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.

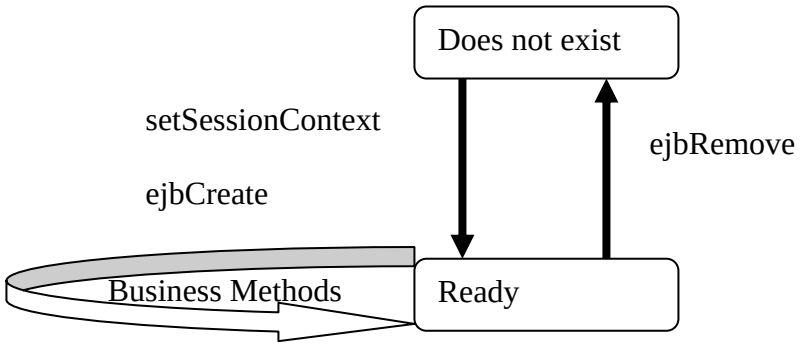

Figure 1: Life Cycle of Stateless Session Bean

- **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.

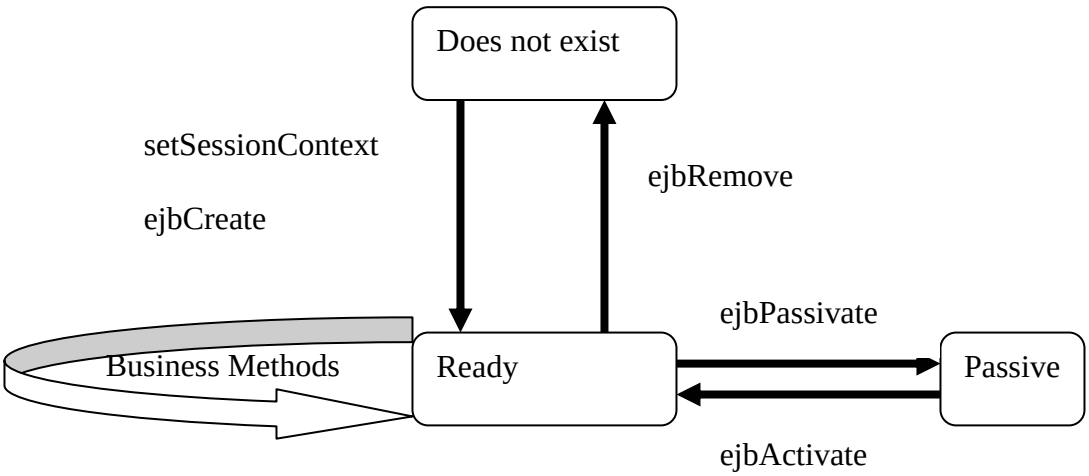

Figure 2: Life cycle of a stateful session bean

- **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.

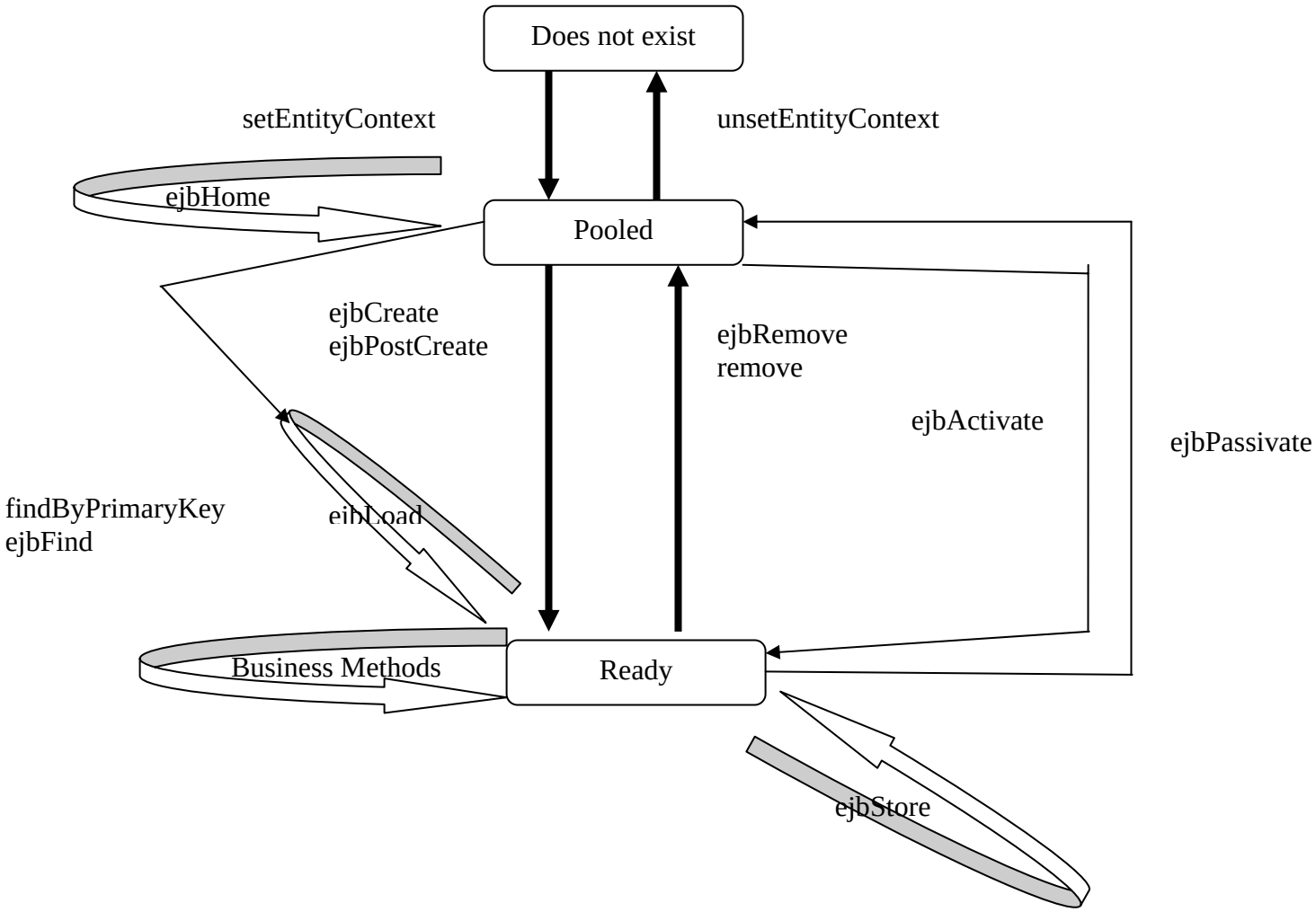

Figure 3: Life cycle of an entity bean

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:

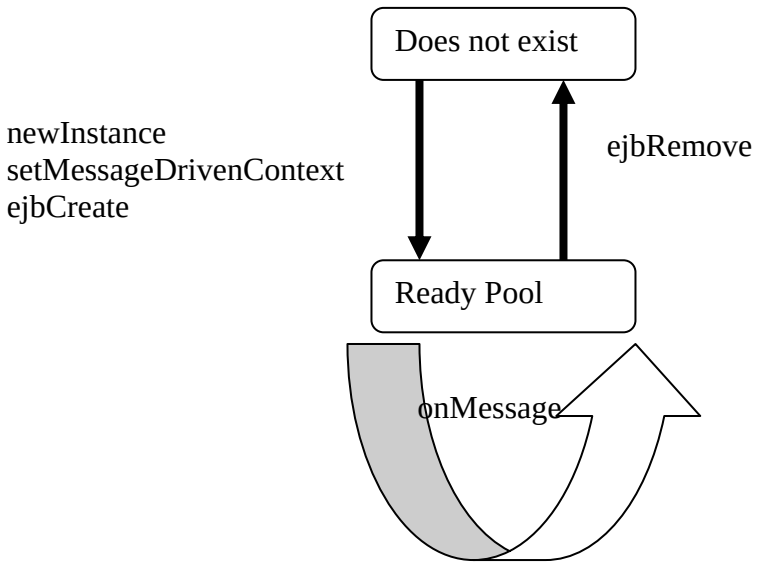

Figure 4: Life Cycle of a Message Driven Bean

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

UNIT 2 ENTERPRISE JAVA BEANS: ARCHITECTURE

Structure **Page Nos.**

2.0 INTRODUCTION

In the previous unit, we have learnt the basics of Enterprise java beans and different type of java beans. In this unit, we shall learn about the architecture of Enterprise java beans. “Enterprise JavaBeans” (EJBs) are distributed network aware components for developing secure, scalable, transactional and multi-user components in a J2EE environment. Enterprise JavaBeans (EJB) is suitable architecture for developing, deploying, and managing reliable enterprise applications in production environments. In this unit, we will study the benefits of using EJB architecture for enterprise applications. Enterprise application architectures have evolved through many phases. Such architectures inevitably evolve because, the underlying computer support and delivery systems have changed enormously and will continue to change in the future. With the growth of the Web and the Internet, more and more enterprise applications, including intranet and extranet applications, are now Web based. Enterprise application architectures have undergone an extensive evolution. The first generation of enterprise applications consisted of centralised mainframe applications. In the late 1980s and early 1990s, most new enterprise applications followed a two-tier, or client/server, architecture. Later, enterprise architecture evolved to a three-tier architecture and then to a Web-based architecture. The current evolutionary state is now represented by the J2EE application architecture. The J2EE architecture provides comprehensive support for two- and three-tier applications, as well as Web-based and Web services applications. Now, we will learn, about EJB architecture in detail.

2.1 OBJECTIVES

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

- list the different goals of Enterprise java beans in the business environment;
- list the benefits offered by EJB architecture to application developers and customers;
- differentiate the services offered by EJB architecture;
- list the various kinds of restrictions for the services offered by EJB;
- differentiate the between programming style and life cycle between stateful sessions;
- make a choice between Entity beans or stateful session beans, and
- install and run application like Jboss for EJB.

2.2 GOALS OF JAVA ENTERPRISE BEANS

The EJB Specifications try to meet several goals which are described as following:

- EJB is designed to make it easy for developers to create applications, freeing them from low-level system details of managing transactions, threads, load balancing, and so on. Application developers can concentrate on business logic and leave the details of managing the data processing to the framework. For specialised applications, though, it's to customise these lower-level services.

- The EJB Specifications define the major structures of the EJB framework, and then specifically define the contracts between them. The responsibilities of the client, the server, and the individual components are all clearly spelled out. A developer creating an Enterprise JavaBean component has a very different role from someone creating an EJB-compliant server, and the specification describes the responsibilities of each.

- EJB aims to be the standard way for client/server applications to be built in the Java language. Just as the original JavaBeans (or Delphi component etc.) from different vendors can be combined to produce a custom client, EJB server components from different vendors can be combined to produce a custom server. EJB components, being Java classes, will of course run in any EJB-compliant server without recompilation. This is a benefit that platform-specific solutions cannot offer.

- Finally, the EJB is compatible with and uses other Java APIs, can interoperate with non-Java applications, and is compatible with CORBA.

2.3 ARCHITECTURE OF AN EJB/ CLIENT SYSTEM

There are various architectures of EJB (J2EE) available which are used for various purposes. J2EE is a standard architecture specifically oriented to the development and deployment of enterprise Web-oriented applications that use Java programming language. ISVs and enterprises can use the J2EE architecture for only not the development and deployment of intranet applications, thus effectively replacing the two-tier and three-tier models, but also for the development of Internet applications, effectively replacing the cgi-bin-based approach. The J2EE architecture provides a flexible distribution and tiering model that allows enterprises to construct their applications in the most suitable manner.

Now, let us understand how an EJB client/server system operates and the underlying architecture of EJB, we need to understand the basic parts of an EJB system: *the EJB component, the EJB container, and the EJB object.*

The Enterprise JavaBeans Component

An Enterprise JavaBean is a component, just like a traditional JavaBean. Enterprise JavaBeans execute within an EJB container, which in turn executes within an EJB server. Any server that can host an EJB container and provide it with the necessary services can be an EJB server. (Hence, many existing servers are being extended to be EJB servers.) An EJB component is the type of EJB class most likely to be considered an “Enterprise JavaBean”. It’s a Java class, written by an EJB developer, that implements business logic. All the other classes in the EJB system either support client access to or provide services (like persistence, and so on) to EJB component classes.

The Enterprise JavaBeans Container

The EJB container is where the EJB component “lives”. The EJB container provides services such as transaction and resource management, versioning, scalability, mobility,

persistence, and security to the EJB components it contains. Since the EJB container handles all these functions, the EJB component developer can concentrate on business rules, and leave database manipulation and other such fine details to the container. For example, if a single EJB component decides that the current transaction should be aborted, it simply tells its container (container is responsible for performing all rollbacks, or doing whatever is necessary to cancel a transaction in progress). Multiple EJB component instances typically exist inside a single EJB container.

The EJB Object and the Remote Interface

Client programs execute methods on remote EJBs by way of an EJB object. The EJB object implements the "remote interface" of the EJB component on the server. The remote interface represents the "business" methods of the EJB component. The remote interface does the actual, useful work of an EJB object, such as creating an order form or deferring a patient to a specialist. EJB objects and EJB components are separate classes, though from the outside (i.e., by looking at their interfaces), they look identical. This is because, they both implement the same interface (the EJB component's remote interface), but they do very different things. An EJB component runs on the server in an EJB container and implements the business logic. *The EJB object runs on the client and remotely executes the EJB component's methods.*

Let us consider an analogy to understand the concept of EJB. Assume your VCD player is an EJB component. The EJB object is then analogous to your remote control: both the VCD and the remote control have the same buttons on the front, but they perform different functions. By pushing the Rewind button on your VCD player's remote control is equivalent to pushing the Rewind button on the VCD player itself, even though it's the VCD player -- and not the remote -- that actually rewinds a tape.

Working of EJB

The actual implementation of an EJB object is created by a code generation tool that comes with the EJB container. The EJB object's interface is the EJB component's remote interface. The EJB object (created by the container and tools associated with the container) and the EJB component (created by the EJB developer) implement the same remote interface. To the client, an EJB object looks just like an object from the application domain -- an order form, for example. But the EJB object is just a stand-in for the actual EJB, running on the server inside an EJB container. When the client calls a method on an EJB object, the EJB object method communicates with the remote EJB container, requesting that the same method be called, on the appropriate (remote) EJB, with the same arguments, on the client's behalf. This is explained with the help of the *Figure1*. This is the core concept behind how an EJB client/server system works.

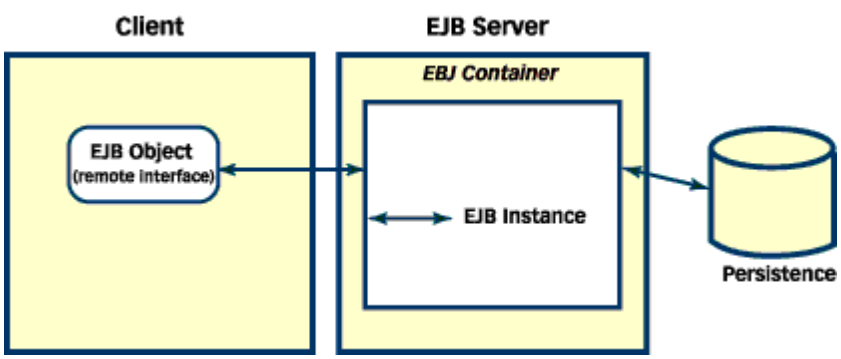

Figure 1: Enterprise JavaBeans in the Client and the Server

2.4 ADVANTAGES OF EJB ARCHITECTURE

Benefits to Application Developers:

The EJB component architecture is the backbone of the J2EE platform. The EJB architecture provides the following benefits to the application developer:

- **Simplicity:** It is easier to develop an enterprise application with the EJB architecture than without it. Since, EJB architecture helps the application developer access and use enterprise services with minimal effort and time, writing an enterprise bean is almost as simple as writing a Java class. The application developer does not have to be concerned with system-level issues, such as security, transactions, multithreading, security protocols, distributed programming, connection resource pooling, and so forth. As a result, the application developer can concentrate on the business logic for domain-specific applications.

