Question 1 a:

```
import java.net.InetAddress;

import java.net.UnknownHostException;


public class GetIPAddress {

   public static void main(String[] args) {

      try {

         InetAddress ip = InetAddress.getLocalHost();

         System.out.println("IP address of this machine is: " + ip.getHostAddress());

      } catch (UnknownHostException e) {

         System.out.println("Unable to get IP address of this machine.");

         e.printStackTrace();

      }

   }

}
```

Question 1 b:

```
import java.sql.*;

class UpdateDemo{

public static void main(String args[]){

try{ Class.forName("com.mysql.jdbc.Driver");

    Connection con=DriverManager.getConnection(

                    "jdbc:mysql://localhost:3306/GTU","root","root");

          Statement stmt=con.createStatement();

          ResultSet rs = stmt.executeQuery("SELECT  * from diet");

          while(rs.next())

          System.out.print(rs.getString(1));

          stmt.executeUpdate("Insert into student values("abc501",601));

              String query="update diet set Name='abc601' where Enr_no=601";
```

```
            int i=stmt.executeUpdate(query);

            System.out.println("total no. of rows updated="+i);

            stmt.close();

            con.close();

            }catch(Exception e){ System.out.println(e);}

} }
```

Question 1 c:

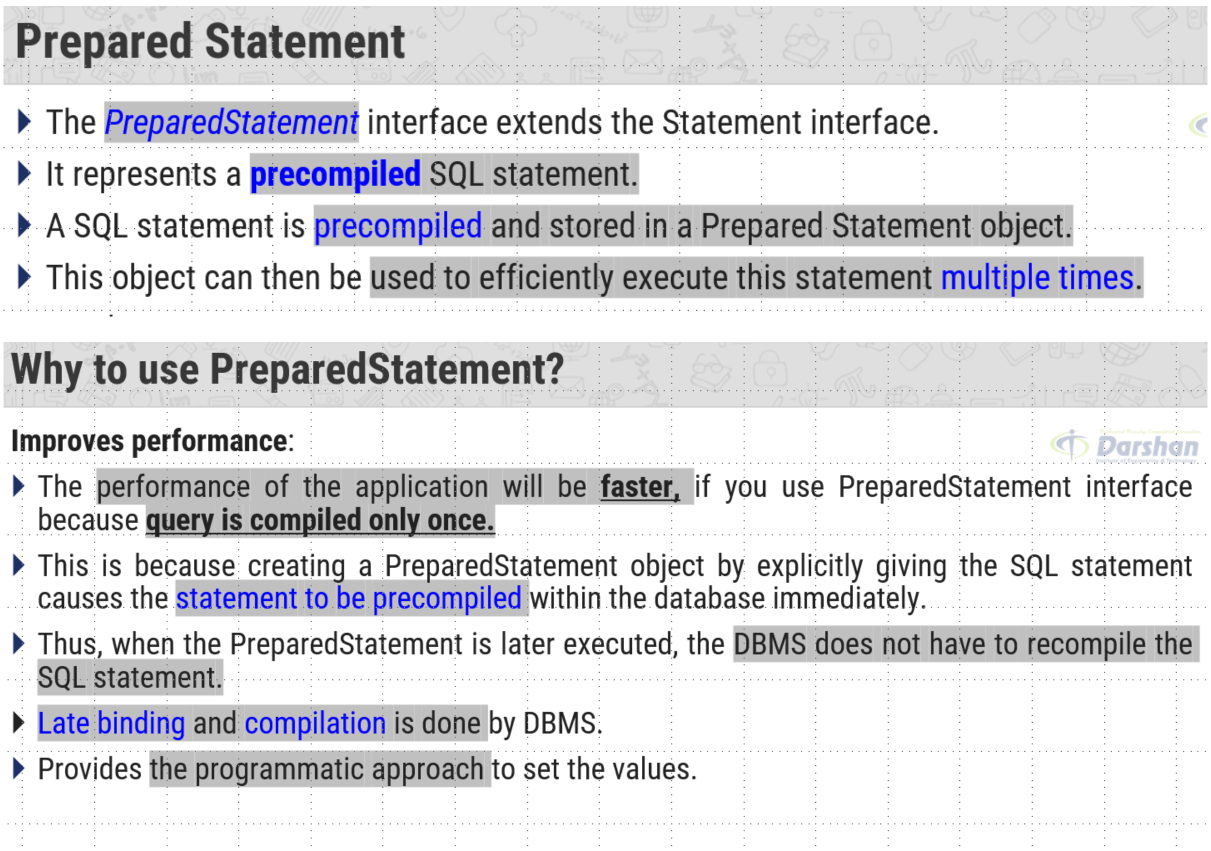
Server.java

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

public class Server {

   public static void main(String[] args) {
      ServerSocket serverSocket = null;

      try {

         serverSocket = new ServerSocket(5000);
         System.out.println("Server is listening on port 5000");

         Socket clientSocket = serverSocket.accept();
         System.out.println("Client connected: " +
clientSocket.getInetAddress().getHostName());

         BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
         PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
true);

         String input = in.readLine();

         String reversedInput = new StringBuilder(input).reverse().toString();
         if (input.equals(reversedInput)) {
            out.println(input + " is a palindrome");
         } else {
            out.println(input + " is not a palindrome");
         }

         in.close();
         out.close();
         clientSocket.close();
         serverSocket.close();

