CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | |

Internal Assessment Test 3 – Jan. 2022

Sub:	**Advances in Java**							**Sub Code:**	**20MCA33**
Date:	**25/1/2022**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**III**	**Branch:**	**MCA**

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

		MARKS	OBE	
			CO	RBT
	PART I			
1	Demonstrate a program to implement an Entity Bean	10	CO6	L4
	OR			
2.a.	How are jar files created and used? Explain its advantages	5	CO5	L2
2.b.	Explain bound and constrained properties of design patterns	5	CO5	L2
	PART II			
3	Demonstrate a program to implement an Message Driven Bean	10	CO6	L4
	OR			
4.a.	What is a manifest file? Mention its importance.	5	CO5	L2
b.	List the differences between stateless session bean and stateful session bean.	5	CO6	L3
	PART III			
5	With a neat block diagram, explain the life cycle of a stateful session bean	10	CO6	L2
	OR			
6.a.	What is entity bean? Explain the java persistence model in detail	6	CO6	L2
b.	Define introspection. Explain simple properties with it.	4	CO5	L2
	PART IV			
7	List all the EJB container services and explain any five in detail	10	CO6	L1
	OR			
8	Discuss the classes of EJB and depict the various components of interaction with a neat diagram	10	CO6	L2
	PART V			
9	Write the short note about the following i)Persistence Context ii)XML Deployment Descriptor	10	CO6	L2
	OR			
10	Write an EJB program that demonstrates session bean with proper business logic	10	CO6	L4

CMR
INSTITUTE OF
TECHNOLOGY

USN

CELEBRATING 25 YEARS
CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 3– Jan 2022

Sub:	Advances in Java					Sub Code:	20MCA33	Branch:	MCA
Date:	25/1/2022	Duration:	90 min’s	Max Marks:	50	Sem	III		

1. Demonstrate a program to implement an Entity Bean.

Student.java

```
package Persist;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String usnno;
    private String name;
    private int mark;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Student)) {
            return false;
        }
        Student other = (Student) object;
        if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "Persist.Student[ id=" + id + " ]";
    }
    public String getUsnno() {
        return usnno;
    }

    public void setUsnno(String usnno) {
        this.usnno = usnno;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public int getMark() {
        return mark;
    }

    public void setMark(int mark) {
        this.mark = mark;
    }

}
```

StudServlet.java

```
package WebClient;
import Persist.Student;
import Persist.StudentFacadeLocal;
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;

public class StudServlet extends HttpServlet
{
    @EJB
    private StudentFacadeLocal studentFacade;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        Student obj=new Student();
        obj.setUsnno(request.getParameter("usnno"));
        obj.setName(request.getParameter("name"));
        obj.setMark(Integer.parseInt(request.getParameter("mark")));
        studentFacade.create(obj);

        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Student Data</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Congrats : Student " + request.getParameter("name") + " Record is created
successfully </h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }

    @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";
    }// </editor-fold>
}
```

<u>Index.html</u>

```
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
         <form method="get" action="StudServlet">
        Enter USN No. :<input type ="text" name ="usnno"/><br/>
        Enter Name    :<input type ="text" name ="name"/><br/>
        Enter Mark    :<input type ="text" name ="mark"/><br/>
            <input type="submit" value="Submit"/>
    </form>
    </body>
</html>
```

2.a. How are jar files created and used? Explain its advantages.

JAR files are packaged with the ZIP file format, so you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking. These tasks are among the most common uses of JAR files, and you can realize many JAR file benefits using only these basic features.

Creating a JAR File

The basic format of the command for creating a JAR file is:

jar cf *jar-file input-file(s)*

The options and arguments used in this command are:

The c option indicates that you want to *create* a JAR file.

The f option indicates that you want the output to go to a *file* rather than to stdout.

jar- file is the name that you want the resulting JAR file to have. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.

