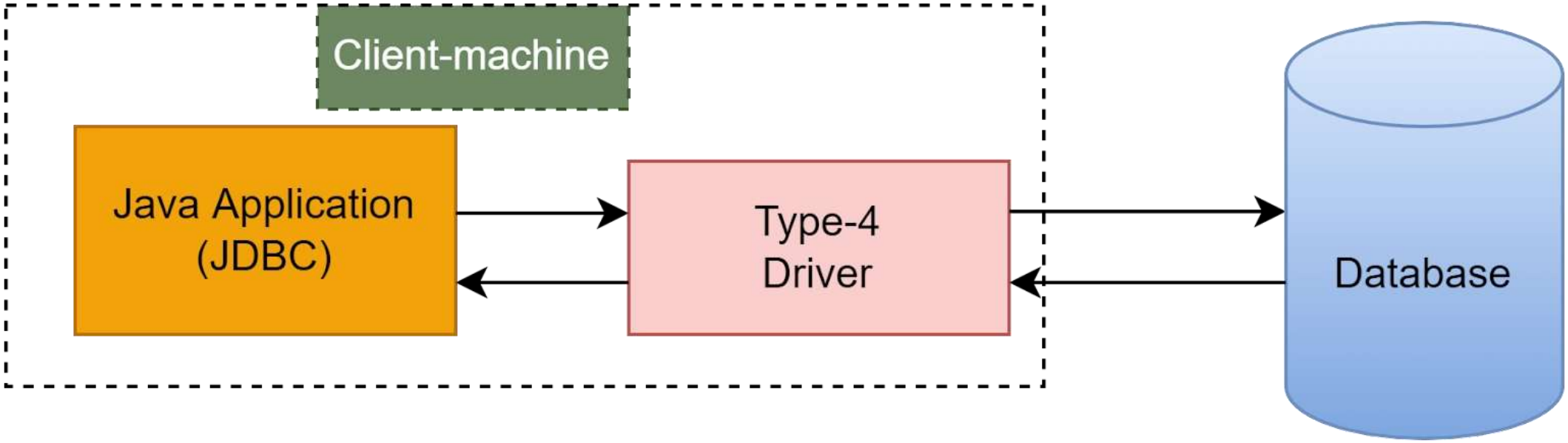
Limitations:

- Because of having Middleware Server in the Middle, there may be a chance of Performance Problems.
- We need to purchase Middleware Server and hence the cost of this Driver is more when compared with remaining Drivers.
  - Ex: IDS Driver (Internet Database Access Server)

Note: The only Driver which is both Platform Independent and Database Independent is Type-3 Driver. Hence it is recommended to use.

Type-4 Driver:

- Also known as Pure Java Driver OR Thin Driver.



- This Driver is developed to communicate with the Database directly without taking Support of ODBC Driver OR Vendor Specific Native Libraries OR Middleware Server.
- This Driver uses Database specific Native Protocols to communicate with the Database.
- This Driver converts JDBC Calls directly into Database specific Calls.
- This Driver developed only in Java and hence it is also known as Pure Java Driver.
- Because of this, Type-4 Driver is Platform Independent Driver. This Driver won't require any Native Libraries at Client side and hence it is light weighted. Because of this it is treated as Thin Driver.

Advantages

- It won't require any Native Libraries, ODBC Driver OR Middleware Server
- It is Platform Independent Driver
- It uses Database Vendor specific Native Protocol and hence Security is more.

Limitations:

The only Limitation of this Driver is, it is Database Dependent Driver because it is communicating with the Database directly.

Ex: Thin Driver for Oracle

Connector/J Driver for **MySQL**

JDBC Connection Steps

There are 5 steps to connect any java application with the database using JDBC.

These steps are as follows:

- ❖Register the Driver class
- ❖Create connection
- ❖Create statement
- ❖Execute queries
- ❖Close connection

Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax:

public static void forName(String className) **throws** ClassNotFoundException

Ex:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Create connection

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax:

public static Connection getConnection(String url) throws SQLException

public static Connection getConnection(String url,String name,String password) throws SQLException

Ex:

```
Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/mlrit","root","root");
```

createStatement() /prepareStatement()

The createStatement() or prepareStatement() method of Connection interface is used to create/prepare statement. The object of statement is responsible to execute queries with the database.

Syntax:

public Statement createStatement() **throws** SQLException

public PreparedStatement prepareStatement() **throws** SQLException

Ex:

```
PreparedStatement statement=con.prepareStatement(qry)
```

Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table

Syntax:

public ResultSet executeQuery(String sql) throws SQLException

public int executeUpdate(String sql) throws SQLException

close()

By closing connection object statement and ResultSet will be closed automatically.

The close() method of Connection interface is used to close the connection.