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

Client.java

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

public class Client {

   public static void main(String[] args) {
      Socket socket = null;
      BufferedReader in = null;
      PrintWriter out = null;

      try {

         socket = new Socket("localhost", 5000);

         in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
         out = new PrintWriter(socket.getOutputStream(), true);

         BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in));
         System.out.print("Enter a string: ");
         String input = userInput.readLine();

         out.println(input);

         String response = in.readLine();
         System.out.println(response);

         userInput.close();
         in.close();
         out.close();
         socket.close();

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

Question 2 a:



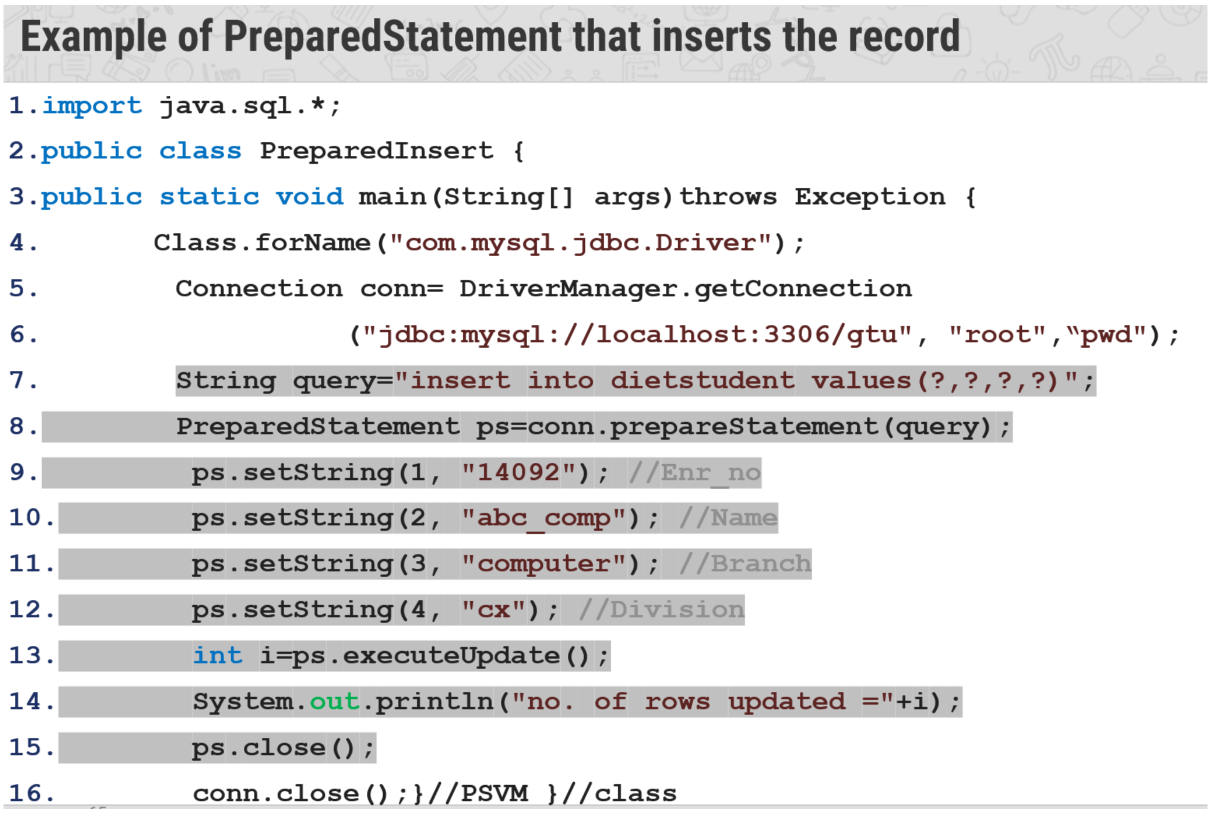
Prepared Statement

- The *PreparedStatement* interface extends the Statement interface.
- It represents a **precompiled** SQL statement.
- A SQL statement is precompiled and stored in a Prepared Statement object.
- This object can then be used to efficiently execute this statement multiple times.

Why to use PreparedStatement?

Improves performance:

