CMR
INSTITUTE OF
TECHNOLOGY

CMRIT

Internal Assessment Test 2 Answer Key

Sub:	Advanced Java Programming							**Sub Code:**	**18MCA41**
Date:	19/05/2021	Duration:	90 min's	Max Marks:	50	**Sem**	**4**	**Branch:**	**MCA**

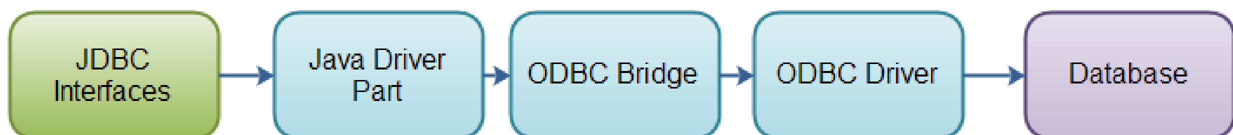
1a. Explain the different type of JDBC drivers **7 CO1 L2**

JDBC driver specification classifies JDBC drivers into four groups

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.

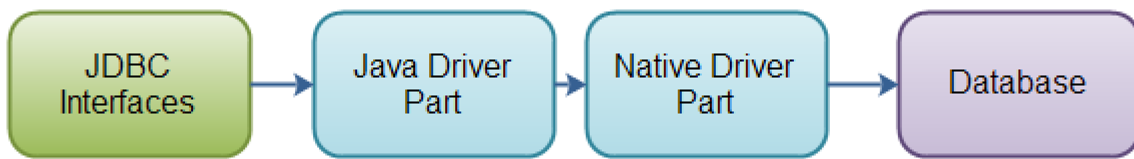
Type-1 ODBC Driver for MS Access and SQL Server

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

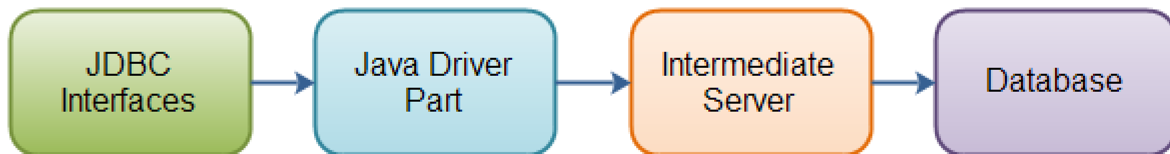


Type 2: Java/Native Code Driver or Native-API Partly Java Driver

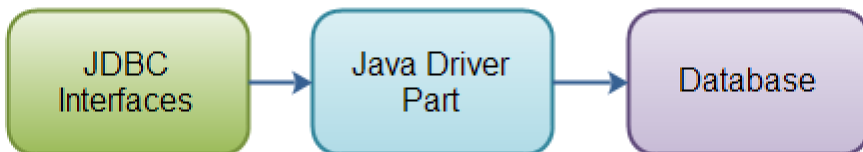
It converts JDBC calls into calls on client API for DBMS.
The driver directly communicates with database servers and therefore some database client software must be loaded on each client machine and limiting its usefulness for internet
The Java/Native Code driver uses Java classes to generate platform- 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:
o Some database client software must be loaded on each client machine
o Loss of some portability of code.
o Limited functionality
o 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



1b. What is Java bean? Write the advantages of Java been

JavaBeans is a
portable,
platform-independent component model
written in the Java programming language.
The JavaBeans architecture was built
through a collaborative industry effort and
enables developers to write reusable components in the Java programming language.
Java Bean components are known as beans.
Beans are dynamic in that they can be changed or customized.

Advantages

Software component architecture provides standard mechanisms to deal with software building blocks. The following list enumerates some of the specific benefits that Java technology provides for a component developer:

- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.

- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Key Concepts

1. Introspection
2. Customizers
3. Properties
4. Methods
5. Events
6. Persistent

2. Explain the various steps of JDBC with code snippet

10 CO1 L2

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.
Register the Driver
Create a Connection
Create SQL Statement
Execute SQL Statement
Closing the connection
Register the Driver
`Class.forName()` is used to load the driver class explicitly.
Example to register with JDBC-ODBC Driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Create a Connection
`getConnection()` method of **DriverManager** class is used to create a connection.
Syntax
getConnection(String url)
getConnection(String url, String username, String password)
getConnection(String url, Properties info)
Example establish connection with Oracle Driver
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:XE","username","password");

