UNIT 1 INTRODUCTION TO JAVA BEANS

Structure **Page Nos.**

1.0 INTRODUCTION

Software has been evolving at a tremendous speed since its inception. It has gone through various phases of development from low level language implementation to high level language implementation to non procedural program models. The designers always make efforts to make programming easy for users and near to the real world implementation. Various programming models were made public like procedural programming, modular programming and non procedural programming model. Apart from these models reusable component programming model was introduced to put programmers at ease and to utilise the reusability as its best. Reusable Component model specifications were adopted by different vendors and they came with their own component model solutions.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- define what is Java Beans;
- list EJB types;
- discuss about Java Architecture;
- make differences between different types of Beans;
- discuss the life cycle of a Beans, and
- describe the use of Message Driven Beans.

1.2 WHAT IS JAVABEAN?

JavaBeans are software component models. A JavaBean is a general-purpose component model. A Java Bean is a reusable software component that can be visually manipulated in builder tools. Their primary goal of a JavaBean is **WORA** (Write Once Run Anywhere). JavaBeans should adhere to portability, reusability and interoperability.

JavaBeans will look a plain Java class written with getters and setters methods. It's logical to wonder: “What is the difference between a Java Bean and an instance of a normal Java class?” What differentiates Beans from typical Java classes is **introspection**. Tools that recognize predefined patterns in method signatures and class definitions can "look inside" a Bean to determine its properties and behaviour.

A Bean’s state can be manipulated at the time it is being assembled as a part within a larger application. The application assembly is referred to as **design time** in contrast to **run time**. For this scheme to work, method signatures within Beans must follow a certain pattern, for introspection tools to recognise how Beans can be manipulated, both at design time, and run time.

In effect, Beans publish their attributes and behaviours through special method signature patterns that are recognised by beans-aware application construction tools. However, you need not have one of these construction tools in order to build or test your beans. Pattern signatures are designed to be easily recognised by human readers as well as builder tools. One of the first things you’ll learn when building beans is how to recognise and construct methods that adhere to these patterns.

Not all useful software modules should be Beans. Beans are best suited to software components intended to be visually manipulated within builder tools. Some functionality, however, is still best provided through a programatic (textual) interface, rather than a visual manipulation interface. For example, an SQL, or JDBC API would probably be better suited to packaging through a class library, rather than a Bean.

1.3 JAVABEAN CONCEPTS

JavaBeans is a complete component model. It supports the standard component architecture features of properties, events, methods, and persistence. In addition, JavaBeans provides support for introspection (to allow automatic analysis of a JavaBeans component) and customisation (to make it easy to configure a JavaBeans component). Typical unifying features that distinguish a Bean are:

- **Introspection:** Builder tools discover a Bean’s features (ie its properties, methods, and events) by a process known as INTROSPECTION. Beans supports introspection in two ways:

1) **Low Level Introspection (Reflection) + Intermediate Level Introspection (Design Pattern):** Low Level Introspection is accomplished using **java.lang.reflect** package API. This API allows Java Objects to discover information about public fields, constructors, methods and events of loaded classes during program execution i.e., at Run-Time. Intermediate Level Introspection (Design Pattern) is accomplished using **Design Patterns**. Design Patterns are bean features naming conventions to which one has to adhere while writing code for Beans. java.beans.Introspector class examines Beans for these design patterns to discover Bean features. The Introspector class relies on the core reflection API. There are two types of methods namely, **accessor methods** and **interface methods**. Accessor methods are used on properties and are of

two sub-types (namely **getter methods and setters methods**). Interface methods are often used to support event handling.

2) **Higest Level or Explicit Introspection (BeanInfo):** It is accomplished by explicitly providing property, method, and event information with a related Bean Information Class. A Bean information class implements the BeanInfo interface. A BeanInfo class explicitly lists those Bean features that are to be exposed to the application builder tools. The Introspector recognises BeanInfo classes by their name. The name of a BeanInfo class is the name of the bean class followed by BeanInfo word e.g., for a bean named "Gizmo" the BeanInfo name would be "GizmoBeanInfo".

   - **Properties:** Are a Bean's appearance and behaviour characteristics that can be changed at design time.

   - **Customisation:** Beans expose properties so they can be customised during the design time.

   - **Events:** Enables Beans to communicate and connect to each other.

   - **Persistence:** The capability of permanently stored property changes is known as Persistence. Beans can save and restore their state i.e., they need to be persistent. It enables developers to customise Beans in an app builder, and then retrieve those Beans, with customised features intact, for future use. JavaBeans uses **Java Object Serialisation** to support persistence. **Serialisation** is the process of writing the current state of an object to a stream. To serialise an object, the class must implement either java.io.Serialisable or java.io.Externalisable interface. Beans that implement Serialisable are automatically saved and beans that implements Externalisable are responsible for saving themselves. The transient and static variables are not serialised i.e., these type of variables are not stored.

Beans can also be used just like any other Java class, manually (i.e., by hand programming), due to the basic Bean property, "Persistence". Following are the two ways:

- Simply instantiate the Bean class just like any other class.