- **Application Portability:** An EJB application can be deployed on any J2EE-compliant server. This means that the application developer can sell the application to any customers who use a J2EE-compliant server. This also means that enterprises are not locked into a particular application server vendor. Instead, they can choose the “best-of-breed” application server that meets their requirements.

- **Component Reusability:** An EJB application consists of enterprise bean components. Each enterprise bean is a reusable building block. There are two essential ways to reuse an enterprise bean:

  1) An enterprise bean not yet deployed can be reused at application development time by being included in several applications. The bean can be customised for each application without requiring changes, or even access, to its source code.

  2) Other applications can reuse an enterprise bean that is already deployed in a customer's operational environment, by making calls to its client-view interfaces. Multiple applications can make calls to the deployed bean.

  In addition, the business logic of enterprise beans can be reused through Java subclassing of the enterprise bean class.

- **Ability to Build Complex Applications:** The EJB architecture simplifies building complex enterprise applications. These EJB applications are built by a team of developers and evolve over time. The component-based EJB architecture is well suited to the development and maintenance of complex enterprise applications. With its clear definition of roles and well-defined interfaces, the EJB architecture promotes and supports team-based development and lessens the demands on individual developers.

- **Separation of Business Logic from Presentation Logic:** An enterprise bean typically encapsulates a business process or a business entity (an object representing enterprise business data), making it independent of the presentation logic. The business programmer need not worry about formatting the output; the Web page designer developing the Web page need be concerned only with the output data that will be passed to the Web page. In addition, this separation makes it possible to develop multiple presentation logic for the same business process or to change the presentation logic of a business process without needing to modify the code that implements the business process.

- **Easy Development of Web Services:** The Web services features of the EJB architecture provide an easy way for Java developers to develop and access Web services. Java developers do not need to bother about the complex details of Web services description formats and XML-based wire protocols but instead can program at the familiar level of enterprise bean components, Java interfaces and data types. The tools provided by the container manage the mapping to the Web services standards.

- **Deployment in many Operating Environments**— The goal of any software development company is to sell an application to many customers. Since, each customer has a unique operational environment, the application typically needs to be customised at deployment time to each operational environment, including different database schemas.

    1) The EJB architecture allows the bean developer to separate the common application business logic from the customisation logic performed at deployment.

    2) The EJB architecture allows an entity bean to be bound to different database schemas. This persistence binding is done at deployment time. The application developer can write a code that is not limited to a single type of database management system (DBMS) or database schema.

    3) The EJB architecture facilitates the deployment of an application by establishing deployment standards, such as those for data source lookup, other application dependencies, security configuration, and so forth. The standards enable the use of deployment tools. The standards and tools remove much of the possibility of miscommunication between the developer and the deployer.

- **Distributed Deployment:** The EJB architecture makes it possible for applications to be deployed in a distributed manner across multiple servers on a network. The bean developer does not have to be aware of the deployment topology when developing enterprise beans but rather writes the same code whether the client of an enterprise bean is on the same machine or a different one.

- **Application Interoperability:** The EJB architecture makes it easier to integrate applications from different vendors. The enterprise bean's client-view interface serves as a well-defined integration point between applications.

- **Integration with non-Java Systems:** The related J2EE APIs, such as the J2EE Connector specification and the Java Message Service (JMS) specification, and J2EE Web services technologies, such as the Java API for XML-based RPC (JAX-RPC), make it possible to integrate enterprise bean applications with various non-Java applications, such as ERP systems or mainframe applications, in a standard way.

- **Educational Resources and Development Tools:** Since, the EJB architecture is an industry wide standard, the EJB application developer benefits from a growing body of educational resources on how to build EJB applications. More importantly, powerful application development tools available from leading tool vendors simplify the development and maintenance of EJB applications.

Benefits to Customers

A customer’s perspective on EJB architecture is different from that of the application developer. The EJB architecture provides the following benefits to the customer:

- **Choice of Server:** Because the EJB architecture is an industry wide standard and is part of the J2EE platform, customer organisations have a wide choice of J2EE-compliant servers. Customers can select a product that meets their needs in terms of scalability, integration capabilities with other systems, security protocols, price, and

so forth. Customers are not locked in to a specific vendor's product. Should their needs change, customers can easily redeploy an EJB application in a server from a different vendor.

- **Facilitation of Application Management:** Because the EJB architecture provides a standardised environment, server vendors have the required motivation to develop application management tools to enhance their products. As a result, sophisticated application management tools provided with the EJB container allow the customer's IT department to perform such functions as starting and stopping the application, allocating system resources to the application, and monitoring security violations, among others.

- **Integration with a Customer's Existing Applications and Data:** The EJB architecture and the other related J2EE APIs simplify and standardise the integration of EJB applications with any non-Java applications and systems at the customer operational environment. For example, a customer does not have to change an existing database schema to fit an application. Instead, an EJB application can be made to fit the existing database schema when it is deployed.

- **Integration with Enterprise Applications of Customers, Partners, and Suppliers:** The interoperability and Web services features of the EJB architecture allows EJB-based applications to be exposed as Web services to other enterprises. This means that enterprises have a single, consistent technology for developing intranet, Internet, and e-business applications.

- **Application Security:** The EJB architecture shifts most of the responsibility for an application's security from the application developer to the server vendor, system administrator, and the deployer. The people performing these roles are more qualified than the application developer to secure the application. This leads to better security of operational applications.

2.5 SERVICES OFFERED BY EJB

As we have learnt earlier, in J2EE all the components run inside their own containers. JSP, Servlets and JavaBeans and EJBs have their own web container. The container of EJB provides certain built-in services to EJBs, which is used by EJBs to perform different functions. The services that EJB container provides are:

- Component Pooling
- Resource Management
- Transaction Management
- Security
- Persistence
- Handling of multiple clients

i) Component Pooling

The EJB container handles the pooling of EJB components. If, there are no requests for a particular EJB then the container will probably contain zero or one instance of that component in the memory. If, the need arises then it will increase component instances to satisfy all incoming requests. Then again, if, the number of requests decrease, the container will decrease the component instances in the pool. The most important thing is that the client is absolutely unaware of this component pooling which the container handles.

ii) Resource Management

The container is also responsible for maintaining database connection pools. It provides us a standard way of obtaining and returning database connections. The container also manages EJB environment references and references to other EJBs. The container manages the following types of resources and makes them available to EJBs:

- JDBC 2.0 Data Sources
- JavaMail Sessions
- JMS Queues and Topics
- URL Resources
- Legacy Enterprise Systems via J2EE Connector Architecture

iii) Transaction Management

This is the most important functionality served by the container. A transaction is a single unit of work, composed of one or more steps. If, all the steps are successfully executed, then, the transaction is committed otherwise it is rolled back. There are different types of transactions and it is absolutely unimaginable not to use transactions in today's business environments.

iv) Security

The EJB container provides it's own authentication and authorisation control, allowing only specific clients to interact with the business process. Programmers are not required to create a security architecture of their own, they are provided with a built-in system, all they have to do is to use it.

v) Persistence

Persistence is defined as saving the data or state of the EJB on non J-C volatile media. The container, if, desired can also maintain persistent data of the application. The container is then responsible for retrieving and saving the data for programmers, while taking care of concurrent access from multiple clients and not corrupting the data.

vi) Handling of Multiple Clients

The EJB container handles multiple clients of different types. A JSP based thin client can interact with EJBs with same ease as that of GUI based thick client. The container is smart enough to allow even non-Java clients like COM based applications to interact with the EJB system.

2.6 RESTRICTIONS ON EJB

Enterprise Java Beans have several restrictions, which they must adhere to:

1. They cannot create or manage threads.
2. They cannot access threads using the java.io package.
3. They cannot operate directly with sockets.
4. They cannot load native libraries.
5. They cannot use the AWT to interact with the user.
6. They can only pass objects and values which are compatible with RMI/IIOP.
7. They must supply a public no argument constructor.
8. Methods cannot be static or final.
9. There are more minor restrictions.

But following can be done with EJB:

1. Subclass another class.
2. Interfaces can extend other interfaces, which are descendants of EJBObject or EJBHome.
3. Helper methods can pass any kind of parameters or return types within the EJB.
4. Helper methods can be any kind of visibility.

2.7 DIFFERENCE BETWEEN SESSION BEANS AND ENTITY BEANS

To understand whether a business process or business entity should be implemented as a stateful session bean or an entity bean, it is important to understand the life-cycle and programming differences between them. These differences pertain principally to object sharing, object state, transactions, and container failure and object recovery.
Object sharing pertains to entity objects. Only a single client can use a stateful session bean, but multiple clients can share an entity object among themselves.

The container typically maintains a stateful session bean's object state in the main memory, even across transactions, although the container may swap that state to secondary storage when deactivating the session bean. The object state of an entity bean is typically maintained in a database, although the container may cache the state in memory during a transaction or even across transactions. Other, possibly non-EJB-based, programs can access the state of an entity object that is externalised in the database. For example, a program can run an SQL query directly against the database storing the state of entity objects. In contrast, the state of a stateful session object is accessible only to the session object itself and the container.

The state of an entity object typically changes from within a transaction. Sinces, its state changes transactionally, the container can recover the state of an entity bean should the transaction fail. The container does not maintain the state of a session object transactionally. However, the bean developer may instruct the container to notify the stateful session objects of the transaction boundaries and transaction outcome. These notifications allow the session bean developer to synchronise manually the session object's state with the transactions. For example, the stateful session bean object that caches changed data in its instance variables may use the notification to write the cached data to a database before the transaction manager commits the transaction.

Session objects are not recoverable; that is, they are not guaranteed to survive a container failure and restart. If, a client has held a reference to a session object, that reference becomes invalid after a container failure. (Some containers implement session beans as recoverable objects, but this is not an EJB specification requirement.) An entity object, on the other hand, survives a failure and restart of its container. If, a client holds a reference to the entity object prior to the container failure, the client can continue to use this reference after the container restarts.

The *Table 1* depicts the significant differences in the life cycles of a stateful session bean and an entity bean.

Table 1: Entity Beans and Stateful Session Beans: Life-Cycle Differences

Functional Area	Stateful Session Bean	Entity Bean
Object state	Maintained by the container in the main memory across transactions. Swapped to secondary storage when deactivated.	Maintained in the database or other resource manager. Typically cached in the memory in a transaction.
Object sharing	A session object can be used by only one client.	An entity object can be shared by multiple clients. A client may pass an object reference to another client.
State externalisation	The container internally maintains the session object's state. The state is inaccessible to other programs.	The entity object's state is typically stored in a database. Other programs, such as an SQL query, can access the state in the database.
Transactions	The state of a session object can be synchronised with a transaction but is not recoverable.	The state of an entity object is typically changed transactionally and is recoverable.
Failure recovery	A session object is not guaranteed to survive failure and restart of its container. The references to session objects held by a client becomes invalid after the failure.	An entity object survives the failure and the restart of its container. A client can continue using the references to the entity objects after the container restarts.

2.8 CHOOSING ENTITY BEANS OR STATEFUL SESSION BEANS

Now, we will learn when and where we should use entity beans or stateful session beans. The architect of the application chooses how to map the business entities and processes to enterprise beans. No prescriptive rules dictate whether a stateful session bean or an entity bean should be used for a component: Different designers may map business entities and processes to enterprise beans differently.

We can also combine the use of session beans and entity beans to accomplish a business task. For example, we may have a session bean represent an ATM withdrawal that invokes an entity bean to represent the account.

The following guidelines outline the recommended mapping of business entities and processes to entity and session beans. The guidelines reflect the life-cycle differences between the session and entity objects.

- A bean developer typically implements a business entity as an entity bean.
- A bean developer typically implements a conversational business process as a stateful session bean. For example, developers implement the logic of most Web application sessions as session beans.
- A bean developer typically implements, as an entity bean a collaborative business process: a business process with multiple actors. The entity object's state represents the intermediate steps of a business process that consists of multiple steps. For example, an entity object's state may record the changing information—state—on a loan application as it moves through the steps of the loan-approval process. The object's state may record that the account representative entered the information on the loan application, the loan officer reviewed the application, and application approval is still waiting on a credit report.
- If, it is necessary for any reason to save the intermediate state of a business process in a database, a bean developer implements the business process as an entity bean. Often, the saved state itself can be considered a business entity. For example, many e-commerce Web applications use the concept of a shopping cart, which stores the

items that the customer has selected but not yet checked out. The state of the shopping cart can be considered to be the state of the customer shopping business process. If, it is desirable that the shopping process span extended time periods and multiple Web sessions, the bean developer should implement the shopping cart as an entity bean. In contrast, if the shopping process is limited to a single Web session, the bean developer can implement the shopping cart as a stateful session bean.

2.9 INSTALLING AND RUNNING APPLICATION SERVER FOR EJB

Depending on the requirements of the organisation or company, one may select an appropriate Application server. If organisation's business processes span hundreds or thousands of different computers and servers then, one can select good application servers like BEA Web Logic, IBM Web Sphere and Oracle 9i Application Server etc. which are licensed servers. But, on the other hand, if, the organisation is really small but still wants to use EJBs due to future scalability requirements then, there are quite a few EJB application servers from free and open source to the ones, which are fast, but with reasonably low license fees. Examples of application servers, which are popular in small organisations, include **JBoss** and **Orion** Application Server.

An application server is a conglomeration of software services that provide a runtime environment for any number of containers, as shown in the *Figure 2*. A typical J2EE application server, such as WebLogic, WebSphere, JBoss, and Sun's J2EE Reference Server, houses a multitude of containers. WebLogic, for example, supports an EJB container and a servlet container.

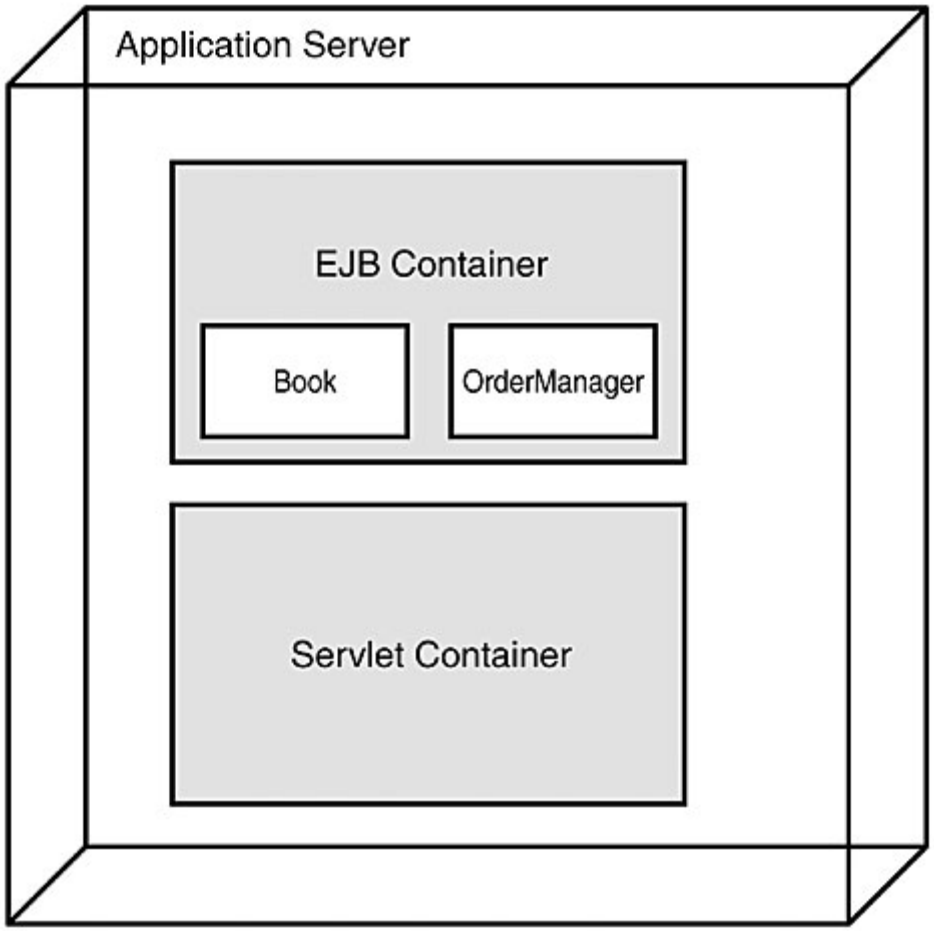

Figure 2: Architecture of EJB container

The EJB container provides basic services, including transactions, life-cycle management, and security, to the EJBs deployed into it. By shouldering much of this burdensome lower-level functionality, the EJB container significantly reduces the responsibilities of the EJBs deployed into it. Because EJBs no longer contain the code to provide these fundamental behaviours, EJB developers are free to concentrate on writing code that solves business problems instead of computer science problems.

