Internal Assessment Test 2 – Nov. 2017
Solution Scheme

Sub:	JAVA & J2EE					Sub Code:	10IS753	Branch:	ISE
Date:	09/11/2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	A&B		OBE

Answer any FIVE FULL Questions

MARKS

1 What is Synchronization? Explain the role of synchronization with producer and consumer problem. [10M]

Solution:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

Producer-Consumer problems

In producer/consumer synchronizations, producer processes make items available to consumer processes. Examples are a message sender and a message receiver, or two machines working on items in sequence. The synchronization here must ensure that the consumer process does not consume more items than have been produced. If necessary, the consumer process is blocked (must wait) if no item is available to be consumed. x Producer and consumer processes are coupled by a buffer to allow asynchronous production and consumption. x The buffer can be bounded (have a capacity limit) or unbounded (be able to store an unlimited number of items).

Consider the following sample program that incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: Q, the queue that you're trying to synchronize. Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

```
// An incorrect implementation of a producer and consumer.
class Q {
int n;
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out.println("Put: " + n);
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PC {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Although the put( ) and get( ) methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

After the producer put 1, the consumer started and got the same 1 five times
in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer
have a chance to consume them. The proper way to write this program in Java is to use wait( ) and
notify( ) to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
while(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
while(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Inside get( ), wait( ) is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside get( ) resumes. After the data has been obtained, get( ) calls notify( ). This tells Producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify( ) is called. This tells the Consumer that it should now remove it. Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

2. Briefly explain the role of any two event classes. [10M]

Solution:

Event Classes: The classes that represent events are at the core of Java's event handling mechanism. The following are the list of Event classes.

1) The Action Event Class: An Action Event is generated when a button is pressed, a list item is double clicked, or a menu item is selected.
2) The Adjustment Event Class: An Adjustment Event is generated by a scroll bar.
3) The Component Event Class: A Component Event is generated when the size, position, or visibility of a component is changed. There are four types of component events.
4) The Container Event Class A Container Event is generated when a component is added to or removed from a container x The Focus Event Class : A Focus Event is generated when a component gains or losses input focus

5) The Input Event Class : The abstract class Input Event is a subclass of Component Event and is the Super class for component input events. Its subclasses are Key Event and Mouse Event.
6) The Item Event Class : An Item Event is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected
7) The Key Event Class A Key Event is generated when keyboard input occurs.
8) The Mouse Event Class There are eight types of mouse events

9) The Mouse Wheel Event Class The Mouse Wheel Event class encapsulates a mouse wheel event.
10) The Text Event Class Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.
11) The Window Event Class There are ten types of window events. The Window Event class defines integer constants that can be used to identify them.

The Mouse Event Class:

There are eight types of mouse events. The **Mouse Event** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED The user clicked the mouse.
MOUSE_DRAGGED The user dragged the mouse.
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.
MOUSE_MOVED The mouse moved.
MOUSE_PRESSED The mouse was pressed.
MOUSE_RELEASED The mouse was released.
MOUSE_WHEEL The mouse wheel was moved.

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors:
MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,int *x*, int *y*, int *clicks*, boolean *triggersPopup*)
Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.
Two commonly used methods in this class are **getX( )** and **getY( )**.
This return the X and Y coordinates of the mouse within the component when the event occurred.
int getX( )
int getY( )
Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse.
Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.
The **translatePoint( )** method changes the location of the event.

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.
The **getClickCount( )** method obtains the number of mouse clicks for this event.
Its signature is shown here:
int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )
Also available is the **getButton( )** method, shown here:
int getButton( )
It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **Mouse Event**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

The **NOBUTTON** value indicates that no button was pressed or released.
Java SE 6 added three methods to **MouseEvent** that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:
Point getLocationOnScreen( )
int getXOnScreen( )
int getYOnScreen( )

The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The Key Event Class:

A **Key Event** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.**Key Event** is a subclass of **Input Event**. Here is one of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers*argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**.For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.

The **Key Event** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

