Idris Ratlamwala
C31 - 2003145

Java Assingment 2

a. Thread synchronization:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. 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.

As Java is a multi_threaded language, thread synchronization has a lot of importance in Java as multiple threads execute in parallel in an application.

We use keywords “synchronized” and “volatile” to achieve Synchronization in Java

We need synchronization when the shared object or resource is mutable. If the resource is immutable, then the threads will only read the resource either concurrently or individually.

In this case, we do not need to synchronize the resource. In this case, JVM ensures that Java synchronized code is executed by one thread at a time.

Most of the time, concurrent access to shared resources in Java may introduce errors like “Memory inconsistency” and “thread interference”. To avoid these errors we need to go for synchronization of shared resources so that the access to these resources is mutually exclusive.

We use a concept called Monitors to implement synchronization. A monitor can be accessed by only one thread at a time. When a thread gets the lock, then, we can say the thread has entered the monitor.

When a monitor is being accessed by a particular thread, the monitor is locked and all the other threads trying to enter the monitor are suspended until the accessing thread finishes and releases the lock.

Going forward, we will discuss synchronization in Java in detail in this tutorial. Now, let us discuss some basic concepts related to synchronization in Java.

Race Condition In Java
In a multithreaded environment, when more than one thread tries to access a shared resource for writing simultaneously, then multiple threads race each other to finish accessing the resource. This gives rise to 'race condition'.

One thing to consider is that there is no problem if multiple threads are trying to access a shared resource only for reading. The problem arises when multiple threads access the same resource at the same time.

Race conditions occur due to a lack of proper synchronization of threads in the program. When we properly synchronize the threads such that at a time only one thread will access the resource, and the race condition ceases to exist.

b. Abstract classes :

A class that is declared using "abstract" keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods.

Lets say we have a class Animal that has a method sound() and the subclasses(see inheritance) of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say "Roar" in this method and Dog class will say "Woof".

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Rules
1: As there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.
A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

c. JDBC Drivers and Architecture

The term JDBC stands for Java Database Connectivity. JDBC is a specification from Sun microsystems. JDBC is an API(Application programming interface) in Java that helps users to interact or communicate with various databases.

The classes and interfaces of JDBC API allow the application to send the request to the specified database.

Using JDBC, we can write programs required to access databases. JDBC and the database driver are capable of accessing databases and spreadsheets. JDBC API is also helpful in accessing the enterprise data stored in a relational database(RDB).

There are some enterprise applications created using the JAVA EE(Enterprise Edition) technology. These applications need to interact with databases to store application-specific information.

