CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 2 – Dec. 2021

Sub:	Advances in Java							Sub Code:	20MCA33
Date:	17/12/2021	Duration:	90 min's	Max Marks:	50	Sem:	III	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

		MARKS	OBE	
			CO	RBT
	PART I			
1	Explain the various steps of JDBC with code snippet.	10	CO4	L2
	OR			
2	Develop a program to insert, delete, select and update music data into database. Table consists of music_id int(5),music_name varchar(20),music_author varchar(20).	10	CO4	L6
	PART II			
3	Describe the basic JDBC datatypes and advanced JDBC datatypes.	10	CO4	L1
	OR			
4.	Explain the different type of JDBC drivers	10	CO4	L2
	PART III			
5	Explain built-in annotations with example.	10	CO3	L1
	OR			
6.a.	Explain creation of packages and sub packages with an example.	5	CO3	L2
b.	Differentiate between abstract class and interface	5	CO3	L2
	PART IV			
7	Write a java JSP program to create Java bean from a HTML form data and display it in a JSP page.	10	CO5	L4
	OR			
8	Discuss the types of JDBC statements with an example	10	CO4	L2
	PART V			
9	Write a JSP program to implement all the attributes of page directive tag	10	CO5	L6
	OR			
10	Write a JSP Program which uses jsp:include and jsp:forward action to display a Webpage.	10	CO5	L6

CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

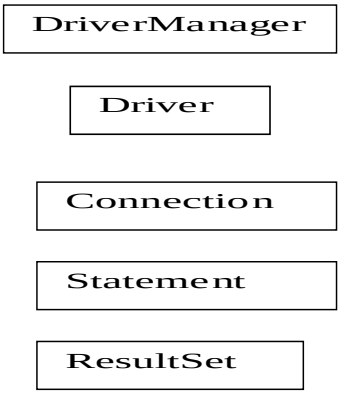
Internal Assessment Test 2– Dec 2021

Sub:	Advances in Java					Sub Code:	20MCA33	Branch:	MCA
Date:	17/11/2021	Duration:	90 min's	Max Marks:	50	Sem	III		

1. Explain the various steps of JDBC with code snippet.

Seven Basic Steps in Using JDBC

1. Load the Driver
2. Define the Connection URL
3. Establish the Connection
4. Create a Statement Object
5. Execute a query
6. Process the results
7. Close the Connection



1. Load the JDBC driver

When a driver class is first loaded, it registers itself with the driver Manager Therefore, to register a driver, just load it!
Example:

String driver = "sun.jdbc.odbc.JdbcOdbcDriver"; Class.forName(driver);
Or Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);

2. Define the Connection URL

The purpose of loading and registering the JDBC driver is to bring the JDBC driver into the Java Virtual Machine (JVM).

jdbc : subprotocol : source

- each driver has its own subprotocol

each subprotocol has its own syntax for the source

jdbc:msql://host[:port]/database

Ex: jdbc:msql://foo.nowhere.com:4333/accounting

3. Establish the Connection

- DriverManager Connects to given JDBC URL with given **user name** and **password**
- **Throws** java.sql.SQLException
- **returns** a Connection object
- A Connection represents a session with a specific database.
- The connection to the database is established by **getConnection()**, which requests access to the database from the DBMS.
- A Connection object is returned by the getConnection() if access is granted; else getConnection() throws a SQLException.
- If username & password is required then those information need to be supplied to access the database.

  String url = jdbc : odbc : Employee;

  Connection c = DriverManager.getConnection(url,userID,password);
- Sometimes a DBMS requires extra information besides userID & password to grant access to the database.
- This additional information is referred as properties and must be associated with Properties or Sometimes DBMS grants access to a database to anyone without using username or password.

  Ex: Connection c = DriverManager.getConnection(url) ;

4. Create a Statement Object

A Statement object is used for executing a static SQL statement and obtaining the results produced by it.

Statement stmt = con.createStatement();

This statement creates a Statement object, *stmt* that can pass SQL statements to the DBMS using connection, *con*.