```
char getKeyChar( )
int getKeyCode( )
```

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When
a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

3. Explain the different types of statement objects with example. [10M]

Solution:

Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection. To execute SQL statements, you need to instantiate a Statement object from your connection object by using the create Statement () method.
Statement statement = dbConnection.createStatement ();

A statement object is used to send and execute SQL statements to a database.
Three kinds of Statements

Statement: Execute simple sql queries without parameters. Statement create Statement () Creates an SQL Statement object.

e.g.
```
Statement st =
conn.createStatement():
ResultSet rs = st.executeQuery(
“ select * from Authors” );
:
st.close()
```

Prepared Statement: Execute precompiled sql queries with or without parameters. Prepared Statement prepare Statement (String sql) returns a new Prepared Statement object. Prepared Statement objects are precompiled SQL statements.

Eg:
```
PreparedStatement pstmt = null;
try {
String SQL = "Update Employees SET age = ? WHERE id = ?";
pstmt = conn.prepareStatement(SQL);
. . .
}
catch (SQLException e) {
. . .
}
finally {
```

```
. . .
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException. Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0. All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
String SQL = "Update Employees SET age = ? WHERE id = ?";
pstmt = conn.prepareStatement(SQL);
. . .
}
catch (SQLException e) {
. . .
```

```
}
finally {
pstmt.close();
}
```

Callable Statement: Execute a call to a database stored procedure. Callable Statement prepare Call(String sql) returns a new Callable Statement object. Callable Statement objects are SQL stored procedure call statements.

Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
(EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
SELECT first INTO EMP_FIRST
FROM Employees
WHERE ID = EMP_ID;
END;
```

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

IN: A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.

OUT: A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.

INOUT: A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure:

```
CallableStatement cstmt = null;
try {
String SQL = "{call getEmpName (?, ?)}";
cstmt = conn.prepareCall (SQL);
. . .
}
catch (SQLException e) {
. . .
}
finally {
. . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders. Using the Callable Statement objects is much like using the Prepared Statement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQL Exception. If you have IN parameters, just follow the same rules and techniques that apply to a Prepared Statement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional Callable Statement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

4. Describe the various steps of JDBC with code snippets. [10M]

<u>Solution:</u>

The Concept of JDBC :

• The JDBC (Java Database Connectivity) API defines interfaces and classes for writing database applications in Java by making database connections. • Using JDBC you can send SQL, PL/SQL statements to almost any relational database. JDBC is a Java API for executing SQL statements and supports basic SQL functionality. • It provides RDBMS access by allowing you to embed SQL inside Java code. **Overview of JDBC Process**
Before you can create a java jdbc connection to the database, you must first import the java.sql package.
import java.sql.*; The star (*) indicates that all of the classes in the package java.sql are to be imported.
Java application calls the JDBC library. JDBC loads a driver which talks to the database. We can change database engines without changing database code. Establishing Database Connection and Associating JDBC/ODBC bridge

1. Loading a database driver,
• In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself. A client can connect to

Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.
• The return type of the Class.forName (String ClassName) method is "Class". Class is a class in java.lang package.

```
Class.forName(”sun.jdbc.odbc.JdbcOdbcDriver”); //Or any other driver
}
catch(Exception x) {
System.out.println( “Unable to load the driver class!” );
}
```

2. Creating a oracle jdbc Connection

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system.
• Its getConnection() method is used to establish a connection to a database. It uses a username, password, and a jdbc url to establish a connection to the database and returns a connection object.

• A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases.

• A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

•Each subprotocol has its own syntax for the source. We’re using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

```
try{
Connection dbConnection=DriverManager.getConnection(url,”loginName”,”Pas
sword”)
}
 catch( SQLException x ){
System.out.println( “Couldn’t get connection!” );
}
```

3. Creating a JDBC Statement object
• Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection.

• To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.

• Statement statement = dbConnection.createStatement();

• A statement object is used to send and execute SQL statements to a database. Three kinds of Statements

• **Statement:** Execute simple sql queries without parameters.
Statement createStatement()
Creates an SQL Statement object.
• **Prepared Statement:** Execute precompiled sql queries with or without parameters.
Prepared Statement prepare Statement(String sql) returns a new PreparedStatement object. Prep aredStatement objects are precompiled SQL statements.

• **Callable Statement:** Execute a call to a database stored procedure.
CallableStatement prepareCall(String sql) returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

4. Executing a SQL statement with the Statement object, and returning a jdbc resultSet.

• Statement interface defines methods that are used to interact with database via the execution of SQL statements.

• The Statement class has three methods for executing statements:
executeQuery(), executeUpdate(), and execute().

• For a SELECT statement, the method to use is executeQuery . For statements that create or modify tables, the method to use is executeUpdate.

Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. execute() executes an SQL statement that is written as String object.
ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSetMetaData Interface holds information on the types and properties of the columns in a Result Set. It is constructed from the Connection object.

5. What are transactions? Write a Java program to execute database transactions. [10M]

Solution:
If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion. That may be fine for simple applications, but there are three reasons why you may want to turn off the auto-commit and manage your own transactions:

 To increase performance,
 To maintain the integrity of business processes,
 To use distributed transactions.
Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.
For example, if you have a Connection object named conn, code the following to turn off auto-commit:

```
conn.setAutoCommit(false);
```

Commit & Rollback
Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows:

```
conn.commit( );
```
Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback( );
```

The following example illustrates the use of a commit and rollback object:

```
try{
//Assume a valid connection object conn
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String SQL = "INSERT INTO Employees " +
"VALUES (106, 20, 'Rita', 'Tez')";
stmt.executeUpdate(SQL);
//Submit a malformed SQL statement that breaks
String SQL = "INSERTED IN Employees " +
"VALUES (107, 22, 'Sita', 'Singh')";
stmt.executeUpdate(SQL);
// If there is no error.
conn.commit();
}catch(SQLException se){
// If there is any error.
conn.rollback();
}
```

Using Savepoints

The new JDBC 3.0 Savepoint interface gives you an additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL. When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints:
setSavepoint(String savepointName): Defines a new savepoint. It also returns a Savepoint object.
 releaseSavepoint(Savepoint savepointName): Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.
There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.
The following example illustrates the use of a Savepoint object:

```
try{
//Assume a valid connection object conn
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
//set a Savepoint
Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
String SQL = "INSERT INTO Employees " +
"VALUES (106, 20, 'Rita', 'Tez')";
stmt.executeUpdate(SQL);
//Submit a malformed SQL statement that breaks
String SQL = "INSERTED IN Employees " +
"VALUES (107, 22, 'Sita', 'Tez')";
stmt.executeUpdate(SQL);
// If there is no error, commit the changes.
conn.commit();
}catch(SQLException se){
// If there is any error.
conn.rollback(savepoint1);
```

}
In this case, none of the above INSERT statement would success and everything would be rolled back.

6. Define JSP. Explain different types of JSP tags [10M]

Solution:

JavaServer Pages (JSP) is a Sun Microsystems specification for combining Java with HTML to provide dynamic content forWeb pages. When you create dynamic content, JSPs are more convenient to write than
HTTP servlets because they allow to embed Java code directly into HTML pages, in contrast with HTTP servlets, in which you embed HTML inside Java code.JSP is part of the Java 2 Enterprise Edition (J2EE). x JSP enables to separate the dynamic content of a Web page from its presentation. It caters to two different types of developers: HTML developers, who are responsible for the graphical design of the page, and Java developers, who handle the development of software to create the dynamic content. The following table describes the basic tags that you can use in a JSP page. Each shorthand tag has an XML equivalent. **JSP Tag**

1. JSP Tags	**Syntax**	**Description**
Scriptlet	<% java_code %> . . . or use the XML equivalent: <jsp:scriptlet> java_code </jsp:scriptlet>	Embeds Java source code Scriptlet in yourHTML page. The Java code is executed and its output is inserted in sequence with the rest of the HTML in the page.
Directive	<% @ *dir-type dir-attr* %> . . . or use the XML equivalent: <jsp:directive.*dir_type dir_attr* />	*Directives* contain messages to the application server. A directive can also contain name/value pair attributes in the form attr=‖value‖, which provides additional instructions to
Declarations	<%! declaration %> . . . or use XML equivalent... <jsp:declaration> declaration;	Declares a variable or method that can be referenced by other declarations, scriptlets,
Expression	<%= expression %>	Defines a Java
. . . or use XML	that is	
Actions	<jsp:useBean ... >	Provide access to

Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed under the Java Community Process.
Apache Tomcat is developed in an open and participatory environment and released under the Apache Software

Example:

```
<% @ page info="a hello world example" %>
<html>
<head>
<title>Hello, World</title>
</head>
<body bgcolor="#ffffff" background="background.gif">
<% @ include file="dukebanner.html" %>
 <table>
<tr>
 <td width=150> &nbsp; </td>
<td width=250 align=right>
 <h1>Hello, World !</h1>
</td>
</tr>
</table>
</body>
</html>
```