- If you have a customised Bean (through some graphic tool) saved into a serialised file (say mybean.ser file), then use the following to create an instance of the Customised Bean class...

```
try {
          MyBean mybean = (MyBean)
          Beans.instantiate(null, "mybean");
     } catch (Exception e) {
     }
```

- **Connecting Events:** Beans, being primarily GUI components, generate and respond to events. The bean generating the event is referred to as *event source* and the bean listening for (and handling) the event is referred to as the *event listener*.

- **Bean Properties:** Bean properties can be categorised as follows...

  1) **Simple Property** are basic, independent, individual prperties like width, height, and colour.

  2) **Indexed Property** is a property that can take on an array of values.

  3) **Bound Property** is a property that alerts other objects when its value changes.

4) **Constrained Property** differs from Bound Property in that it notifies other objects of an impending change. Constrained properties give the notified objects the power to veto a property change.

Accessor Methods

1. **Simple Property :**

   If, a bean has a property named **foo** of type **fooType** that can be read and written, it should have the following accessor methods:

```
public fooType getFoo( ) { return foo; }
public void setFoo(fooType fooValue) {
      foo = fooValue; ...
}
```

   If a property is boolean, getter methods are written using **is** instead of **get** eg isFoo( ).

2. **Indexed Property :**

```
public widgetType getWidget(int index)
public widgetType[] getWidget( )
public void setWidget(int index, widgetType widgetValue)
public void setWidget(widgetType[] widgetValues)
```

3. **Bound Property :**

   Getter and setter methods for bound propery are as described above based on whether it is simple or indexed. Bound properties require certain objects to be notified when they change. The change notification is accomplished through the generation of a **PropertyChangeEvent** (defined in java.beans). Objects that want to be notified of a property change to a bound property must register as listeners. Accordingly, the bean that's implementing the bound property supplies methods of the form:

```
public void addPropertyChangeListener(ropertyChangeListener l)
public void removePropertyChangeListener(PropertyChangeListener l)
```

   The preceding listener registeration methods do not identify specific bound properties. To register listeners for the PropertyChangeEvent of a specific property, the following methods must be provided:

```
public void addPropertyNameListener(PropertyChangeListener l)
public void emovePropertyNameListener(PropertyChangeListener l)
```

   In the preceding methods, **PropertyName** is replaced by the name of the bound property.

   Objects that implement the PropertyChangeListener interface must implement the **PropertyChange( )** method. This method is invoked by the bean for all registered listeners to inform them of a property change.

4. **Constrained Property :**

   The previously discussed methods used with simple and indexed properties also apply

to the constrained properties. In addition, the following event registeration methods provided:

```
public void addVetoableChangeListener(VetoableChangeListener l)
public void removeVetoableChangeListener(VetoableChangeListener l)
public void addPropertyNameListener(VetoableChangeListener l)
public void removePropertyNameListener(VetoableChangeListener l)
```

Objects that implement the VetoableChangeListener interface must implement the vetoableChange( ) method. This method is invoked by the bean for all of its registered listeners to inform them of a property change. Any object that does not approve of a property change can throw a PropertyVetoException within its vetoableChange( ) method to inform the bean whose constrained property was changed that the change was not approved.

Inside java.beans package

The classes and packages in the java.beans package can be categorised into three types (NOTE: following is not the complete list).

1) **Design Support**

Classes - Beans, PropertyEditorManager, PropertyEditorSupport
Interfaces - Visibility, VisibilityState, PropertyEditor, Customizer

2) **Introspection Support**.

Classes - Introspector, SimpleBeanInfo, BeanDescriptor, EventSetDescriptor, FeatureDescriptor, IndexedPropertyDescriptor, MethodDescriptor, ParameterDescriptor, PropertyDescriptor
Interfaces - BeanInfo

3) **Change Event-Handling Support**.

Classes - PropertyChangeEvent, VetoableChangeEvent, PropertyChangeSupport, VetoableChangeSupport
Interfaces - PropertyChangeListener, VetoableChangeListener

1.4 WHAT IS EJB?

Enterprise JavaBeans are software component models, their purpose is to build/support enterprise specific problems. EJB - is a reusable server-side software component. Enterprise JavaBeans facilitates the development of distributed Java applications, providing an object-oriented transactional environment for building distributed, multi-tier enterprise components. An EJB is a remote object, which needs the services of an EJB container in order to execute.

The primary goal of a EJB is **WORA** (Write Once Run Anywhere). Enterprise JavaBeans takes a high-level approach to building distributed systems. It frees the application developer and enables him/her to concentrate on programming only the business logic while removing the need to write all the “plumbing” code that's required in any enterprise application. For example, the enterprise developer no longer needs to write code that handles transactional behaviour, security, connection pooling, networking or threading. The architecture delegates this task to the server vendor.

1.5 EJB ARCHITECTURE

The Enterprise JavaBeans spec defines a server component model and specifies, how to create server-side, scalable, transactional, multiuser and secure enterprise-level components. Most important, EJBs can be deployed on top of existing transaction processing systems including traditional transaction processing monitors, Web, database and application servers.