5. Execute a query

Execute a SQL query such as SELECT, INSERT, DELETE, UPDATE Example

String SelectStudent= "select * from STUDENT";

6. **Process the results**

- A ResultSet provides access to a table of data generated by executing a Statement.
- Only one ResultSet per Statement can be open at once.
- The table rows are retrieved in sequence.
- A ResultSet maintains a cursor pointing to its current row of data.
- The 'next' method moves the cursor to the next row.

7. **Close the Connection**

connection.close();

- Since opening a connection is expensive, postpone this step if additional database operations are expected

2. Develop a program to insert, delete, select and update music data into database. Table consists of music_id int(5),music_name varchar(20),music_author varchar(20).

```
package j2ee.p9;
import java.sql.*;
import java.io.*;

public class Studentdata {

public static void main(String[] args) {
Connection con;
PreparedStatement pstmt;
Statement stmt;
ResultSet rs;
String mid,mname, aname;
Integer marks,count;
try
{
Class.forName("com.mysql.jdbc.Driver"); // type1 driver

try{
con=DriverManager.getConnection("jdbc:mysql://127.0.0.1/mca","root","system");  //  type1  access connection
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
   do
   {


System.out.println("\n1. Insert.\n2. Select.\n3. Update.\n4. Delete.\n5. Exit.\nEnter your choice:");
int choice=Integer.parseInt(br.readLine());
switch(choice)
{
case 1: System.out.print("Enter music id :");
```

```
mid=br.readLine();
System.out.print("Enter mname :");
mname=br.readLine();
System.out.print("Enter aname :");
aname=br.readLine();

pstmt=con.prepareStatement("insert into music values(?,?,?)");
pstmt.setString(1,mid);
pstmt.setString(2,mname);
pstmt.setString(3,aname);
pstmt.execute();
System.out.println("\nRecord Inserted successfully.");
break;
case 2:
stmt=con.createStatement();
rs=stmt.executeQuery("select *from music");
if(rs.next())
{
System.out.println("Music Id\tMusic album Name\tAuthor Name\n--------------------------------");
do
{
mid=rs.getString(1);
mname=rs.getString(2);
aname= rs.getString(3);

System.out.println(mid+"\t"+mname+"\t"+aname);
}while(rs.next());
}
else
System.out.println("Record(s) are not available in database.");
break;
case 3:
System.out.println("Enter mid to update :");
mid=br.readLine();
System.out.println("Enter mname :");
mname=br.readLine();
stmt=con.createStatement();
count=stmt.executeUpdate("update music set mname='"+mname+"'where  mid='"+mid+"'");
System.out.println("\n"+count+" Record Updated.");
break;
case 4: System.out.println("Enter mid  to delete record:");
mid=br.readLine();
stmt=con.createStatement();
count=stmt.executeUpdate("delete from music where username='"+uname+"'");


if(count!=0)
System.out.println("\nRecord "+mid+" has deleted.");
```

```
else
System.out.println("\nInvalid id, Try again.");
break;

case 5: con.close(); System.exit(0);
default: System.out.println("Invalid choice, Try again.");
}//close of switch
}while(true);
}//close of nested try
catch(SQLException e2)
{
System.out.println(e2);
}
catch(IOException e3)
{
System.out.println(e3);
}
}//close of outer
catch(ClassNotFoundException e1)
{
System.out.println(e1);
}
}
}
```

3. List and discuss the basic data types and advanced data types of JDBC.

1. CHAR, VARCHAR, and LONGVARCHAR

CHAR

Represents a small, fixed-length character string

SQL CHAR type corresponding to JDBC CHAR is defined in SQL-92 and is supported by all the major databases

CHAR(12) defines a 12-character string.

All the major databases support CHAR lengths up to at least 254 characters. To retrieve the data from CHAR, ResultSet.getString method will be used.

VARCHAR

VARCHAR represents a small, variable-length character string

It takes a parameter that specifies the maximum length of the string.

VARCHAR(12) defines a string whose length may be up to 12 characters.
All the major databases support VARCHAR lengths up to 254 characters.

When a string value is assigned to a VARCHAR variable, the database remembers the length of the assigned string and on a SELECT, it will return the exact original string.

To retrieve the data from VARCHAR, ResultSet.getString method will be used.

LONGVARCHAR

LONGVARCHAR represents a large, variable-length character string No consistent SQL mapping for the JDBC LONGVARCHAR type.