The input-file(s) argument is a space-separated list of one or more files that you want to include in your JAR file. The input- file(s)argument can contain the wildcard * symbol. If any of the "input- files" are directories, the contents of those directories are added to the JAR archive recursively.

The c and f options can appear in either order, but there must not be any space between them.

This command will generate a compressed JAR file and place it in the current directory.

c	-	creates a new or empty archive pm the Std output
t	-	lists the table of contents from std output

X file	–	it extracts all files or just the named files
f	-	The argument following this option specifies a JAR file to work
v	-	It generates verbose output on stderr
m	-	It includes manifest information from a specified manifest file
0	-	it indicates ‘store only’ without using ZIP compression
M	-	it specifies that a manifest file should not be created for the entries
u	-	It updates an exisiting JAR file by adding files or changing the Manifest

To create a JAR file	jar cf *jar-file input-file(s)*

To view the contents of a JAR file	jar tf *jar-file*
To extract the contents of a JAR file	jar xf *jar-file*
To extract specific files from a JAR file	jar xf *jar-file archived-file(s)*
To run an application packaged as a JAR file (requires the Main-class manifest header)	java -jar *app.jar*

Benefits of JAR

- Security: You can digitally sign the contents of a JAR file.
- Decreased download time: for Applets and Java Web Start
- Compression: efficient storage
- Packaging for extensions: extend JVM (example Java3D)
- Package Sealing: enforce version consistency
  - o all classes defined in a package must be found in the same JAR file
- Package Versioning: hold data like like vendor and version information
- Portability: the mechanism for handling JAR files is a standard part of the Java platform's core API

2.b. Explain bound and constrained properties of design patterns.

Bound Properties

- Bound properties generates an event when their values change.
- This event is of type **PropertyChangeEvent** and is sent to all registered event listeners of this type.
- To make a property a bound property, use the **setBound** method like

PropertyName.setBound(true)

For example **filled.setBound(true);**

- When bound property changes, an event is of type **PropertyChangeEvent**, is generated and a notification is sent to interested listeners.
- There is a standard listener class for this kind of event. Listeners need to implement this interface **PropertyChangeListener**

It has one method:

public void propertyChange(PropertyChangeEvent)

- A class that handles this event must implement the **PropertyChangeListene**r interface

Implement Bound Property in a Bean

- Declare and Instantiate a **PropertyChangeSupport** object that provides the bulk of bound property's functionality,

private PropertyChangeSupport changes = new PropertyChangeSupport(this);

- Implement registration and unregistration methods . The BeanBox will call these methods when a connection is made.

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

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

Constrained Properties:

- An object with constrained properties allows other objects to veto a constrained property value change.
- A bean property for which a change to the property results in validation byanother bean. The other bean may reject the change if it is not appropriate.
- Constrained property listeners can veto a change by throwing a **PropertyVetoException.**
- It generates an event called **PropertyChangeEvent** when an attempt is made to change its value
- This event is sent to objects that previously registered an interest in receiving an such notification
- Those objects have the ability to veto the proposed change
- This allows a bean to operate differently according to the runtime

environment Bean with constrained property must

1. Allow **VetoableChangeListener** object to register and unregister its interest in receiving notifications
2. Fire property change at those registered listeners. The event is fired before the actual property change takes place

Implementation of Constrained Property in a Bean

1. To support constrained properties the Bean class must instantiate the a **VetoableChangeSupport** object

```
    private VetoableChangeSupport vetos=new
                                         VetoableChangeSupport(this);
```

2. Define registration methods to add and remove vetoers.

```
public void addVetoableChangeListener(VetoableChangeListener v)
    {
          vetos.addVetoableChangeListener(v);
    }
    public void removeVetoableChangeListener(VetoableChangeListener v)
    {
         vetos.removeVetoableChangeListener(v);
    }
```

3. Write a property's setter method to fire a property change event and setter method throws a **PropertyVetoException**. In this method change the property and send change event to listeners.