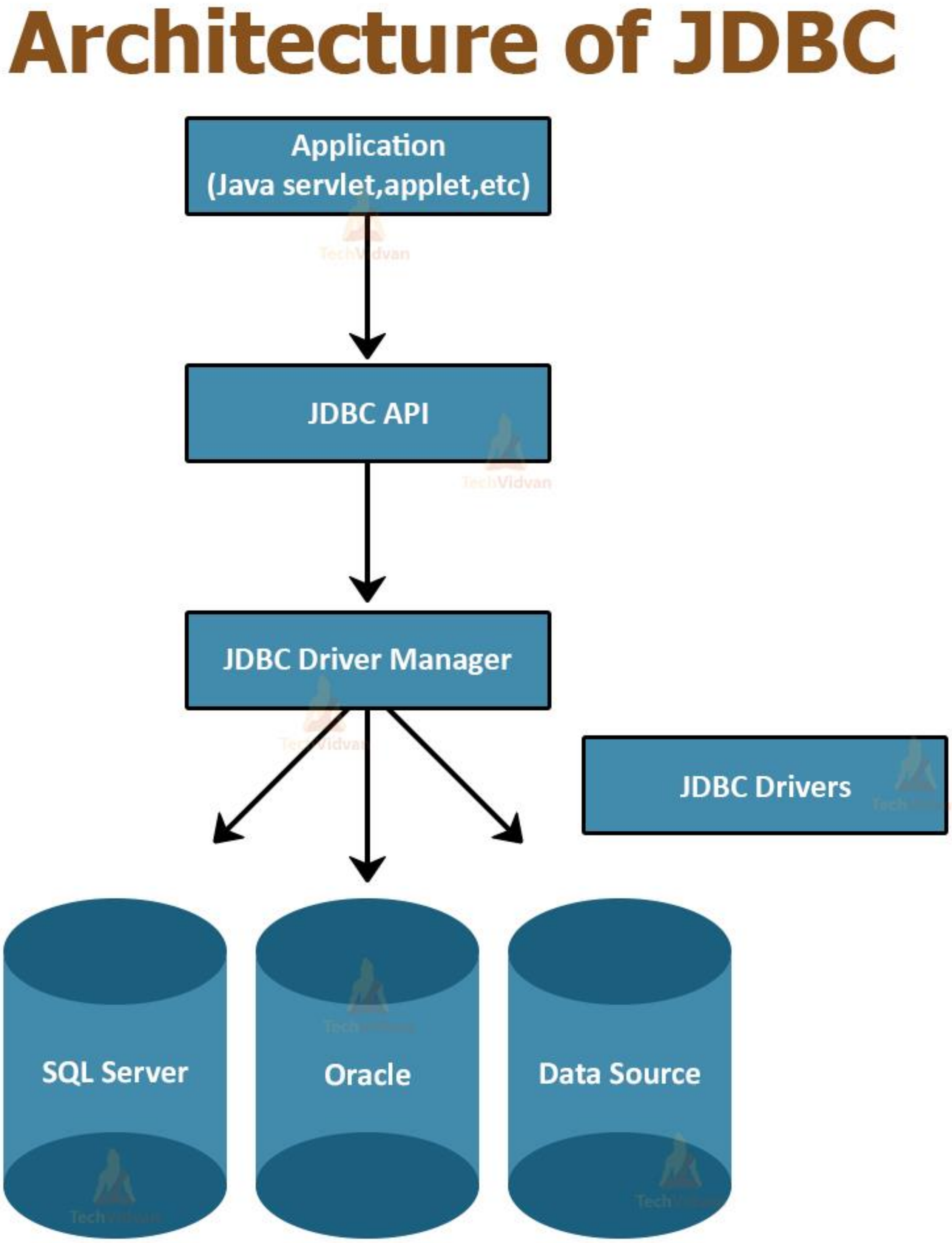
Interacting with the database requires efficient database connectivity, which we can achieve using ODBC(Open database connectivity) driver. We can use this ODBC Driver with the JDBC to interact or communicate with various kinds of databases, like Oracle, MS Access, Mysql, and SQL, etc.

Applications of JDBC
JDBC is fundamentally a specification that provides a complete set of interfaces. These interfaces allow for portable access to an underlying database.

We can use Java to write different types of executables, such as:

Java Applications
Java Applets
Enterprise JavaBeans (EJBs)
Java Servlets
Java ServerPages (JSPs)
All these different executables can use a JDBC driver to access a database, and take advantage of the stored data. JDBC provides similar capabilities as ODBC by allowing Java programs to contain database-independent code.

Architecture of JDBC
The following figure shows the JDBC architecture:



Description of the 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 Architecture

There are two types of processing models in JDBC architecture: two-tier and three-tier. These models help us to access a database. They are:

1. Two-tier model

In this model, a Java application directly communicates with the data source. JDBC driver provides communication between the application and the data source. When a user sends a query to the data source, the answers to those queries are given to the user in the form of results.

We can locate the data source on a different machine on a network to which a user is connected. This is called a client/server configuration, in which the user machine acts as a client, and the machine with the data source acts as a server.