All the major databases support some kind of very large variable-length string supporting up to at least a gigabyte of data, but the SQL type names vary

These methods are getAsciiStream and getCharacterStream, which deliver the data stored in a LONGVARCHAR column as a stream of ASCII or Unicode characters.

2. BINARY, VARBINARY, and LONGVARBINARY

BINARY

Represents a small, fixed-length binary value

SQL BINARY type corresponding to JDBC BINARY is defined in SQL-92 and is supported by all the major databases

BINARY(12) defines a 12-byte binary value.

To retrieve the data from BINARY, ResultSet.getBytes method will be used.

VARBINARY

VARBINARY represents a small, variable-length binary value

It takes a parameter that specifies the maximum binary bytes.
BINARY(12) defines a 12-byte binary type.

BINARY values are limited to 254 bytes.

To retrieve the data from BINARY, ResultSet.getBytes method will be used

LONGVARBINARY

LONGVARBINARY represents a large, variable-length byte value

No consistent SQL mapping for the JDBC LONGVARBINARY type.

JDBC LONGVARBINARY stores a byte array that is many megabytes long, however, the method getBinaryStream is recommended

3. BIT

The JDBC type BIT represents a single bit value that can be zero or one. SQL-92 defines an SQL BIT type.

Portable code may use the JDBC SMALLINT type, which is widely supported.

The recommended Java mapping for the JDBC BIT type is as a Java boolean.

4. TINYINT

The JDBC type TINYINT represents an 8-bit integer value between 0 and 255 that may be signed or unsigned.

Portable code may use the JDBCSMALLINT type, which is widely supported.

Java mapping for the JDBC TINYINT type is as either a Java byte or a Java short.
Represents a signed value from -128to 127

16-bit Java short will always be able to hold all TINYINT values.

5. SMALLINT

Represents a 16-bit signed integer value between -32768 and 32767.

The recommended Java mapping for the JDBC SMALLINT type is as a Java short.

6. INTEGER

Represents a 32-bit signed integer value ranging between -2147483648 and 2147483647.

SQL type, INTEGER, is defined in SQL-92 and is widely supported by all the major databases.

Recommended Java mapping for the INTEGER type is as a Java int.\

7. BIGINT

Represents a 64-bit signed integer value between - 9223372036854775808 and 9223372036854775807.

The corresponding SQL type BIGINT is a nonstandard extension to SQL.

Recommended Java mapping for the BIGINT type is as a Java long.

8. REAL

The JDBC type REAL represents a "single precision" floating point number that supports 7 digits of mantissa.

4 bytes(32 bits) will be allocated. 1 bit for sign, 8 bits for exponent and 23 for fraction.

Recommended Java mapping for the REAL type is as a Java float.

9. DOUBLE

The JDBC type DOUBLE represents a "double precision" floating point number that supports 15 digits of mantissa.

8 bytes (64 bits) will be allocated. 1 bit for sign, 11 bits for exponent and 52 for fraction.

Recommended Java mapping for the DOUBLE type is as a Java double.

10. FLOAT

The JDBC type FLOAT is basically equivalent to the JDBC type DOUBLE.

FLOAT represents a "double precision" floating point number that supports 15 digits of mantissa.

Both FLOAT and DOUBLE in a possibly misguided attempt at consistency with previous database APIs.

Use the JDBC DOUBLE type in preference to FLOAT.

11. DECIMAL and NUMERIC

The JDBC types DECIMAL and NUMERIC are very similar. They

both represent fixed-precision decimal values.

The precision is the total number of decimal digits supported

The scale is the number of decimal digits after the decimal point.

For example, the value "12.345" has a precision of 5 and a scale of 3, and the value ".11" has a precision of 2 and a scale of 2.

NUMERIC(12,4) will always be represented with exactly 12 digits

DECIMAL(12,4) might be represented by some larger number of digits.

Java mapping for the DECIMAL and NUMERIC types is java.math.BigDecimal.