3. Demonstrate a program to implement an Message Driven Bean.

<u>Mbean.java</u>

```
package com.message;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
```

```
@MessageDriven(mappedName = "jms/dist", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue")
})
public class MBean implements MessageListener
{
    public MBean()
   {
    }
        @Override
    public void onMessage(Message message)
    {
          TextMessage tmsg=null;
        tmsg=(TextMessage)message;
        try {
            System.out.println(tmsg.getText());
        } catch (JMSException ex) {
            Logger.getLogger(MBean.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

<u>NewServlet.java</u>

```
package com.web;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.annotation.Resource;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class NewServlet extends HttpServlet {
    @Resource(mappedName = "jms/dist")
    private Queue dist;
    @Resource(mappedName = "jms/queue")
    private ConnectionFactory queue;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        String str=request.getParameter("msg");
        try {
            sendJMSMessageToDist(str);
        } catch (JMSException ex) {
            Logger.getLogger(NewServlet.class.getName()).log(Level.SEVERE, null, ex);
        }
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet NewServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Your msg " + str + "has been sent to server pls check the log</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }

    @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";
    }// </editor-fold>

    private Message createJMSMessageForjmsDist(Session session, Object messageData) throws
JMSException {
        TextMessage tm = session.createTextMessage();
        tm.setText(messageData.toString());
        return tm;
    }

    private void sendJMSMessageToDist(Object messageData) throws JMSException {
        Connection connection = null;
        Session session = null;
        try {
            connection = queue.createConnection();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer = session.createProducer(dist);
            messageProducer.send(createJMSMessageForjmsDist(session, messageData));
        } finally {
            if (session != null) {
                try {
                    session.close();
                } catch (JMSException e) {
                    Logger.getLogger(this.getClass().getName()).log(Level.WARNING, "Cannot close
session", e);
                }
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```
<u>index.jsp</u>
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <form action="NewServlet">
            Your Message :
            <input type="text" name="msg">

            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

4.a. What is a manifest file? Mention its importance.

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

Applications Bundled as JAR Files: If an application is bundled in a JAR file, the Java Virtual Machine needs to be told what the entry point to the application is. An entry point is any class with a public static void main(String[] args) method. This information is provided in the Main-Class header, which has the general form:

```
Main-Class: classname
```

The value classname is to be replaced with the application's entry point.

Download Extensions: Download extensions are JAR files that are referenced by the manifest files of other JAR files. In a typical situation, an applet will be bundled in a JAR file whose manifest references a JAR file (or several JAR files) that will serve as an extension for the purposes of that applet. Extensions may reference each other in the same way. Download extensions are specified in the Class-Path header field in the manifest file of an applet, application, or another extension. A Class-Path header might look like this, for example:

```
Class-Path: servlet.jar infobus.jar acme/beans.jar
```

With this header, the classes in the files servlet.jar, infobus.jar, and acme/beans.jar will serve as extensions for purposes of the applet or application. The URLs in the Class-Path header are given relative to the URL of the JAR file of the applet or application.

Package Sealing: A package within a JAR file can be optionally sealed, which means that all classes defined in that package must be archived in the same JAR file. A package might be sealed to ensure version consistency among the classes in your software or as a security measure. To seal a package, a Name header needs to be added for the package, followed by a Sealed header, similar to this:

```
Name: myCompany/myPackage/
Sealed: true
```

The Name header's value is the package's relative pathname. Note that it ends with a '/' to distinguish it from a filename. Any headers following a Name header, without any intervening blank lines, apply to the file or package specified in the Name header. In the above example, because the Sealed header occurs after the Name: myCompany/myPackage header, with no blank lines between, the Sealed header will be interpreted as applying (only) to the package myCompany/myPackage.