A typical EJB architecture consists of:

- **EJB clients:** EJB client applications utilise the Java Naming and Directory Interface (JNDI) to look up references to home interfaces and use home and remote EJB interfaces to utilise all EJB-based functionality.
- **EJB home interfaces (and stubs):** EJB home interfaces provide operations for clients to create, remove, and find handles to EJB remote interface objects. Underlying stubs marshal home interface requests and unmarshal home interface responses for the client.
- **EJB remote interfaces (and stubs):** EJB remote interfaces provide business-specific client interface methods defined for a particular EJB. Underlying stubs marshal remote interface requests and unmarshal remote interface responses for the client.
- **EJB implementations:** EJB implementations are the actual EJB application components implemented by developers to provide any application-specific business method invocation, creation, removal, finding, activation, passivation, database storage, and database loading logic.
- **Container EJB implementations (skeletons and delegates):** The container manages the distributed communication skeletons used to marshal and unmarshal data sent to and from the client. Containers may also store EJB implementation instances in a pool and use delegates to perform any service-management operations related to a particular EJB before calls are delegated to the EJB implementation instance.

Some of the advantages of pursuing an EJB solution are:

- EJB gives developers architectural independence.
- EJB is WORA for server-side components.
- EJB establishes roles for application development.
- EJB takes care of transaction management.
- EJB provides distributed transaction support.
- EJB helps create portable and scalable solutions.
- EJB integrates seamlessly with CORBA.
- EJB provides for vendor-specific enhancements.

1.6 BASIC EJB EXAMPLE

To create an EJB we need to create Home, Remote and Bean classes.

Home Interface

```
import java.rmi.*;
import javax.ejb.*;
import java.util.*;
public interface HelloObject extends EJBObject {
  public String sayHello() throws RemoteException;
}
```

Remote Interface

```
import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface HelloHome extends EJBHome {
  public HelloObject create() throws RemoteException,
CreateException;
}
```

Bean Implementation

```
import java.rmi.RemoteException;
import javax.ejb.*;

public class HelloBean implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext)
{
    this.sessionContext = sessionContext;
  }
  public String sayHello() throws java.rmi.RemoteException {
    return "Hello World!!!!!";
  }
}
```

Deployment Descriptor

```
<ejb-jar>
<description>HelloWorld deployment descriptor</description>
<display-name>HelloWorld</display-name>
<enterprise-beans>
<session>
<description> HelloWorld deployment descriptor
</description>
<display-name>HelloWorld</display-name>
<ejb-name>HelloWorld</ejb-name>
<home>HelloWorldHome</home>
<remote>HelloWorld</remote>
<ejb-class>HelloWorldBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>HelloWorld</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

EJB Client

```
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.transaction.UserTransaction;
import javax.rmi.PortableRemoteObject;

public class HelloClient {
        public static void main(String args[]) {
            try {
                Context initialContext = new InitialContext();
                Object objref =
initialContext.lookup("HelloWorld");
                HelloWorldHome home =