The method recommended for retrieving DECIMAL and NUMERIC values is ResultSet.getBigDecimal

12. DATE, TIME, and TIMESTAMP

There are three JDBC types relating to time:

The JDBC DATE type represents a date consisting of day, month, and year. The corresponding SQL DATE type is defined

The JDBC TIME type represents a time consisting of hours, minutes, and seconds. The corresponding SQL TIME type is defined

JDBC TIMESTAMP type represents DATE plus TIME plus a nanosecond field. The corresponding SQL TIMESTAMP type is defined

Advanced JDBC Data Types

1. BLOB

- The JDBC type BLOB represents an SQL3 BLOB (Binary Large Object).
- A JDBC BLOB value is mapped to an instance of the Blob interface in the Java programming language.
- A Blob object logically points to the BLOB value on the server rather than containing its binary data, greatly improving efficiency.
- The Blob interface provides methods for materializing the BLOB data on the client when that is desired.

2. CLOB

- The JDBC type CLOB represents the SQL3 type CLOB (Character Large Object).
- A JDBC CLOB value is mapped to an instance of the Clob interface in the Java programming language.
- A Clob object logically points to the CLOB value on the server rather than containing its character data, greatly improving efficiency.
- Two of the methods on the Clob interface materialize the data of a CLOB object on the client.

3. ARRAY

- The JDBC type ARRAY represents the SQL3 type ARRAY.
- An ARRAY value is mapped to an instance of the Array interface in the Java programming language.
- An Array object logically points to an ARRAY value on the server rather than containing the elements of the ARRAY object, which can greatly increase efficiency.
- The Array interface contains methods for materializing the elements of the ARRAY object on the client in the form of either an array or a ResultSet object.

Example : ResultSet rs = stmt.executeQuery("SELECT NAMES FROM STUDENT");
rs.next();
Array stud_name=rs.getArray("NAMES");

4. DISTINCT

- The JDBC type DISTINCT represents the SQL3 type DISTINCT.
- For example, a DISTINCT type based on a CHAR would be mapped to a String object, and a DISTINCT type based on an SQL INTEGER would be mapped to an int.
- The DISTINCT type may optionally have a custom mapping to a class in the Java programming language.
- A custom mapping consists of a class that implements the interface SQLData and an entry in a java.util.Map object.

5. STRUCT

- The JDBC type STRUCT represents the SQL3 structured type.
- An SQL structured type, which is defined by a user with a CREATE TYPE statement, consists of one or more attributes. These attributes may be any SQL data type, built-in or user-defined.
- A Struct object contains a value for each attribute of theSTRUCT value it represents.

- A custom mapping consists of a class that implements the interface SQLData and an entry in a java.util.Map object.

6. REF

- The JDBC type REF represents an SQL3 type REF<structured type>.
- An SQL REF references (logically points to) an instance of an SQL structured type, which the REF persistently and uniquely identifies.
- In the Java programming language, the interface Ref represents an SQL REF.

7. JAVA_OBJECT

- The JDBC type JAVA_OBJECT, makes it easier to use objects in the Java programming language as values in a database.
- JAVA_OBJECT is simply a type code for an instance of a class defined in the Java programming language that is stored as a database object.
- The JAVA_OBJECT value may be stored as a serialized Java object, or it may be stored in some vendor-specific format.
- The type JAVA_OBJECT is one of the possible values for the column DATA_TYPE in the ResultSet objects returned by various DatabaseMetaData methods, including getTypeInfo, getColumns, and getUDTs.
- Values of type JAVA_OBJECT are stored in a database table using the method PreparedStatement.setObject.
- They are retrieved with They are retrived with the methods ResultSet.getObject or CallableStatement.getObject and updated with the ResultSet.updateObject method.

For example, assuming that instances of the class Engineer are stored in the column ENGINEERS in the table PERSONNEL, the following code fragment, in which stmt is a Statement object, prints out the names of all of the engineers.

4.Explain the different type of JDBC drivers.

- JDBC driver specification classifies JDBC drivers into four groups. They are…

Type 1: JDBC-to-ODBC Driver

- Microsoft created ODBC (Open Database Connection), which is the basis from which Sun created JDBC. Both have similar driver specifications and an API.
- The JDBC-to-ODBC driver, also called the JDBC/ODBC Bridge, is used to translate DBMS calls between the JDBC specification and the ODBC specification.
- MS Access and SQL Server contains ODBC driver written in C language using pointers, but java does not support the mechanism to handle pointers.
- So JDBC-ODBC Driver is created as a bridge between the two so that JDBC-ODBC bridge driver translates the JDBC API to the ODBC API.

