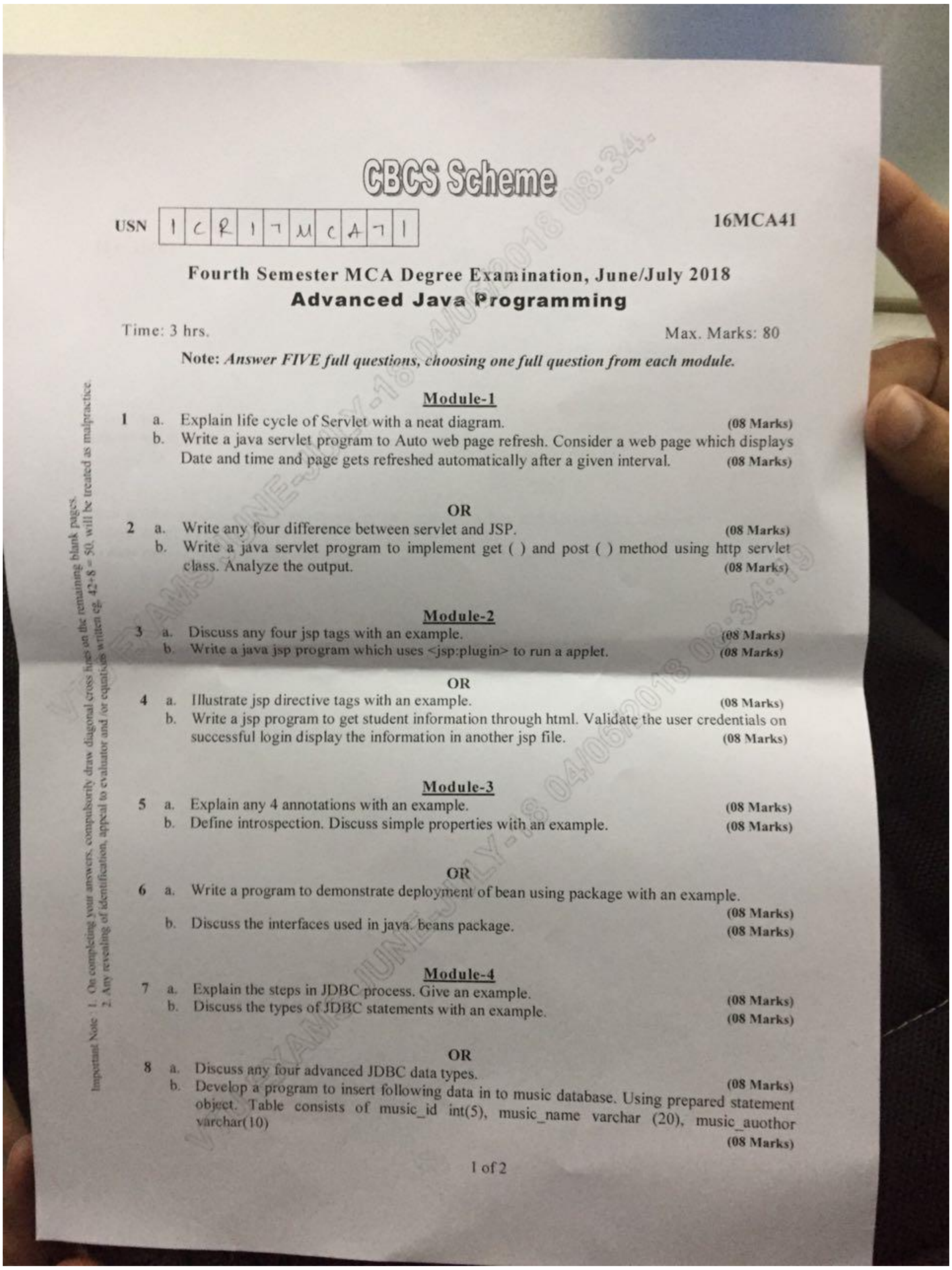


CBCS Scheme

USN | 1 | C | R | 1 | 7 | M | C | A | 7 | 1 | **16MCA41**

Fourth Semester MCA Degree Examination, June/July 2018
Advanced Java Programming

Time: 3 hrs.   Max. Marks: 80

Note: ***Answer FIVE full questions, choosing one full question from each module.***

Module-1

1 a. Explain life cycle of Servlet with a neat diagram. **(08 Marks)**
b. Write a java servlet program to Auto web page refresh. Consider a web page which displays Date and time and page gets refreshed automatically after a given interval. **(08 Marks)**

OR

2 a. Write any four difference between servlet and JSP. **(08 Marks)**
b. Write a java servlet program to implement get ( ) and post ( ) method using http servlet class. Analyze the output. **(08 Marks)**

Module-2

3 a. Discuss any four jsp tags with an example. **(08 Marks)**
b. Write a java jsp program which uses <jsp:plugin> to run a applet. **(08 Marks)**

OR

4 a. Illustrate jsp directive tags with an example. **(08 Marks)**
b. Write a jsp program to get student information through html. Validate the user credentials on successful login display the information in another jsp file. **(08 Marks)**

Module-3

5 a. Explain any 4 annotations with an example. **(08 Marks)**
b. Define introspection. Discuss simple properties with an example. **(08 Marks)**