Every application server vendor has its own way of deploying EJBs. They all share some common traits, however, that are illustrated in *Figure 3* and described here:

- An EJB's class (or sometimes its source code) files and its deployment descriptor are placed into an archive, which is typically a JAR file. Deployment descriptors are described in more depth in Part II, "Reference".

- A deployment tool of some sort creates a deployable archive file (typically, but not always, a JAR) from the contents of the archive created in Step 1.

- The deployable archive file is deployed into the EJB container by editing the container's configuration file or by running an administrative interface program.

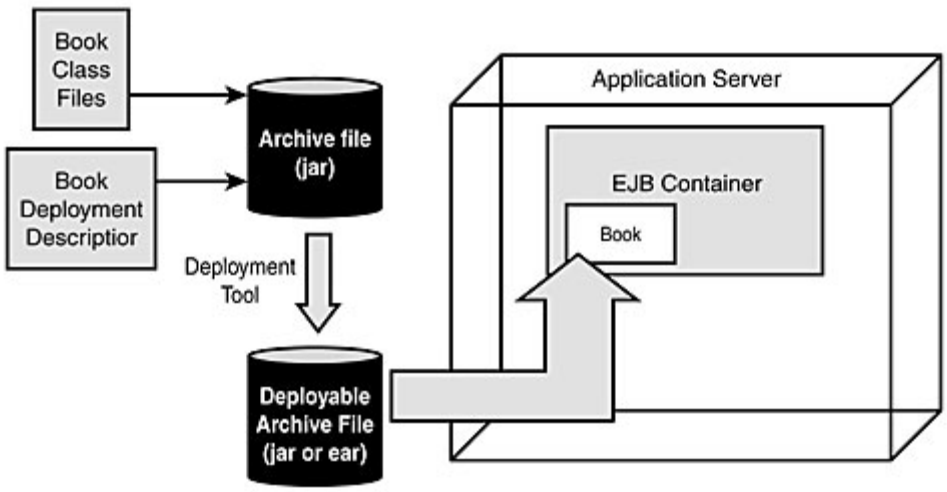

Figure 3: EJBs are typically bundled into archive files, processed by a deployment tool, and then deployed into the EJB container

There are many free application servers like Sun's J2EE Reference Application Server, which is available free at http://www.javasoft.com. Or Jboss, which may be downloaded JBoss from JBoss web site: www.jboss.org. Current stable version is 2.4.3. Download it from their web site. Once you have downloaded it, unzip the JBoss zip file into some directory e.g. C:\JBoss. The directory structure should be something like the following:

```
C:\JBoss
 admin
 bin
 client
 conf
  db
```

```
deploy
lib
log
tmp
```

Now, to start JBoss with default configuration go to JBoss/bin directory and run the following command at the DOS prompt :

```
C:\JBoss\bin>run
```

run.bat is a batch file which starts the JBoss Server. Once JBoss Server starts, you should see huge lines of text appearing on your command prompt screen. These lines show that JBoss Server is starting. Once JBoss startup is complete you should see a message like following one on your screen :

```
[Default] JBoss 2.4.3 Started in 0m:11s
```

Now, we have successfully installed and run JBoss on your system. To stop JBoss, simply press Ctrl + C on the command prompt and JBoss will stop, after displaying huge lines of text.

The client for our EJB will be a JSP page / Java Servelet running in a separate Tomcat Server. We have already learnt in an earlier Block 1, how to create and install Tomcat server for running JSP page or Servlet

Configuring and Running Tomcat :

Create a new folder under the main C:\ drive and name it “Projects”. Now, create a new sub-folder in the C:\Projects folder and name it "TomcatJBoss". The directory structure should look like the following:

```
C:\Projects
    TomcatJBoss
```

Now, open conf/Server.xml file from within the Tomcat directory where you have installed it. By default this location will be :

```
C:\Program Files\Apache Tomcat 4.0\conf\server.xml
```

Somewhere in the middle where you can see multiple <Context> tags, add following lines between other <Context> tags :

```
<!-- Tomcat JBoss Context -->
<Context path=”/jboss” docBase=“C:\Projects\TomcatJBoss\” debug=“0” reloadable= “true” />
```

Now, save Server.xml file. Go to Start -> Programs -> Apache Tomcat 4.0 -> Start Tomcat, to start Tomcat Server. If everything has been setup correctly, you should see the following message on your command prompt:

Starting service Tomcat-Standalone

Apache Tomcat/4.0.1

Note: Creating C:\Projects\TomcatJBoss folder and setting up new /jboss context in Tomcat is NOT required as far as accessing EJBs is concerned. We are doing it only to put our JSP client in a separate folder so as not to intermingle it with your other projects.

Check Your Progress 1

1) State True or False

a) EJB can manage threads. T ☐ F ☐

F

b) In EJB Methods can be static or final. T ☐ F ☐

c) A transaction is a single work, composed of one or more steps. T ☐ F ☐

d) The EJB container handles multiple clients of different types. T ☐ F ☐

2) Describe the layered architecture of EJB and explain all its components briefly.

……..…

…..……..…

……

3) Explain the different benefits of EJB architecture to Application Developers and Customers.

……..…

…..……..…

……

4) How does the EJB container assist in transaction management and persistence?

……..…

…..……..…

……

5) Explain the different types of restrictions on EJB.

……..…

…..……..…

……

6) How does the Session Bean differ from the Entity Bean in terms of object sharing, object state and failure recovery?

……..…

…..……..…

……

7) What criteria should a developer keep in mind while choosing between a session bean and an entity bean?

……..…

…..……..…

……

2.10 SUMMARY

In this unit, we have learnt basic architecture of EJB and the various kind of benefits offered by it to application developers and customers. We have also highlighted services offered by EJB, restriction emposed on EJB. The difference between session beans and entitiy beans and choasing between entity beans and session beans. Finally we have discussed how to install and run application server for EJB.

2.11 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) True/ False

a) False

b) False

c) True

d) True

Explanatory Answers

2) EJB is a layered architecture consisting of:

- Enterprise bean component which contains methods that implements the business logic
- EJB container
- EJB server, which contains the EJB container
- EJB object.

The Enterprise Java Beans Component

An Enterprise JavaBean is a component, just like a traditional JavaBean. Enterprise JavaBeans execute within an EJB container, which in turn executes within an EJB server. Any server that can host an EJB container and provide it with the necessary services can be an EJB server. EJB is a Java class, written by an EJB developer, that implements business logic. All the other classes in the EJB system either support client access to or provide services (like persistence, and so on) to EJB component classes.

The Enterprise Java Beans Container

The EJB container acts as an interface between an Enterprise bean and clients. The client communicates with the Enterprise bean through the remote and home interfaces provided by the customer. The EJB container is where the EJB component "lives." Since the EJB container handles all these functions, the EJB component developer can concentrate on business rules, and leave database manipulation and other such fine details to the container. The EJB container provides services such as security, transaction and resource management, versioning, scalability, mobility, persistence, and security to the EJB components it contains.

EJB Server

The EJB server provides some low level services like network connectivity to the container. In addition to this, it also provides the following services:

- Instance Passivation
- Instance pooling

- Database Connection Pooling
- Preached Instances

The EJB Object and the Remote Interface

Client programs execute methods on remote EJBs by way of an EJB object. The EJB object implements the "remote interface" of the EJB component on the server. The remote interface represents the "business" methods of the EJB component. The remote interface does the actual, useful work of an EJB object, such as creating an order form or referring a patient to a specialist.

EJB objects and EJB components are separate classes, though from the outside (i.e., by looking at their interfaces), they look identical. This is because, they both implement the same interface (the EJB component's remote interface), but they do very different things. An EJB component runs on the server in an EJB container and implements the business logic. *The EJB object runs on the client and remotely executes the EJB component's methods.*

3) The EJB component architecture is the backbone of the J2EE platform. The EJB architecture provides the following benefits to the application developer:

   - **Simplicity:** It is easier to develop an enterprise application with EJB architecture than without it. Since, EJB architecture helps the application developer access and use enterprise services with minimal effort and time, writing an enterprise bean is almost as simple as writing a Java class.

   - **Application Portability:** An EJB application can be deployed on any J2EE-compliant server. Application developer can choose the "best-of-breed" application server that meets their requirements.

   - **Component Reusability:** An EJB application consists of enterprise bean components and each of the enterprise Bean is a reusable building block. Business logic of enterprise beans can be reused through Java sub classing of the enterprise bean class.

   - **Ability to Build Complex Applications:** EJB architecture simplifies building complex enterprise applications. The component-based EJB architecture is well suited to the development and maintenance of complex enterprise applications.

   - **Separation of Business Logic from Presentation Logic:** An enterprise bean typically encapsulates a business process or a business entity (an object representing enterprise business data), making it independent of the presentation logic.

   - **Easy development of Web Services:** The Web services features of the EJB architecture provide an easy way for Java developers to develop and access Web services. The tools provided by the container manage the mapping to the Web services standards.

   - **Deployment in many Operating Environments:** The EJB architecture allows the bean developer to separate common application business logic from the customisation logic performed at deployment to me. The EJB architecture allows an entity bean to be bound to different database schemas. The EJB architecture facilitates the deployment of an application by establishing deployment standards, such as those for data source lookup, other application dependencies, security configuration, and so forth.

- **Distributed Deployment:** The EJB architecture makes it possible for applications to be deployed in a distributed manner across multiple servers on a network.
- **Application Interoperability:** The EJB architecture makes it easier to integrate applications from different vendors. The enterprise bean's client-view interface serves as a well-defined integration point between applications.
- **Integration with non-Java systems:** The related J2EE APIs, such as the J2EE Connector specification and the Java Message Service (JMS) specification, and J2EE Web services technologies, such as the Java API for XML-based RPC (JAX-RPC), make it possible to integrate enterprise bean applications with various non-Java applications, such as ERP systems or mainframe applications, in a standard way.
- **Educational Resources and Development Tools:** Because EJB architecture is an industry wide standard, the EJB application developer benefits from a growing body of educational resources on how to build EJB applications.

Benefits to Customers

A customer's perspective on EJB architecture is different from that of the application developer. The EJB architecture provides the following benefits to the customer:

- **Choice of Server:** Because the EJB architecture is an industry wide standard and is part of the J2EE platform, customer organisations have a wide choice of J2EE-compliant servers. Customers can select a product that meets their needs in terms of scalability, integration capabilities with other systems, security protocols, price, and so forth.
- **Facilitation of Application Management:** Sophisticated application management tools provided with the EJB container allow the customer's IT department to perform such functions as starting and stopping the application, allocating system resources to the application, and monitoring security violations, among others.
- **Integration with a Customer's Existing Applications and Data:** EJB architecture and the other related J2EE APIs simplify and standardize the integration of EJB applications with any non-Java applications and systems at the customer operational environment.
- **Integration with Enterprise Applications of Customers, Partners, and Suppliers:** The interoperability and Web services features of the EJB architecture allow EJB-based applications to be exposed as Web services to other enterprises.
- **Application Security:** EJB architecture shifts most of the responsibility for an application's security from the application developer to the server vendor, system administrator, and the deployer.

4) EJB container offers various kinds of services. Transaction Management is the most important functionality served by the container. The container provides the support according to the transaction specified and we can specify following types of transaction support for the bean:

- The bean manages its own transaction
- The bean doesn't support any transaction.
- When the client invokes the bean's method and if, the client has a transaction in progress, then the bean runs within the client's transaction. Otherwise, the container starts a new transaction for the bean.

- The bean must always start a new transaction, even if a transaction is open.
- The bean requires the client to have a transaction open before the client invokes the bean's method.

Persistence is another important function, which is supported by the EJB container. Persistence is the permanent storage of state of an object in a non-volatile media. This allows an object to be accessed at any time, without having to recreate the object every time it is required. The container if desired can also maintain persistent data of the application.

5) Enterprise Java Beans have several restrictions, which they must adhere to:

1. They cannot create or manage threads.
2. They cannot access threads using the java.io package.
3. They cannot operate directly with sockets.
4. They cannot load native libraries.
5. They cannot use the AWT to interact with the user.
6. They can only pass objects and values, which are compatible with RMI/IIOP.
7. They must supply a public no argument constructor.
8. Methods cannot be static or final.
9. There are more minor restrictions.

6) Session Beans and Entity beans differ in respect to object sharing, object state, transactions and container failure and object recovery.

Object Sharing: A session object can be used by only one client whereas, an entity object can be shared by multiple clients. A client may pass an object reference to another client.

Object State: Object state of stateful session bean is maintained by the container in the main memory across transactions and swapped to secondary storage when deactivated whereas, the object state of an entity bean is maintained in the database or other resource manager and typically cached in the memory in a transaction.

Failure Recovery: A session object is not guaranteed to survive failure and restart of its container. The references to session objects held by a client become invalid after the failure whereas, an entity object survives the failure and the restart of its container. A client can continue using the references to the entity objects after the container restarts.

7) There could be various guidelines, which can assist application developer choose between a session bean and entity bean, which are described below:

- A bean developer may typically implement a business entity as an entity bean.
- A bean developer may typically implement a conversational business process as a stateful session bean. For example, developers implement the logic of most Web application sessions as session beans.
- A bean developer typically implements as an entity bean a collaborative business process: a business process with multiple actors.
- If, it is necessary for any reason to save the intermediate state of a business process in a database, a bean developer implements the business process as an entity bean. Often, the saved state itself can be considered a business entity.

2.12 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

UNIT 3 EJB: DEPLOYING ENTERPRISE JAVA BEANS

Structure **Page Nos.**

3.0 INTRODUCTION

In the previous unit, we have learnt the basics of Enterprise java beans and different type of java beans. In this unit we shall learn how to create and deploy Enterprise java beans. “Enterprise JavaBeans” (EJBs) are distributed network aware components for developing secure, scalable, transactional and multi-user components in a J2EE environment. EJBs are a collection of Java classes, interfaces and XML files adhering to given rules. In J2EE all the components run inside their own containers. As we have learnt, JSP Servlets have their own web container and run inside that container. Similarly, EJBs run inside EJB container. The container provides certain built-in services to EJBs which the EJBs use to function. Now, we will learn how to create our first Enterprise JavaBean and we will then deploy this EJB on a production class, open source, and free EJB Server; JBoss.

3.1 OBJECTIVES

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

- provide an overview of XML;
- discuss how XML differs from HMTL;
- understand the SGML and its use;
- analyse the difference between SGML and XML;
- understand the basic goals of development of XML;
- explain the basic structure of XML document and its different components;
- understand what DTD is and how do prepare DTD for an XML document;
- learn what an XML parser is and its varieties, and
- differentiate between the different types of entities and their uses.

