CMR INSTITUTE OF TECHNOLOGY

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

Internal Assessment Test 3 – Mar. 2023

Sub:	Advances in Java							Sub Code:	20MCA33
Date:	14/3/2023	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.	Discuss the interfaces used in java.beans package.	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 package? With an example, explain usage of sub packages	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.	Explain any four annotations used in EJB along with their meaning.	6	CO6	L2
b.	Define introspection. Explain simple properties with it.	4	CO5	L2
	PART IV			
7	Briefly explain the following terms: a) MDB b) Dependency Injection c) Interceptors d) Naming and Object Stores	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 | | | | | | | | | | |

Internal Assessment Test 3– Mar 2023

Sub:	Advances in Java					Sub Code:	20MCA33	Branch:	MCA
Date:	14/3/2023	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>
}
```

Index.html

```
<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. Discuss the interfaces used in java.beans package.

Contains classes related to developing *beans* -- components based on the JavaBeansTM architecture
Interface Summary
AppletInitializer This interface is designed to work in collusion with java.beans.Beans.instantiate.
BeanInfo
A bean implementor who wishes to provide explicit information about their bean may provide a 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.
DesignMode
This interface is intended to be implemented by, or delegated from, instances of 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.

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 by another 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.

Mbean.java

```
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);
      }
   }
}
```

NewServlet.java

```
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();
            }
        }
    }
}
```

index.jsp

```
<%@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 package? With an example, explain usage of sub packages

java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc. Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on. package importpackage.subpackage;

```
public class HelloWorld {
 public void show(){
 System.out.println("This is the function of the class
HelloWorld!!");
 }
 }
```

Now import the package "subpackage" in the class file "CallPackage" shown as:

```
import importpackage.subpackage.*;
class CallPackage{
 public static void main(String[] args){
 HelloWorld h2=new HelloWorld();
 h2.show();
 }
 }
```

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.

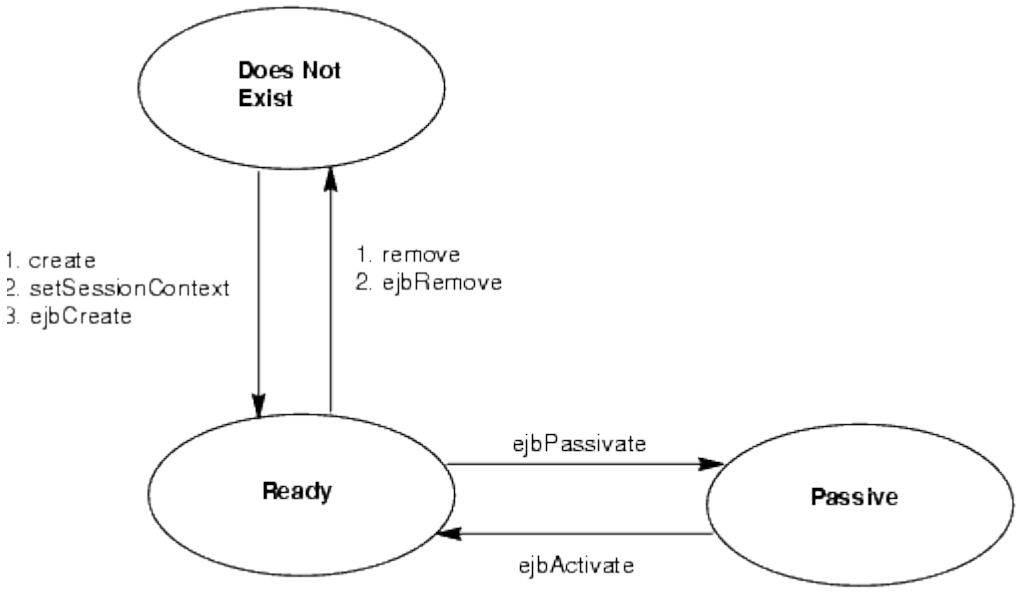


Figure Life Cycle of a Stateful Session Bean

6.a. Explain any four annotations used in EJB along with their meaning.

	Name	Description
	javax.ejb.Stateless	Specifies that a given EJB class is a stateless session bean. Attributes name − Used to specify name of the session bean. mappedName − Used to specify the JNDI name of the session bean. description − Used to provide description of the session bean.

	javax.ejb.Stateful	Specifies that a given EJB class is a stateful session bean. Attributes name − Used to specify name of the session bean. mappedName − Used to specify the JNDI name of the session bean. description − Used to provide description of the session bean.
	javax.ejb.Message DrivenBean	Specifies that a given EJB class is a message driven bean. Attributes name − Used to specify name of the message driven bean. messageListenerInterface− Used to specify message listener interface for the message driven bean. activationConfig − Used to specify the configuration details of the message-driven bean in an operational environment of the message driven bean. mappedName − Used to specify the JNDI name of the session bean. description − Used to provide description of the session bean.

	javax.ejb.EJB	Used to specify or inject a dependency as EJB instance into another EJB. Attributes name − Used to specify name, which will be used to locate the referenced bean in the environment. beanInterface − Used to specify the interface type of the referenced bean. beanName − Used to provide name of the referenced bean. mappedName − Used to specify the JNDI name of the referenced bean. description − Used to provide description of the referenced bean.

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 an application 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. Briefly explain the following terms:

e) MDB

f) Dependency Injection

g) Interceptors

h) Naming and Object Stores

a) MDB:

MDB are

- stateless,
- server-side
- transaction-aware components
- for processing asynchronous message delivered via JMS
- Asynchronous message is a paradigm in which
- two or more applications communicate via a message describing a business event.

A message driven bean (MDB) is a bean that contains business logic. But, it is invoked by passing the message. Hence , it is like JMS Receiver.

- MDB asynchronously receives the message and processes it.
- A message driven bean receives message from queue or topic
- A message driven bean is like stateless session bean that encapsulates the business logic and doesn't maintain state.Messaging is a technique to communicate applications or software components.
- JMS is mainly used to send and receive message from one application to another.
- JMS (Java Message Service) is an API that provides the facility to create, send and read messages.

- It provides loosely coupled, reliable and asynchronous communication. JMS is also known as a messaging service.
- There are two types of messaging domains in JMS.

1. Queue Model - Point-to-Point Messaging Domain
2. Topic Model -Publisher/Subscriber Messaging Domain

b)Dependency Injection

Dependency Injection

- EJB – Component-centric Architecture, it provides a means to

reference dependent modules in decoupled fashion.

- Result – container provide the implementation at deployment time

```
prototype UserModule
{
// Instance Member
@DependentModule
MailModule mail;
// A function
function mailUser()
{
mail.sendMail("me@ejb.somedomain");
}
}
```

Depend Module annotation serves two purposes

i. Defines a dependency upon some service of type MailModule. UserModule may not deploy until this dependency is satisfied.

ii. Marks the instance member mail as a candidate for injection. The container will populate this field during deployment. Dependency injection encourages coding to interfaces

c)Interceptors

EJB provides aspectized handling of many of the container

• services, the specification cannot possibly identify all cross-cutting concerns facing your project.

• For this reason, EJB makes it possible to define custom interceptors upon business methods and lifecycle callbacks.

• This makes it easy to contain some common code in a centralized location and have it applied to the invocation chain without impacting your core logic.