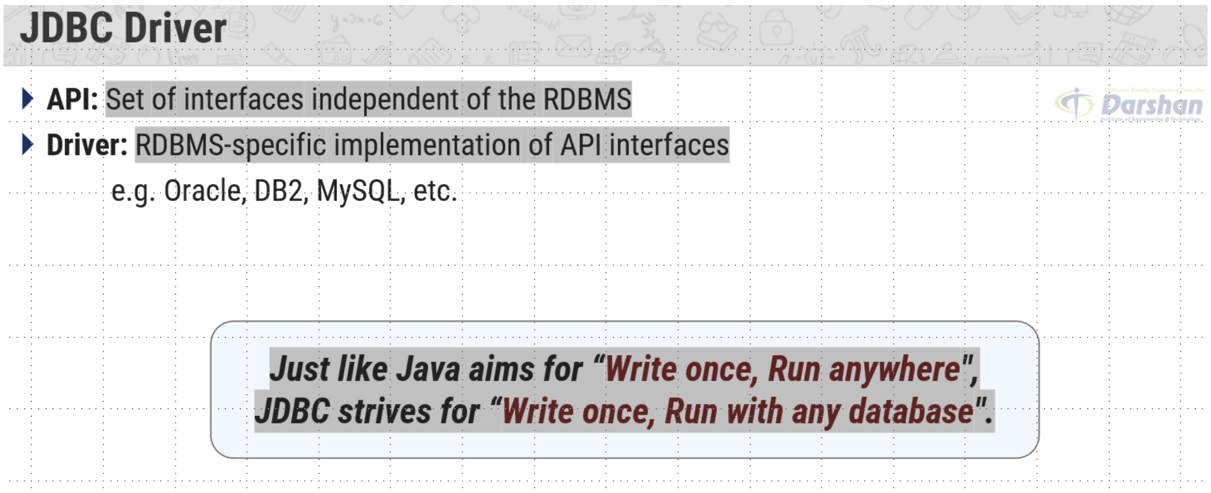
- The performance of the application will be **faster,** if you use PreparedStatement interface because **query is compiled only once.**
- This is because creating a PreparedStatement object by explicitly giving the SQL statement causes the statement to be precompiled within the database immediately.
- Thus, when the PreparedStatement is later executed, the DBMS does not have to recompile the SQL statement.
- Late binding and compilation is done by DBMS.
- Provides the programmatic approach to set the values.



Example of PreparedStatement that inserts the record

```java
import java.sql.*;
public class PreparedInsert {
public static void main(String[] args)throws Exception {
        Class.forName("com.mysql.jdbc.Driver");
         Connection conn= DriverManager.getConnection
                 ("jdbc:mysql://localhost:3306/gtu", "root",“pwd");
         String query="insert into dietstudent values(?,?,?,?)";
         PreparedStatement ps=conn.prepareStatement(query);
          ps.setString(1, "14092"); //Enr_no
          ps.setString(2, "abc_comp"); //Name
          ps.setString(3, "computer"); //Branch
          ps.setString(4, "cx"); //Division
          int i=ps.executeUpdate();
          System.out.println("no. of rows updated ="+i);
          ps.close();
          conn.close();}//PSVM }//class
```

Question 2 b:



JDBC Driver

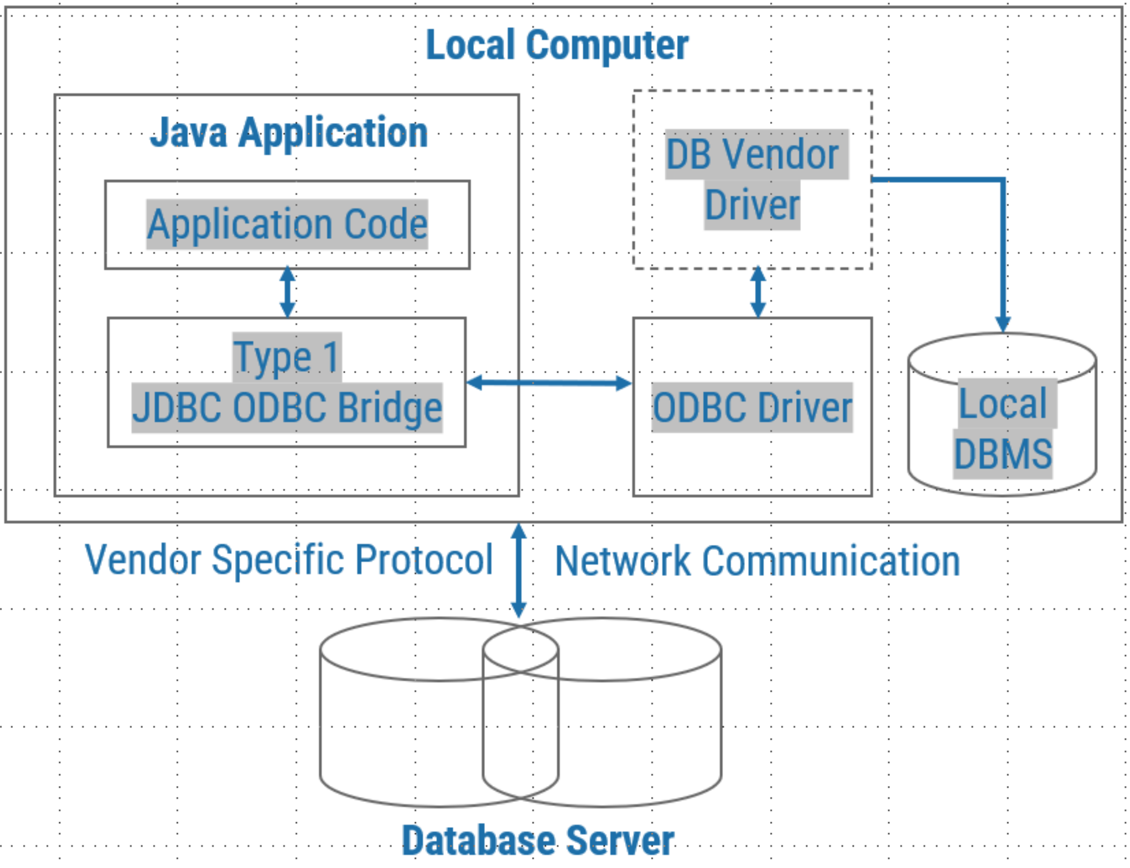
- **API:** Set of interfaces independent of the RDBMS
- **Driver:** RDBMS-specific implementation of API interfaces
  e.g. Oracle, DB2, MySQL, etc.