3.2 DEVELOPING THE FIRST SESSION EJB

We have already learnt in the previous unit about Session EJBs which are responsible for maintaining business logic or processing logic in our J2EE applications. Session beans are the simplest beans and are easy to develop so, first, we will develop Session beans.
Every EJB class file has two accompanying interfaces and one XML file. The two interfaces are :

- **Remote Interfaces:** Remote interface is what the client gets to work with, in other words Remote interface should contain the methods you want to expose to your clients.

- **Home Interfaces:** Home interface is actually EJB builder and should contain methods used to create Remote interfaces for your EJB. By default, the Home interface must contain at least one create() method.
  The actual implementation of these interfaces, our Session EJB class file remains hidden from the clients.

The XML file will be named as “ejb-jar.xml” and will be used to configure the EJBs during deployment. So, in essence our EJB, FirstEJB, which we are going to create consists of the following files :

Ejb

session

First.java

FirstHome.java

FirstEJB.java

META-INF

ejb-jar.xml

“First.java” will be the Remote interface we talked about. “FirstHome.java” is our Home interface and “FirstEJB.java” is the actual EJB class file. The ejb-jar.xml deployment descriptor file goes in the META-INF folder.

Now, create a new folder under the C:\Projects folder which we had created earlier, and name it “EJB”. Now, create a new sub-folder under C:\Projects\EJB folder and name it “FirstEJB”. Create a new folder "src" for Java source files in the "FirstEJB' folder. The directory structure should look like the following :

C:\Projects

TomcatJBoss

EJB

FirstEJB

Src

Now, create directory structure according to the package ignou.mca.ejb.session in the "src" folder. If, you know how package structure is built, it shouldn't be a problem. Anyhow, the final directory structure should like the following :

C:\Projects

TomcatJBoss

```
EJB
    FirstEJB
       src
          ignou
             mca
                ejb
                   session
```

First.java :

Now, we shall create the First.java source file in ignou/mca/ejb/session folder. Type the following code in it:

```
/* First.java */

package ignou.mca.ejb.session;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface First extends EJBObject {

   public String getTime() throws RemoteException;
}
```

Now save this file as first.java

First line is the package statement which indicates that the first interface belongs to the ignou.mca.ejb.session package:

```
package com.stardeveloper.ejb.session;
```

The next two lines are import statements for importing the required classes from the packages.

import javax.ejb.EJBObject;

import java.rmi.RemoteException;

Then, comes the interface declaration line which depicts that this is an interface with name “First” which extends an existing interface javax.ejb.EJBObject.

```
Note:  Every Remote interface *must* always extend EJBObject interface. It is a
       requirement, not an option.
```

public interface First extends EJBObject {

Now, we have to declare methods in Remote interface which we want to be called for the client (which the client can access and call). For simplicity, we will only declare a single method, getTime(); which will return a String object containing current time.

```
public String getTime() throws RemoteException;
```

We should notice that there are no {} parenthesis in this method as has been declared in an interface. Remote interface method's must also throw RemoteException because EJBs are distributed components and during the call to an EJB, due to some network problem, exceptional events can arise, so all Remote interface method's must declare that they can throw RemoteException in Remote interfaces.

FirstHome.java :

Now, let us create the home interface for our FirstEJB. Home interface is used to create and get access to Remote interfaces. Create a new FirstHome.java source file in ignou/mca/ejb/session package. Type the following code in it:

```
/* FirstHome.java */
package com.stardeveloper.ejb.session;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface FirstHome extends EJBHome {

   public First create() throws CreateException, RemoteException;
}
```

Now, save this file as FirstHome.java.

First few lines are package and import statements. Next, we have declared our FirstHome interface which extends javax.ejb.EJBHome interface.

Note: All Home interfaces *must* extend EJBHome interface.

```
public interface FirstHome extends EJBHome {
```

Then, we declared a single create() method which returns an instance of First Remote interface. Notice that all methods in Home interfaces as well must also declare that they can throw RemoteException. One other exception that must be declared and thrown is CreateException.

```
public First create() throws CreateException, RemoteException;
```

We are done with creating Remote and Home interfaces for our FirstEJB. These two are the only things which our client will see, the client will remain absolutely blind as far as the actual implementation class, FirstEJB is concerned.

FirstEJB.java :

FirstEJB is going to be our main EJB class. Create a new FirstEJB.java source file in

ignou/mca/ejb/session folder. Type the following code in it:

```
/* FirstEJB.java */
package ignou.mca.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.EJBException;
import javax.ejb.SessionContext;
import java.rmi.RemoteException;
import java.util.Date;

public class FirstEJB implements SessionBean {

   public String getTime() {
      return "Time is : " + new Date().toString();
   }
   public void ejbCreate() {}
   public void ejbPassivate() {}
   public void ejbActivate() {}
   public void ejbRemove() {}
   public void setSessionContext(SessionContext context) {}
}
```

Now, save this file as FirstEJB.java.

The first few lines are again the package and import statements. Next, we have declared our FirstEJB class and made it implement javax.ejb.SessionBean interface.

Note: All Session bean implementation classes must implement SessionBean interface.

```
public class FirstEJB implements SessionBean
```

Our first method is getTime() which had been declared in our First Remote interface. We implement that method here in our FirstEJB class. It simply returns get date and time as can be seen below:

```
public String getTime() {

      return "Time is : " + new Date().toString();

   }
```

Then comes 5 callback methods which are part of SessionBean interface and since we are implementing SessionBean interface, we have to provide empty implementations of these methods.

ejb-jar.xml :

Let's now create the EJB deployment descriptor file for our FirstEJB Session bean. Create a new ejb-jar.xml file in the FirstEJB/META-INF folder. Copy and paste the following text in it:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
   Inc.//DTD Enterprise JavaBeans 2.0//EN"
   "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
```

```
  <description></description>
  <enterprise-beans>
    <session>
      <display-name>FirstEJB</display-name>
      <ejb-name>First</ejb-name>
      <home>ignou.mca.ejb.session.FirstHome</home>
      <remote>ignou.mca.ejb.session.First</remote>
      <ejb-class>ignou.mca.ejb.session.FirstEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>First</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Supports</trans-attribute>
    </container-transaction>

    <security-role>
      <description>Users</description>
      <role-name>users</role-name>
    </security-role>
  </assembly-descriptor>
</ejb-jar>
```

One or more EJBs are packaged inside a JAR (.jar) file. There should be only one ejb-jar.xml file in an EJB JAR file. So ejb-jar.xml contains deployment description for one or more than one EJBs. Now as we learned in an earlier unit, there are 3 types of EJBs so ejb-jar.xml should be able to contain deployment description for all 3 types of EJBs.

Our ejb-jar.xml file for FirstEJB contains deployment description for the only EJB we have developed; FirstEJB. Since it is a Session bean, it's deployment description is contained inside <session></session> tags.

```
<ejb-jar>
  <description></description>
  <enterprise-beans>
    <session></session>
  </enterprise-beans>
</ejb-jar>
```

Now, let us discuss different deployment descriptor tags inside the <session></session> tag. First is the <ejb-name> tag. The value of this tag should be name of EJB i.e. any name you think should point to your Session EJB. In our case it's value is "First". Then, comes <home>, <remote> and <ejb-class> tags which contain complete path to Home, Remote and EJB implementation classes. Then comes <session-type> tag whose value is either "Stateless" or "Stateful". In our case it is "Stateless" because our Session bean is stateless. Last tag is <transaction-type>, whose value can be either "Container" or "Bean". The Transactions for our FirstEJB will be managed by the container.

```
<ejb-jar>
  <description></description>
```

```
  <enterprise-beans>
    <session>
      <display-name>FirstEJB</display-name>
      <ejb-name>First</ejb-name>
      <home>ignou.mca.ejb.session.FirstHome</home>
      <remote>ignou.mca.ejb.session.First</remote>
      <ejb-class>ignou.mca.ejb.session.FirstEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Then there is a <container-transaction> tag, which indicates that all methods of FirstEJB "support" transactions.

Compiling the EJB Java Source Files :

Now, our directory and file structure till now looks something like the following:

```
C:\Projects
   TomcatJBoss
   EJB
      FirstEJB
         src
            ignou
               mca
                  ejb
                     session
                        First.java
                        FirstHome.java
                        FirstEJB.java
         META-INF
            ejb-jar.xml
```

We can compile all the Java source file by using a command like the following on the command prompt:

```
C:\Projects\EJB\FirstEJB\src\ignou\mca\ejb\session>
            javac -verbose -classpath %CLASSPATH%;C:\JBoss\client\jboss-j2ee.jar
      -d C:\Projects\EJB\FirstEJB *.java
```

Note: The point to remember is to make sure jboss-j2ee.jar (which contains J2EE package classes) in the CLASSPATH, or you will get errors when trying to compile these classes.

If, you have installed JBoss Server in a separate directory then, substitute the path to jboss-j2ee.jar with the one present on your system. The point is to put jboss-j2ee.jar in the CLASSPATH for the javac, so that all EJB source files compile successfully.

3.3 PACKAGING EJB SOURCE FILES INTO A JAR FILE

Till now our directory and file structure should look something like the following:

```
C:\Projects
   TomcatJBoss
   EJB
      FirstEJB
         ignou
            mca
               ejb
                  session
                     First.class
                     FirstHome.class
                     FirstEJB.class
         META-INF
            ejb-jar.xml
         src
            ignou
               mca
                  ejb
                     session
                        First.java
                        FirstHome.java
                        FirstEJB.java
```

Now, to package the class files and XML descriptor files together, we will use the concept of jar file, which act as a container. For creating the jar file, run the following command at the command prompt:

```
C:\Projects\EJB\FirstEJB>jar cvfM FirstEJB.jar com META-INF
```

After running this command, EJB JAR file will be created with the name FirstEJB.jar in the FirstEJB folder. After this Jboss configuration file is to be added.

Adding JBoss specific configuration file :

Till now our FirstEJB.jar file contains generic EJB files and deployment description. To run it on JBoss we will have to add one other file into the META-INF folder of this JAR file, called jboss.xml. This file contains the JNDI mapping of FirstEJB.
So, create a new jboss.xml file in the FirstEJB/META-INF folder where ejb-jar.xml file is present and type the following text in it:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
          "http://www.jboss.org/j2ee/dtd/jboss.dtd">
<jboss>
          <enterprise-beans>
                    <session>
                              <ejb-name>First</ejb-name>
                              <jndi-name>ejb/First</jndi-name>
                    </session>
          </enterprise-beans>
</jboss>
```

Now, we will add this new jboss.xml file into our existing FirstEJB.jar file by running the following command at the DOS prompt:

```
C:\Projects\EJB\FirstEJB>jar uvfM FirstEJB.jar META-INF
```

Now, we will deploy the FirstEJB.jar on the JBoss Server.

3.4 DEPLOYING JAR FILE ON JBOSS SERVER

Now, we will deploy the FirstEJB.jar file on the Jboss Server by copying the FirstEJB.jar file and pasting it into the C:\JBoss\deploy folder. If JBoss is running, you should see text messages appearing on the console that this EJB is being deployed and finally deployed and started.

Creating the Client JSP Page:

Now we will create a new WEB-INF folder in C:\Projects\TomcatJBoss folder. Then create a new web.xml file and type the following text in it :

```
<?xml version= “1.0” encoding= “ISO-8859-1”?>

<!DOCTYPE web-app
   PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN”
   “http://java.sun.com/j2ee/dtds/web-app_2.3.dtd”>

<web-app>
</web-app>
```

As we can see, this web.xml is almost empty and is not doing anything useful. We still created it because Tomcat will throw an error if you try to access this /jboss context that we had created earlier without creating a /WEB-INF/web.xml file.

FirstEJB.jsp JSP Client page :

We will now create a new JSP page in the C:\Projects\TomcatJBoss folder and save it as "firstEJB.jsp". Now type the code in it :

```
<%@ page import= “javax.naming.InitialContext,
            javax.naming.Context,
            java.util.Properties,
            ignou.mca.ejb.session.First,
            ignou.mca.ejb.session.FirstHome"%>
<%
   long t1 = System.currentTimeMillis();
   Properties props = new Properties();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
         "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.PROVIDER_URL, "localhost:1099");

   Context ctx = new InitialContext(props);
   FirstHome home = (FirstHome)ctx.lookup("ejb/First");
   First bean = home.create();
   String time = bean.getTime();
   bean.remove();
```

```
   ctx.close();
   long t2 = System.currentTimeMillis();
%>
<html>
<head>
   <style>p { font-family:Verdana;font-size:12px; }</style>
</head>
<body>
<p>Message received from bean = "<%= time %>".<br>Time taken :
   <%= (t2 - t1) %> ms.</p>
</body>
</html>
```

Now, save this file as firstEJB.jsp as JSP page.

As we can see, the code to connect to an *external* JNDI/EJB Server is extremely simple. First, we have created a Properties object and put certain values for Context.INITIAL_CONTEXT_FACTORY and Context.PROVIDER_URL properties.

The value for Context.INITIAL_CONTEXT_FACTORY is the interface provided by JBoss and the value for Context.PROVIDER_URL is the location:port number where JBoss is running. Both these properties are required to connect to an external JNDI Server.

```
   Properties props = new Properties();
      props.put(Context.INITIAL_CONTEXT_FACTORY,
         "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.PROVIDER_URL, "localhost:1099");
```

We then get hold of that external JNDI Context object by creating a new InitialContext() object, it's argument being the Properties object we had created earlier.

```
Context ctx = new InitialContext(props);
```

Next, we have used that external JNDI Context handle to lookup our FirstEJB running on JBoss. Notice that the argument to Context.lookup("ejb/First") is the same value we had put in the jboss.xml file to bind our FirstEJB to this name in the JNDI context. We have used that same value again to look for it. Once our lookup is successful, we cast it to our FirstHome Home interface.

```
   FirstHome home = (FirstHome)ctx.lookup("ejb/First");
```

We have then used create method of our FirstHome Home interface to get an instance of the Remote interface; First. We will now use the methods of our EJB's remote interface (First).

```
   First bean = home.create();
```

We then called the getTime() method we had created in our EJB to get the current time from JBoss Server and saved it in a temporary String object.

```
String time = bean.getTime();
```

Once we are done with our Session EJB, we use the remove method to tell the JBoss Server that we no longer need this bean instance. After this we close the external JNDI Context.

```
bean.remove();
ctx.close();
```

After this retrieved value from getTime() method displayed on the user screen.

3.5 RUNNING THE CLIENT/JSP PAGE

Before trying to run firstJSP.jsp page, we have to do one other thing. Copy following files from C:\JBoss\client folder and paste them in the C:\Projects\TomcatJBoss\WEB-INF\lib folder. It is a must, without it firstEJB.jsp page will not run.