        (HelloWorldHome)PortableRemoteObject.narrow(objref,
                HelloWorldHome.class);
                HelloWorld myHelloWorld = home.create();
                String message = myHelloWorld.sayHello();
                System.out.println(message);
                } catch (Exception e) {
                        System.err.println(" Erreur : " + e);
                        System.exit(2);
            }
        }
}
```

1.7 EJB TYPES

EJBs are distinguished along three main functional roles. Within each primary role, the EJBs are further distinguished according to subroles. By partitioning EJBs into roles, the programmer can develop an EJB according to a more focused programming model than, if, for instances such roles were not distinguished earlier. These roles also allow the EJB container to determine the best management of a particular EJB based on its programming model type.

There are three main types of beans:

- Session Bean
- Entity Beans
- Message-driven Beans

1.7.1 Session Bean

A session EJB is a non persistent object. Its lifetime is the duration of a particular interaction between the client and the EJB. The client normally creates an EJB, calls methods on it, and then removes it. If, the client fails to remove it, the EJB container will remove it after a certain period of inactivity. There are two types of session beans:

- **Stateless Session Beans:** A stateless session EJB is shared between a number of clients. It does not maintain conversational state. After each method call, the container may choose to destroy a stateless session bean, or recreate it, clearing itself out, of all the information pertaining the invocation of the last method. The algorithm for creating new instance or instance reuse is container specific.

- **Stateful Session Beans:** A stateful session bean is a bean that is designed to service business processes that span multiple method requests or transaction. To do this, the stateful bean retains the state for an individual client. If, the stateful bean's state is changed during method invocation, then, that same state will be available to the same client upon invocation.

1.7.1.1 Life Cycle of a Stateless Session Bean

The *Figure 1* shows the life cycle of a Stateless Session Bean.

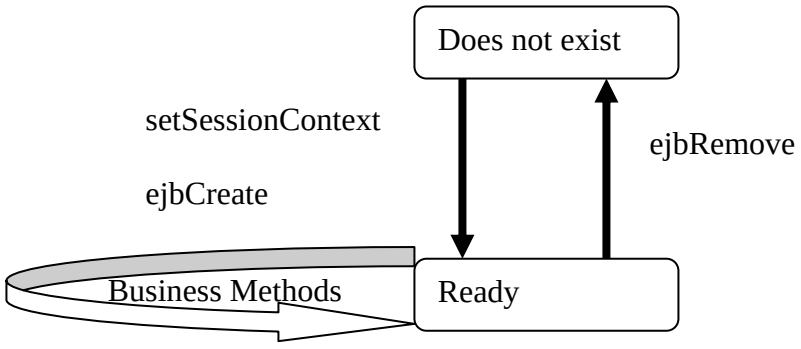

Figure 1: Life Cycle of Stateless Session Bean

- **Does not exist:** In this state, the bean instance simply does not exist.
- **Ready state:** When EJB Server is first started, several bean instances are created and placed in the Ready pool. More instances might be created by the container as and when needed by the EJB container

1.7.1.2 Life Cycle of a Stateful Session Bean

The *Figure 2* shows the life cycle of a Stateful Session Bean.

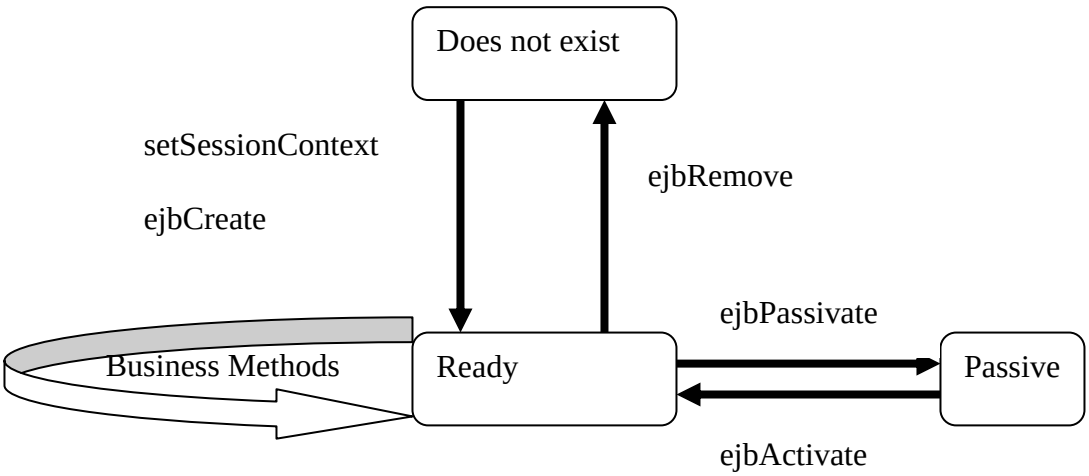

Figure 2: Life cycle of a stateful session bean

- **Does not exist:** In this state, the bean instance simply does not exist.
- **Ready state:** A bean instance in the ready state is tied to a particular client and engaged in a conversation.
- **Passive state:** A bean instance in the passive state is passivated to conserve resources.

1.7.1.3 Required Methods in Session Bean

The following are the required methods in a Session Bean:

setSessionContext(SessionContext ctx) :
Associate your bean with a session context. Your bean can make a query to the context about its current transactional state, and its current security state.

ejbCreate(…) :

Initialise your session bean. You would need to define several ejbCreate (…) methods and then, each method can take up different arguments. There should be at least one ejbCreate() in a session bean.

ejbPassivate():

This method is called for, just before the session bean is passivated and releases any resource that bean might be holding.

ejbActivate():

This method is called just for, before the session bean is activated and acquires the resources that it requires.

ejbRemove():

This method is called for, by the ejb container just before the session bean is removed from the memory.

1.7.1.4 The use of a Session Bean

In general, one should use a session bean if the following circumstances hold:

- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period and therefore.
- The bean implements a web service.

Stateful session beans are appropriate if, any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans.

To improve performance, one might choose a stateless session bean if, it has any of these traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send a promotional email to several registered users.

1.7.2 Entity Bean

Entity EJBs represent persistent objects. Their lifetimes is not related to the duration of interaction with clients. In nearly all cases, entity EJBs are synchronised with relational databases. This is how persistence is achieved. Entity EJBs are always shared amongst clients. A client cannot get an entity EJB to itself. Thus, entity EJBs are nearly always used as a scheme for mapping relational databases into object-oriented applications. An important feature of entity EJBs is that they have identity—that is, one can be distinguished from another. This is implemented by assigning a primary key to each instance of the EJB, where 'primary key' has the same meaning as it does for database management. Primary keys that identify EJBs can be of any type, including programmer-defined classes.

