5 : Enterprise Java Beans

IT 4206 – Enterprise Application Development

Level II - Semester 4

Overview

- Identify what is Enterprise Java Beans and its key features.
- Identify how to convert a Plain Old Java Object (POJO) to an EJB.

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Describe what is an Enterprise Java Beans
  - Describe Stateful beans, Stateless beans, Message passing beans, Singleton beans, Entity beans
  - Deploy a Java Bean
  - Generate an EJB
  - Convert a Plain Old Java Object(POJO) to an EJB

List of sub topics

1.1 Enterprise Java Beans

- Specification for developing large-scale, distributed business applications on the Java platform.
- Bean is a another word for component.
- With Enterprise JavaBeans you can develop building blocks that you or someone else can assemble and reassemble into different applications.
- Need only to focus on business logic of the application.
- EJB lets you focus on the business logic for your business and leave the underlying services to the EJB server vendor.

Enterprise Java Beans

- EJB is a component-based development model.
- Components are reusable chunks of functionality you can modify for different applications without touching the java source code.

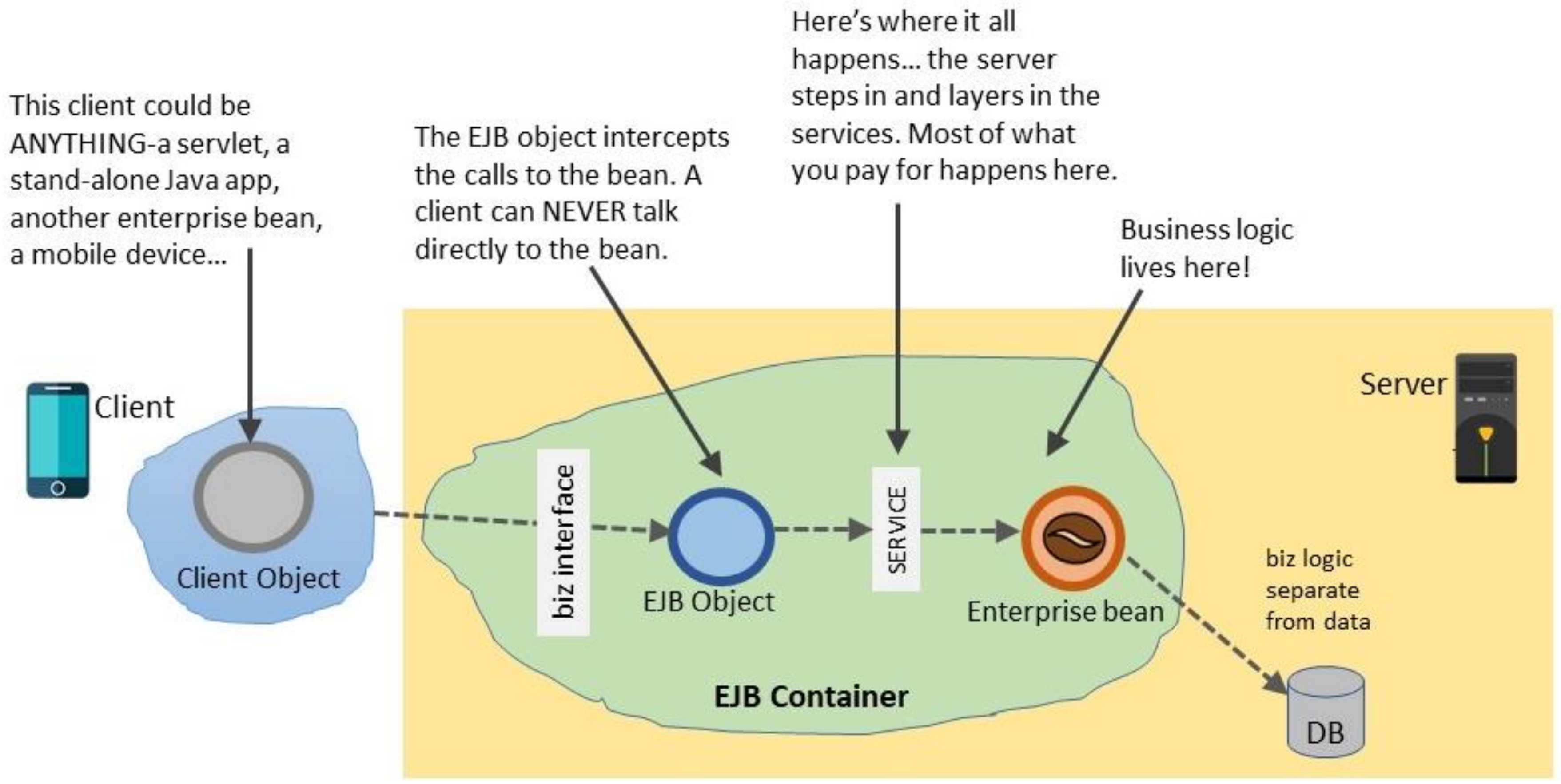
EJB Servers give several services

- Transaction Management
- Security
- Concurrency
- Networking
- Resource Management
- Persistence
- Messaging
- Deploy-time Customization