OR

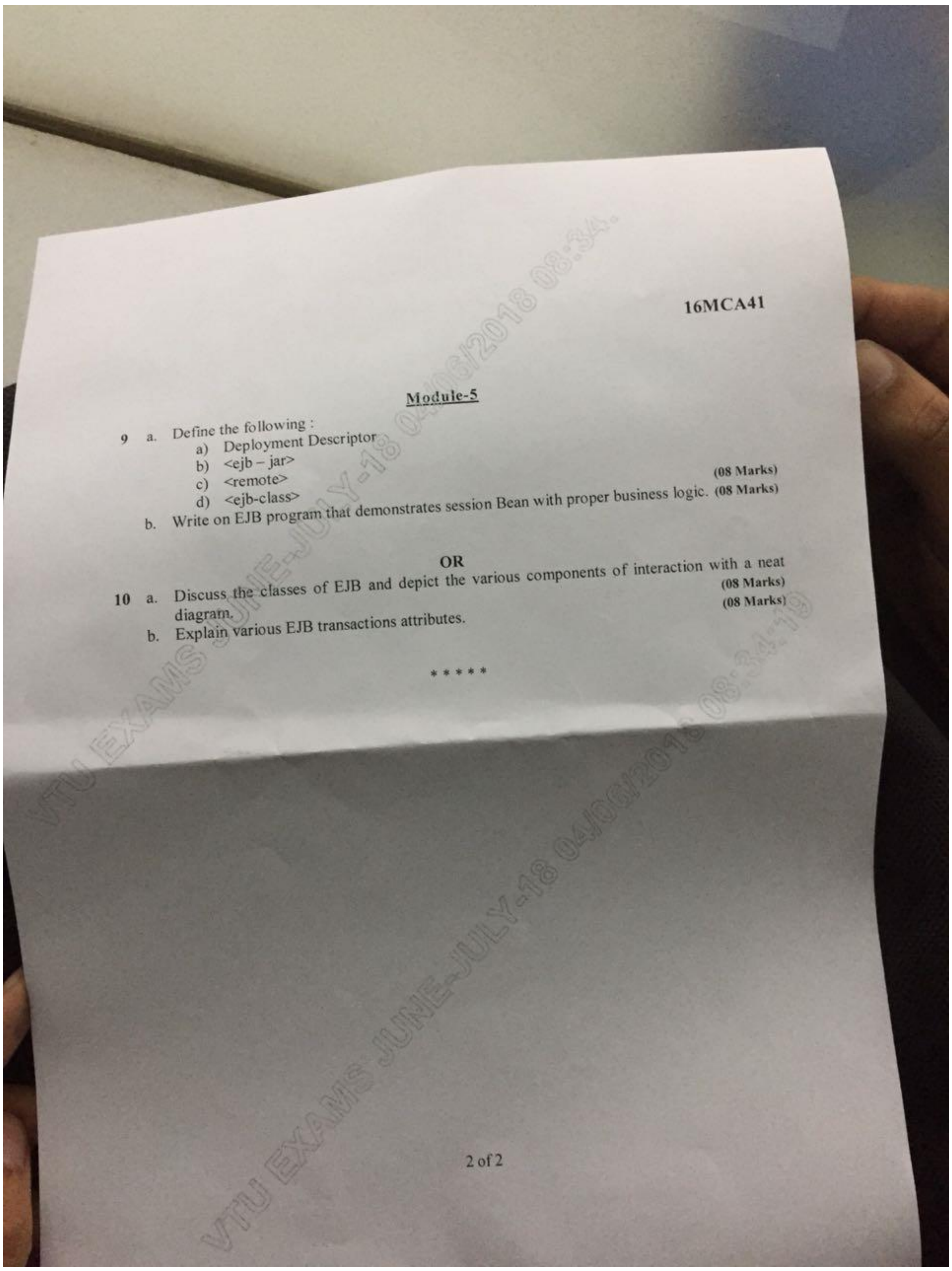
6 a. Write a program to demonstrate deployment of bean using package with an example. **(08 Marks)**
b. Discuss the interfaces used in java. beans package. **(08 Marks)**

Module-4

7 a. Explain the steps in JDBC process. Give an example. **(08 Marks)**
b. Discuss the types of JDBC statements with an example. **(08 Marks)**

OR

8 a. Discuss any four advanced JDBC data types. **(08 Marks)**
b. Develop a program to insert following data in to music database. Using prepared statement object. Table consists of music_id int(5), music_name varchar (20), music_auothor varchar(10) **(08 Marks)**



Module-5

9 a. Define the following :
   a) Deployment Descriptor
   b) <ejb – jar>
   c) <remote>
   d) <ejb-class> (08 Marks)

b. Write on EJB program that demonstrates session Bean with proper business logic. (08 Marks)

OR

10 a. Discuss the classes of EJB and depict the various components of interaction with a neat diagram. (08 Marks)

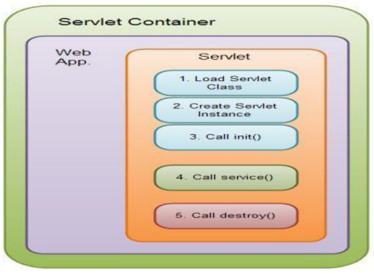
b. Explain various EJB transactions attributes. (08 Marks)

* * * * *

Module-1

1. a. Explain life cycle of servlet with a neat diagram. (08 Marks)

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet



 The servlet is initialized by calling the init () method.
 The servlet calls service() method to process a client's request.
 The servlet is terminated by calling the destroy() method.
 Finally, servlet is garbage collected by the garbage collector of the JVM.
Now let us discuss the life cycle methods in details.
The init() method :
 The init method is designed to be called only once.
 It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.
 The servlet is normally created when a user first invokes a URL corresponding to the servlet,but you can also specify that the servlet be loaded when the server is first started.
 The init() method simply creates or loads some data that will be used throughout the life of the servlet.
The init method definition looks like this:

```
public void init() throws ServletException {
// Initialization code...
}
```

The service() method :
 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 formatted response back to the client.
 Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE,etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.
Signature of service method:

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

 The service () method is called by the container and service method invokes doGet, doPost,doPut, doDelete, etc.methods as appropriate.
 So you have nothing to do with service() method but you override either doGet() or doPost()
depending on what type of request you receive from the client.