Package Versioning: The Package Versioning specification defines several manifest headers to hold versioning information. One set of such headers can be assigned to each package. The versioning headers should appear directly beneath the Name header for the package. This example shows all the versioning headers:

```
Name: java/util/
Specification-Title: "Java Utility Classes"
Specification-Version: "1.2"
Specification-Vendor: "Sun Microsystems, Inc.".
Implementation-Title: "java.util"
Implementation-Version: "build57"
Implementation-Vendor: "Sun Microsystems, Inc."
```

4.b. List the differences between stateless session bean and stateful session bean.

Stateless:

1) Stateless session bean maintains across method and transaction
2) The EJB server transparently reuses instances of the Bean to service different clients at the per-method level (access to the session bean is serialized and is 1 client per session bean per method.
3) Used mainly to provide a pool of beans to handle frequent but brief requests. The EJB server transparently reuses instances of the bean to service different clients.
4) Do not retain client information from one method invocation to the next. So many require the client to maintain client side which can mean more complex client code.
5) Client passes needed information as parameters to the business methods.
6) Performance can be improved due to fewer connections across the network.

Stateful:

1) A stateful session bean holds the client session's state.
2) A stateful session bean is an extension of the client that creates it.

3) Its fields contain a conversational state on behalf of the session object's client. This state describes the conversation represented by a specific client/session object pair.
4) Its lifetime is controlled by the client.
5) Cannot be shared between clients.

5. With a neat block diagram, explain the life cycle of a stateful session bean.

Below Figure illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by invoking the create method.The EJB container instantiates the bean and then invokes the setSessionContext and ejbCreate methods in the session bean. The bean is now ready to have its business methods invoked.

While in the ready stage, the EJB container may decide to deactivate, or passivate, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the bean's ejbPassivate method immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the

EJB container activates the bean, moving it back to the ready stage, and then calls the bean's ejbActivate method.

At the end of the life cycle, the client invokes the remove method and the EJB container calls the bean's ejbRemove method. The bean's instance is ready for garbage collection.

Your code controls the invocation of only two life cycle methods-the create and remove methods in the client. All other methods in Figure are invoked by the EJB container. The ejbCreate method, for example, is inside the bean class, allowing you to perform certain operations right after the bean is instantiated. For instance, you may wish to connect to a database in the ejbCreate method.

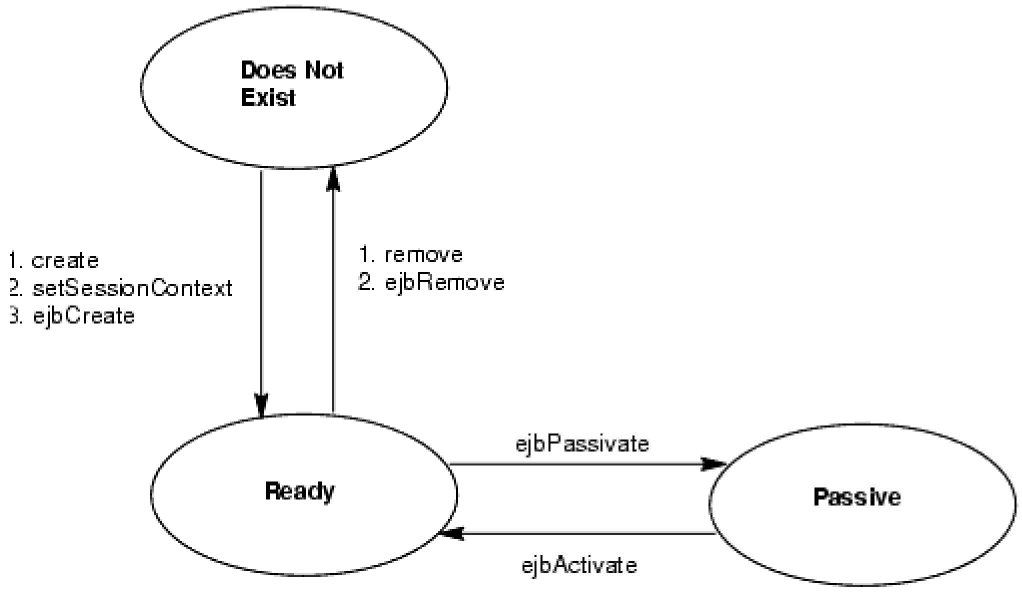


Figure Life Cycle of a Stateful Session Bean

6.a. What is entity bean? Explain the java persistence model in detail.