Request string

```
 <%@ page import="hello.NameHandler" %>
<jsp:useBean id="mybean" scope="page" class="hello.NameHandler" />
 <jsp :setProperty name="mybean" property="*" />
<html>
<head>
<title>Hello, User</title>
</head>
<body bgcolor="#ffffff" background="background.gif">
 <% @ include file="dukebanner.html" %>
<table border="0" width="700"> <tr>
  <td width="150"> &nbsp; </td>
<td width="550">
<h1>My name is Duke. What's yours?</h1>
 </td>
 </tr>
<tr>
<td width="150" &nbsp; </td>
<td width="550">
<form method="get">
 <input type="text" name="username" size="25">
 <br> <input type="submit" value="Submit">
<input type="reset" value="Reset">
</td>
</tr>
</form>
 </table>
 <% if ( request.getParameter("username") != null )
```

```
{ %>
 <% @ include file="response.jsp" %>
 <% } %>
</body>
</html>
```

The data the user enters is stored in the request object, which usually implements javax.servlet.HttpServletRequest (or if your implementation uses a different protocol, another interface that is subclassed from javax.servlet.ServletRequest). You can access the request object directly within a scriptlet.

7. What is RMI? Explain server side and client side methods. [10M]

Solution:

RMI x RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates a number of remote objects, makes references to those remote objects accessible, and waits for clients to invoke methods on those remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an applications is sometimes referred to as a distributed object application.

The java.rmi.Remote Interface :

In RMI, a remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. A remote interface must satisfy the following requirements:

- A remote interface must at least extend, either directly or indirectly, the interface java.rmi.Remote.
- Each method declaration in a remote interface must satisfy the requirements **Server side and Client side** The interface ServerRef represents the server-side handle for a remote object implementation.

```
package java.rmi.server;
public interface ServerRef extends RemoteRef
{
RemoteStub exportObject(java.rmi.Remote obj, Object data) throws java.rmi.RemoteException;
String getClientHost() throws ServerNotActiveException;
}
```

The method exportObject finds or creates a client stub object for the supplied Remote object implementation *obj.* x The parameter *data* contains information necessary to export the object (such as port number).

The method getClientHost returns the host name of the current client. When called from a thread actively handling a remote method invocation, the host name of the client invoking the call is returned. If a remote method call is not currently being service, then ServerNotActiveException is called. There is no special configuration necessary to enable the client to send RMI calls through a firewall. The client can, however, disable the packaging of RMI calls as HTTP requests by setting the java.rmi.server.disableHttp property to equal the boolean value true.

8 a.What is thread? Explain two ways of creating a thread in Java. [5M]

Solution:

A class can be made threadable in one of the following ways

(1) Implement the Runnable Interface and apply its run() method.

(2) Extend the Thread class itself.

Implementing Runnable Interface:

1) Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared as

public void run( )

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.

Create a class that implements Runnable, you will instantiate an object of typeThread from within that class. Thread defines several constructors.

Thread(Runnable *threadOb*, String *threadName*)
In this constructor, *threadOb* is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*. After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is

void start( )

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
```

```
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
```

2) Extending Threads :

You can inherit the Thread class as another way to create a thread in your program. When you declare an instance of your class, you‘ll also have access to members of the Thread class. Whenever your class inherits the Thread class, you must override the run() method, which is an entry into the new thread. The following example shows how to inherit the Thread class and how to override the run() method. This example defines the MyThread class, which inherits the Thread class. The constructor of the MyThread class calls the constructor of the Thread class by using the super keyword and passes it the name of the new thread, which is My thread. It then calls the start() method to activate the new thread. The start() method calls the run() method of the MyThread class

```
class MyThread extends Thread
{
MyThread()
{
super("My thread");
 start();
}
 public void run()
{
 System.out.println("Child thread started");
System.out.println("Child thread terminated");
 } }
class Demo
{
public static void main (String args[])
{
new MyThread();
System.out.println("Main thread started");
 System.out.println("Main thread terminated");
 } }
```

b. Write a java program that creates multiple threads and also ensure that the main thread is the last to stop.

[5M]

Solution:

```
// Create multiple threads.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}
}
class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
```

```
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```