Just like Java aims for "Write once, Run anywhere",
JDBC strives for "Write once, Run with any database".

JDBC Driver: Type1 (JDBC-ODBC Driver)

- Depends on support for ODBC(Open database connectivity is required)
- Not portable
- Translate JDBC calls into ODBC calls and use Windows ODBC built in drivers
- ODBC must be set up on every client
  - for server side servlets ODBC must be set up on web server
- driver sun.jdbc.odbc.JdbcOdbc provided by JavaSoft with JDK
- No support from JDK 1.8 (Java 8)

E.g. MS Access



JDBC Driver: Type 1 (JDBC-ODBC Driver)

Advantages :

- Allow to communicate with all database supported by ODBC driver
- It is vendor independent driver

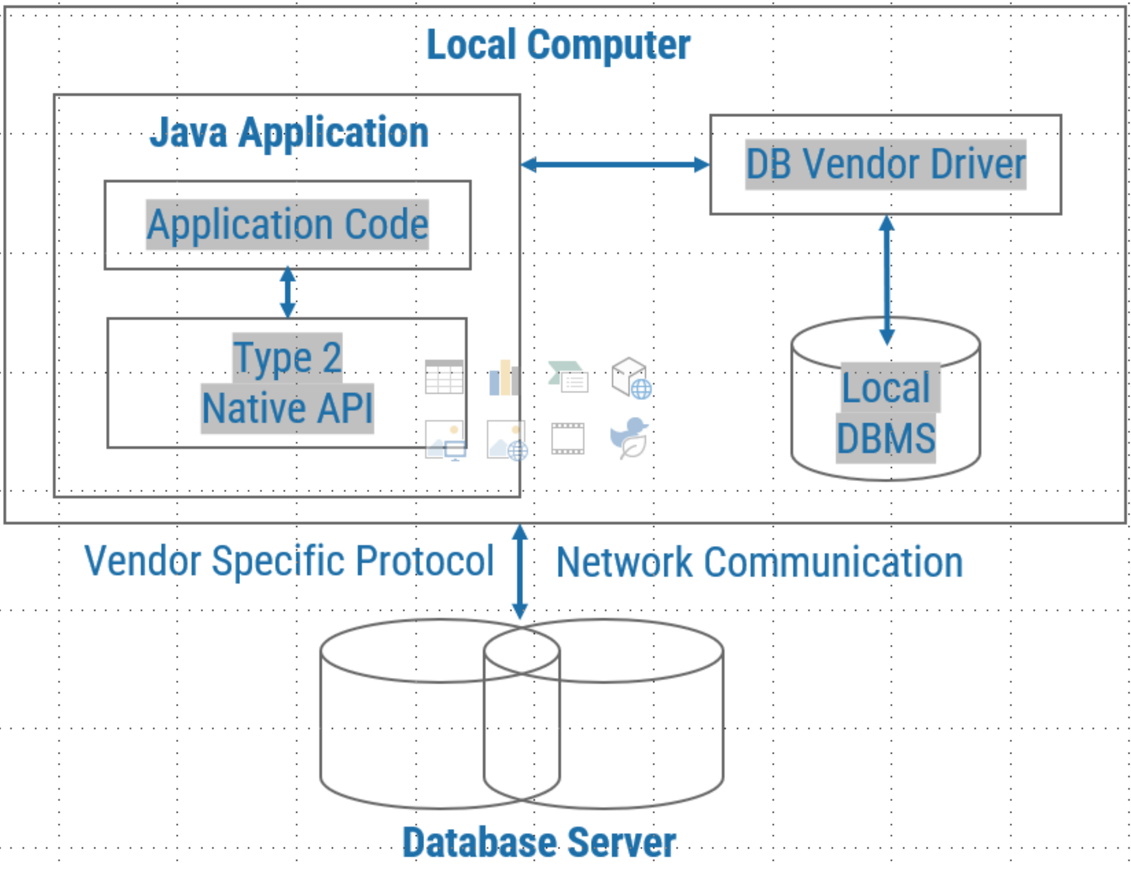
Disadvantages:

- Due to large number of translations, **execution speed** is decreased
- Dependent on the ODBC driver(If ODBC Driver is not present in pc you cant use this)
- ODBC binary code or ODBC client **library to be installed** in every client machine
- Uses java native interface to make ODBC call

Because of listed disadvantage, type1 driver is not used in production environment. It can only be used, when database doesn't have any other JDBC driver implementation.