 The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.
The doGet() Method
A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Servlet code
}
```

The doPost() Method
A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
// Servlet code
}
```

The destroy() method :
 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 database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
 After the destroy() method is called, the servlet object is marked for garbage collection.
The destroy method definition looks like this:

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

b. Write a java servlet program to auto web page refresh. Consider a web page which displays Date and Time and page gets refreshed automatically after a given interval. (08 Marks)

```
package j2ee.prg2;

import java.io.*;
import java.util.Date;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.*;
/**
 * Servlet implementation class program2
 */
@WebServlet("/program2")
public class program2 extends HttpServlet {
	private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public program2() {
        super();
        // TODO Auto-generated constructor stub
    }

	/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
response)
	 */
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		// TODO Auto-generated method stub
		performTask(request,response);
	}

	/**
	 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
response)
	 */
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		// TODO Auto-generated method stub
		performTask(request,response);
	}

	private void performTask(HttpServletRequest request,HttpServletResponse
response)throws ServletException,IOException
	{
		// Setting the HTTP Content-Type response header to text/html
		response.setContentType("text/html");
		//Adds a response header to refresh the webpage in every one second.
		response.addHeader("Refresh","1");
		//	Returns a PrintWriter object that can send character text to the client.
		PrintWriter out=response.getWriter();
		//writing the output in the html format
		out.println("Text servlet says hi at "+new Date());
	}

}
```

2. a. Write any four differences between servlet and jsp.(08 Marks)

JSP	Servlets
JSP is a webpage scripting language that can generate dynamic content.	Servlets are Java programs that are already compiled which also creates dynamic web content.
JSP run slower compared to Servlet as it takes	Servlets run faster compared to JSP.

compilation time to convert into Java Servlets.	
It's easier to code in JSP than in Java Servlets.	Its little much code to write here.
In MVC, jsp act as a view.	In MVC, servlet act as a controller.
JSP are generally preferred when there is not much processing of data required.	servlets are best for use when there is more processing and manipulation involved.
The advantage of JSP programming over servlets is that we can build custom tags which can directly call Java beans.	There is no such custom tag facility in servlets.
We can achieve functionality of JSP at client side by running JavaScript at client side.	There are no such methods for servlets.

b. Write a java servlet program to implement get() and post() method using HTTPServlet class.Analyse the output.(08 Marks)

prg3.java

```
package prg3.j2ee;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class prg3
 */
@WebServlet("/prg3")
public class prg3 extends HttpServlet {
        private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public prg3() {
        super();
        // TODO Auto-generated constructor stub
    }

        /**
         * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
response)
         */
        protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
```

```
                    // Setting the HTTP Content-Type response header to text/html
                    response.setContentType("text/html");
                    // Returns a PrintWriter object out that can send character text to the client.
                    PrintWriter out=response.getWriter();
                    // To retrieve the optional values (color) from HTML page and store in the
string color
                    String col = request.getParameter("color");
                    out.println("<html><body bgcolor="+col+">");
                    out.println("You have selected "+col);
                    out.println("</body></html>");
                    out.close();
          }
}
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<!-- send the form data to url mapping "prg3" and the get method is used  -->
<form method ="post" action="prg3">
<!--Display 3 Colors RED, BLUE, GREEN in the dropdown Box -->
<select name="color" size="1">
<Option value="red">RED</Option>
<Option value="green">GREEN</Option>
<Option value="blue">BLUE</Option>
</select>
<input type="Submit" value="Enter">
</form>
</body>
</html>
```

Module-2

3.a. Discuss any four jsp tags with an example.(08 Marks)

1) Expression Tag: (<%= … %>)

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.
Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.

The expression element can contain any expression that is valid according to the Java Language Specification but you cannot use a semicolon to end an expression.

Syntax two forms:
<%= expr %>

<jsp:expression> expr </jsp:expression> (XML form)

2) Scriptlet Tag (<% … %>)

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

Embeds Java code in the JSP document that will be executed each time the JSP page is processed.

Code is inserted in the service() method of the generated Servlet

Syntax two forms:
<% any java code %>

<jsp:scriptlet> ... </jsp:scriptlet>. (XML form)

Example
– <% if (Math.random() < 0.5) { %>

Have a <B>nice</B> day! <% } else { %>

Have a <B>lousy</B> day! <% } %>

• Representative result

– if (Math.random() < 0.5) { out.println("Have a <B>nice</B> day!"); } else {

out.println("Have a <B>lousy</B> day!");
}
3) Declaration Tag (<%! … %>)

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

Code is inserted in the body of the servlet class, outside the service method.
o May declare instance variables.

o May declare (private) member functions.