```
connector.jar
deploy.jar
jaas.jar
jboss-client.jar
jboss-j2ee.jar
jbossmq-client.jar
jbosssx-client.jar
jndi.jar
jnp-client.jar
```

we will also copy the FirstEJB.jar file from C:\Projects\EJB\FirstEJB folder to the C:\Projects\TomcatJBoss\WEB-INF\lib folder. Without it the Tomcat will not be able to compile firstEJB.jsp JSP page.

We are now ready to run our firstEJB.jsp page

Running firstEJB.jsp JSP page :

Now, start JBoss Server if, it is not already running. Also start Tomcat Server. If it is already running, then stop it and restart it again. Open your browser and access the following page :

```
http://localhost:8080/jboss/firstEJB.jsp
```

Please substitute port number “8080” above with the port number where your Tomcat Server is running. By default it is “8080” you will see a result like the following :

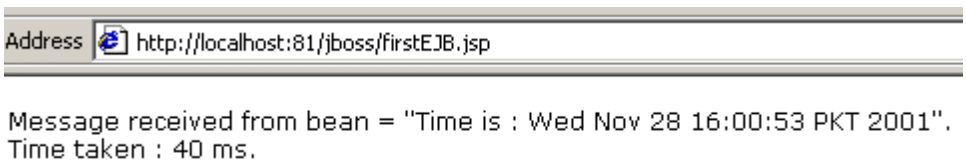

Figure 1: firstEJB.jsp Client View

3.6 MESSAGE-DRIVEN BEAN

Message-driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java™ API for XML Messaging (JAX-M). Message-driven beans asynchronously consume messages from a message destination, such as a JMS queue or topic.

Message-driven beans are components that receive incoming enterprise messages from a messaging provider. The primary responsibility of a message-driven bean is to process messages, because the bean's container automatically manages other aspects of the message-driven bean's environment. Message-driven beans contain business logic for handling received messages. A message-driven bean's business logic may key off the contents of the received message or may be driven by the mere fact of receiving the message. Its business logic may include such operations as initiating a step in a workflow, doing some computation, or sending a message.

A message-driven bean is essentially an application code that is invoked when a message arrives at a particular destination. With this type of messaging, an application—either a J2EE component or an external enterprise messaging client—acts as a message producer and sends a message that is delivered to a message destination. The container activates an instance of the correct type of message-driven bean from its pool of message-driven beans, and the bean instance consumes the message from the message destination. As a message-driven bean is stateless, any instance of the matching type of message-driven bean can process any message. Thus, message-driven beans are programmed in a similar manner as stateless session beans.

The advantage of message-driven beans is that they allow a loose coupling between the message producer and the message consumer, thus, reducing the dependencies between separate components. In addition, the EJB container handles the setup tasks required for asynchronous messaging, such as registering the bean as a message listener, acknowledging message delivery, handling re-deliveries in case of exceptions, and so forth. A component other than a message-driven bean would otherwise have to perform these low-level tasks.

3.7 IMPLEMENTING A MESSAGE-DRIVEN BEAN

Now, we will learn how to implement message driven beans. Message-driven beans are much like stateless session beans, having the same life cycle as stateless session beans but not having a component or home interface. The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. A message-driven bean class must also implement an ejbCreate method, even though the bean has no home interface. As they do not expose a component or home interface, clients cannot directly access message-driven beans. Like session beans, message-driven beans may be used to drive workflow processes. However, the arrival of a particular message initiates the process.

Implementing a message-driven bean is fairly straightforward. A message-driven bean extends two interfaces: javax.ejb.MessageDrivenBean and a message listener interface corresponding to the specific messaging system. (For example, when using the JMS messaging system, the bean extends the javax.jms.MessageListener interface.) The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

We can provide an empty implementation of the ejbCreate and setMessageDrivenContext methods. These methods are typically used to look up objects from the bean's JNDI environment, such as references to other beans and resource references. If the message-driven bean sends messages or receives synchronous communication from another destination, you use the ejbCreate method to look up the JMS connection factories and destinations and to create the JMS connection. The implementation of the ejbRemove method can also be left empty. However, if the ejbCreate method obtained any resources, such as a JMS connection, you should use the ejbRemove method to close those resources.

The methods of the message listener interface are the principal methods of interest to the developer. These methods contain the business logic that the bean executes upon receipt of a message. The EJB container invokes these methods defined on the message-driven bean class when a message arrives for the bean to service.

A developer decides how a message-driven bean should handle a particular message and codes this logic into the listener methods. For example, the message-driven bean might simply pass the message to another enterprise bean component via a synchronous method invocation, send the message to another message destination, or perform some business logic to handle the message itself and update a database.

A message-driven bean can be associated with configuration properties that are specific to the messaging system it uses. A developer can use the bean's XML deployment descriptor to include the property names and values that the container can use when connecting the bean with its messaging system.

3.8 JMS AND MESSAGE-DRIVEN BEANS

The EJB architecture requires the container to support message-driven beans that can receive JMS messages. You can think of message-driven beans as message listeners that consume messages from a JMS destination. A JMS destination may be a queue or a topic. When the destination is a queue, there is only one message producer, or sender, and one message consumer. When the destination is a topic, a message producer publishes messages to the topic, and any number of consumers may consume the topic's messages.

A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class. The onMessage method contains the business logic that the message-driven bean executes upon receipt of a message. The bean typically examines the message and executes the actions necessary to process it. This may include invoking other components.

The onMessage method has one parameter, the JMS message itself, and this parameter may be any valid JMS message type. The method tests whether the message is the expected type, such as a JMS TextMessage type, and then casts the message to that type and extracts from the message the information it needs.

Because the method does not include a throws clause, no application exceptions may be thrown during processing.

The EJB architecture defines several configuration properties for JMS-based message-driven beans. These properties allow the container to appropriately configure the bean and link it to the JMS message provider during deployment. These properties include the following:

- **DestinationType:** Either a javax.jms.Queue if the bean is to receive messages from a JMS queue, or a javax.jms.Topic if the bean is to receive messages from a JMS topic.
- **SubscriptionDurability:** Used for JMS topics to indicate whether the bean is to receive messages from a durable or a nondurable subscription. A durable subscription has the advantage of receiving a message even if the EJB server is temporarily offline.
- **AcknowledgeMode:** When message-driven beans use bean-managed transactions, this property indicates to the container the manner in which the delivery of JMS messages is to be acknowledged. (When beans use container-managed transactions, the message is acknowledged when the transaction commits.) Its values may be Auto-acknowledge, which is the default, or Dups-ok-acknowledge. The Auto-acknowledge mode indicates that the container should acknowledge messages as soon as the onMessage method returns. The Dups-ok-acknowledge mode indicates that the

container may lazily acknowledge messages, which could cause duplicate messages to be delivered.

- **MessageSelector**— Allows a developer to provide a JMS message selector expression to filter messages in the queue or topic. Using a message selector expression ensures that only messages that satisfy the selector are delivered to the bean.

3.9 MESSAGE-DRIVEN BEAN AND TRANSACTIONS

As messaging systems often have full-fledged transactional capabilities, message consumption can be grouped into a single transaction with such other transactional work as database access. This means that a bean developer can choose to make a message-driven bean invocation part of a transaction. Keep in mind, however, that if the message-driven bean participates in a transaction, you must also be using container-managed transaction demarcation. The deployment descriptor transaction attribute, which for a message-driven bean can be either Required or NotSupported, determines whether the bean participates in a transaction.

When a message-driven bean's transaction attribute is set to Required, the message delivery from the message destination to the message-driven bean is part of the subsequent transactional work undertaken by the bean. By having the message-driven bean be part of a transaction, you ensure that message delivery takes place. If the subsequent transaction fails, the message delivery is rolled back along with the other transactional work. The message remains available in the message destination until picked up by another message-driven bean instance. Note that, the message sender and the message receiver, which is the message-driven bean, do not share the same transaction. Thus, the sender and the receiver communicate in a loosely coupled but reliable manner.

If the message-driven bean's transactional attribute is NotSupported, it consumes the message outside of any subsequent transactional work. Should that transaction not complete, the message is still considered consumed and will be lost.

It is also possible to use bean-managed transaction demarcation with a message-driven bean. With bean-managed transaction demarcation, however, the message delivery is not be part of the transaction, because the transaction starts within the onMessage method.

3.10 MESSAGE-DRIVEN BEAN USAGE

Bean developers should consider using message-driven beans under certain circumstances:

- To have messages automatically delivered.
- To implement asynchronous messaging.
- To integrate applications in a loosely coupled but reliable manner.
- To have message delivery drive other events in the system workflow.
- To create message selectors, whereby specific messages serve as triggers for subsequent actions.

3.11 EXAMPLE OF MESSAGE-DRIVEN BEAN

Now, we will learn how to write a message driven bean. Consider,PayrollMDB, a message-driven bean that follows the requirements of the EJB 2.1 architecture, consisting

of the PayrollMDB class and associated deployment descriptors. The following Code shows the complete code for the PayrollMDB implementation class:

```
Public class PayrollMDB implements MessageDrivenBean,
        MessageListener {

   private PayrollLocal payroll;

   public void setMessageDrivenContext(MessageDrivenContext mdc) {
      try {
         InitialContext ictx = new InitialContext();
         PayrollLocalHome payrollHome = (PayrollLocalHome)
               ictx.lookup("java:comp/env/ejb/PayrollEJB");
         payroll = payrollHome.create();
      } catch ( Exception ex ) {
         throw new EJBException("Unable to get Payroll bean", ex);
      }
   }

   public void ejbCreate() { }

   public void ejbRemove() { }

   public void onMessage(Message msg) {
      MapMessage map = (MapMessage)msg;
      try {
         int emplNumber = map.getInt("Employee");
         double deduction = map.getDouble("PayrollDeduction");

         payroll.setBenefitsDeduction(emplNumber, deduction);
      } catch ( Exception ex ) {
         throw new EJBException(ex);
      }
   }
}
```

The PayrollMDB class implements two interfaces: javax.ejb.MessageDrivenBean and javax.jms.MessageListener. The JMS MessageListener interface allows the bean to receive JMS messages with the onMessage method. The MessageDrivenBean interface defines a message-driven bean's life-cycle methods called for, by the EJB container.

Of three such life-cycle methods, only one is of interest to PayrollMDB—setMessageDrivenContext. The container calls for setMessageDrivenContext immediately after the PayrollMDB bean instance is created. PayrollMDB uses this method to obtain a local reference to the Payroll stateless session bean, first looking up the PayrollLocalHome local home object and then invoking its create method. The setMessageDrivenContext method then stores the local reference to the stateless bean in the instance variable payroll for later use in the onMessage method.

The ejbCreate and ejbRemove life-cycle methods are empty. They can be used for any initialisation and cleanup that the bean needs to do.

The real work of the PayrollMDB bean is done in the onMessage method. The container calls the onMessage method when a JMS message is received in the PayrollQueue queue. The msg parameter is a JMS message that contains the message sent by the Benefits Enrollment application or other enterprise application. The method typecasts the received message to a JMS MapMessage message type, which is a special JMS message type that

contains property-value pairs and is particularly useful when receiving messages sent by non-Java applications. The EJB container's JMS provider may convert a message of MapMessage type either from or to a messaging product–specific format.

Once the message is in the proper type or format, the onMessage method retrieves the message data: the employee number and payroll deduction amount, using the Employee and PayrollDeduction properties, respectively. The method then invokes the local business method setBenefitsDeduction on the Payroll stateless session bean method to perform the update of the employee's payroll information in PayrollDatabase.

The PayrollMDB bean's deployment descriptor declares its transaction attribute as Required, indicating that the container starts a transaction before invoking the onMessage method and to make the message delivery part of the transaction. This ensures that the Payroll stateless session bean performs its database update as part of the same transaction and that message delivery and database update are atomic. If, an exception occurs, the transaction is rolled back, and the message will be delivered again. By using the Required transaction attribute for the message-driven bean, the developer can be confident that the database update will eventually take place.

PayrollEJB Local Interfaces

In the PayrollMDB means we have noticed that it uses the local interfaces of the PayrollEJB stateless session bean. These interfaces provide the same functionality as the remote interfaces described earlier. Since, these interfaces are local, they can be since accessed only by local clients deployed in the same JVM as the PayrollEJB bean. As a result, the PayrollMDB bean can use these local interfaces because it is deployed together with the PayrollEJB bean in the payroll department's application server. The following is the code for the local interfaces :

```
public interface PayrollLocal extends EJBLocalObject {
   void setBenefitsDeduction(int emplNumber, double deduction)
      throws PayrollException;
   double getBenefitsDeduction(int emplNumber)
      throws PayrollException;
   double getSalary(int emplNumber)
      throws PayrollException;
   void setSalary(int emplNumber, double salary)
      throws PayrollException;
}

public interface PayrollLocalHome extends EJBLocalHome {
   PayrollLocal create() throws CreateException;