Create SQL Statement
`createStatement()` method is invoked on current **Connection** object to create a SQL Statement.
Syntax
public Statement createStatement() throws SQLException
Example to create a SQL statement
Statement s=con.createStatement();

Execute SQL Statement
`executeQuery()` method of **Statement** interface is used to execute SQL statements.
Syntax
public ResultSet executeQuery(String query) throws SQLException
Example to execute a SQL statement

```
ResultSet rs=s.executeQuery("select * from user");
  while(rs.next())
  {
   System.out.println(rs.getString(1)+" "+rs.getString(2));
```

}

Closing the connection
After executing SQL statement you need to close the connection and release the session.
The `close()` method of **Connection** interface is used to close the connection.
Syntax
public void close() throws SQLException
Example of closing a connection
con.close();

```
import java.sql.*;
class OracleCon{
public static void main(String args[]){
try{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");
  //step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
//step3 create the statement object
Statement stmt=con.createStatement();
//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
//step5 close the connection object
con.close();
  }catch(Exception e){ System.out.println(e);}
}
}
```

3. Describe the basic JDBC datatypes and advanced JDBC datatypes

10 CO1

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 the`STRUCT` 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. Develop a program to insert following data into music database. Using prepared Statement object. Table consists of music_id int(5),music_name varchar(20),music_author varchar(20) (08 Marks).

10 CO1 L5

```
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 music_name,music_author;
		Integer music_id,
		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.5. Exit.\nEnter your choice:");
				int choice=Integer.parseInt(br.readLine());
				switch(choice)
				{
					case 1: System.out.print("Enter music id :");
						music_id =Integer.parseInt(br.readLine());
						System.out.print("Enter music name :");
						music_name=br.readLine();
						System.out.print("Enter music author :");
						music_author=br.readLine();
						pstmt=con.prepareStatement("insert into music
values(?,?,?)");
						pstmt.setInt(1,music_id);
						pstmt.setString(2,music_name);
						pstmt.setString(3,music_author);
						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 \t Music Name \t Music
author\n--------------------------------");
						do
						{
							music_id=rs.getInt(1);
							music_name=rs.getString(2);
							music_author=rs.getString(3);

			System.out.println(music_id+"\t"+music_name+"\t"+music_author);
						}while(rs.next());
						}
						else
							System.out.println("Record(s) are not available in
database.");
						break;
```

```
                    case 3: 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 try
            catch(ClassNotFoundException e1)
            {
                System.out.println(e1);
            }
        }
}
```

5. Discuss built-in annotations with example program

10 CO 1 L2

Java Annotation is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
Annotations in java are used to provide additional information, so it is an alternative option for XML and java marker interfaces.

Example for annotation

```
public @interface RequestForEnhancement {
      int identification();
      String abstract();
      String operator() default "[unassigned]";
      String date(); default "[unimplemented]";
}
```

Built-In Java Annotations

Java defines seven built-in annotations. From them, four were imported from `java.lang.annotation` and three more were included from `java.lang`:

- `@Retention`
- `@Documented`
- `@Target`
- `@Inherited`
- `@Overdrive`
- `@Deprecated`
- `@SuppressWarnings`

- These four are the annotations imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**,and **@Inherited**.
- **@Override**, **@Deprecated**, and **@SuppressWarnings** are included in **java.lang**.

1. @Retention

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

the retention policy.

- A retention policy determines at what point annotation should be discarded.
- Java defined 3 types of retention policies through java.lang.annotation.RetentionPolicy enumeration. It has SOURCE, CLASS and RUNTIME.
- Annotation with retention policy SOURCE will be retained only with source code, and discarded during compile time.
- Annotation with retention policy CLASS will be retained till compiling the code, and discarded during runtime.
- Annotation with retention policy RUNTIME will be available to the JVM through runtime.
- The retention policy will be specified by using java built-in annotation @Retention, and we have to pass the retention policy type.

The default retention policy type is CLASS.

@Retentation (RetentionPolity.RUNTIME)

#interface classsica{ }

Code:

```
1  package com.java2novice.annotations;
2
3  import java.lang.annotation.Retention;
4  import java.lang.annotation.RetentionPolicy;
5
6  @Retention(RetentionPolicy.RUNTIME)
7  public @interface MySampleAnn {
8
9      String name();
10     String desc();
11 }
12
13 class MyAnnTest{
14
15     @MySampleAnn(name = "test1", desc = "testing annotations")
16     public void myTestMethod(){
17         //method implementation
18     }
19 }
```

2. @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. By default, annotation are not included in javadoc(is a documentation generator). But if @document is used, it then will be processed by javadoc like toolas and the annotation type information will also be included in generated document .