Syntax two forms:
<%! declaration %>

<jsp:declaration> declaration(s)</jsp:declaration>

Example for declaration of Instance Variable:

```
<html>
<body>
<%! private int accessCount = 0; %>
<p> Accesses to page since server reboot:
<%= ++accessCount %> </p>
</body></html>
```

4) Directive Tag (<%@ … %>)
Directives are used to convey special processing information about the page to the JSP container.
The Directive tag commands the JSP virtual engine to perform a specific task, such as importing a Java package required by objects and methods.

Directive	Description
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
<%@ include ... %>	Includes a file during the translation phase.
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page

The page directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.
Following is the basic syntax of page directive:

```
<%@ page attribute="value" %>
```

You can write XML equivalent of the above syntax as follows:

```
<jsp:directive.page attribute="value" />
```

b. Write a java jsp program which uses <jsp:plugin> to run a applet.(08 Marks)

index.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

<!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=ISO-8859-1">
```

```
<title>Applet Index</title>

</head>

<body>

<jsp:plugin type="applet" code="applet11.class" codebase=" " width="400" height="400">

<jsp:fallback><p>unable to load applet</p>p>

</jsp:fallback>

</jsp:plugin>

</body>

</html>
```

<u>Applet11.class</u>

```
package com;

importjava.awt.*;

importjava.applet.*;

importjava.awt.event.*;

publicclass applet11  extends Applet

 {

publicvoid paint(Graphics g)

 {

        setBackground(Color.pink);

        setForeground(Color.black);

        g.drawString("welcome jsp-Applet",100,100);

         }

}
```

4.a. Illustrate jsp directive tags with an example(08 Marks)

The **jsp directives** are messages that tells the web container how to translate a JSP page into the corresponding servlet.

There are three types of directives:

- page directive
- include directive
- taglib directive

page directive

The page directive defines attributes that apply to an entire JSP page.

```
<%@ page attribute="value" %>
```

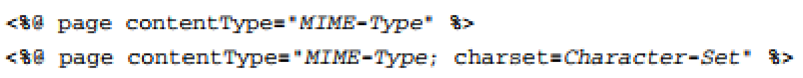
i) import
The import attribute of the page directive lets us specify the packages that should be imported by the servlet into which the JSP page gets translated.
By default, the servlet imports java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.http.*, and possibly some number of server-specific entries. Never write JSP code that relies on any server-specific classes being imported automatically; doing so makes your code nonportable. Use of the import attribute takes one of the following form
<% @ page import =”package.classname” %>
Ex: <% @ page import=”java.util.*” %>
ii) errorPage and isErrorPage
The errorPage attribute is used to define the error page, if exception occurs in the current page, it will be redirected to the error page.

```
//index.jsp
<html>
<body>

<%@ page errorPage="myerrorpage.jsp" %>
<%= 100/0 %>
</body>
</html>
```

The isErrorPage attribute is used to declare that the current page is the error page.

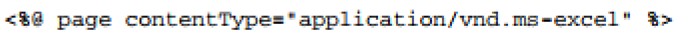
```
<html>
<body>
<%@ page isErrorPage="true" %>
 Sorry an exception occured!<br/>
The exception is: <%= exception %>
 </body>
</html>
```

iii) Content Type
The contentType attribute sets the Content-Type response header, indicating the MIME type of the document being sent to the client.



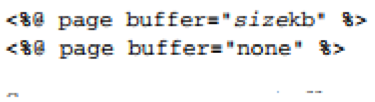
```
<%@ page contentType="MIME-Type" %>
<%@ page contentType="MIME-Type; charset=Character-Set" %>
```

Ex:



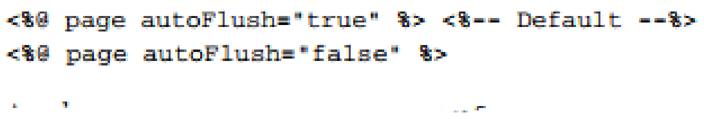
```
<%@ page contentType="application/vnd.ms-excel" %>
```

iv) buffer and autoflush
The buffer attribute specifies the size of the buffer used by the out variable, which is of type JspWriter. Use of this attribute takes one of two forms:



```
<%@ page buffer="sizekb" %>
<%@ page buffer="none" %>
```

The autoFlush attribute controls whether the output buffer should be automatically flushed when it is full (the default) or whether an exception should be raised when the buffer overflows (autoFlush="false"). Use of this attribute takes one of the following two forms.



```
<%@ page autoFlush="true" %> <%-- Default --%>
<%@ page autoFlush="false" %>
```

include directive:

- Includes the contents of any resource(may be jsp file, html file or text file
- It includes the original content of the included resources at page translation time
- Reusability is the advantage
- Syntax:
  <%@include file="resourcename" %>
  Note : this tag includes the original content, so the actual page size grows at run time

taglib directive:

- Includes the contents of any resource(may be jsp file, html file or text file

- It includes the original content of the included resources at page translation time

- Reusability is the advantage

- Syntax:

  <%@include file="resourcename" %>

  Note : this tag includes the original content, so the actual page size grows at run time

b. Write a jsp program to get student information through html. Validate the user credentials on successful login display the information in another jsp fie.