JDBC Driver: Type 2 (Native Code Driver)

- JDBC API calls are converted into **native API calls**, which are unique to the database.
- These drivers are typically provided by the database vendors(Oracal,MySQL) and used in the same manner as the JDBC-ODBC Bridge.
- Native code Driver are usually written in **C, C++.**
- The vendor-specific driver must be installed on each client machine.
- Type 2 Driver is suitable to use with server side applications.
- E.g. Oracle OCI driver, Weblogic OCI driver, Type2 for Sybase



Advantages

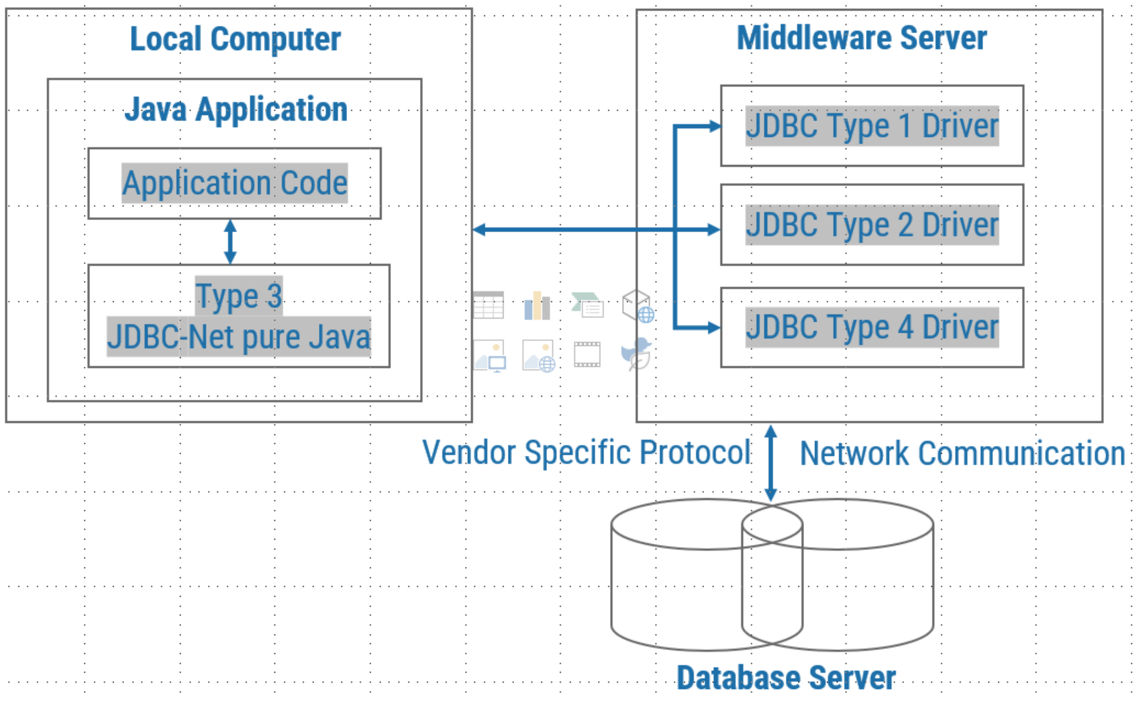
- As there is no implementation of JDBC-ODBC bridge, it may be considerably **faster than a Type 1 driver**.

Disadvantages

- The vendor client library needs to be installed on the client machine.
- This driver is **platform dependent**.
- This driver supports all java applications except **applets**.
- It may **increase cost of application**, if it needs to run on different platform (since we may require buying the native libraries for all of the platform).

JDBC Driver: Type 3 (Java Protocol)

- Pure Java Driver
- Depends on Middleware server
- Can interface to multiple databases – Not vendor specific.
- Follows a three-tier communication approach.
- The JDBC clients use standard network sockets to communicate with a middleware application server.
- The socket information is then translated by the middleware application server into the call format required by the DBMS.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client.
- A single driver can handle any database, provided the middleware supports it.
- We can switch from one database to other without changing the client-side driver class, by just changing configurations of middleware server.
- E.g.: IDS Driver, Weblogic RMI Driver