}
```

Check Your Progress 1

1) State True/ False

   a) By default Home interface must contain at least one create() method. T ☐ F ☐

   b) Message-driven beans always synchronously consume messages from a message destination, such as a JMS queue. T ☐ F ☐

   c) The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. T ☐ F ☐

d) A message-driven bean that consumes JMS messages is not always required to implement the javax.jms.MessageListener interface. T ☐ F ☐

2) Explain two interfaces associated with EJB class files.

……..…

…..……..…

……

3) Explain the Message Driven bean and its advantages.

……..…

…..……..…

……

4) Explain the methods required to implement the Message Driven Bean.

……..…

…..……..…

……

5) How does Message Driven Bean consumes the messages from the JMS?

……..…

…..……..…

……

6) Explain the various configuration properties of JMS based Message Driven Beans offered by EJB architecture.

……..…

…..……..…

……

7) Explain the various circumstances under which Message Driven Bean can be used ?

……..…

…..……..…

……

3.12 SUMMARY

In this unit we have learnt what comprises an Enterprise JavaBean and how to develop and deploy an EJB. We created an EJB, deployed it on JBoss Server and called it from a JSP page running on Tomcat Server in a separate process. EJBs are collection of Java classes, interfaces and XML files adhering to given rules. In J2EE all the components run inside their own containers. EJBs run inside EJB container. The container provides certain built-in services to EJBs which the EJBs use to function.

Every EJB class file has two accompanying interfaces and one XML file. The two interfaces are Remote Interfaces and home interface. Remote interface contains the methods that developer wants to expose to the clients. Home interface is actually EJB builder and should contain methods used to create Remote interfaces for the EJB. By default Home interface must contain at least one create() method. The actual implementation remains hidden from the clients.

Every Remote interface must always extend EJBObject interface. It is a requirement, not an option. Message driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging

system, such as the Java API for XML Messaging (JAX-M). Message-driven beans are components that receive incoming enterprise messages from a messaging provider. Message-driven beans contain business logic for handling received messages. A message-driven bean's business logic may key off the contents of the received message or may be driven by the mere fact of receiving the message. Its business logic may include such operations as initiating a step in a workflow, doing some computation, or sending a message. The advantage of message-driven beans is that they allow a loose coupling between the message producer and the message consumer, thus reducing the dependencies between separate components.

Message-driven beans are much like stateless session beans, having the same life cycle as stateless session beans but not having a component or home interface. The implementation class for a message-driven bean must implement the javax.ejb.MessageDrivenBean interface. A message-driven bean class must also implement an ejbCreate method, even though the bean has no home interface. The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

The EJB architecture requires the container to support message-driven beans that can receive JMS messages. Message-driven beans act as message listeners that consume messages from a JMS destination. A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class.

Messaging systems often have full-fledged transactional capabilities, message consumption can be grouped into a single transaction with such other transactional work as database access. This means that a bean developer can choose to make a message-driven bean invocation part of a transaction.

Message driven can be used under the various circumstances like when messages to automatically delivered, to implement asynchronous messaging and to create message selectors.

3.13 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) True/ False

a) True
b) False
c) True
d) False

Explanatory Answers

2) There are two interfaces associated with every EJB class file :

Remote Interfaces: Remote interface is what the client gets to work with, in other words Remote interface should contain the methods you want to expose to your clients.

Home Interfaces: Home interface is actually the EJB builder and should contain methods used to create Remote interfaces for your EJB. By default, Home interface must contain at least one create() method.

The actual implementation of these interfaces, our Session EJB class file remains hidden from the clients.

3) Message-driven beans are EJB components that process asynchronous messages. These messages may be delivered via JMS or by using any other messaging system, such as the Java API for XML Messaging (JAX-M). The primary responsibility of a message-driven bean is to process messages, because the bean's container automatically manages other aspects of the message-driven bean's environment. Message-driven beans contain business logic for handling received messages.

   A message-driven bean is essentially an application code that is invoked when a message arrives at a particular destination. With this type of messaging, an application—either a J2EE component or an external enterprise messaging client—acts as a message producer and sends a message that is delivered to a message destination.

   Advantages

   The advantage of message-driven beans is that they allow loose coupling between the message producer and the message consumer, thus, reducing the dependencies between separate components. In addition, the EJB container handles the setup tasks required for asynchronous messaging, such as registering the bean as a message listener, acknowledging message delivery, handling re-deliveries in case of exceptions, and so forth. A component other than a message-driven bean would otherwise have to perform these low-level tasks.

4) The message-driven bean must implements two interfaces: javax.ejb.MessageDrivenBean and a message listener interface corresponding to the specific messaging system. (For example, when using the JMS messaging system, the bean extends the javax.jms.MessageListener interface.) The container uses the MessageDrivenBean methods ejbCreate, ejbRemove, and setMessageDrivenContext to control the life cycle of the message-driven bean.

   There can be empty implementation of the ejbCreate and setMessageDrivenContext methods. These methods are typically used to look up objects from the bean's JNDI environment, such as references to other beans and resource references.

5) Message-driven beans act as message listeners that consume messages from a JMS destination. A JMS destination may be a queue or a topic. A message-driven bean that consumes JMS messages needs to implement the javax.jms.MessageListener interface, which contains the single method onMessage that takes a JMS message as a parameter. When a message arrives for the bean to service, the container invokes the onMessage method defined on the message-driven bean class. The onMessage method contains the business logic that the message-driven bean executes upon receipt of a message. The bean typically examines the message and executes the actions necessary to process it. This may include invoking other components.

6) The onMessage method has one parameter, the JMS message itself, and this parameter may be any valid JMS message type. The method tests whether the message is the expected type, such as a JMS TextMessage type, and then casts the message to that type and extracts from the message the information it needs.

7) The EJB architecture defines several configuration properties for JMS-based message-driven beans, which are described as the following:

   DestinationType: Either a javax.jms.Queue if the bean is to receive messages from a JMS queue, or a javax.jms.Topic if the bean is to receive messages from a JMS topic.

SubscriptionDurability: It is used for JMS topics to indicate whether the bean is to receive messages from a durable or a nondurable subscription. A durable subscription has the advantage of receiving a message even if the EJB server is temporarily offline.

AcknowledgeMode: When message-driven beans use bean-managed transactions, this property indicates to the container the process of acknowledging the delivery of JMS messages. (When beans use container-managed transactions, the message is acknowledged once the transaction commits.) Its values may be Auto-acknowledge, which is the default, or Dups-ok-acknowledge.

MessageSelector: It allows a developer to provide a JMS message selector expression to filter messages in the queue or topic. Using a message selector expression ensures that only messages that satisfy the selector are delivered to the bean.

Bean developers may consider using message-driven beans under certain circumstances:

- To have messages automatically delivered.
- To implement asynchronous messaging.
- To integrate applications in a loosely coupled but reliable manner.
- To have message delivery drive other events in the system workflow.
- To create message selectors, whereby specific messages serve as triggers for subsequent actions.

3.14 FURTHER READINGS/REFERENCES

- Justin Couch and Daniel H. Steinberg, *Java 2 Enterprise Edition Bible*, Hungry Minds, Inc.

- 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.javaworld.com
- www.j2eeolympus.com
- www.samspublishing.com
- www.oreilly.com
- www.stardeveloper.com
- www.roseindia.net
- www.e-docs.bea.com
- www.java.sun.com

UNIT 4 XML : EXTENSIBLE MARKUP LANGUAGE

Structure **Page Nos.**

4.0 INTRODUCTION

In the previous blocks, we have already learnt about Java Servlets, Enterprise Java Beans and Java Server pages. In this unit, we shall cover some basics aspects of XML, which stands for Extensible Markup language, and SGML. XML is a structured document containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of the role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure. SGML, stands for both a language and an ISO standard for describing information, embedded within a document. XML is a meta-language written in SGML that allows one to design a markup language, used for the *easy interchange of documents* on the World Wide Web. In this unit, we shall also learn to discuss the differences between XML, SGML, and DTD, which stands for Document Type Definition and the requirements of DTD for the XML document.

4.1 OBJECTIVES

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

- provide an overview of XML;
- distinguish between XML and HTML;
- define SGML and its use;
- discuss the difference between SGML and XML;
- understand the basic goals of the development of XML;
- define the basic structure of the XML document and its different components;
- define DTD and the method of preparing DTD for an XML document;
- understand what is XML parser and its varieties, and
- understand the different types of entities and their uses.

4.2 AN OVERVIEW OF XML

XML stands for Extensible Markup Language (often written as extensible Markup Language to justify the acronym). A markup language is a specification that adds new information to existing information while keeping two sets of information separate and XML is a set of rules for defining semantic tags that break a document into parts and identifies the different parts of the document. It is a meta-markup language that defines a syntax that is in turn used to define other domain-specific, semantic, structured markup languages. With XML we can store the information in a structured manner.

Comparison of HTML and XML

XML differs from HTML in many aspects. As, we already know, HTML is a markup language used for displaying information, while XML markup is used for describing data of virtually any type. In other words, we can say that HTML deals with how to present whereas XML deals with what to present. Actually, HTML is a markup language whereas XML is a markup language and a language for creating markup languages. HTML limits you to a fixed collection of tags and these tags are primarily used to describe how content will be displayed, such as, making text bold or italiced or headings etc., whereas with XML you can create new or any user defined tags. Hence, XML enables the creation of new markup languages to markup anything imaginable (such as Mathematical formulas, chemical formulas or reactions, music etc.)

Let us, understand the difference between HTML and XML with the help of an example. In HTML a song might be described using a definition title, definition data, an unordered list, and list items. But none of these elements actually have anything to do with music. The HTML might look something like this:

```
<HTML>
<body>
<dt>Indian Classical </dt>
<dd> by HariHaran , Ravi shankar  and Shubha Mudgal</dd>
<ul>
<li>Producer: Rajesh
<li>Publisher: T-Series Records
<li>Length: 6:20
<li>Written: 2002
<li>Artist: Village People
</ul>
<body>
</html>
```

In XML, the same data might be marked up like this:

```
<XML>
<SONG>
<TITLE>Indian Classical</TITLE>
<COMPOSER>Hariharan</COMPOSER>
<COMPOSER>Ravi Shankar</COMPOSER>
<COMPOSER>Shubha Mudgal</COMPOSER>
<PRODUCER>Rajesh</PRODUCER>
<PUBLISHER>T-Series</PUBLISHER>
```

```
<LENGTH>6:20</LENGTH>
<YEAR>2002</YEAR>
<ARTIST>Village People</ARTIST>
</SONG></XML>
```

Instead of generic tags like <dt> and <li>, this listing uses meaningful tags like <SONG>, <TITLE>, <COMPOSER>, and <YEAR>. This has a number of advantages, including the fact that its easier for a human to read the source code to determine what the author intended.

4.3 AN OVERVIEW OF SGML

Now, we shall learn how SGML acts as a base for all the markup languages. SGML is a system or meta-language for defining markup languages, organising and tagging elements of a document. SGML was developed and standardised by the International Organisation for Standards (ISO). SGML itself does not specify any particular formatting; rather, it specifies the rules for tagging elements. These tags can then be interpreted to format elements in different ways. HTML and XML are based on SGML.

Authors mark up their documents by representing structural, presentational, and semantic information alongside content. Each markup language defined in SGML is known as SGML application. An SGML application is generally characterised by:

1) An SGML declaration. The SGML declaration specifies which characters and delimiters may appear in the application. By means of an SGML declaration (XML also has one), the SGML application specifies which characters are to be interpreted as data and which characters are to be interpreted as markup. (They do not have to include the familiar < and > characters; in SGML they could just as easily be {and} instead).

2) A document type definition (DTD). The DTD defines the syntax of markup constructs. It has rules for SGML/XML document just like there is grammar for English. The DTD may include additional definitions such as character entity references. We shall study later in this unit how we can create DTD for an XML document.

3) A specification that describes the semantics to be ascribed to the markup. This specification also imposes syntax restrictions that cannot be expressed within the DTD.

4) Document instances containing data (content) and markup. Each instance contains a reference to the DTD to be used to interpret it.
Using the rules given in the SGML declaration and the results of the information analysis (which ultimately creates something that can easily be considered an information model), the SGML application developer identifies various types of documents—such as reports, brochures, technical manuals, and so on—and develops a DTD for each one. Using the chosen characters, the DTD identifies information objects (elements) and their properties (attributes). The DTD is the very core of an SGML application; how well it is made largely determines the success or failure of the whole activity. Using the information objects (elements), defined in the DTD, the actual information is then marked up using the tags identified for it in the application. If, the development of the DTD has been rushed, it might need continual improvement, modification, or correction. Each time the DTD is changed, the information that has been marked up with it might also need to be modified because it may be incorrect. Very quickly, the quantity of data that needs modification (now called legacy

data) can become a far more serious problem—one that is more costly and time-consuming than the problem that SGML was originally introduced to solve.

Why not SGML?

The SGML on the Web initiative existed a long time before XML was even considered. Somehow, though, it was never really successful. Basically, SGML is just too expensive and complicated for Web use on a large scale. It isn't that it can't be used its just that it won't be used. Using SGML requires too much of an investment in time, tools, and training.

Why do we need XML?

XML adds a list of features that make it far more suitable than either SGML or HTML for use on an increasingly complex and diverse Web:

- **Modularity:** Although HTML appears to have no DTD, there is an implied DTD hard-wired into Web browsers. SGML has a limitless number of DTDs, on the other hand, but there's only one for each type of document. XML enables us to leave out the DTD altogether or, using sophisticated resolution mechanisms, combine multiple fragments of either XML instances or separate DTDs into one compound instance.
- **Extensibility:** XML's powerful linking mechanisms allow you to link to the material without requiring the link target to be physically present in the object. This opens up exciting possibilities for linking together things like material to which you do not have write access, CD-ROMs, library catalogue, the results of database queries, or even non-document media such as sound fragments or parts of videos. Furthermore, it allows you to store the links separately from the objects they link. This makes long-term link maintenance a real possibility.
- **Distribution:** In addition to linking, XML introduces a far more sophisticated method of including link targets in the current instance. This opens the doors to a new world of composite documents—documents composed of fragments of other documents that are automatically (and transparently) assembled to form what is displayed at that particular moment. The content can be instantly tailored to the moment, to the media, and to the reader, and might have only a fleeting existence: a virtual information reality composed of virtual documents.
- **Internationality:** Both HTML and SGML rely heavily on ASCII, which makes using foreign characters very difficult. XML is based on Unicode and requires all XML software to support Unicode as well. Unicode enables XML to handle not just Western-accented characters, but also Asian languages.
- **Data orientation:** XML operates on data orientation rather than readability by humans. Although being humanly readable is one of XML's design goals, electronic commerce requires the data format to be readable by machines as well. XML makes this possible by defining a form of XML that can be more easily created by a machine, but it also adds tighter data control through the more recent XML schema.

4.4 DIFFERENCE BETWEEN SGML AND XML

Now, we shall study what are the basic differences between SGML and XML. SGML, the Standard Generalised Markup Language, is the international standard for defining descriptions of structure and content in electronic documents whereas XML is a simplified version of SGML and it was designed to maintain the most useful parts of SGML.

SGML requires that structured documents reference a Document Type Definition (DTD) to be "valid", XML allows for "well-formed" data and can be delivered with and without a DTD. XML was designed so that SGML can be delivered, as XML, over the Web.

What does XML mean to SGML product vendors? On the technology front, SGML products should be able to read valid XML documents as they sit, as long as they are in 7-bit ASCII. To read internationalised XML documents, (for example in Japanese) SGML software will need modification to handle the ISO standard 10646 character set, and probably also a few industry-but-not-ISO standard encoding such as JIS and Big5. To write XML, SGML products will have to be modified to use the special XML syntax for empty elements.

On the business front, much depends on whether the Web browsers learn XML. If they do, SGML product vendors should brace for a sudden, dramatic demand for products and services from all the technology innovators who are, at the moment, striving to get their own extensions into HTML, and will (correctly) see XML as the way to make this happen. If the browsers remain tied to the fixed set of HTML tags, then XML will simply be an easy on-ramp to SGML, important probably more so because, the spec is short and simple than because of its technical characteristics. This will probably still generate an increase in market size, but not at the insane-seeming rate that would result from the browers' adoption of XML.

Check Your Progress 1

1) State True or False

   a) In HTML, we can define our own tags. T ☐ F ☐

   b) XML deals with what to present on the web page. T ☐ F ☐

   c) With XML one can create new markup languages like math markup language. T ☐ F ☐

2) How does HTML differs from XML? Explain with the help of an example.

   ……...

   ……...

   ……...

3) What are the basic characterstics of SGML?

   ……...

   ……...

   ……...

4) What are the advantages of XML over HTML?

   ……...

   ……...

   ……...

5) Explain the basic differences between XML and SGML.

   ……...

   ……...

   ……...

4.5 XML DEVELOPMENT GOALS

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

The design and development goals for XML are:

- XML shall be straightforwardly usable over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents. In practice, this will only be possible when XML browsers are as robust and widely available as HTML browsers, but the principle remains intact.
- XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications: authoring, browsing, content analysis, etc.
- XML shall be compatible with SGML. Most of the people involved in the XML effort come from organisations that have a large, in some cases staggering, amount of material in SGML. XML was designed pragmatically, to be compatible with existing standards while solving the relatively new problem of sending richly structured documents over the web.
- It shall be easy to write programs, which process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum, ideally zero. Optional features inevitably raise compatibility problems when users want to share documents and sometimes lead to confusion and frustration.
- XML documents should be human-legible and reasonably clear. If you don't have an XML browser and you've received a hunk of XML from somewhere, you ought to be able to look at it in your favourite text editor and actually figure out what the content means.
- The XML design should be prepared quickly. XML was needed immediately and was developed as quickly as possible. It must be possible to create XML documents in other ways: directly in a text editor, with simple shell and Perl scripts, etc
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.
- Terseness in XML markup is of minimal importance. Several SGML language features were designed to minimise the amount of typing required to manually key in SGML documents. These features are not supported in XML. From an abstract point of view, these documents are indistinguishable from their more fully specified forms, but supporting these features adds a considerable burden to the SGML parser (or the person writing it, anyway). In addition, most modern editors offer better facilities to define shortcuts when entering text.

4.6 THE STRUCTURE OF THE XML DOCUMENT

In this section, we shall create a simple XML document and learn about the structure of the XML document. Example 4.1 is about the simplest XML document I can imagine, so start with it. This document can be typed in any convenient text editor, such as Notepad, vi, or emacs.

Example 4.1: Hello XML