```
<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"))
{
 %>
          <jsp:forward page="main.jsp"></jsp:forward>

<%
}
else
{
          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>
```

Module-3

5. a. Explain any 4 annotations with an example. (08 Marks)

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..

b. Define introspection. Discuss simple properties with an example. (08 Marks)

Introspection:

Introspection can be defined as the technique of obtaining information about bean properties, events and methods.

Basically introspection means analysis of bean capabilities.

Introspection is the automatic process by which a builder tool finds out which properties, methods, and events a bean supports.

Introspection describes how methods, properties, and events are discovered in the beans that you write.
This process controls the publishing and discovery of bean operations and properties
Without introspection, the JavaBeans technology could not operate.

Simple Properties:

Simple properties refer to the private variables of a JavaBean that can have only a single value.
Simple properties are retrieved and specified using the get and set methods respectively.

A read/write property has both of these methods to access its values. The **get method** used to read the value of the property .The **set method** that sets the value of the property.

The setXXX() and getXXX() methods are the heart of the java beans properties mechanism. This is also
called getters and setters. These accessor methods are used to set the property .

The syntax of get method is:

public return_type
get<PropertyName>() public T getN();

public void setN(T arg)

N is the name of the property and T is its type
Ex:

public double getDepth()

```
 {
 return depth;

 }
```
Read only property has only a get method.
The syntax of set method is:
 public void set<PropertyName>(data_type value)
Ex:
```
public void setDepth(double d)

{

Depth=d;

}
```
Write only property has only a set method.

6. a. Write a program to demonstrate deployment of bean using package with an example. (08 Marks)

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>
```

b. Discuss the interfaces used in java. Beans package. (08 Marks)

Interface Summary

AppletInitializer This interface is designed to work in collusion with java.beans.Beans.instantiate.

A bean implementor who wishes to provide explicit information about their bean may provide a
BeanInfo BeanInfo class that implements this BeanInfo interface and provides explicit information about the

methods, properties, events, etc, of their bean.
Customizer A customizer class provides a complete custom GUI for customizing a target Java Bean.
This interface is intended to be implemented by, or delegated from, instances of
DesignMode java.beans.beancontext.BeanContext, in order to propagate to its nested hierarchy of java.beans.beancontext.BeanContextChild instances, the current "designTime" property.
ExceptionListener An ExceptionListener is notified of internal exceptions.
PropertyChangeListener A "PropertyChange" event gets fired whenever a bean changes a "bound" property.
PropertyEditor
A PropertyEditor class provides support for GUIs that want to allow users to edit a property value of a given type.
VetoableChangeListener A VetoableChange event gets fired whenever a bean changes a "constrained" property.
Visibility Under some circumstances a bean may be run on servers where a GUI is not available.

Module-4

7. a. Explain the steps in JDBC process. Give an example. (08 Marks)

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

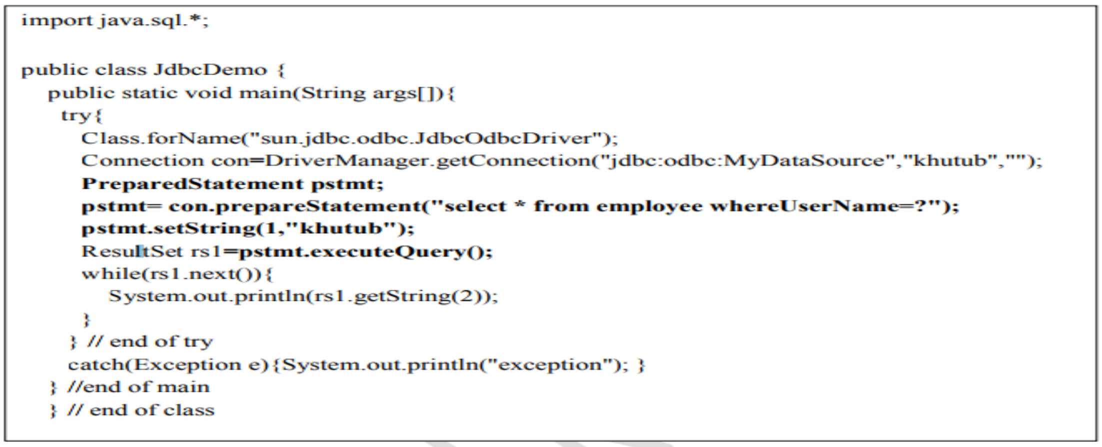
```
ResultSet rs=s.executeQuery("select * from user");
  while(rs.next())
  {
   System.out.println(rs.getString(1)+" "+rs.getString(2));
  }
```

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);}
}
}
```

b. Discuss the types of JDBC statements with an example. (08 Marks)

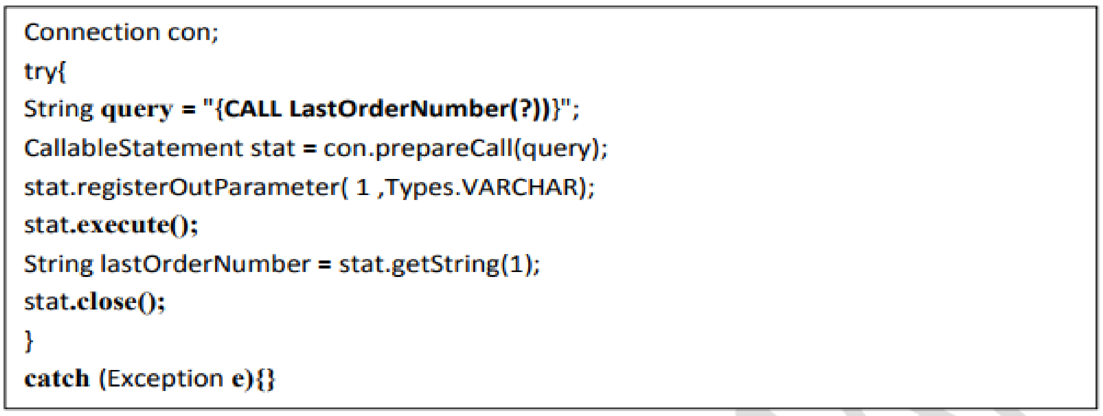
The preparedStatement object allows you to execute parameterized queries.
A SQL query can be precompiled and executed by using the PreparedStatement object.
• Ex: Select * from publishers where pub_id=?
Here a query is created as usual, but a question mark is used as a placeholder for a value• that is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