Advantages

- EJBs provide low-level system services.
- Can easily load balance requests when a large number of clients access the application concurrently.
- EJBs provide transactional capabilities to enterprise applications.
- Client code is simplified.
- EJB components can be secured for access.
- EJBs can be accessed by multiple different types of clients.

EJB Architecture


Here's where it all happens... the server steps in and layers in the services. Most of what you pay for happens here.
This client could be ANYTHING-a servlet, a stand-alone Java app, another enterprise bean, a mobile device...
The EJB object intercepts the calls to the bean. A client can NEVER talk directly to the bean.
Business logic lives here!
Client
Server
Client Object
biz interface
EJB Object
SERVICE
Enterprise bean
biz logic separate from data
EJB Container
DB

EJB Architecture

- EJB architecture uses an EJBObject intercept client calls to a bean. This gives the server/container a chance to step in and add services.
- EJB services include transactions,security,resource management, networking and persistence.

Types of EJB

- Java EE specification defines three different types of EJBs:
  - **Session Bean:** Performs an operation when called from a client. A session bean typically represents a process.
  - **Message Driven Bean (MDB):** Used for asynchronous communication between components in a Java EE application and can be used to receive Java Messaging Service (JMS) compatible messages and take some action based on the content of the received messages. A typical message-driven bean might be a NewCustomerNotification subscriber.
  - **Entity Bean:** It encapsulates the state that can be persisted in the database. It is deprecated. Use an entity bean to represent a thing in a persistent store.

Session Beans

- A session bean performs operations for the client.
- Provides an interface to clients.
- Encapsulates business logic methods.
- It can be transactional.
- Session EJBs can be clustered and deployed across multiple machines in a client transparent manner.

Stateful Beans

- Maintain conversational state with clients across multiple calls.
- Associated client state in its instance variables.
- One-to-one relationship between the number of stateful bean instances and the number of clients.
- EJB Container creates a separate stateful session bean to process client's each request.
- When a client completes the interaction with the bean and disconnects, the bean instance is destroyed.

Stateless Beans

- Does not maintain conversational state with clients between calls.
- Stateless session bean and invoke methods on it.
- The application server allocates an instance from a pool of stateless session beans.
- Useful in scenarios where the application has to serve a large number of clients concurrently accessing the bean's business methods.

Singleton Beans

- Instantiated once per application and exists for the lifecycle of the application.
- Every client request for a singleton bean goes to the same instance.
- Used in scenarios where a single enterprise bean instance is shared across multiple clients.

Message Passing Beans

- Enables Java EE applications to process messages asynchronously.
- Provide an event driven loosely coupled model for application development.
- Stateless and does not maintain any conversational state with clients.

Entity Beans

- Remote object that manages persistent data.
- Performs complex business logic.
- Uses several dependent Java objects.
- Entity beans are retained after the end of a session, unlike session beans.
- Entity beans permit shared data access.
- Entity beans have a primary key or a unique identifier.

Summary

- Beans come in three flavours: Entity, Session and Message-driven.
- Entity beans represent a uniquely identifiable thing in a persistent store; usually that means a row in a database table.
- Message-driven beans are JMS messaging service consumers.
- Session beans are everything else.
- Session beans can be stateful or stateless.
- Stateful beans can remember "conversational state" with a client, while stateless beans cannot.

Steps to build a Bean:

1. Code the bean class with all of the business methods.
2. Code two interfaces for the bean: home and component.
3. Create an XML deployment descriptor that tells the server what your bean is and how it should be managed. You must name it ejb-jar-xml.
4. Put the bean, the interfaces and the deployment descriptor into an ejb-jar file.
5. Deploy the bean into the server, using the tools provided by the server vendor.