2. Three-tier model

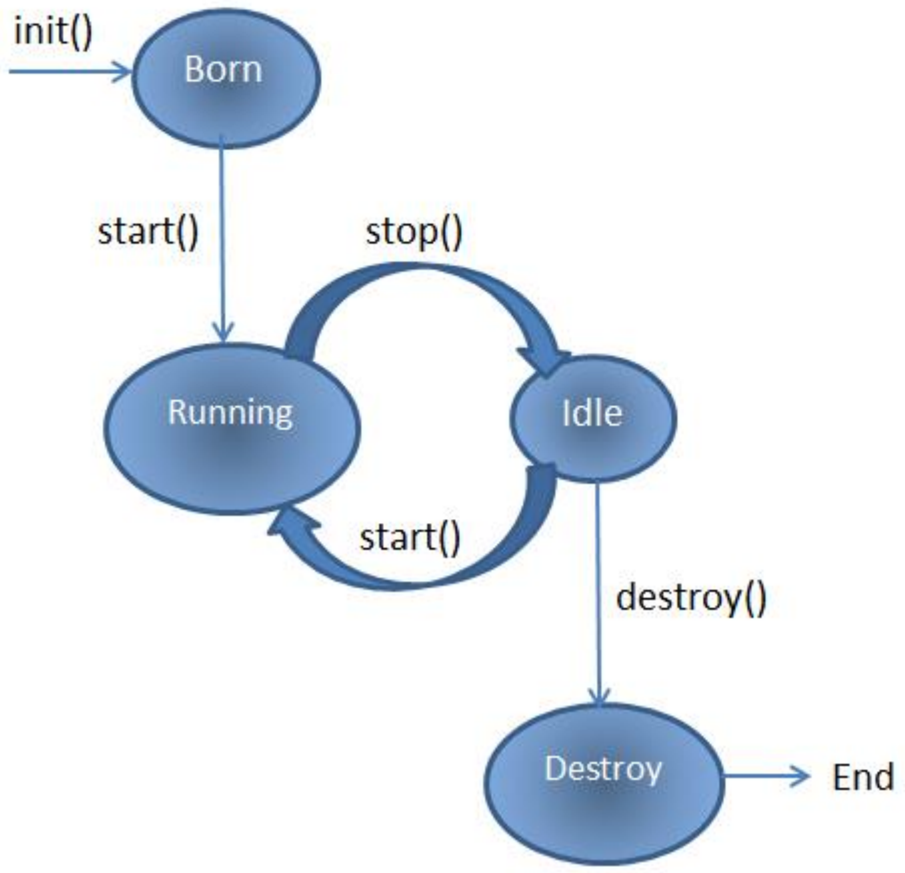
In the three-tier model, the query of the user queries goes to the middle-tier services. From the middle-tier service, the commands again reach the data source. The results of the query go back to the middle tier.

d. Life Cycle of an applet

Applets are small java programs that are primarily used in internet computing. It can be transported over the internet from one computer to another and run using the Applet Viewer or any web browser that supports Java.

An applet is like an application program, it can do many things for us. It can perform **arithmetic operations, display graphics, create animation & games, play sounds** and accept user input, etc. Every Java applet inherits a set of default behaviours from the Applet class. As a result, when an applet is loaded, it undergoes a series of changes in its states.

Applet Life Cycle:
Applet Life Cycle is defined as how the applet created, started, stopped and destroyed during the entire execution of the application. The methods to execute in the Applet Life Cycle are **init()**, **start()**, **stop()**, **destroy()**. The Applet Life Cycle Diagram is given as below:



- Born or Initialization State
- Running State
- Idle State
- Dead State
-

1. Born or Initialization State: Applets enters the initialization state when it is first loaded. It is achieved by **init()** method of Applet class. Then applet is born.

```
public void init()
{
............
............
(Action)
}
```

2. Running State: Applet enters the running state when the system calls the **start()** method of Applet class. This occurs automatically after the applet is initialized.

```
public void start()
{
............
............
(Action)
}
```

3. Idle State: An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. This is achieved by calling the **stop()** method explicitly.

```
public void stop()
{
............
............
(Action)
}
```

4. Dead State: An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the **destroy()** method when we want to quit the browser.

```
public void destroy()
{
............
............
(Action)
}
```