There are two type of persistence that entity EJB supports. These persistence types are:

- **Bean-managed persistence (BMP):** The entity bean's implementation manages persistence by coding database access and updating statements in callback methods.
- **Container-managed persistence (CMP):** The container uses specifications made in the deployment descriptor to perform database access and update statements automatically.

1.7.2.1 Life Cycle of an Entity Bean

The *Figure 3* shows the life cycle of an entity bean.

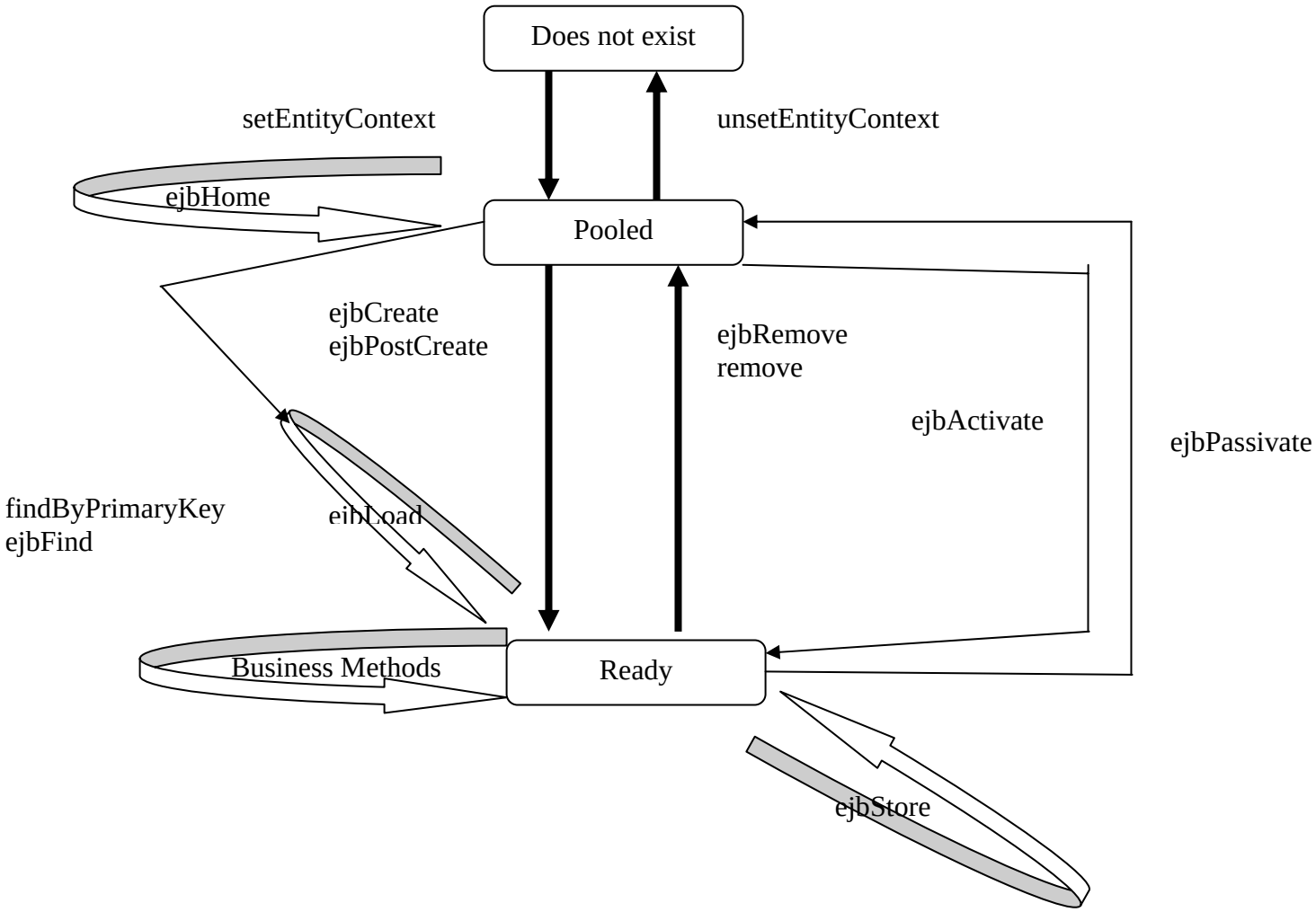

Figure 3: Life cycle of an entity bean

An entity bean has the following three states:

- **Does not exist:** In this state, the bean instance simply does not exist.
- **Pooled state:** When the EJB server is first started, several bean instances are created and placed in the pool. A bean instance in the pooled state is not tied to a particular data, that is, it does not correspond to a record in a database table. Additional bean instances can be added to the pool as needed, and a maximum number of instances can be set.
- **Ready state:** A bean instance in the ready state is tied to a particular data, that is, it represents an instance of an actual business object.

1.7.2.2 Required Methods in Entity Bean

Entity beans can be bean managed or container managed. Here, are the methods that are required for entity beans:

setEntityContext():

This method is called for, if a container wants to increase its pool size of bean instances, then, it will instantiate a new entity bean instance. This method associates a bean with context information. Once this method is called for, then, the bean can access the information about its environment

ejbFind(..):

This method is also known as the Finder method. The Finder method locates one or more existing entity bean data instances in underlying persistent store.

ejbHome(..):

The Home methods are special business methods because they are called from a bean in the pool before the bean is associated with any specific data. The client calls for, home methods from home interface or local home interface.

ejbCreate():

This method is responsible for creating a new database data and for initialising the bean.

ejbPostCreate():

There must be one ejbPostCreate() for each ejbCreate(). Each method must accept the same parameters. The container calls for, ejbPostCreate() right after ejbCreate().

ejbActivate():

When a client calls for, a business method on a EJB object but no entity bean instance is bound to EJB object, the container needs to take a bean from the pool and transition into a ready state. This is called Activation. Upon activation the ejbActivate() method is called for by the ejb container.

ejbLoad():

This method is called for, to load the database in the bean instance.

ejbStore():

This method is used for, to update the database with new values from the memory. This method is also called for during ejbPassivate().

ejbPassivate():

This method is called for, by the EJB container when an entity bean is moved from the ready state to the pool state.

ejbRemove():
This method is used to destroy the database data. It does not remove the object. The object is moved to the pool state for reuse.

unsetEntityContext():

This method removes the bean from its environment. This is called for, just before destroying the entity bean.

1.7.2.3 The Use of the Entity Bean

You could probably use an entity bean under the following conditions:

- The bean represents a business entity and not a procedure. For example, BookInfoBean would be an entity bean, but BookInfoVerifierBean would be a session bean.

- The bean’s state must be persistent. If the bean instance terminates or if the Application Server is shut down, the bean's state still exists in persistent storage (a database).

1.7.3 Message Driven Bean

A message-driven bean acts as a consumer of asynchronous messages. It cannot be called for, directly by clients, but is activated by the container when a message arrives. Clients interact with these EJBs by sending messages to the queues or topics to which they are listening. Although a message-driven EJB cannot be called for, directly by clients, it can call other EJBs itself.

1.7.3.1 Life Cycle of a Message Driven Bean

The *Figure 4* shows the life cycle of a Message Driven Bean:

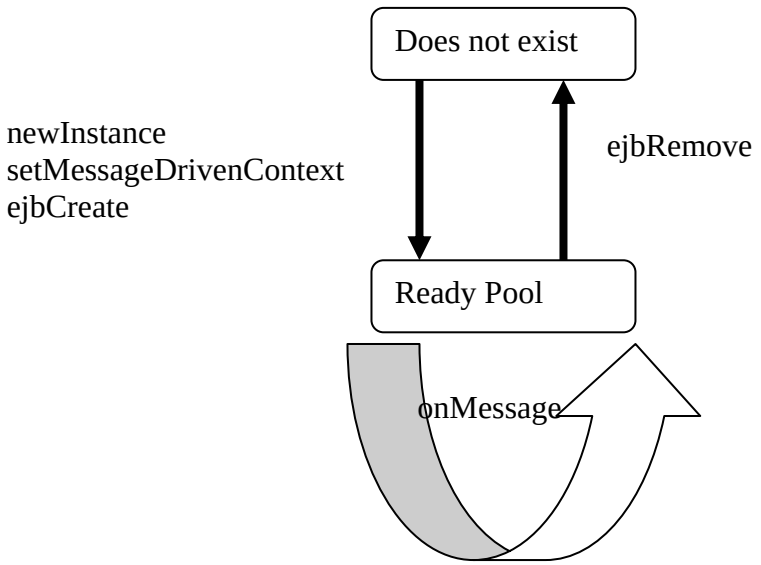

Figure 4: Life Cycle of a Message Driven Bean

A message driven bean has the following two states:

- **Does not exist:** In this state, the bean instance simply does not exist. Initially, the bean exists in the; does not exist state.

- **Pooled state:** After invoking the ejbCreate() method, the MDB instance is in the ready pool, waiting to consume incoming messages. Since, MDBs are stateless, all instances of MDBs in the pool are identical; they're allocated to process a message and then return to the pool.

1.7.3.2 Method for Message Driven Bean

onMessage(Message):

This method is invoked for each message that is consumed by the bean. The container is responsible for serialising messages to a single message driven bean.

ejbCreate():

When this method is invoked, the MDB is first created and then, added to the ‘to pool’.