•An entity bean is a remote object that

–manages persistent data,

–performs complex business logic,

–uses several dependent Java objects, and

–can be uniquely identified by a primary key.

•Entity beans are normally coarse-grained persistent objects,

They utilize persistent data stored within several fine- grained persistent Java objects. Persistence

•JPA (Java Persistent API) - handles the plumbing between Java and SQL. EJB provides convenient integration with JPA via the entity bean.

•Persistence is a key piece of the Java EE platform.

–Older versionJ2EE, the EJB 2.x specification was responsible for defining this layer.

–In Java EE 5, persistence was spun off into its own specification.

–EE6, we have a new revision called the Java Persistence API, Version 2.0, or JPA.

•Persistence

–provides an ease-of-use abstraction on top of JDBC

–so that your code may be isolated from the database

–and vendor-specific peculiarities and optimizations.

•It can also be described as an object-to-relational mapping engine (ORM), which means that the Java Persistence API can automatically map your Java objects to and from a relational database.

•Entity - A persistent object representing the data-store record. It is good to be serializable.

•EntityManager - Persistence interface to do data operations like add/delete/update/find on persistent object(entity). It also helps to execute queries using Query interface.

•Persistence unit (persistence.xml) - Persistence unit describes the properties of persistence mechanism.

•Data Source (*ds.xml) - Data Source describes the data-store related properties like connection url. user-name,password etc.

Persistence Context

•A persistence context is a set of managed entity object instances .

•Persistence contexts are managed by an entity manager.

•The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database using the flush mode rules

•Once a persistence context is closed, all managed entity object instances become detached and are no longer managed.

•Once an object is detached from a persistence context, it can no longer be managed by an entity manager, and any state changes to this object instance will not be synchronized with the database.

•When a persistence context is closed, all managed entity objects become detached and are unmanaged.

•There are two types of persistence contexts:

•transaction-scoped persistence context

•extended persistence context.

Transaction Scoped Persistence context

•Everything executed

•Either fully succeed or fully fail

6.b. Define introspection. Explain simple properties with it.

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.

2 ways by which the developer of a Bean can indicate which of its properties, events and methods should be exposed by anapplication builder tool.

- Simple naming conventions are used, which allows to infer information about a bean
- Additional class that extends the BeanInfo interface that explicitly supplies this information

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;