 Type-1 ODBC Driver for MS Access and SQL Server Drawbacks of Type-I

Driver:

- ODBC binary code must be loaded on each client.
- Transaction overhead between JDBC and ODBC.
- It doesn‟t support all features of Java.
- It works only under Microsoft, SUN operating systems.

Type 2: Java/Native Code Driver or Native-API Partly Java Driver

- It converts JDBC calls into calls on client API for DBMS.
- The driver directly communicates with database servers and therefore some database client software must be loaded on each client machine and limiting its usefulness for internet
- The Java/Native Code driver uses Java classes to generate platform- specific code that is code only understood by a specific DBMS.

Ex: Driver for DB2, Informix, Intersoly, Oracle Driver, WebLogic drivers Drawbacks

of Type-I Driver:

- Some database client software must be loaded on each client machine
- Loss of some portability of code.
- Limited functionality
- The API classes for the Java/Native Code driver probably won‟t work with another manufacturer‟s DBMS.

Type 3: Net-Protocol All-Java Driver

- It is completely implemented in java, hence it is called pure java driver. It translates the JDBC calls into vendor‟s specific protocol which is translated into DBMS protocol by a middleware server
- Also referred to as the Java Protocol, most commonly used JDBC driver.
- The Type 3 JDBC driver converts SQL queries into JDBC- formatted statements, in-turn they are translated into the format required by the DBMS.

Ex: Symantec DB

Drawbacks:

- It does not support all network protocols.
- Every time the net driver is based on other network protocols.

Type 4: Native-Protocol All-Java Driver or Pure Java Driver

- Type 4 JDBC driver is also known as the Type 4 database protocol.
- The driver is similar to Type 3 JDBC driver except SQL queries are translated into the format required by the DBMS.
- SQL queries do not need to be converted to JDBC-formatted systems.
- This is the fastest way to communicated SQL queries to the DBMS.
- Here the driver uses network protocol this protocol is already built-into the database engine; here the driver talks directly to the database using java sockets. This driver is better than all other drivers, because this driver supports all network protocols.
- Use Java networking libraries to talk directly to database engines

Ex: Oracle, MYSQL

Only disadvantage: need to download a new driver for each database engine

5. List and describe the built in annotations in detail.

Built in Annotations:

Java defines many built-in annotations.

These four are the annotations imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**,and **@Inherited**.

@Override, **@Deprecated**, and **@SuppressWarnings** are included in **java.lang**.

@Retention

@Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy.

@Documented

The **@Documented** annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

@Target

The **@Target** annotation specifies the types of declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which must be a constant from the **ElementType** enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond.

Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, interface, or enumeration

we can specify one or more of these values in a **@Target** annotation. To specify multiple values, we must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, we can use this **@Target** annotation:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

@Inherited

@Inherited is a marker annotation that can be used only on another annotation declaration. it affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass,then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited**, then that annotation will be returned.

@Override

@Override is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

@Deprecated

@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

@SuppressWarnings

@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form. This annotation can be applied to any type of declaration.

6.Explain creation of packages and sub packages with an example.

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7. }
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
1. javac -d directory javafilename
```

For **example**

```
1. javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

---

run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Example of Subpackage

```
1. package com.javatpoint.core;
2. class Simple{
```

```
3.   public static void main(String args[]){
4.    System.out.println("Hello subpackage");
5.    }
6.  }
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

6.b. Differentiate between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8 can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstr class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyw "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawab
	void draw
	}

7. Write a java JSP program to create Java bean from a HTML form data and display it in a JSP page.

student.java