ejbCreate():

When this method is invoked, the MDB is removed from the ‘to pool’.

setMessageDrivenContext(MessageDrivenContext):

This method is called for, as a part of the event transition that message driven bean goes through, when it is being added to the pool. This is called for, just before the ejbCreate().

1.7.3.3 The Use of the Message Driven Bean

Session beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, you may prefer not blocking synchronous receives in a server-side component. To receive messages asynchronously, use a message-driven bean.

Check Your Progress 1

1) What is the relationship between Enterprise JavaBeans and JavaBeans?
……...…
…..……...…
……

2) Explain the different types of Enterprise beans briefly.
……...…
……..……
……..

3) What is the difference between Java Bean and Enterprise Java Bean?
……...…
…..……...…
……

4) Can Entity Beans have no create() methods?
……...…
……..……
……..

5) What are the call back methods in the Session Bean?
……...…
……..……
……..

6) What are the call back methods of Entity Beans?
……...…
……..……
……..

7) Can an EJB send asynchronous notifications to its clients?
……...…
……..……
……..

8) What is the advantage of using an Entity bean for database operations, over directly using JDBC API to do database operations? When would I need to use one over the other?
……...…
…..……...…
……

9) What are the callback methods in Entity beans?
……...…
…..……...…
……

10) What is the software architecture of EJB?

……...…

….……...…

……

11) What are session Beans? Explain the different types.

……...…

….……...…

……

1.8 SUMMARY

Java bean and enterprise java beans are most widely used java technology. Both technologies contribute towards component programming. GUI JavaBeans can be used in visual tools. Currently most of the Java IDE and applications are using GUI JavaBeans. For enterprise application we have to choose EJB. Based on the requirements we can either use Session Bean, Entity Bean or Message Driven Bean. Session and Entity beans can be used in normal scenarios where we have synchronous mode. For asynchronous messaging like Publish /Subscribe we should use message driven beans.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Enterprise JavaBeans extends the JavaBeans component model to handle the needs of transactional business applications.

   JavaBeans is a component model for the visual construction of reusable components for the Java platform. Enterprise JavaBeans extends JavaBeans to middle-tier/server side business applications. The extensions that Enterprise JavaBeans adds to JavaBeans include support for transactions, state management, and deployment time attributes.

   Although applications deploying Enterprise JavaBeans architecture are independent of the underlying communication protocol, the architechture of the Enterprise JavaBeans specifies how communication among components, maps into the underlying communication protocols, such as CORBA/IIOP.

2) Different types of Enterprise Beans are following :

   - **Stateless Session Bean:** An instance of these non-persistent EJBs provides service without storing an interaction or conversation state between methods. Any instance can be used for any client.
   - **Stateful Session Bean**: An instance of these non-persistent EJBs maintains state across methods and transactions. Each instance is associated with a particular client.\
   - **Entity Bean:** An instance of these persistent EJBs represents an object view of the data, usually rows in a database. They have a primary key as a unique identifier. Entity bean persistence can be either container-managed or bean-managed.
   - **Message:Driven Bean:** An instance of these EJBs is integrated with the Java Message Service (JMS) to provide the ability for message-driven beans to act as a standard JMS message consumer and perform asynchronous processing between the server and the JMS message producer.

3) Java Bean is a plain java class with member variables and getter setter methods. Java Beans are defined under JavaBeans specification as Java-Based software component model which includes features such as introspection, customisation, events, properties and persistence.

Enterprise JavaBeans or EJBs for short are Java-based software components that comply with Java's EJB specification. EJBs are delpoyed on the EJB container and execute in the EJB container. EJB is not that simple, it is used for building distributed applications.

Examples of EJB are Session Bean, Entity Bean and Message Driven Bean. EJB is used for server side programming whereas java bean is a client side programme. While Java Beans are meant only for development the EJB is developed and then deployed on EJB Container.

4) Entity Beans can have no create() methods. Entity Beans have no create() an method, when an entity bean is not used to store the data in the database. In this case, entity bean is used to retrieve the data from the database.

5) Callback methods are called for, by the container to notify the important events to the beans in its life cycle. The callback methods are defined in the javax.ejb.EntityBean interface.The callback methods example are ejbCreate(), ejbPassivate(), and ejbActivate().

6) An entity bean consists of 4 groups of methods:

- **Create methods**: To create a new instance of a CMP entity bean, and therefore, insert data into the database, the create() method on the bean's home interface must be invoked. They look like this: EntityBeanClass ejbCreateXXX(parameters), where EntityBeanClass is an Entity Bean you are trying to instantiate, ejbCreateXXX(parameters) methods are used for creating Entity Bean instances according to the parameters specified and to some programmer-defined conditions.

  A bean's home interface may declare zero or more create() methods, each of which must have corresponding ejbCreate() and ejbPostCreate() methods in the bean class. These creation methods are linked at run time, so that when a create() method is invoked on the home interface, the container delegates the invocation to the corresponding ejbCreate() and ejbPostCreate() methods on the bean class.

- **Finder methods**: The methods in the home interface that begin with "find" are called the find methods. These are used to query to the EJB server for specific entity beans, based on the name of the method and arguments passed. Unfortunately, there is no standard query language defined for find methods, so each vendor will implement the find method differently. In CMP entity beans, the find methods are not implemented with matching methods in the bean class; containers implement them when the bean is deployed in a vendor specific manner. The deployer will use vendor specific tools to tell the container how a particular find method should behave. Some vendors will use object-relational mapping tools to define the behaviour of a find method while others will simply require the deployer to enter the appropriate SQL command.

  There are two basic kinds of find methods: single-entity and multi-entity. Single-entity find methods return a remote reference to the one specific entity bean that matches the find request. If, no entity beans are found, the method throws an ObjectNotFoundException . Every entity bean must define the

single-entity find method with the method name findByPrimaryKey(), which takes the bean's primary key type as an argument.

The multi-entity find methods return a collection (Enumeration or Collection type) of entities that match the find request. If, no entities are found, the multi-entity find returns an empty collection.

- **Remove methods**: These methods (you may have up to 2 remove methods, or don't have them at all) allow the client to physically remove Entity beans by specifying either Handle or a Primary Key for the Entity Bean.

- **Home methods**: These methods are designed and implemented by a developer, and EJB specifications do not require them as such, except when, there is the need to throw a RemoteException in each home method.

7) Asynchronous notification is a known hole in the first versions of the EJB spec. The recommended solution to this is to use JMS (Java Messaging Services), which is now, available in J2EE-compliant servers. The other option, of course, is to use client-side threads and polling. This is not an ideal solution, but it’s workable for many scenarios.

8) Entity Beans actually represents the data in a database. It is not that Entity Beans replaces JDBC API. There are two types of Entity Beans - Container Managed and Bean Mananged. In a Container Managed Entity Bean – Whenever, the instance of the bean is created, the container automatically retrieves the data from the DB/Persistance storage and assigns to the object variables in the bean for the user to manipulate or use them. For this, the developer needs to map the fields in the database to the variables in deployment descriptor files (which varies for each vendor). In the Bean Managed Entity Bean - the developer has to specifically make connection, retrive values, assign them to the objects in the ejbLoad() which will be called for, by the container when it instatiates a bean object. Similarly, in the ejbStore() the container saves the object values back to the persistance storage. ejbLoad and ejbStore are callback methods and can only be invoked by the container. Apart from this, when you are use Entity beans you do not need to worry about database transaction handling, database connection pooling etc. which are taken care of by the ejb container. But, in case of JDBC you have to explicitly take care of the above features. The great thing about the entity beans is that container managed is, that, whenever the connection fail during transaction processing, the database consistancy is mantained automatically. The container writes the data stored at persistant storage of the entity beans to the database again to provide the database consistancy. Whereas in jdbc api, developers need to maintain the consistency of the database manually.

9) The bean class defines create methods that match methods in the home interface and business methods that match methods in the remote interface. The bean class also implements a set of callback methods that allow the container to notify the bean of events in its life cycle. The callback methods are defined in the javax.ejb.EntityBean interface that is implemented by all entity beans.The EntityBean interface has the following definition. Notice that, the bean class implements these methods.