```
prototype MetricsInterceptor
```

```
{
function intercept(Invocation invocation)
{
// Get the start time
Time startTime = getTime();
// Carry out the invocation
invocation.continue();
// Get the end time
Time endTime = getTime();
// Log out the elapsed time
log("Took " + (endTime − startTime));
}
}
```

Then we could apply this to methods as we'd like:

```
@ApplyInterceptor(MetricsInterceptor.class)
function myLoginMethod{ ... }
@ApplyInterceptor(MetricsInterceptor.class)
function myLogoutMethod{ ... }
```

e) Naming and Object Stores

Provide clients with a mechanism for locating distributed objects or resources.

Naming service much fulfill two requirements:

1. object binding and a lookup API.

Object binding is the association of a distributed object with a natural language name or identifier.

2. A lookup API provides the client with an interface to the naming system;

Allows us to connect with a distributed service and request a remote reference to a specific object.

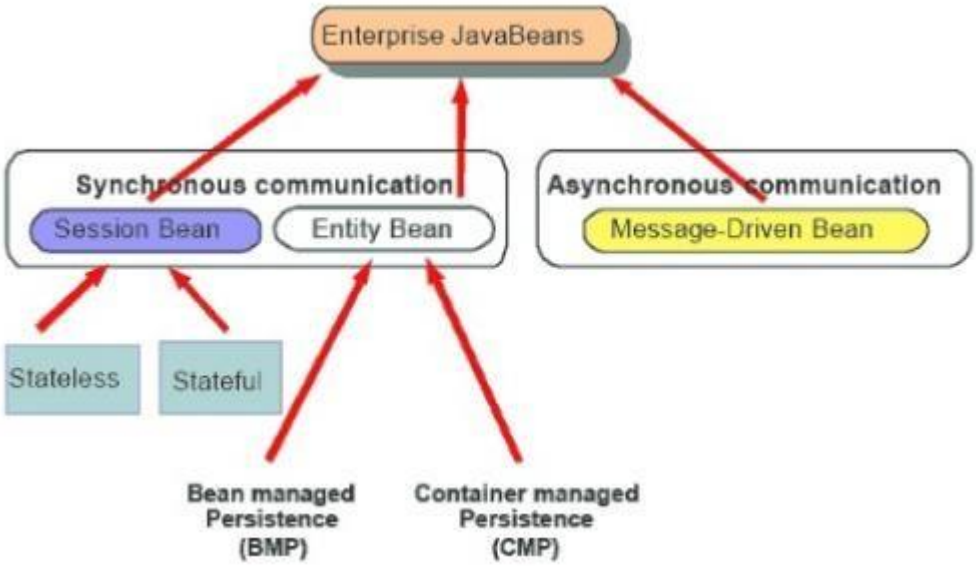
Enterprise JavaBeans mandates the use of Java Naming and Directory Interface (JNDI)as a lookup API on Java clients.

• JNDI supports any kind of naming and directory service.

• It is complex but, the way JNDI is used in Java Enterprise Edition (EE) applications is usually simple.

• Java client applications can use JNDI to initiate a connection to an EJB server and locate a specific EJB.