3. @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 })

4. @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.

```
java.lang.annotation.Inherited

@Inherited
public @interface MyAnnotation {

}
@MyAnnotation
public class MySuperClass { ... }
public class MySubClass extends MySuperClass { ... }
```

In this example the class MySubClass inherits the annotation @MyAnnotation because MySubClassinherits from MySuperClass, and MySuperClass has a @MyAnnotation annotation.

5. @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.

```
class ParentClass
{
        public void displayMethod(String msg){
                System.out.println(msg);
        }
}
class SubClass extends ParentClass
{
        @Override
        public void displayMethod(String msg){
                System.out.println("Message is: "+ msg);
        }
        public static void main(String args[]){
                SubClass obj = new SubClass();
                obj.displayMethod("Hey!!");
        }
}
```

6. @Deprecated

@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form. This annotation is used to mark a class, method or field as deprecated, meaning it should on longer be used If your code uses deprecated classes, methods or fields the compiler will give you a warning.

```
@Deprecated
Public class MyComponent
{
}
```

The use of the @Deprecated annotation above the class declaration marks the class as deprecated.
The use of the @Deprecated annotation above the fieldclass declaration marks the field as deprecated.

7. **@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.

   @SuppressWarnings
   - Makes the compiler suppress warnings for a given methods
   - If a method class a deprecated method, or makes an insecure type case, the compiler may generate a warning.
   - You can suppress these warnings by annotating the method containing the code with the @SuppressWarnings annotation

```
@ SuppressWarnings
public void methodWithWarning()
{
}
```

6. Write a short note about
a)Bean Implementation class
b)Batch updates

10 CO 2 L2

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
```

Batch Updates

A batch update is a batch of updates grouped together, and sent to the database in one "batch", rather than sending the updates one by one.

Sending a batch of updates to the database in one go, is faster than sending them one by one, waiting for each one to finish. There is less network traffic involved in sending one batch of updates (only 1 round trip), and the database might be able to execute some of the updates in parallel. The speed up compared to executing the updates one by one, can be quite big.

You can batch both SQL inserts, updates and deletes. It does not make sense to batch select statements.

There are two ways to execute batch updates:

1. Using a Statement
2. Using a PreparedStatement

i) Add Batch
ii) Clear Batch
iii) Execute Batch

`Statement` object is used to execute batch updates. You do so using the `addBatch()` and `executeBatch()` methods. Here is an example:

```
Statement statement = null;

try{
    statement = connection.createStatement();

    statement.addBatch("update people set firstname='John' where id=123");
    statement.addBatch("update people set firstname='Eric' where id=456");
    statement.addBatch("update people set firstname='May'  where id=789");

    int[] recordsAffected = statement.executeBatch();
} finally {
    if(statement != null) statement.close();
}
```

First you add the SQL statements to be executed in the batch, using the `addBatch()` method.

Then you execute the SQL statements using the `executeBatch()`. The `int[]` array returned by the `executeBatch()` method is an array of `int` telling how many records were affected by each executed SQL statement in the batch

7. Write a java JSP program to create Java bean from a HTML form data and display it in a JSP page.
10 CO2 L4

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;
```

```java
	}
	//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

```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

```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

```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

10 CO1 L2

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=?")

```java
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
```

Callable Statement:
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. Write a JSP program to implement all the attributes of page directive tag

10 CO2 L6

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>
<formaction="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("name");
dept=request.getParameter("dept");
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(""))
thrownew RuntimeException("FieldBlank");
}
else
{
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"><%= newDate() %></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. Summarise about bean key concepts

Main Components of JavaBeans:

- Introspection [naming convention and BeanInfo]
- Bound and Constraints properties
- Persistence
- Customizers

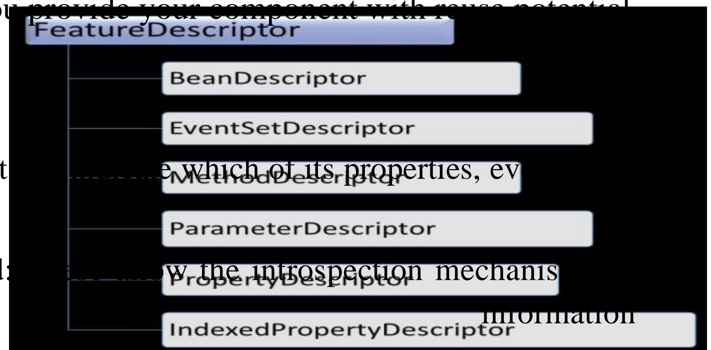
Introspection:

Introspection is the automatic process of analysing a bean's design patterns to reveal the bean's properties, events, and methods.