```
public interface javax.ejb.EntityBean {
public void setEntityContext();
public void unsetEntityContext();
   public void ejbLoad();
   public void ejbStore();
   public void ejbActivate();
   public void ejbPassivate();
```

```
public void ejbRemove();
}
```

The setEntityContext() method provides the bean with an interface to the container called the EntityContext. The EntityContext interface contains methods for obtaining information about the context under which the bean is operating at any particular moment. The EntityContext interface is used to access security information about the caller; to determine the status of the current transaction or to force a transaction rollback; or to get a reference to the bean itself, its home, or its primary key. The EntityContext is set only once in the life of an entity bean instance, so its reference should be put into one of the bean instance's fields if it will be needed later.

The unsetEntityContext() method is used at the end of the bean's life cycle before the instance is evicted from the memory to dereference the EntityContext and perform any last-minute clean-up.

The ejbLoad() and ejbStore() methods in CMP entities are invoked when the entity bean's state is being synchronised with the database. The ejbLoad() is invoked just after the container has refreshed the bean container-managed fields with its state from the database.

The ejbStore() method is invoked just before the container is about to write the bean container-managed fields to the database. These methods are used to modify data as it is being synchronised. This is common when the data stored in the database is different than the data used in the bean fields.

The ejbPassivate() and ejbActivate() methods are invoked on the bean by the container just before the bean is passivated and just after the bean is activated, respectively. Passivation in entity beans means that the bean instance is disassociated with its remote reference so that the container can evict it from the memory or reuse it. It's a resource conservation measure that the container employs to reduce the number of instances in the memory. A bean might be passivated if it hasn't been used for a while or as a normal operation performed by the container to maximise reuse of resources. Some containers will evict beans from the memory, while others will reuse instances for other more active remote references. The ejbPassivate() and ejbActivate() methods provide the bean with a notification as to when it's about to be passivated (disassociated with the remote reference) or activated (associated with a remote reference).

10) Session and Entity EJBs consist of 4 and 5 parts respectively:

a) A remote interface (a client interacts with it),

b) A home interface (used for creating objects and for declaring business methods),

c) A bean object (an object, which actually performs business logic and EJB-specific operations).

d) A deployment descriptor (an XML file containing all information required for maintaining the EJB) or a set of deployment descriptors (if you are using some container-specific features).

e) A Primary Key class - that is only Entity bean specific.

11) A session bean is a non-persistent object that implements some business logic running on the server. One way to think of a session object is as a logical extension of the client program that runs on the server.

Session beans are used to manage the interactions of entity and other session beans, access resources, and generally perform tasks on behalf of the client.

There are two basic kinds of session bean: Stateless and Stateful.

Stateless session beans are made up of business methods that behave like procedures; they operate only on the arguments passed to them when they are invoked. Stateless beans are called stateless because they are transient; they do not maintain business state between method invocations.Each invocation of a stateless business method is independent of any previous invocations. Because stateless session beans are stateless, they are easier for the EJB container to manage, so they tend to process requests faster and use less resources.

Stateful session beans encapsulate business logic and are state specific to a client. Stateful beans are called “tateful” because they do maintain business state between method invocations, held the in memory and not persistent. Unlike stateless session beans, clients do not share stateful beans. When a client creates a stateful bean, that bean instance is dedicated to the service of only that client. This makes it possible to maintain conversational state, which is business state that can be shared by methods in the same stateful bean.

1.10 FURTHER READINGS/REFERENCES

- Paco Gomez and Peter Zadronzy, *Professional Java 2 Enterprise Edition with BEA Weblogic Server*, WROX Press Ltd
- Budi Kurniawan, *Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions*, New Riders Publishing
- Richard Monson-Haefel, *Enterprise JavaBeans (3rd Edition)*, O'Reilly
- Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns, *Applying Enterprise JavaBeans™:Component-Based Development for the J2EE™ Platform*, Second Edition, Pearson Education
- Robert Englander, *Developing Java Beans*, O'Reilly Media

Reference websites:

- www.j2eeolympus.com
- www.phptr.com
- www.samspublishing.com
- www.oreilly.com
- www.roseindia.net
- www.caucho.com
- www.tutorialized.com
- www.stardeveloper.com