}
```

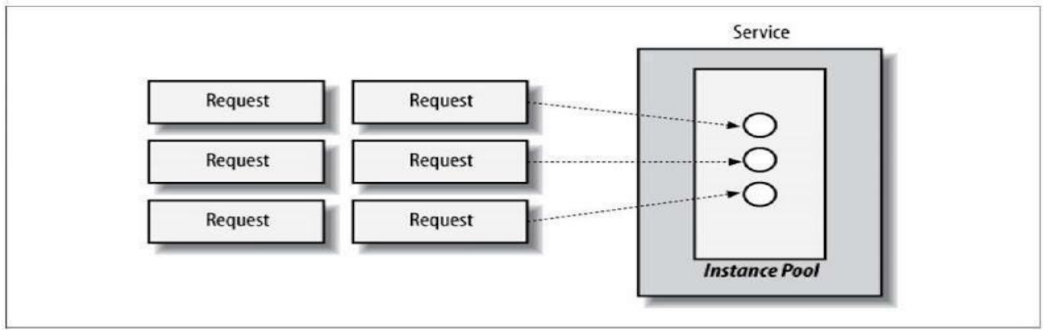
7. List all the EJB container services and explain any five in detail.

1. Dependency Injection
2. Concurrency
3. Instance Polling and Caching
4. Transactions
5. Security
6. Timers
7. Naming and Object Stores, JNDI
8. Lifecycle callbacks
9. Interoperability
10. Interceptors

1. Instance Pooling/Caching

Because of the strict concurrency rules enforced by the Container, an intentional bottleneck is often introduced where a service instance may not be available for processing until some other request has completed.

If the service was restricted to a singular instance, all subsequent requests would have to queue up until their turn was reached



addresses this problem through a technique called instance pooling, in which each odule is allocated some number of instances with which to serve incoming requests Many vendors provide configuration options to allocate pool sizes appropriate to the work being performed, providing the compromise needed to achieve optimal throughput.

2. Transactions

Transactions provide a means for the developer to easily delegate the creation and control of transactions to the container.

- When a bean calls createTimer(), the operation is performed in the scope of the current transaction. If the transaction rolls back, the timer is undone and it’s not created
- The timeout callback method on beans should have a transaction attribute of RequiresNew.
- This ensures that the work performed by the callback method is in the scope of container-initiated transactions.

3. Security

Most enterprise applications are designed to serve a large number of clients, and users are not necessarily equal in terms of their access rights.