```
<?xml version=”1.0”?>
<FOO>
Hello XML!
</FOO>
```

Example-4.1 is not very complicated, but it is a good XML document. To be more precise, it is a well-formed XML document. (“Wellformed” is one of the terms, we shall study about it later)

Saving the XML file

After you have typed in *Example 4.1*, save it in a file called hello.xml or some other name. The three-letter extension .xml is fairly standard. However, do make sure that you save it in plain-text format, and not in the native format of a word processor such as WordPerfect or Microsoft Word.

Loading the XML File into a Web Browser

Now that you’ve created your first XML document, you’re going to want to look at it. The file can be opened directly in a browser that supports XML such as Internet Explorer 5.0. *Figure 1*a shows the result.

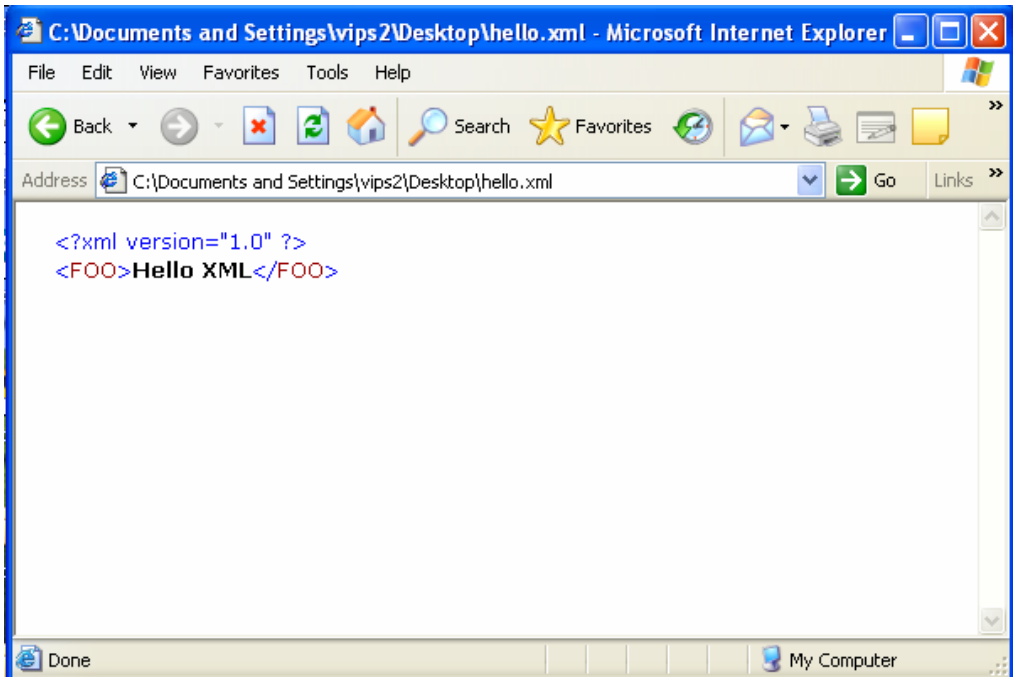

Figure 1: Output of XML Document

XML document may consists of the following parts:

a) The XML Prolog

XML file always starts with a Prolog as shown in the above example. An attribute is a name-value pair separated by an equals sign. Every XML document should begin with an XML declaration that specifies the version of XML in use. (Some XML documents omit this for reasons of backward compatibility, but you should include a version declaration unless you have a specific reason to leave it out). The minimal prolog contains a declaration that identifies the document as an XML document, like this:

<?xml version= “1.0”?>
The declaration may also contain additional information, like this:
<?xml version= “1.0” encoding= “ISO-8859-1” standalone= “yes”?>

The XML declaration may contain the following attributes:
Version Identifies the version of the XML markup language used in the data. This attribute is not optional.

Encoding

Identifies the character set used to encode the data. " ISO-8859-1" is "Latin-1" the Western European and English language character set. (The default is compressed Unicode: UTF-8.).

Standalone

This indicates whether or not this document references an external entity or an external data type specification (see below). If, there are no external references, then "yes" is appropriate.

The first line of the simple XML document in Listing 4.1 is the XML declaration:
<?xml version= "1.0"?>

<?xml version= "1.0" standalone= "no"?>

Identifying the version of XML ensures that future changes to the XML specification will not alter the semantics of this document. The standalone declaration simply makes explicit the fact that this document cannot "stand alone," and that it relies on an external DTD.

b) Elements and Attributes

Each Tag in a XML file can have Element and Attributes. Here's how a Typical Tag looks like,

<Email to= "admin@mydomain.com"
from= "user@mySite.com"
subject= "Introducing XML">
</Email>

In this Example, Email is known as an Element. This Element called E-mail has three attributes, to, from and subject.

The Following Rules need to be followed while declaring the XML Elements Names:

- Names can contain letters, numbers, and other characters
- Names must not start with a number or "_" (underscore)
- Names must not start with the letters xml (or XML or Xml).
- Names cannot contain spaces

Any name can be used, no words are reserved, but the idea is to make names descriptive. Names with an underscore separator are nice.
Examples: <author_name>, <published_date>.

Avoid "-" and "." in names. It could be a mess if your software tried to subtract name from first (author-name) or thought that "name" was a property of the object "author" (author.name).

Element names can be as long as you like, but don't exaggerate. Names should be short and simple, like this: <author_name> not like this: <name_of_the_author>.

XML documents often have a parallel database, where fieldnames are parallel to element names. A good rule is to use the naming rules of your databases.

The ":" should not be used in element names because it is reserved to be used for something called namespaces

In the above example-4.1, the next three lines after the prologue form a FOO element. Separately, <FOO> is a start tag; </FOO> is an end tag; and Hello XML! is the content of the FOO element. Divided another way, the start tag, end tag, and XML declaration are all markup. The text Hello XML! is character data.

c) Empty Tags:

In cases where you don't have to provide any sub tags, you can close the Tag, by providing a "/" to the Closing Tag. For example declaring
<Text></Text> is same a declaring <Text />

d) Comments in XML File:

Comments in XML file are declared the same way as Comments in HTML File.

<Text>Welcome To XML Tutorial </Text>

<!-- This is a comment -->

<Subject />

e) Processing Instructions

An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

<?target instructions?>

Where the target is the name of the application that is expected to do the processing, and instructions is a string of characters that embodies the information or commands for the application to process.

Well Formed Tags

One of the most important Features of a XML file is, it should be a Well Formed File. What it means that is all the Tags should have a closing tag. In a HTML file, for some tags like
 we don't have to specify a closing tag called </br>. Whereas in a XML file, it is compulsory to have a closing tag. So we have to declare
</br>. These are what are known as Well Formed Tags.

We shall take another *Example 4.2* ignou.xml to understand all the components of the XML document.

Example 4.2

```
<?xml version= "1.0"?>
<?xml version= "1.0" encoding= "ISO-8859-1" standalone= "yes"?>
<!-- ignou.xml-- >
<!-- storing the student details in XML document-- >
<!DOCTYPE ignou SYSTEM ignou.dtd">
<student>
    < course type = "undergraduate">
    <name>
    <fname> Saurabh </fname>
    <lname> Shukla </lname>
    <city> New Delhi </city>
    <state> Delhi </state>
    <zip> 110035 </zip>
    <status id = "P">
</course>
< course type = "Post Graduate">
    <name>
        <fname> Sahil</fname>
```

```
        <lname> Shukla </lname>
    <city> Kanpur </city>
    <state> U.P.</state>
    <zip> 110034 </zip>
    <status id = "D">
</course>
<content> IGNOU is world class open university which is offering undergraduate
and postgraduate course.
</content>
</student>
```

In the above XML document, student is the root element and it contains all other elements (e.g. course, content). Lines preceeding the root element is the prolog which we have discussed above.

<!DOCTYPE ignou SYSTEM ignou.dtd">

The above given line specifies the document type definition for this xml document. Document files define the rules for the XML document. This tag contains three items: the name of the root element to which the DTD is applied, the SYSTEM flag (which denotes an external DTD), and the DTD's name and location. We shall study more about DTD in the next section.

4.7 USING DTD WITH XML DOCUMENT

DTD stands for **document type definition**. A DTD defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. Document type definition lists the elements, attributes, entities, and notations that can be used in a document, as well as their possible relationships to one another. In brief, we may say that DTD specifies, a set of rules for the structure of a document.

If present, the document type declaration must be the first, in the document after optional processing instructions and comments. The document type declaration identifies the root element of the document and may contain additional declarations. All XML documents must have a single root element that contains all the contents of the document. Additional declarations may come from an external definition (a DTD) or be included directly in the document.

Creating DTD is just like creating a table in a database. In DTDs, you specify the structure of the data by declaring the element to denote the data. You can also specify whether providing a value for the element is mandatory or optional.

Declaring Elements in a DTD

After identifying the elements that can be used for storing structured data, they can be declared in a DTD. The XML document can be checked against the DTD. We can declare Element with the following syntax:

<!Element elementname(content-type or content-model)>

In the above syntax, elementname specifies the name of the element and content-type or content-model specifies whether the element contains textual data or other elements. For e.g. #PCDATA which specifies that element can store parsed character data(i.e. text). An element can be empty, unrestricted or container example. In the above mentioned example ignou.xml, the status is empty element whereas city is the unrestricted element.

Element Type	Description
Empty	Empty elements have no content and are marked up as <empty-elements>
Unrestricted	The opposite of an empty element is an unrestricted element, which can contain any element declared elsewhere in the DTD

In DTD various symbols are used to specify whether an element is mandatory or a optional or number of occurrences. The following is the list of symbols that can be used:

Symbol	Meaning	Example	Description
+	It indicates that there can be at least one or multiple occurrences of the element	Course+	There can multiple occurrences of course element.
*	It indicates that there can be either zero or any number of occurrences of the element	Content *	Any number of content element can be present.
?	It indicates that there can be either zero or exactly one occurrence.	Content ?	Content may not be present or present then only once
|	Or	City | state	City or state

We can declare attribute in DTD using the following syntax:

<!ATTLIST elementname att_name val_type [att_type] [“default”]>

att_name is the name of the attribute and val_type is the type of value which attribute can hold like CDATA defines that this attribute can contains a string.

We can also discuss whether attribute is mandatory or optional. We can do this with the help of att_type which can have the following values:

Attribute Type	Description
REQUIRED	If the attribute of an element is specified as # REQUIRED then, the value of that attribute must be specified if, it is not be specified then, the XML document will be invalid.
FIXED	If attribute of an element is specified as #FIXED then, the value of attribute cannot be changed in the XML document.
IMPLIED	If attribute of an element is specified as #IMPLIED then, attribute is optional i.e. this attribute need not be used every time the associated element is used.

Now, we are in the position to create the DTD named ignou.dtd, as shown in the *Example 4.2*

```
<?XML version= “1.0” rmd= “internal”?>
<!ELEMENT student( course +, content *)>
<!Element course (name, city, state, zip,status)>
<!ATTLIST course type CDATA #IMPLIED>
<!Element name (fname, lname)>
<!Element  fname (#PCDATA)>
```

```
<!Element  lname (#PCDATA)>
<!Element  city (#PCDATA)>
<!Element  state (#PCDATA)>
<!Element  zip (#PCDATA)>
<!Element  content (#PCDATA)>
```

This example, references an external DTD, ignou.dtd, and includes element and attribute declarations for the course element. In this case, course is being given the semantics of a simple link from the XLL specification.

In order to determine if a document is valid, the XML processor must read the entire document type declaration (both internal and external). But for some applications, validity may not be required, and it may be sufficient for the processor to read only the internal declaration. In the example given above, if validity is unimportant and the only reason to read the doctype declaration is to identify the semantics of course, then, reading the external definition is not necessary.

4.8 XML PARSER

An XML parser (or XML processor) is the software that determines the content and structure of an XML document by combining XML document and DTD (if any present). *Figure 2* shows a simple relationship between XML documents, DTDs, parsers and applications. XML parser is the software that reads XML files and makes the information from those files available to applications and other programming languages. The XML parser is responsible for testing whether a document is well-formed and, if, given a DTD or XML schema, whether will also check for validity (i.e., it determines if the document follows the rules of the DTD or schema). Although, there are many XML parsers we shall discuss only Micorosoft's parser used by the Internet explorer and W3C's parser that AMAYA uses.

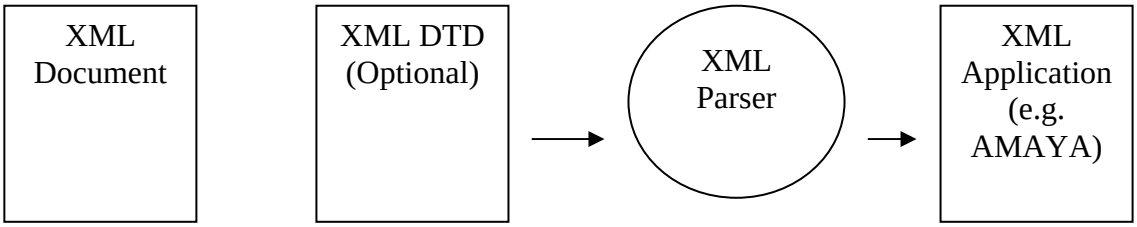

Figure 2: XML Documents and their Corresponding DTDs are Parsed and sent to Application.

XML Parser builds tree structures from XML documents. For example, an XML parser would build the tree structure shown in *Figure 2* for the previously mentioned example ignou.xml. If, this tree structure is created successfully without using a DTD, the XML document is considered **well-formed.** If, the tree stucture is created successfully and DTD is used, the XML document is considered valid. Hence, there can be two types of XML parsers : validating (i.e., enforces DTD rules) and non-validating (i.e., ignores DTD rules).

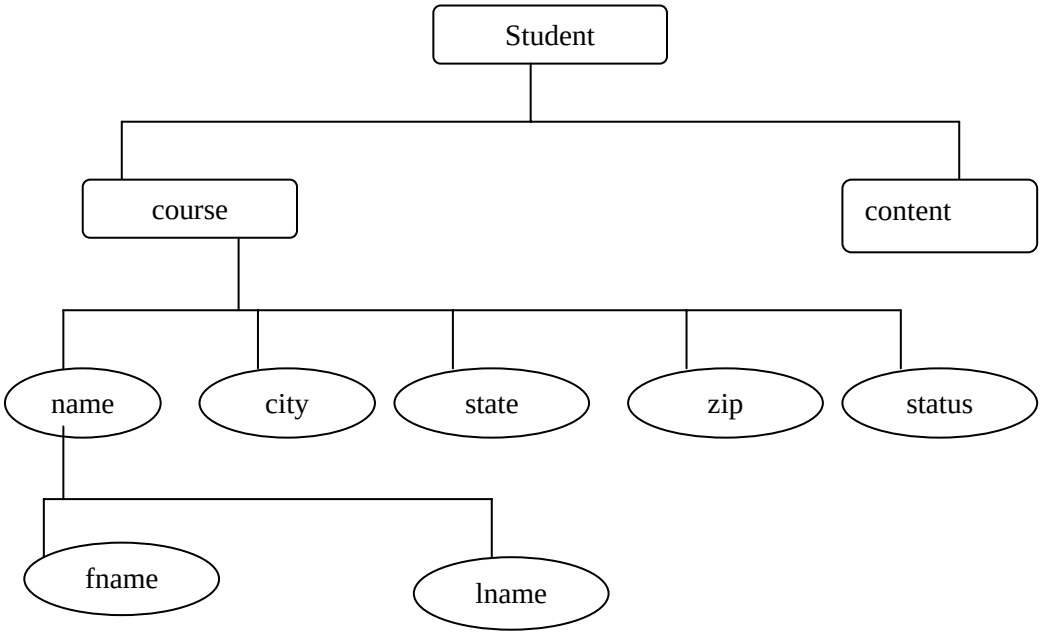

Figure 3: Tree Structure for ignou.xml

4.9 XML ENTITIES

Now, we shall learn about entity. Entities are normally external objects such as graphics files that are meant to be included in the document. Entity declarations [Section 4.3] allow you to associate a name with some other fragment of the document. That construct can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data.There are three varities of entities in XML.

Here are a few typical entity declarations:

<!ENTITY BCA “Bachelor of Computer Application.”>

<!ENTITY course SYSTEM “/standard/course1.xml”>

External Entity and Internal Entity : The first entity in the example given above is an internal entity because the replacement text is stored in the declaration. Using &BCA; anywhere in the document insert “Bachelor of computer Application.” at that location. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document. You must declare an internal entity before you can use it. It can save you a lot of unnecessary typing.

The declaration of an internal entity has this form:

<!ENTITY name “replacement text”>

Now, everytime the string &ampname; appears in your XML code, the XML processor will automatically replace it with the replacement text (which can be just as long as you like).

The XML specification predefines five internal entities:

- < produces the left angle bracket, <
- > produces the right angle bracket, >
- & produces the ampersand, &

- ' produces a single quote character (an apostrophe),
- " produces a double quote character,

External Entities: The second example id external entity. Using &course; will have - insert the contents of the file /standard/legalnotice.xml at that location in the document when it is processed. The XML processor will parse the content of that file as if its content were typed at the location of the entity reference. To reference external entities, you must have a DTD for your XML document.

External entities allow an XML document to refer to an external file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document.

Parameter Entities

Parameter entities can only occur in the document type declaration. A parameter entity is identified by placing "% " (percent-space) in front of its name in the declaration. The percent sign is also used in references to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded.

Check Your Progress 2

1) Explain the structure of an XML document.

……...

……..

……..

2) Explain the different development goals of XML document.

……...

……...

……...

3) What is DTD? Why do we use it? Explain the different components of DTD.

……...

……...

……...

4) Differentiate between validating and non-validating parser.

……...

……...

……...……

5) Explain the need/use of entities in XML document. Describe all three types of entities with the help of an example.

……...

……

……...

6) Where would you declare entities?

……...

……..

……..

7) Why is an XML declaration needed?

……...

……

……...

8) Can entities in attribute values as well as in content be used?

……...

……

……...

9) Is the use of binary data permitted in a CDATA section?

……...

……

……...

10) How many elements should you have in a DTD?

……...……

…….……

……...11) Is it necessary to validate XML documents?

……...

……

……...

12) How can we check whether the DTD is correct?

……...

……

……...

13) How can you put unparsed (non-XML) text in an external entity?

……...

……

……...

4.10 SUMMARY

XML, which stands for Extensible Markup Language, is defined as a structured document containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays. XML is a meta-language written in SGML that allows one to design a markup

language, used to allow for the easy interchange of documents on the World Wide Web. XML is a set of rules for defining semantic tags that break a document into parts and identifies the different parts of the document

XML differs from HTML in many aspects. As we know, HTML is a markup language and is used for displaying information, while XML markup is used for describing data of virtually any type. XML can be used to create other markup languages whereas HTML cannot be used for that purpose.

SGML acts as a base for all the markup languages. SGML is a system or meta-language for defining markup languages, organising and tagging elements of a document. SGML was developed and standardised by the International Organisation for Standards (ISO). SGML application is generally characterised by SGML declaration, Document Type Definition, specification and document instance.

XML is more suitable than SGML or HTML and is increasingly used for web application because of features like, modularity, extensibility distribution and data orientation. XML differs from SGML in some ways such as, SGML, is the international standard for defining descriptions of structure and content in electronic documents whereas XML is a simplified version of SGML and it was designed to maintain the most useful parts of SGML.

XML was developed by an XML Working Group with few design and development goals. Some of them like XML shall be straightforwardly usable over the Internet, XML shall be compatible with SGML, the number of optional features in XML is to be kept to the absolute minimum, ideally zero. XML documents should be human-legible and reasonably clear etc. These two qualities are of utmost importance and have contributed to the success of XML.

XML document may consist of many parts like XML prolog, elements, attributes, empty tags, comments and processing instructions. XML document should begin with an XML declaration, which is known as an XML prolog that specifies the version of XML in use. There can be two types of elements in XML container elements and empty elements. An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data.

DTD stands for document type definition, it defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. Document type definition lists the elements, attributes, entities, and notations that can be used in a document, as well as their possible relationships with one another. The document type declaration identifies the root element of the document and may contain additional declarations. In DTDs, the structure of the data can be specified by declaring the element as denoting the data.

An XML parser (or XML processor) is the software that determines the content and structure of an XML document by combining XML document and DTD. The XML parser is responsible for testing whether a document is well-formed and, if given a DTD or XML schema, it also checks for the validity. XML Parser builds the tree structure from XML documents. If this tree structure is created successfully without using a DTD, the XML document is considered well-formed and if the tree structure is created successfully and DTD is used, the XML document is considered valid. Entities are normally external objects such as graphics files that are meant to be included in the document. Entity declarations allow you to associate a name with some other fragment of the document. There are three varities of entities in XML. Internal Entity, where the replacement text is stored in the declaration. Internal entities allow you to define shortcuts for frequently typed text. External entities allow

an XML document to refer to an external file. External entities contain either text or binary data. Parameter entities can only occur in the document type declaration. A parameter entity is identified by placing "% " (percent-space) in front of its name in the declaration

4.11 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) True/ False

   a) False

   b) True

   c) True

2) XML differs from HTML in many aspects. As we know, HTML is a markup language that is used for displaying information, while XML markup is used for describing data of virtually any type. HTML deals with How to present whereas XML deals with what to present. XML can be used for creating other markup languages whereas HTML cannot be used for creating other markup languages.

   In HTML a song might be described using a definition title, definition data, an unordered list, and list items. But none of these elements actually have anything to do with music. The HTML example:

```
<HTML>
<body>
<dt>Indian Classical </dt>
<dd> by HariHaran , Ravi shankar  and Shubha Mudgal</dd>
<ul>
<li>Producer: Rajesh
<li>Publisher: T-Series Records
<li>Length: 6:20
<li>Written: 2002
<li>Artist: Village People
</ul>
<body>
</html>
```

In XML the same data might be marked up like this:

```
<XML>
<SONG>
<TITLE>Indian Classical</TITLE>
<COMPOSER>Hariharan</COMPOSER>
<COMPOSER>Ravi Shankar</COMPOSER>
<COMPOSER>Shubha Mudgal</COMPOSER>
<PRODUCER>Rajesh</PRODUCER>
<PUBLISHER>T-Series</PUBLISHER>
<LENGTH>6:20</LENGTH>
<YEAR>2002</YEAR>
<ARTIST>Village People</ARTIST>
</SONG></XML>
```

Instead of generic tags like <dt> and <li>, this listing uses meaningful tags like <SONG>, <TITLE>, <COMPOSER>, and <YEAR>.

3) SGML stands for Standard Generalised Markup Language, SGML is a system or meta-language for defining markup languages, organising and tagging elements of a document. Each markup language defined in SGML is known an SGML application. An SGML application is generally characterised by the following:

   a) **An SGML Declaration:** The SGML declaration specifies which characters and delimiters may appear in the application. By means of an SGML declaration (XML also has one), the SGML application specifies which characters are to be interpreted as data and which characters are to be interpreted as markup.

   b) **A Document Type Definition (DTD):** The DTD defines the syntax of markup constructs. It has rules for SGML/XML document just as there are rules of grammar for English. The DTD may include additional definitions such as character entity references.

   c) **A Specification:** This describes the semantics to be ascribed to the markup. This specification also imposes syntax restrictions that cannot be expressed within the DTD.

   d) **Document Instances:** Contain data (content) and markup. Each instance contains a reference to the DTD to be used to interpret it.

4) The advantages of XML over HTML and SGML are the following:

   - ***Modularity :*** Although HTML appears to have no DTD, there is an implied DTD hard-wired into Web browsers. XML enables us to leave out the DTD altogether or, using sophisticated resolution mechanisms, combine multiple fragments of either XML instances or separate DTDs into one compound instance.

   - ***Extensibility:*** XML's powerful linking mechanisms allow you to link to material without requiring the link target to be physically present in the object. This helps in linking together things like material to which you do not have write access, CD-ROMs, library catalogue, the results of database queries etc.

   - ***Distribution :***XML introduces a far more sophisticated method of including link targets in the current instance. This opens the doors to a new world of composite documents—documents composed of fragments of other documents that are automatically (and transparently) assembled to form what is displayed at that particular moment. The content can be instantly tailored to the moment, to the media, and to the reader.

   - ***Internationality:*** Both HTML and SGML rely heavily on ASCII, which makes using foreign characters very difficult. XML is based on Unicode and requires all XML software to support Unicode as well. Unicode enables XML to handle not just Western-accented characters, but also Asian languages.

   - ***Data orientation:*** XML operates on data orientation rather than readability by humans. Although being humanly readable is one of XML's design goals, electronic commerce requires the data format to be readable by machines as well.

5) XML differs from SGML in many respects. SGML, is the international standard for defining descriptions of structure and content in electronic documents whereas XML is a simplified version of SGML and it was designed to maintain the most useful parts of SGML. SGML requires that structured documents reference a Document Type Definition (DTD) to be “valid”, XML allows for “well-formed” data and can be delivered with and without a DTD.

Check Your Progress 2

1) XML document may consists of the following parts:

a) The XML Prolog

XML file always starts with a Prolog. Every XML document should begin with an XML declaration that specifies the version of XML in use. The minimal prolog contains a declaration that identifies the document as an XML document, like this:

<?xml version="1.0"?>
The declaration may also contain additional information, like this:
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

b) Elements and Attributes:

Each Tag in a XML file can have Element and Attributes. Here’s what a Typical Tag looks like,

<Email to="admin@mydomain.com"
from="user@mySite.com"
subject="Introducing XML">
</Email>

In this Example, Email is called an Element. This Element called E-mail has three attributes: to, from and subject.

c) Empty Tags:

In cases where you don't have to provide any sub tags, you can close the Tag, by providing a "/" to the Closing Tag. For example declaring

<Text></Text> is same a declaring <Text />

d) Comments in XML File:

Comments in XML file are declared the same way as Comments in HTML File.
<Text>Welcome To XML Tutorial </Text>
<!-- This is a comment -->
<Subject />

e) Processing Instructions

An XML file can also contain processing instructions that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

<?target instructions?>

Where the target is the name of the application that is expected to do the processing, and instructions is a string of characters that embodies the information or commands for the application to process.

2) XML was developed by an XML Working Group formed under the auspices of the World Wide Web Consortium (W3C) in 1996.

The design and development goals of XML are:

- XML shall be straightforwardly usable over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents.
- XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications: authoring, browsing, content analysis, etc.
- XML shall be compatible with SGML. Most of the people involved in the XML effort come from organisations that have a large, in some cases staggering, amount of material in SGML.
- It shall be easy to write programs, which process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
- XML documents should be human-legible and reasonably clear.
- The XML design should be prepared quickly. XML was needed immediately and was developed as quickly as possible.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.
- Terseness in XML markup is of minimal importance.

3) DTD stands for document type definition. A DTD defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. Document type definition lists the elements, attributes, entities, and notations that can be used in a document, as well as their possible relationships with one another. The document type declaration identifies the root element of the document and may contain additional declarations. To determine whether document is valid or not, the XML processor must read the entire document type declaration.

In DTDs, the structure of the data can be specified by declaring element to denote the data. DTD can be declared with the following syntax:

<!Element elementname(content-type or content-model)>

In the above syntax, elementname specifies the name of the element & content-type or content-model specifies whether the element contains textual data or other elements. . For e.g. #PCDATA which specifies that element can store parsed character data (i.e. text). DTD can describe two types of elements

Empty : Empty elements have no content and are marked up as <empty-elements>

Unrestricted: The opposite of an empty element is an unrestricted element, which can contain any element declared elsewhere in the DTD.
DTD may contain various symbols. These varied symbols are used to specify an element and may be mandatory, or optional, or number. For example '+' is used indicate that there can be at least one or multiple occurrences of the element, '*' indicates that there can be either zero or any number of occurrences of the element, '?' indicates that there can be either zero or exactly one occurrence etc.
Attribute can be declared in DTD by the following syntax:
<!ATTLIST elementname att_name val_type [att_type] ["default"]>
att_name is the name of the attribute and val_type is the type of value which an attribute can hold such as CDATA. CDATA defines that this attribute can contains a string.
An Attribute type may be defined as REQUIRED, FIXED or IMPLIED.

4) XML parser, which enforces the DTD rules on the XML document, is known as the validating parser, whereas the XML document which ignores DTD rules, is known as the non-validating parser.

5) Entity declarations are used in XML document to associate a name with some other fragment of the document. It is also used to create shortcuts for the frequently typed text. That construct/text can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data. There are three varieties of entities in XML.

Internal Entity: In internal entity the replacement text is stored in the declaration. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document. The declaration of an internal entity has this form:

<!ENTITY name "replacement text">
for e.g,
<!ENTITY BCA "Bachelor of Computer Application.">
Using &BCA; anywhere in the document insert "Bachelor of computer Application." at that location.

External Entities: External entities allow an XML document to refer to an external file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document. For e.g.,

<!ENTITY course SYSTEM "/standard/course1.xml">
Using &course; will have insert the contents of the file /standard/legalnotice.xml at that location in the document when it is processed.

Parameter Entities :
Parameter entities can only occur in the document type declaration. A parameter entity is identified by placing "% " (percent-space) in front of its name in the declaration. The percent sign is also used with reference to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded.

6) You can declare entities inside either the internal subset or the external subset of the DTD. If , you have an external DTD, you will have to create a complete DTD. If, you need only the entities then, you can get away with an internal DTD subset.

Entity references in XML documents that have external DTD subsets are only replaced when the document is validated.

7) As such we do not need an XML declaration. XML has also been approved as a MIME type, which means that if you add the correct MIME header (xml/text or xml/application), a Web server can explicitly identify the data that follows as being an XML document, regardless of what the document itself says. MIME, (Multipurpose Internet Mail Extensions), is an Internet standard for the transmission of data of any type via electronic mail. It defines the way messages are formatted and constructed, can indicate the type and nature of the contents of a message, and preserve international character set information. MIME types are used by Web servers to identify the data contained in a response to a retrieval request.

The XML declaration is not mandatory for some practical reasons; SGML and HTML code can often be converted easily into perfect XML code (if it isn't already). If the XML declaration was compulsory, this wouldn't be possible.

8) You can use entity references in attribute values, but an entity cannot be the attribute value. There are strict rules on where entities can be used and when they are recognised. Sometimes they are only recognised when the XML document is validated.

9) Technically, there's nothing which stops you from using binary data in a CDATA section, even though it's really a character data section, particularly as, the XML processor doesn't consider the contents of a CDATA section to be part of the document's character data. However, this may cause increase in file size and all the transportation problems that may be implied. Ultimately, it would could create problems for the portability of the XML documents when there is a far more suitable feature of XML you can use for this purpose. Entities, allow you to declare a format and a helper application for processing a binary file (possibly displaying it) and associating it with an XML document by reference.

10) It all depends on your application. HTML has about 77 elements, and some of the industrial DTDs have several hundred. It isn't always the number of elements that determine the complexity of the DTD. By using container elements in the DTD (which add to the element count), authoring software is able to limit possible choices and automatically enter required elements. Working with one of the major industrial DTDs can actually be far easier than creating HTML. Because HTML offers you more free choice, you have to have a much better idea of the purpose of all the elements rather than just the ones you need.

11) No, but unless you are certain that your documents are valid you cannot predict what will happen at the receiving end. Although the XML specification lays down rules of conduct for XML processors and specifies what they must do when certain invalid content is parsed, the requirements are often quite loose. Validation certainly won't do any harm—though it might create some extra work.

12) Simply validate an XML document that uses the DTD. There aren't as many tools specifically intended for checking DTDs as there are for validating documents. However, when an XML document is validated against the DTD, the DTD is checked and errors in the DTD are reported.

13) There are several ways. You could declare a TEXT notation, but this would not allow you to physically include the text in the XML document. (It would go to the helper application you designated in the notation declaration.) The best way would probably be to declare the file containing the text as an external text entity and put the text in that file in a CDATA section.

4.12 FURTHER READINGS/REFERENCES

- Brett McLaughlin & Justin Edelson, *Java and XML*, Third Edition, Oreilly
- Simon North, *Sams Teach yourself XML in 21 days,* Macmillan Computer Publishing
- Eric Westermann, *Learn XML in a weekend XML Bible*, Premier press
- Natanya Pitts-Moultis and Cheryl Kirk, *XML Black Book,* The Coriolis Group

References websites:

- www.w3schools.com
- www.xml.com
- www.java.sun.com/xml/tutorial_intro.html
- www.zvon.org
- www.javacommerce.com