This process controls the publishing and discovery of bean operations and properties.

Advantages:

1. **Portability -** Everything is done in the Java platform, so you can write components once, reuse them everywhere. There are no extra specification files that need to be maintained independently from your component code. You get all the advantages of the evolving Java APIs, while maintaining the portability of your components.
2. **Reuse -** By following the JavaBeans design conventions, implementing the appropriate interfaces, and extending the appropriate classes, you provide your component with reuse potential that possibly exceeds your expectations.



There are **two** ways for a Bean developer that ... which of its properties, ev... methods should be exposed.

1. Simple naming conventions are used: ... w the introspection mechanis... infer ... information about a Bean (Introspector API).
2. An additional class that extends the BeanInfo class.

Introspection API:

The JavaBeans API architecture supplies a set of classes and interfaces to provide introspection.

- The **BeanInfo** class of the java.Beans package:
  - Defines a set of methods that allow bean implementors to provide *explicit* information about their beans.
  - By specifying **BeanInfo** for a bean component, a developer can hide methods, specify an icon for the toolbox, provide descriptive names for properties, define which properties are bound properties, and much more.
- The **getBeanInfo()** of the Introspector class can be used to provide detailed information about a bean.

The getBeanInfo() method relies on the naming conventions for the bean's properties, events, and methods.

- The **Introspector** class provides ***Descriptor*** classes with information about properties, events, and methods of a bean. Methods of this class identify the information that has been supplied by the developer through **BeanInfo** classes.

The figure represents a hierarchy of the **FeatureDescriptor** classes:

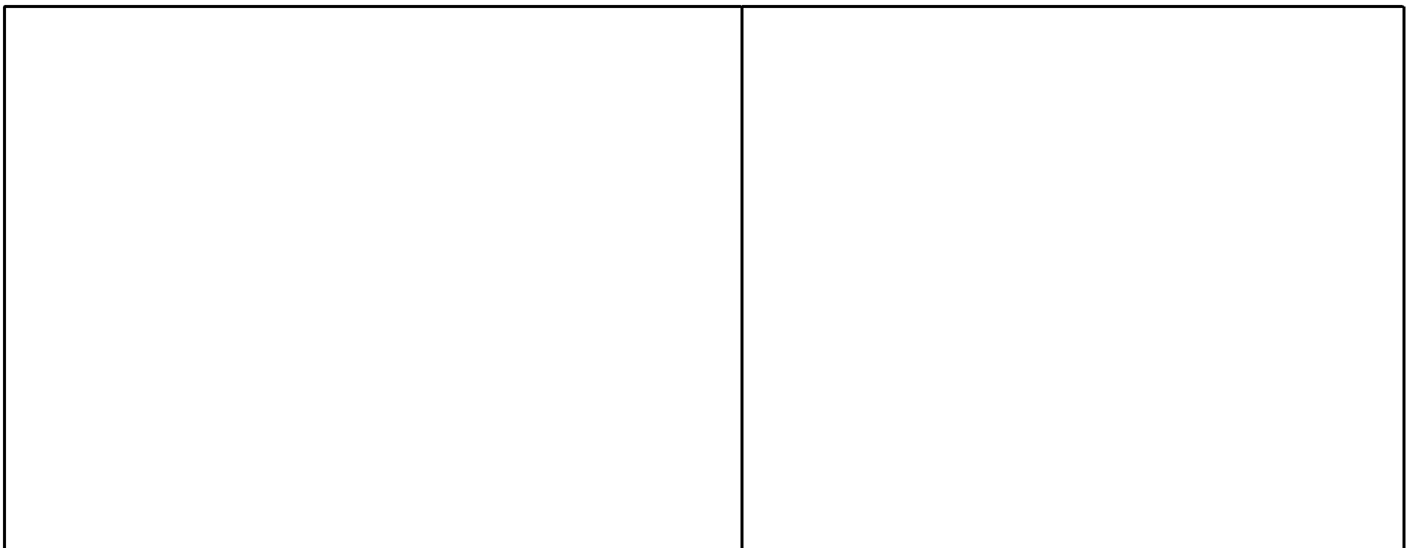
Example - 3.3.1:

SimpleBean.java

```
package  introspectionexample;
import java.io.Serializable;

public class SimpleBean implements
                    Serializable{
  private String Name = "SimpleBean";
  private int size;
  public int getSize() {
       return size;
  }
  public void setSize(int size) {
       this.size = size;
  }
```

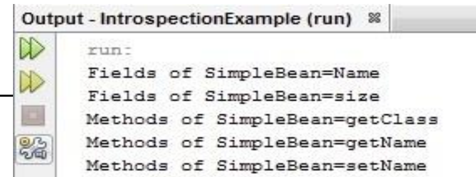
SimpleBean.java(continued)

```
 public String getName() {
        return Name;
  }

public void setName(String Name)
{
      this.Name = Name;
}
}
```

IntrospectionExample.java

```
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.MethodDescriptor;
import java.beans.PropertyDescriptor;
public class IntrospectionExample extends SimpleBeanInfo{
 public static void main(String[] args)throws IntrospectionException {

     PropertyDescriptor pdname = new PropertyDescriptor("Name",
                                               SimpleBean.class);
     System.out.println("Fields of SimpleBean=" + pdname.getName());
     //Getting Method information
     BeanInfo  info=Introspector.getBeanInfo(SimpleBean.class);
     MethodDescriptor[] mdname = info.getMethodDescriptors();
     for(MethodDescriptor md : mdname)
        System.out.println("Methods of SimpleBean=" + md.getName());
  }
}
```



Bean Customization

Customization provides a means for modifying the appearance and behavior of a bean within an application builder so it meets your specific needs. The following links are useful for learning about property editors and customizers:

- **PropertyEditor** interface
- **PropertyEditorSupport** class
- **PropertyEditorManager** class
- **BeanInfo** interface
- **Customizer** interface

A bean's ***appearance and behavior can be customized*** at design time within beans-compliant builder tools. There are **two** ways to ***customize*** a bean:

- By using a **property editor:** Each bean property has its own property editor. The property editor that is associated with a particular property type edits that property type.
- By using **customizers:** Customizers give you complete GUI control over bean customization. Customizers are used where property editors are not applicable.

Customizers:

- Customizers gives you complete GUI control over bean customization.
- When you use a bean *Customizer*, you have complete control over how to configure or edit a bean.
- A Customizer is an application that specifically targets a bean's customization.
- It provides a ***higher level*** of customization when compared to property editors.

All Customizer must:

- Extend `java.awt.Component` or one of its subclasses.
- Implement the `java.beans.Customizer` interface. This means implementing methods to register `PropertyChangeListener` objects, and firing property change events at those listeners when a change to the target bean has occurred.
- Implement a *default constructor*.
- Associate the customizer with its target class via `BeanInfo.getBeanDescriptor`.

The Customizer interface is simple and has only three methods:

`void setObject(Object bean)`	builder tool calls this method to pass the target bean instance to the customizer. It is called only once, which is before the builder tool launches the customizer.
`void addPropertyChangeListener PropertyChangeListener lis)`	Register a lis(listener) for the PropertyChange event. The customizer should fire a PropertyChange event whenever it changes the target bean in a way that might require the displayed properties to be refreshed.
`void removePropertyChangeListener PropertyChangeListener lis)`	Remove a lis(listener) for the PropertyChange event.

Persistence:

A bean has the property of ***persistence*** when its properties, fields, and state information are ***saved*** to and ***retrieved*** from storage. Component models provide a mechanism for persistence that enables the state of components to be stored in a non-volatile place for later retrieval.

- The mechanism that makes persistence possible is called ***serialization***. Object serialization means converting an object into a data stream and writing it to storage. Any applet, application, or tool that uses that bean can then "reconstitute" it by deserialization. The object is then restored to its original state.

  For example: A Java application can serialize a Frame window on a Microsoft Windows machine, the serialized file can be sent with e-mail to a Solaris machine, and then a Java application can restore the Frame window to the exact state which existed on the Microsoft Windows machine.

- All beans must persist. To persist, your beans must support serialization by implementing either the `java.io.Serializable` interface, or the `java.io.Externalizable` interface.
- These interfaces offer you the choices of ***automatic*** serialization and ***customized*** serialization. If any class in a class's inheritance hierarchy implements `Serializable` or `Externalizable`, then that class is **serializable**.