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



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.

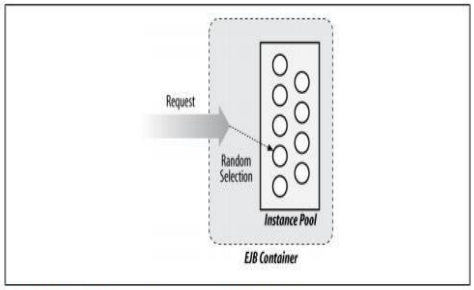


Figure 2-2. An SLSB Instance Selector picking an instance at random

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.

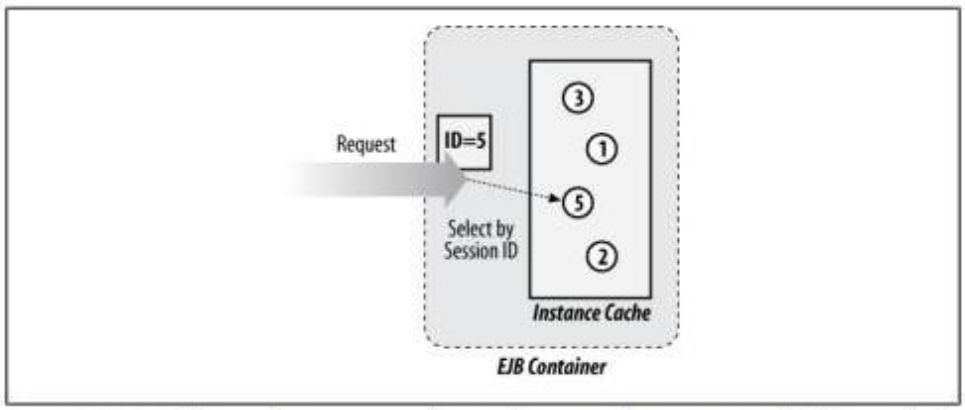


Figure 2-3. Stateful session bean creating and using the correct client instance, which lives inside the EJB Container, to carry out the invocation

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.

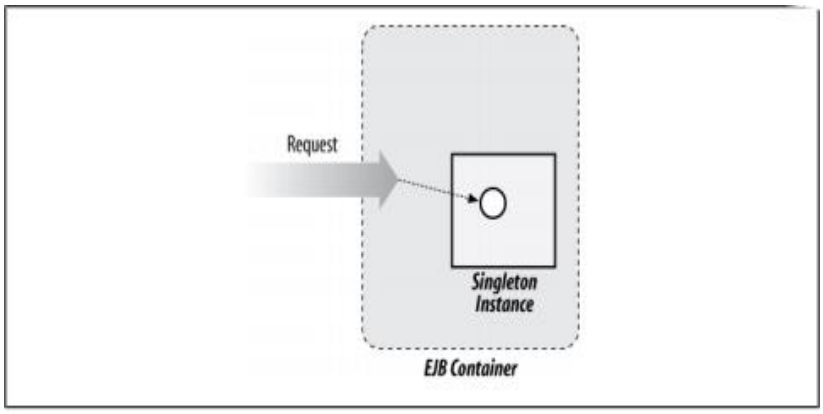


Figure 2-4. Conceptual diagram of a singleton session bean with only one backing bean instance

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

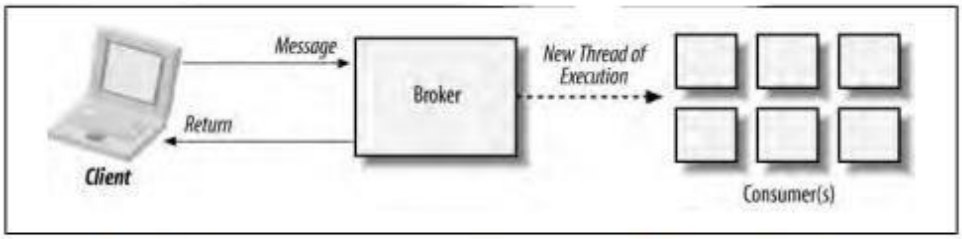


Figure 2-5. Asynchronous invocation of a message-driven bean, which acts as a listener for incoming events

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.

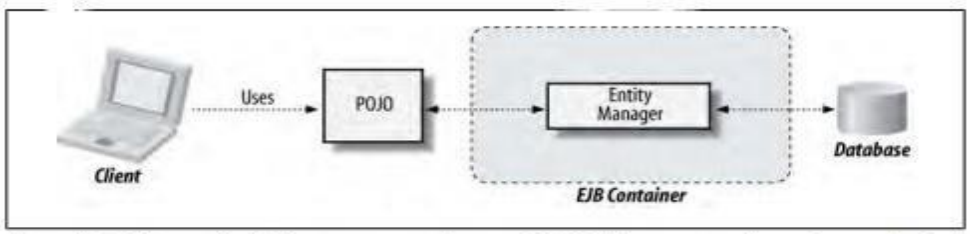


Figure 2-6. Using an EntityManager to map between POJO object state and a persistent relational database

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:

1) transaction-scoped persistence context
2) extended persistence context.

Transaction Scoped Persistence context

ii) Everything executed
iii) 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>
```