```
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
```

The preparedStatement() method of Connection object is called to return the• PreparedStatement object.Ex: PreparedStatement stat; stat= con.prepareStatement("select * from publisher wherpub_id=?")
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){}
```

8.a. Discuss any 4 advanced JDBC data types. (08 Marks)

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.

b. 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)

```
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);
                        }
                }
        }
```

Module-5

9.a. Define the following: (08 Marks)

a) Deployment Descriptor

A deployment descriptor describes how EJBs are managed at runtime and enables the customization of EJB behavior without modification to the EJB code.
A deployment descriptor is written in a file using XML syntax.
Add file is packed in the Java Archive (JAR) file along with the other files that are required to deploy the EJB.
It includes classes and component interfaces that are necessary for each EJB in the package.

An EJB container references the deployment descriptor file to understand how to deploy and manage EJBs contained in package.
 The deployment descriptor identifies the types of EJBs that are contained in the package as well as other attributes, such as how transactions are managed.

b) <ejb-jar>

We can use this descriptor as an alternative to annotations, to augment metadata that is not declared as an annotation, or to override an annotation

c) <remote>

It is used to identify the local business interfaces of the bean.

d) <ejb-class>

It defines the bean class which we want to deploy

b. Write an EJB program that demonstrates session bean with proper business logic.(08 Marks)

Calculator.java

```
package package1;

import javax.ejb.Stateless;


@Stateless

public class Calculator implements CalculatorLocal

{

   @Override

   public Integer Addition(int a, int b) {

      return a+b;

   }


   @Override

   public Integer Subtract(int a, int b) {

      return a-b;

   }

   @Override

   public Integer Multiply(int a, int b) {

      return a*b;

   }
```

```
    @Override
     public Integer Division(int a, int b) {
        return a/b;
    }
}
```

CalculatorLocal.java

```
package package1;
import javax.ejb.Local;
public interface CalculatorLocal {
        Integer Addition(int a, int b);
        Integer Subtract(int a, int b);
        Integer Multiply(int a, int b);
        Integer Division(int a, int b);
}
```

Servlet1.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import package1.CalculatorLocal;

public class Servlet1 extends HttpServlet {
```

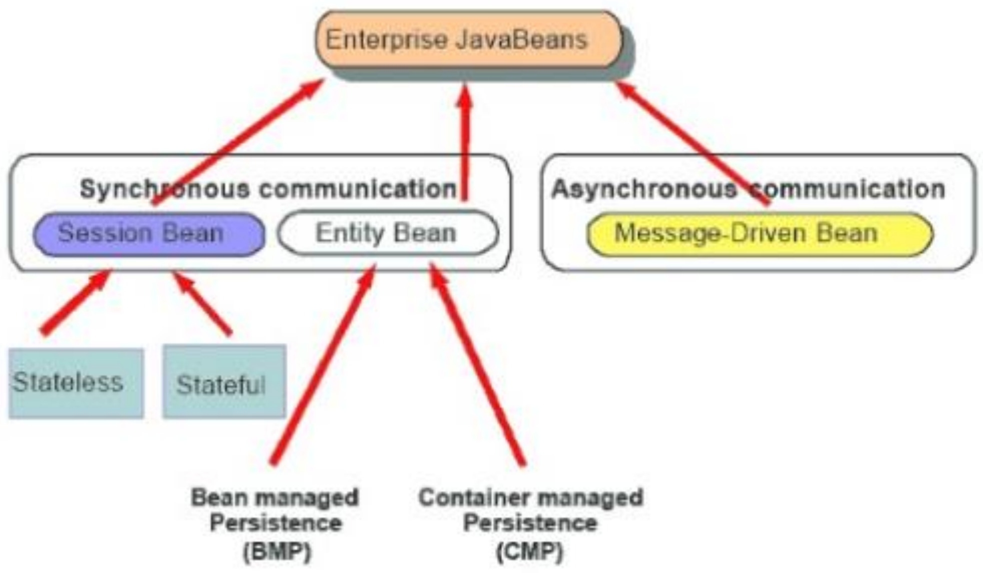
```
@EJB
private CalculatorLocal calculator;
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
{
        response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter())
    {
        out.println("Output :  "+ "<br/>");
        int a;
        a = Integer.parseInt(request.getParameter("num1"));
        int b;
        b=Integer.parseInt(request.getParameter("num2"));
        out.println("Number1 :  " + a + "<br/>");
        out.println("Number2 :  " + b+ "<br/>");
        out.println("Addition  : " + calculator.Addition(a, b)+ "<br/>");
        out.println("Subtraction  :" + calculator.Subtract(a, b)+ "<br/>");
        out.println("Multiplication  :"+calculator.Multiply(a, b)+ "<br/>");
        out.println("Division  :"+calculator.Division(a, b)+ "<br/>");