```
package program8;
public class stud
{
	public String sname;
	public String rno;
	//Set method for Student name
	public void setsname(String name)
	{
		sname=name;
	}
	//Get method for Student name
	public String getsname()
	{
		return sname;
	}
	//Set method for roll no
	public void setrno(String no)
	{
		rno=no;
	}
	//Get method for roll no
	public String getrno()
	{
		return rno;
	}
}
```

display.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<!-- Using the studb bean -->
<jsp:useBean id ="studb" scope = "request" class =
"program8.stud"></jsp:useBean> Student Name : <jsp:getProperty
name="studb" property="sname"/><br/>
Roll No. :  <jsp:getProperty name="studb" property="rno"/><br/>
</body>
</html>
```

first.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<!-- Create the bean studb and set the property -->
<jsp:useBean id="studb" scope="request" class="program8.stud"></jsp:useBean>
<jsp:setProperty name="studb" property='*'/>
<jsp:forward page="display.jsp"></jsp:forward>
</body>
</html>
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<!-- send the form data to first.jsp -->
<form action="first.jsp">
Student Name : <input type="text" name =
"sname">Student Roll no : <input type="text" name
= "rno">
<input type = "submit" value="Submit"/>
</form>
</body>
</html>
```

8. Discuss the types of JDBC statements with an example.

- The Statement object is used whenever J2EE component needs to immediately execute a query without first having the query compiled.

Statement Object contains 3 methods:

1. **Execute()** □ **(**used for **DDL** commands like, **Create, Alter, Drop)**

2. **executeUpdate()** **(**Used for **DML** commands like, **Insert, Update, Delete)**
3. **exceuteQuery()** **(**Used for **Select** command**)**

- The **execute()** method is used during execution of DDL commands and also used when there may be multiple results returned.
- The **executeUpdate()** executes INSERT, UPDATE, DELETE, and returns an int value specifying the number of rows affected or 0 if zero rows selected
- The **executeQuery()** method, which passes the query as an argument. The query is then transmitted to the DBMS for processing.
- The executeQuery() method executes a simple select query and returns a ResultSet object.
- The ResultSet object contains rows, columns, and metadata that represent data requested by query.

Example-1:

```
Statement stmt = con.createStatement();
ResultSet res = stmt.executeQuery("select * from Employee");
```

OR

Example-2:

```
Statement stmt = con.createStatement();
stmt.executeUpdate("Insert into employee values(„12345“,“sk“,98453));
stmt.executeUpdate("update employee set Mobile=89706 where Mobile=12345 );
```

OR

Example-3:

```
Statement stmt = con.createStatement();
stmt.execute("Drop table Employee");
stmt.execute("Create table Employee (name varhcar(10), age Number(3))");
```

```
import java.sql.*;

public class StatementDemo {
   public static void main(String args[]){
    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con=DriverManager.getConnection("jdbc:odbc:MyDataSource","khutub","");
      Statement stmt;
      stmt= con.prepareStatement("select * from employee where Name='abc'");
      ResultSet rs=stmt.executeQuery();
      while(rs.next()){
         System.out.println(rs.getString(1));
      }
    } // end of try
    catch(Exception e){ System.out.println("exception" + e); }
   } //end of main
   } // end of class
```

The Bold line can be replace
by any of the

1.6.1

PreparedStatement Object

- The preparedStatement object allows you to execute parameterized queries.
- A SQL query can be precompiled and executed by using the PreparedStatement object.
  Ex: Select * from publishers where pub_id=?
- Here a query is created as usual, but a question mark is used as a placeholder for a value that is inserted into the query after the query is compiled.
- The preparedStatement() method of Connection object is called to return the PreparedStatement object.

Ex:
PreparedStatement stat;
stat= con.prepareStatement("select * from publisher where pub_id=?")

```
import java.sql.*;

public class JdbcDemo {
   public static void main(String args[]){
    try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con=DriverManager.getConnection("jdbc:odbc:MyDataSource","khutub","");
      PreparedStatement pstmt;
      pstmt= con.prepareStatement("select * from employee whereUserName=?");
      pstmt.setString(1,"khutub");
      ResultSet rs1=pstmt.executeQuery();
      while(rs1.next()){
         System.out.println(rs1.getString(2));
      }
    } // end of try
    catch(Exception e){System.out.println("exception"); }
   } //end of main
   } // end of class
```

1.6.2 CallableStatement

- The CallableStatement object is used to call a stored procedure from within a J2EE object.
- A Stored procedure is a block of code and is identified by a unique name.
- The type and style of code depends on the DBMS vendor and can be written in PL/SQL, Transact-SQL, C, or other programming languages.
- IN, OUT and INOUT are the three parameters used by the CallableStatement object to call a stored procedure.
- The IN parameter contains any data that needs to be passed to the stored procedure and whose value is assigned using the setxxx() method.
- The OUT parameter contains the value returned by the stored procedures. The OUT parameters must be registered using the registerOutParameter() method, later retrieved by using the getxxx()
- The INOUT parameter is a single parameter that is used to pass information to the stored procedure and retrieve information from the stored procedure.