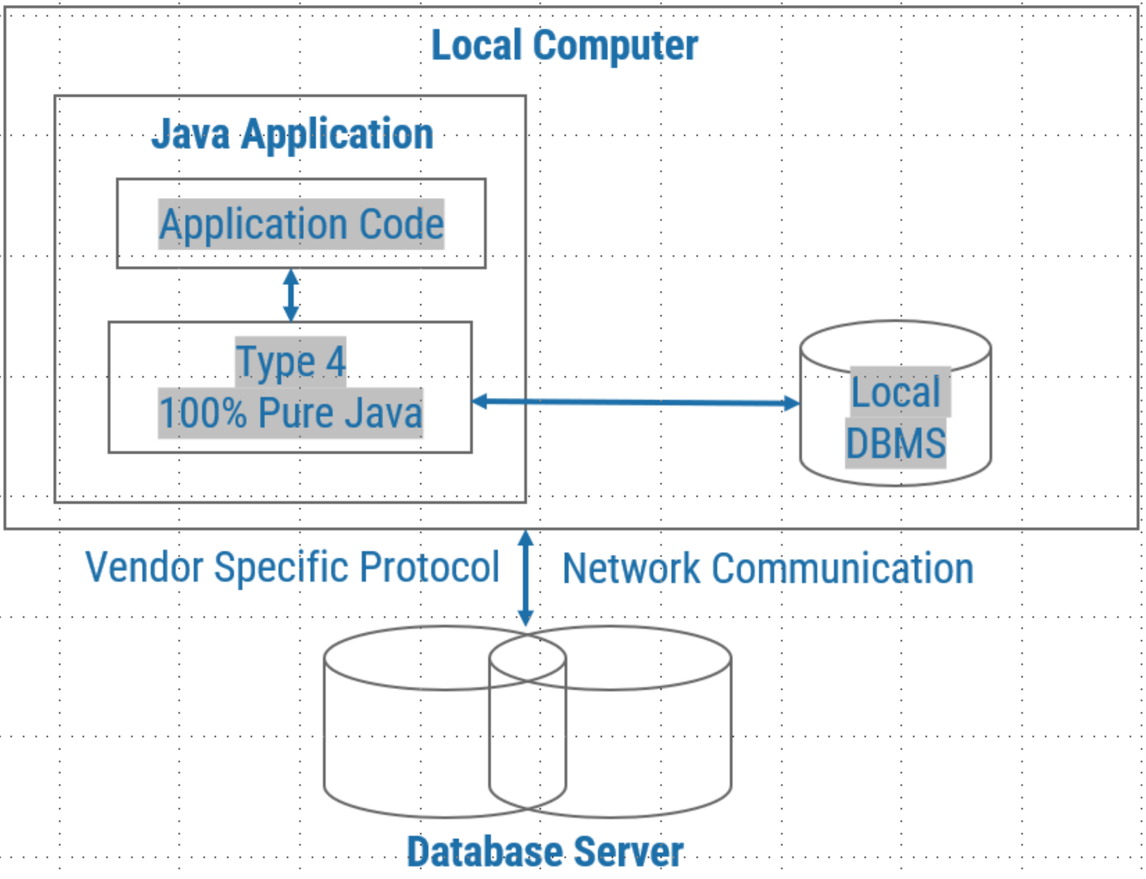
Disadvantages

- Compared to Type 2 drivers, Type 3 drivers are slow due to increased number of network calls.
- Requires database-specific coding to be done in the middle tier.
- The middleware layer added may result in additional latency, but is typically overcome by using better middleware services.

JDBC Driver: Type 4 (Database Protocol)

- It is known as the Direct to Database Pure Java Driver
- Need to download a new driver for each database engine

  e.g. Oracle, MySQL
- Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server.
- Such drivers are implemented by DBMS vendors.



Advantages

- Completely implemented in Java to achieve platform independence.
- No native libraries are required to be installed in client machine.
- These drivers don't translate the requests into an intermediary format (such as ODBC).
- Secure to use since, it uses database server specific protocol.
- The client application connects directly to the database server.
- No translation or middleware layers are used, improving performance.
- The JVM manages all the aspects of the application-to-database connection.

Disadvantage

- This Driver uses database specific protocol and it is DBMS vendor dependent.

Question 2 a:

Callable Statement

- CallableStatement interface is used to call the **stored procedures**.
- We can have business logic on the database by the use of stored procedures that will make the performance better as they are **precompiled**.

Callable Statement

- Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

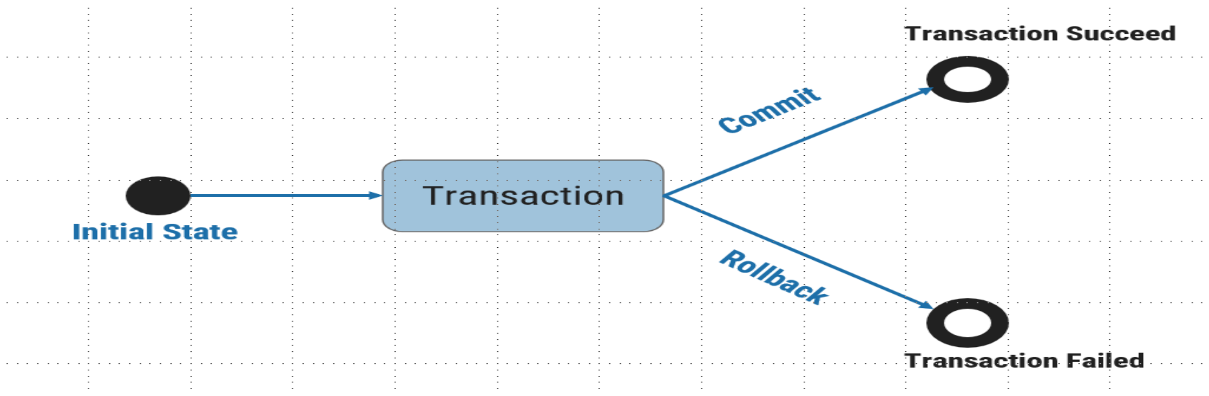
Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

Example CallableStatement

```
import java.sql.*;
public class CallableDemo {
public static void main(String[] args)throws Exception {
        Class.forName("com.mysql.jdbc.Driver");
          Connection conn= DriverManager.getConnection
                  ("jdbc:mysql://localhost:3306/gtu", "root","pwd");
          CallableStatement cs=conn.prepareCall("{call gettitle(?,?)}");
          cs.setInt(1,1201);
          cs.registerOutParameter(2,Types.VARCHAR);
          cs.execute();
          System.out.println(cs.getString(2));
          cs.close();
        conn.close();
}//PSVM}//class
```

Procedure Name

Question 2 b:



- In JDBC, **Connection interface** provides methods to manage transaction.

void **setAutoCommit**(boolean status)	It is true **by default,** means each transaction is committed bydefault.
void **commit**()	commits the transaction.
void **rollback**()	cancels the transaction.