    }
}

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
{
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    public String getServletInfo()
    {
        return "Short description";
    }
}
```

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Calculator</title>
    </head>
    <body>
        <form method="get" action="Servlet1">
            Enter Number1 :  <input type="text" name="num1"/><br/>
```

```
          Enter Number2 :  <input type="text" name="num2"/><br/>
          <input type="submit" value="Submit"/>
       </form>
   </body>
</html>
```

10. a. Discuss the classes of EJB and depict the various components of interaction with a neat diagram. (08 Marks)



Enterprise JavaBeans
Synchronous communication
Session Bean
Entity Bean
Asynchronous communication
Message-Driven Bean
Stateless
Stateful
Bean managed Persistence (BMP)
Container managed Persistence (CMP)

Session Beans If EJB is a grammar, session beans are the verbs.• Session beans contain business methods.• The client does not access the EJB directly, which allows the Container to perform all sorts of• magic before a request finally hits the target method. It’s this separation that allows for the client to be completely unaware of the location of the server,• concurrency policies, or queuing of requests to manage resources.

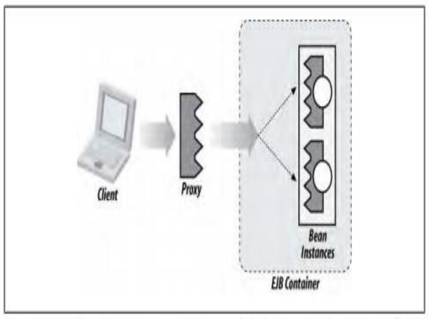


Figure 2-1. Client invoking upon a proxy object, responsible for delegating the call along to the EJB Container

Types of Session Bean

There are 3 types of session bean.

1) Stateless Session Bean: It doesn't maintain state of a client between multiple method calls.

2) Stateful Session Bean: It maintains state of a client across multiple requests.

3) Singleton Session Bean: One instance per application, it is shared between clients and supports concurrent access.

Stateless session beans (SLSBs) Stateless session beans are useful for functions in which state does not need to be carried from• invocation to invocation. The Container will often create and destroy instances however it feels will be most efficient• How a Container chooses the target instance is left to the vendor's discretion.• Because there's no rule linking an invocation to a particular target bean instance, these instances may be used interchangeably and shared by many clients. This allows the Container to hold a much smaller number of objects in service, hence keeping• memory footprint down.

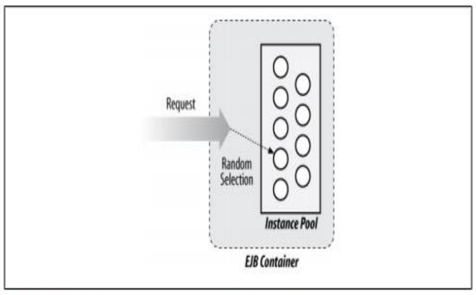


Figure 2-2. An SLSB Instance Selector picking an instance at random

Stateful session beans (SFSBs) Stateful session beans differ from SLSBs in that every request upon a given proxy reference is• guaranteed to ultimately invoke upon the same bean instance. SFSB invocations share conversational state.• Each SFSB proxy object has an isolated session context, so calls to one session will not affect• another. Stateful sessions, and their corresponding bean instances, are created sometime before the first• invocation upon a proxy is made to its target instance (Figure 2-

3). They live until the client invokes a method that the bean provider has marked as a remove event,• or until the Container decides to remove the session

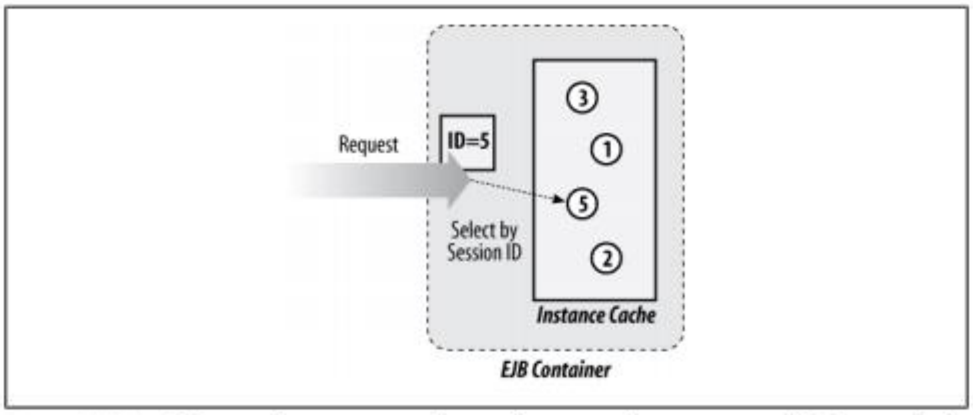


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

Singleton beans Sometimes we don't need any more than one backing instance for our business objects.• All requests upon a singleton are destined for the same bean instance,• The Container doesn't have much work to do in choosing the target (Figure 2-4).• The singleton session bean may be marked to eagerly load when an application is deployed;• therefore, it may be leveraged to fire application lifecycle events. This draws a relationship where deploying a singleton bean implicitly leads to the invocation of its• lifecycle callbacks. We'll put this to good use when we discuss singleton beans.