```
Connection
con; try{
String query = "{CALL
LastOrderNumber(?))}"; CallableStatement
stat = con.prepareCall(query);
stat.registerOutParameter( 1
,Types.VARCHAR); stat.execute();
String lastOrderNumber =
stat.getString(1); stat.close();
}
catch (Exception e){}
```

9. Create a JSP application to implement all the attributes of page directive tag.

student.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Student Information System</title>
<h4>Enter the details</h4>
</head>
<body>
<form action="process.jsp" method="post">
<table boder=1>
<tr><td>Usn No.</td><td><input type="text" name="usn"/></td></tr>
<tr><td>Student Name</td><td><input type="text" name="name"/></td></tr>
<tr><td>Department</td><td><input type="text" name="dept"/></td></tr>
</table>
<input type="submit" value="Submit"/>
<input type="reset" value="Clear"/>
</form>
</body>
</center>
</html>
```

process.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<%                          String
name="",usn="",dept="";
usn=request.getParameter("usn
");
name=request.getParameter("na
me");
dept=request.getParameter("de
pt");
out.println("<html><center><body bgcolor=grey>"); %>
<%@page errorPage="error.jsp" session="true" isThreadSafe="true" %>

<%synchronized(this)
   {
   wait(1000);
   }
   if(dept.equals("")||name.equals("")||usn.equals(""))
   {
```

```
    }

else
    {
thrownew RuntimeException("FieldBlank");


    session.setAttribute("name",name);
    session.setAttribute("usn",usn);
```

```
session.setAttribute("dept",dept);
request.getRequestDispatcher("display.jsp").forward(request,response);
}
 %>
<%out.println("<body></center></html>");
 %>

</body>
</html>
```

error.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<%@page isErrorPage="true"%>
<%=exception %>
</body>
</html>
```

display.jsp

```
<%@page import="java.util.*" session="true" contentType="text/html;"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
<h3 align="center"> Student information</h3>
<h4 align="right"><%= new Date() %></h4>
</head>
<center>
<body>
<table border=1 cellPadding=10 cellSpacing=10>
<tr>
<th>Name</th>
<th>USN</th>
<th>Dept</th>
</tr>
<tr><td><%=session.getAttribute("usn")%></td>
<td><%=session.getAttribute("name")%></td>
<td><%=session.getAttribute("dept")%></td>
```

```
</tr>
</table>
</body>
<a href="student.jsp"> Back to info</a>
</center>
</html>
```

10.Create a JSP application which uses jsp:include and jsp:forward action to display a Webpage.

index.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<!-- send the form data to login.jsp and the get method is used -->
<form method="get" action="login.jsp">
UserName : <input type="text" name
="name"><br> Password : <input
type="password" name ="pass"><br>
<input type="Submit" value ="Submit"/><br>
</form>
</body>
</html>
```

login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<%
//Getting the input name from the html form and storing in
String ‘uname’ String uname =
request.getParameter("name");
//Getting the input pass from the html form and storing in
```

```
String ‘upass’String upass = request.getParameter("pass");
if(uname.equals("admin") && upass.equals("admin"))
{
 %>


<%
}

else
{
<jsp:forward page="main.jsp"></jsp:forward>
out.println("Wrong Credentials Username and Password"+"<br>");
out.println("Enter Corrects Username and Password.. Try again"
+"<br><br>");%>


<jsp:include page="index.jsp"></jsp:include>
<%
}%>
</body>
</html>
```

main.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
<%
// Getting the input name from the html form and storing in
String ‘un’-->String un=request.getParameter("name");
// Getting the input pass from the html form and storing in
String ‘pw’-->String pw=request.getParameter("pass");
%>
<h1>welcome:<%=un%></h1>
<h1>your user name is:<%=un%></h1>
<h1>your password is:<%=pw%></h1>
</body>
</html>
```