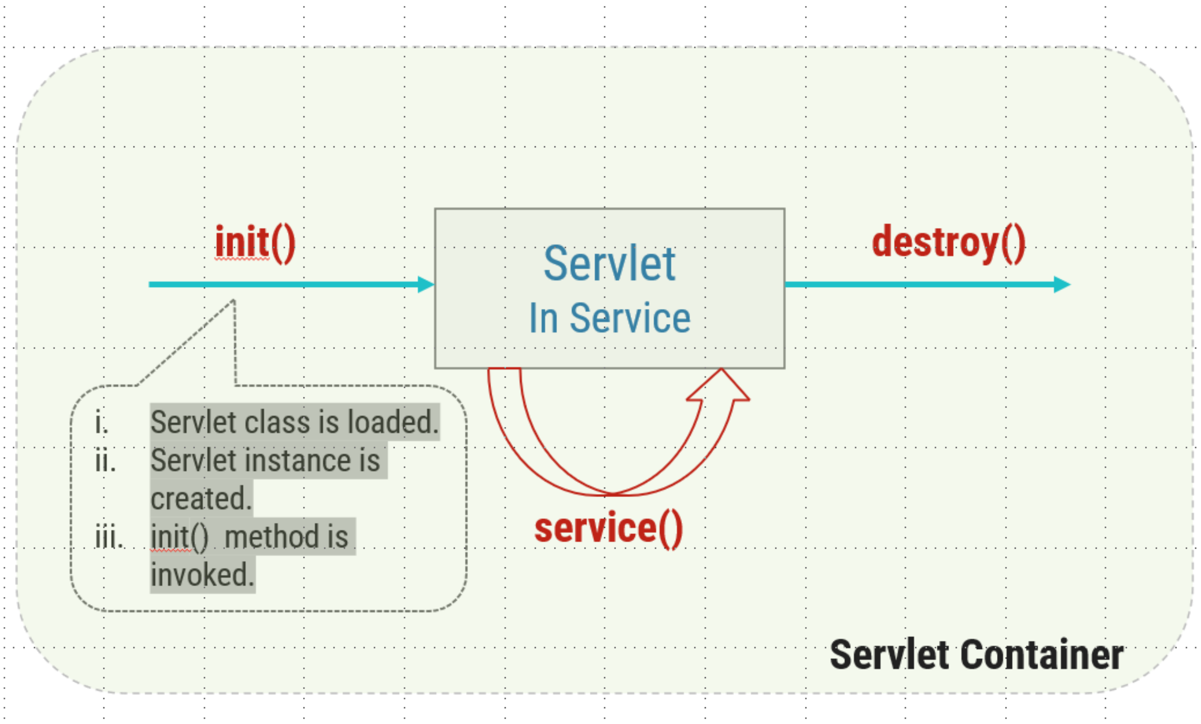
Transaction Management:commit

```
import java.sql.*;
class CommitDemo{
public static void main(String args[]){
Class.forName("com.mysql.jdbc.Driver");
 Connection con=DriverManager.getConnection(
             "jdbc:mysql://localhost:3306/GTU","root","root");
 con.setAutoCommit(false);//bydefault it is true
 Statement stmt=con.createStatement();
 int i=stmt.executeUpdate("insert into diet
                          values(605,'def','ci')");
 System.out.println("no. of rows inserted="+i);
 con.commit();//commit transaction
 con.close();
}}
```

Transaction Management:rollback

```
import java.sql.*;
class RollbackDemo{
public static void main(String args[]){
try{ Class.forName("com.mysql.jdbc.Driver");
  Connection con=DriverManager.getConnection(
                 "jdbc:mysql://localhost:3306/GTU","root","root");
  con.setAutoCommit(false);//bydeafault it is true
  Statement stmt=con.createStatement();
  int i=stmt.executeUpdate("insert into diet
                          values(606,'ghi','ee')");
   con.commit(); //Commit Transaction
   i+=stmt.executeUpdate("insert into diet values(607,'mno','ch')");
   System.out.println("no. of rows inserted="+i);
   con.rollback(); //Rollback Transaction
   con.close();
 }catch(Exception e){ System.out.println(e);}  }}
```

Question 3 a:


init()
Servlet
In Service
destroy()
service()
i. Servlet class is loaded.
ii. Servlet instance is created.
iii. init() method is invoked.
Servlet Container

Servlet Life Cycle: init()

i. Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the **web container.**

A Web application runs within a **Web container** of a Web server. Web container provides runtime environment.

ii. Servlet instance is created

The web container creates the instance(object) of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

iii. Init() method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet.

Syntax:

```
public void init(ServletConfig config)
                         throws ServletException
{
    //initialization…
}
```

A servlet configuration object used by a servlet container to pass information to a servlet during initialization process.

Servlet Life Cycle: Service()

- The service() method is the main method to perform the actual task.
- The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the response back to the client.
- Each time the server receives a request for a servlet, the server spawns a new thread and calls service.

Syntax:

```
public void service(ServletRequest request,
      ServletResponse response)
            throws ServletException, IOException
{
      ...
      ...
}
```

Servlet Life Cycle: Destroy()

- The destroy() method is called only once at the end of the life cycle of a servlet.
- This method gives your servlet a chance to close
  i. database connections,
  ii. halt background threads,
  iii. write cookie lists or hit counts to disk, and
  iv. perform other such cleanup activities.
- After the destroy() method is called, the servlet object is marked for garbage collection.