You can **control the level of serialization** that your beans undergo. **Three ways** to control serilization are:

- **Automatic serialization:** implemented by the `Serializable` interface. The Java serialization software serializes the entire object, except `transient` and `static` fields.
- **Customized serialization:** Selectively exclude fields you do not want serialized by marking with the `transient` or `static` modifier.
- **Customized file format:** implemented by the `Externalizable` interface and its two methods. Beans are written in a specific file format.

Bound properties and Constrained properties:

A ***bound*** **property** notifies listeners when its value changes. This has two implications:

1. The bean class includes `addPropertyChangeListener()` methods and `removePropertyChangeListener()` methods for managing the bean's listeners.

```
public void addPropertyChangeListener(PropertyChangeListener p){
     changes.addPropertyChangeListener(p); }
public void removePropertyChangeListener(PropertyChangeListener p) {
     changes.removePropertyChangeListener(p); }
```

2. When a bound property is changed, the bean sends a `PropertyChangeEvent` to its registered listeners.

```
public interface PropertyChangeListener extends EventListener {
     public void propertyChange(PropertyChangeEvent e );
     ------- }
```

`PropertyChangeEvent` and `PropertyChangeListener` live in the `java.beans` package.

To implement a bound property in your application, follow these steps:

1. Import the `java.beans` package. This gives you access to the `PropertyChangeSupport` class.
2. Instantiate a `PropertyChangeSupport` object. This object keeps track of property listeners and includes a convenience method that fires property change events to all registered listeners.

Example-3.3.2:

FaceBean.java

```
import java.beans.*;
public class FaceBean {
   private int mMouthWidth = 90;
   private PropertyChangeSupport mPcs=new PropertyChangeSupport(this);

   public int getMouthWidth() {
        return mMouthWidth;
   }
   public void setMouthWidth(int mw) {
       int oldMouthWidth = mMouthWidth;
       mMouthWidth = mw;
       mPcs.firePropertyChange("mouthWidth", oldMouthWidth, mw);
   }
   public void addPropertyChangeListener(PropertyChangeListener
                                                    listener) {
        mPcs.addPropertyChangeListener(listener);
   }
   public void removePropertyChangeListener(PropertyChangeListener
                                                    listener) {
        mPcs.removePropertyChangeListener(listener);
   }
   public static void main(String[] args) {
        FaceBean fb = new FaceBean();
        FaceMain fm = new FaceMain();

        //add the fm as a registered listener.
        fb.addPropertyChangeListener(fm);

        //change mouth width and notify any listeners of the change.
        fb.setMouthWidth(10);
        fb.setMouthWidth(30);
        fb.setMouthWidth(50);
    }
}
```

FaceMain.java

```
package BoundConstraints;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
public class FaceMain implements PropertyChangeListener {
   @Override
   //called by PropertyChangeSupport firePropertyChange() in FaceBean
   public void propertyChange(PropertyChangeEvent evt1) {
     System.out.println("mouth width has been modified");
     System.out.println("Old Width: " + evt1.getOldValue());
     System.out.println("New Width: " + evt1.getNewValue());
     System.out.println("");
   }
}
```

Output - IntrospectionExample (run)

```
run:
mouth width has been modified
Old Width: 90
New Width: 10

mouth width has been modified
Old Width: 10
New Width: 30

mouth width has been modified
Old Width: 30
New Width: 50
```

A ***constrained*** **property** is a special kind of bound property and generates an event when an attempt is made to change its value.

- For a constrained property, the bean keeps track of a set of ***veto*** **listeners**. When a constrained property is about to change, the listeners are consulted about the change.
- Import the `java.beans` package, which includes a `VetoableChangeSupport` class that greatly simplifies constrained properties.
- A bean property is constrained if the bean supports the `VetoableChangeListener` and `PropertyChangeEvent` classes, and if the set method for this property throws a `PropertyVetoException`.
- *Constrained properties* are more **complicated** than *bound properties* because they also support property change listeners which happen to be *vetoers*.

The following operations in the `setXXX` method for the constrained property must be implemented:

1. Save the old value in case the change is vetoed.
2. Notify listeners of the new proposed value, allowing them to veto the change.
3. If no listener vetoes the change (no exception is thrown), set the property to the new value.

Syntax:

```
public void setPropertyName(PropertyType pt)
                           throws PropertyVetoException {
        //Code
}
```

Note: The `VetoableChangeSupport` provides the following operations:

- Keeping track of `VetoableChangeListener` objects.
- Issuing the `vetoableChange` method on all registered listeners.
- Catching any vetoes (exceptions) thrown by listeners.
- Informing all listeners of a veto by calling `vetoableChange` again, but with the old property value as the proposed "new" value.