An administrator might require hooks into the configuration of the system, whereas unknown guests may be allowed a read-only view of data.
If we group users into categories with defined roles, we can then allow or restrict access to the role itself, as illustrated in Figure 15-1.

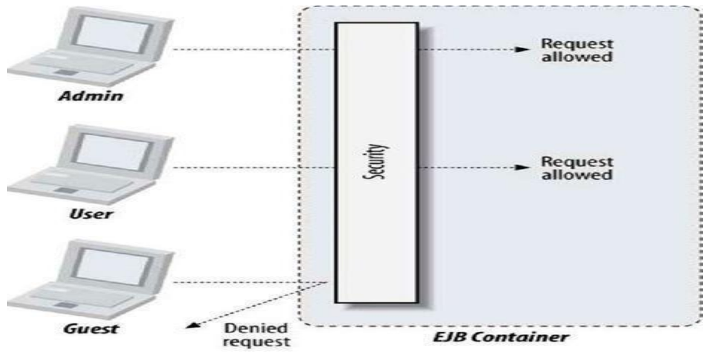


Figure 15-1. EJB security permitting access based upon the caller’s role

This allows the application developer to explicitly allow or deny access at a fine- grained level based upon the caller’s identity

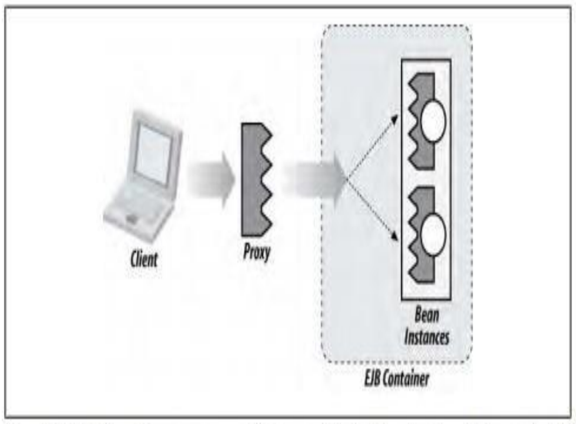


Figure 2-1. Client invoking upon a proxy object, responsible for delegating the call along to the EJB Container

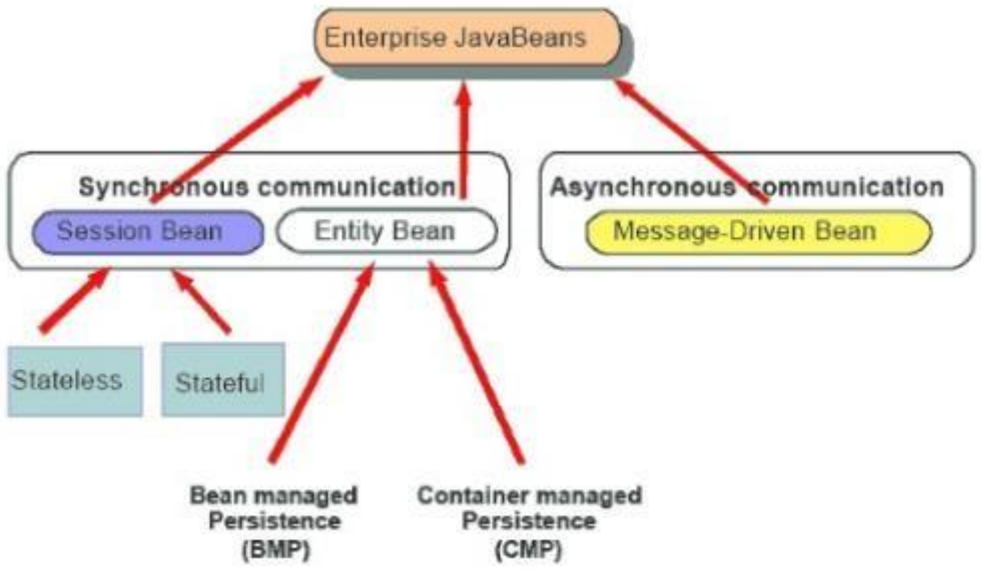
4. Timers

We dealt exclusively with client-initiated requests. While this may handle the bulk of an application's requirements, it doesn't account for scheduled jobs:

• A ticket purchasing system must release unclaimed tickets after some timeout of inactivity.

• An auction house must end auctions on time.
• A cellular provider should close and mail statements each month.
The EJB Timer Service may be leveraged to trigger these events and has been enhanced in the 3.1 specification with a natural- language expression syntax.

8. Discuss the classes of EJB and depict the various components of interaction with a neat diagram


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.

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. This allows the Container to hold a much smaller number of objects in service, hence keeping memory footprint down.

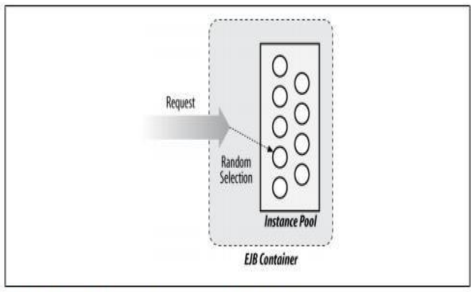


Figure 2-2. An SLSB Instance Selector picking an instance at random

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.

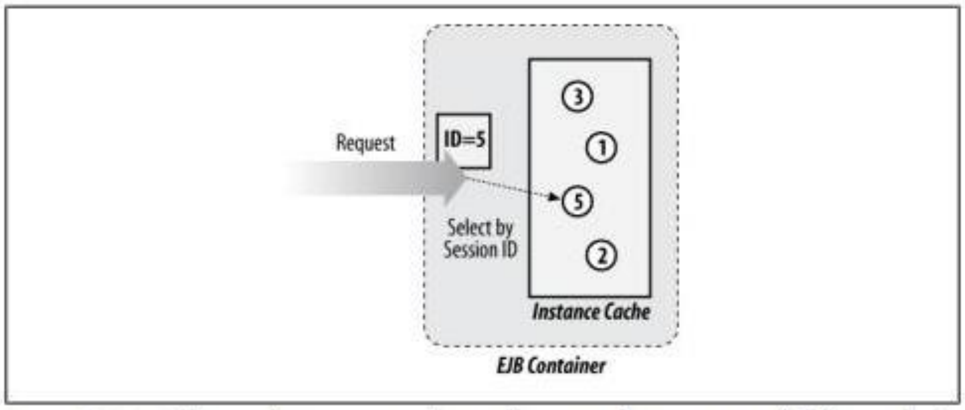


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