Servlet Life Cycle: Destroy()

```
public void destroy()
{
      // Finalization code...
}
```

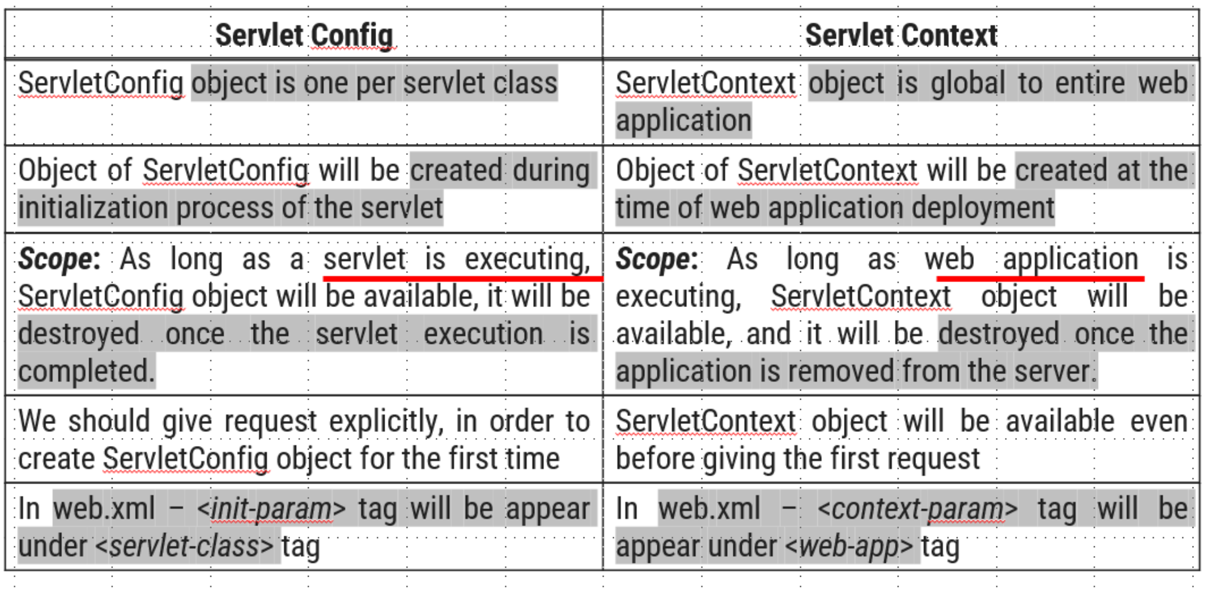
Question 3 b:

```
import java.io.*;

import java.io.servlet.*;

import java.io.servlet.http.*;


public class DoGetDemo extends HttpServlet{

            protected void doGet(HttpSrvletRequest req, HttpservletResponse resp) throws
ServletException, IOEXception{


resp.setContentType("txt/html");

PrintWriter pw = resp.getWriter();

String username = req.getParameter("username");

String password = req.getParameter("password");


            if(username=="admin" && password=="abc")

            {

                        pw.print("logged in successfully");

            }

}

}
```

```
<Html>

<Head>

<Title>

</Title>

</Head>

<Body>

            <form action="DoGetDemo" method="post">
```

```
                Enter username=<input type="text" name="username">

                Enter password=<input type="text" name="password">

                <p><input type="submit></p>

        </form>

</Body>

</Html>
```

Question 3 a:



Servlet Config	Servlet Context
ServletConfig object is one per servlet class	ServletContext object is global to entire web application
Object of ServletConfig will be created during initialization process of the servlet	Object of ServletContext will be created at the time of web application deployment
Scope: As long as a servlet is executing, ServletConfig object will be available, it will be destroyed once the servlet execution is completed.	***Scope***: As long as web application is executing, ServletContext object will be available, and it will be destroyed once the application is removed from the server.
We should give request explicitly, in order to create ServletConfig object for the first time	ServletContext object will be available even before giving the first request
In web.xml – <*init-param*> tag will be appear under <*servlet-class*> tag	In web.xml – <*context-param*> tag will be appear under <*web-app*> tag

Question 3 b:

```
import javax.servlet.*;

import javax.servlet.http.*;


public class PrimeNumbersServlet extends HttpServlet {
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();
    out.println("<html><body>");

    String num1 = request.getParameter("num1");
    String num2 = request.getParameter("num2");


    int n1 = Integer.parseInt(num1);
    int n2 = Integer.parseInt(num2);

    if (n1 <= 1 || n2 <= 1 || n1 >= n2) {
        out.println("Invalid input parameters.");
        out.println("</body></html>");
        return;
    }


    List<Integer> primes = new ArrayList<Integer>();
    for (int i = n1; i <= n2; i++) {
        if (isPrime(i)) {
            primes.add(i);
        }
    }
```

```
            out.println("There are no prime numbers between " + n1 + " and " + n2 + ".");



        out.println("</body></html>");
    }

    private boolean isPrime(int n) {
        if (n <= 1) {
            return false;
        }
        for (int i = 2; i <= n/2; i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }

}
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Prime Numbers</title>
</head>
<body>
```

```
  <form action="prime" method="get">
    <label for="num1">First number:</label>
    <input type="number" id="num1" name="num1" required>
    <br>
    <label for="num2">Second number:</label>
    <input type="number" id="num2" name="num2" required>
    <br><br>
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```