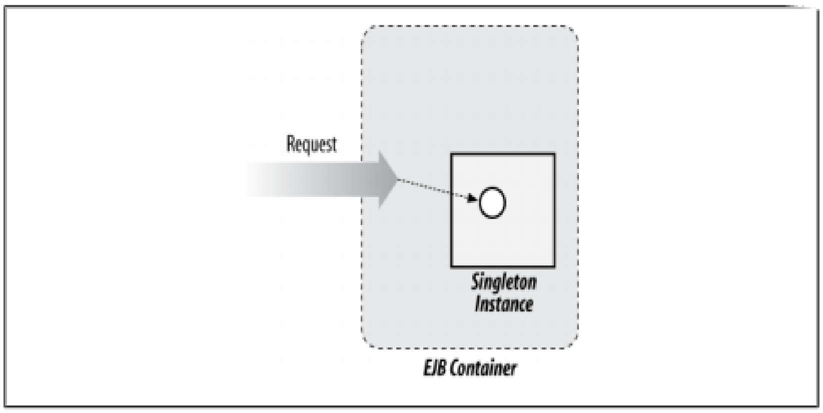


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

MDB

Asynchronous messaging is a paradigm in which two or more applications communicate via a message describing a business event. EJB 3.1 interacts with messaging systems via the Java Connector Architecture (JCA) 1.6 (http://jcp.org/en/jsr/detail?id=322), which acts as an abstraction layer that enables any system to be adapted as a valid sender. The message-driven bean, in turn, is a listener that consumes messages and may either handle them directly or delegate further processing to other EJB components. The asynchronous characteristic of this exchange means that a message sender is not waiting for a response, so no return to the caller is provided

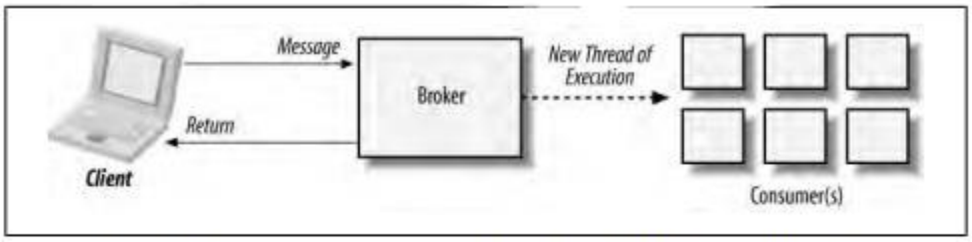


Figure 2-5. Asynchronous invocation of a message-driven bean, which acts as a listener for incoming events

Entity Beans While session beans are our verbs, entity beans are the nouns. Their aim is to express an object view of resources stored within a Relational Database Management System (RDBMS)—a process commonly known as object-relational mapping. Like session beans, the entity type is modeled as a POJO, and becomes a managed object only when associated with a construct called the javax.persistence.EntityManager, a container-supplied service that tracks state changes and synchronizes with the database as necessary. A client who alters the state of an entity bean may expect any altered fields to be propagated to persistent storage. Frequently the EntityManager will cache both reads and writes to transparently streamline performance, and may enlist with the current transaction to flush state to persistent storage automatically upon invocation completion

Unlike session beans and MDBs, entity beans are not themselves a server-side component type. Instead, they are a view that may be detached from management and used just like any stateful object. When detached (disassociated from the EntityManager), there is no database association, but the object may later be re-enlisted with the EntityManager such that its state may again be synchronized. Just as session beans are EJBs only within the context of the Container, entity beans are managed only when registered with the EntityManager. In all other cases entity beans act as POJOs, making them extremely versatile.

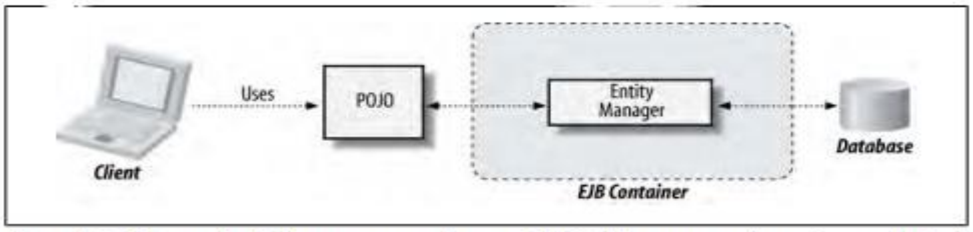


Figure 2-6. Using an EntityManager to map between POJO object state and a persistent relational database

b. Explain various EJB transactions attributes(08 Marks)

When an EJB is deployed, you can set its runtime transaction attribute in the @javax.ejb.TransactionAttribute annotation or deployment descriptor to one of several values:
NotSupported
Supports

Required
RequiresNew
Mandatory
Never

Using the @TransactionAttribute annotation

The @javax.ejb.TransactionAttribute annotation can be used to apply transaction attributes to your EJB's bean class or business methods. The attribute is defined using the javax.ejb.TransactionAttributeType Java enum:

```
public enum TransactionAttributeType
{
 MANDATORY,
 REQUIRED,
 REQUIRES_NEW,
 SUPPORTS,
 NOT_SUPPORTED,
 NEVER
}
@Target({METHOD, TYPE})
public @interface TransactionAttribute
{
 TransactionAttributeType value( ) default TransactionAttributeType.REQUIRED;
}
```