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.

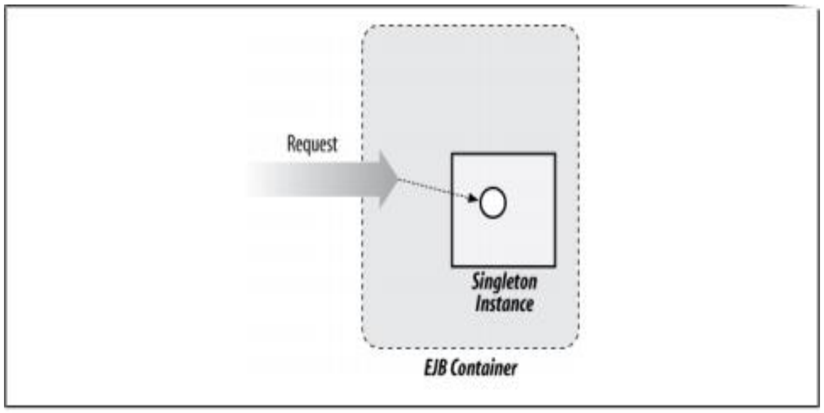


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

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

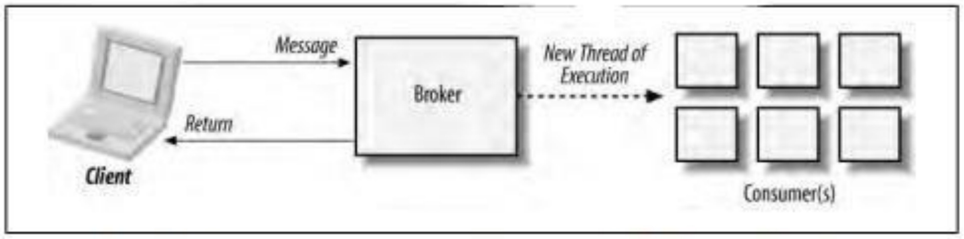


Figure 2-5. Asynchronous invocation of a message-driven bean, which acts as a listener for incoming events

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.

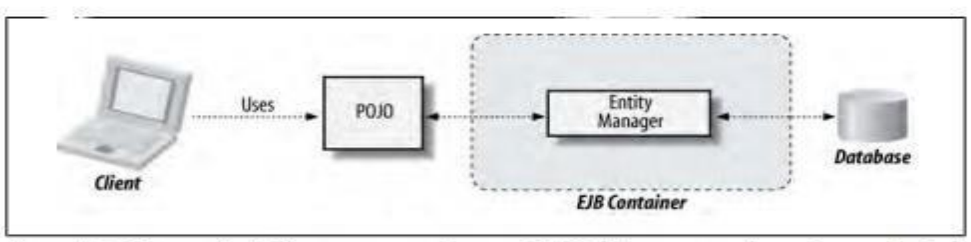


Figure 2-6. Using an EntityManager to map between POJO object state and a persistent relational database

9. Write the short note about the following

i)Persistence Context

ii)XML Deployment Descriptor

(i) Persistence Context

- **A persistence context is a set of managed entity object instances**.
- Persistence contexts are managed by an entity manager.
- The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database using the flush mode rules
- Once a persistence context is closed, all managed entity object instances become detached and are no longer managed.
- Once an object is detached from a persistence context, it can no longer be managed by an entity manager, and any state changes to this object instance will not be synchronized with the database.
- When a persistence context is closed, all managed entity objects become detached and are unmanaged.
- There are two types of persistence contexts:
  - transaction-scoped persistence context
  - extended persistence context.

Transaction Scoped Persistence context

- Everything executed
- Either fully succeed or fully fail

(ii) XML 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.

10. Write an EJB program that demonstrates session bean with proper business logic

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>
```