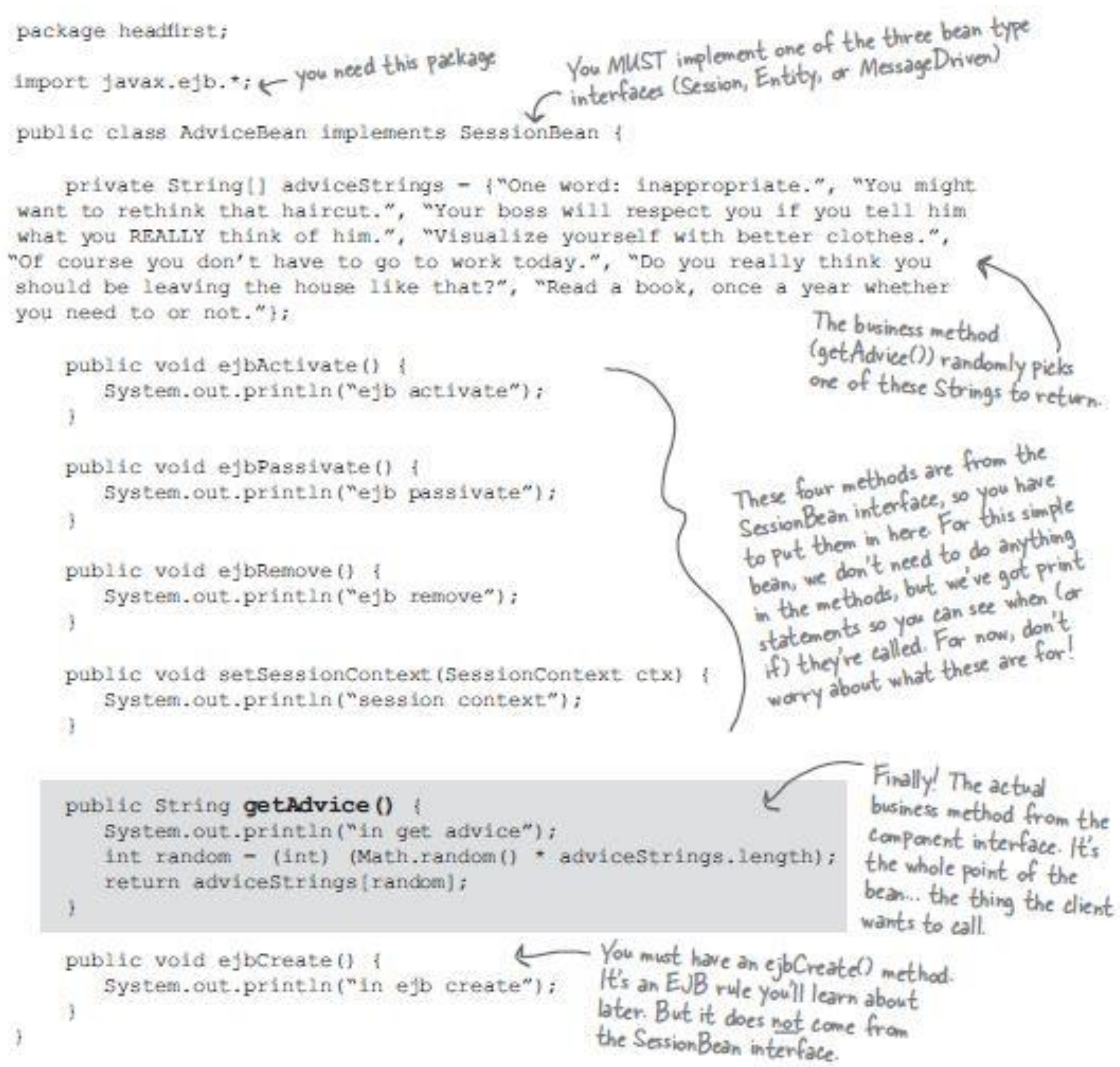
Steps

1. bean class
2. interfaces
3. XML DD
4. ejb-jar
5. deploy

1. bean class

- Write the bean class with the actual business methods the client calls.
- The implementation of your business methods defined in the component interface.
- You write your business logic in the bean class.

Bean Class Example



```
package headfirst;

import javax.ejb.*;

public class AdviceBean implements SessionBean {

    private String[] adviceStrings = {"One word: inappropriate.", "You might
 want to rethink that haircut.", "Your boss will respect you if you tell him
 what you REALLY think of him.", "Visualize yourself with better clothes.",
"Of course you don't have to go to work today.", "Do you really think you
 should be leaving the house like that?", "Read a book, once a year whether
 you need to or not."};

    public void ejbActivate() {
       System.out.println("ejb activate");
    }

    public void ejbPassivate() {
       System.out.println("ejb passivate");
    }

    public void ejbRemove() {
       System.out.println("ejb remove");
    }

    public void setSessionContext(SessionContext ctx) {
       System.out.println("session context");
    }

    public String getAdvice() {
       System.out.println("in get advice");
       int random = (int) (Math.random() * adviceStrings.length);
       return adviceStrings[random];
    }

    public void ejbCreate() {
       System.out.println("in ejb create");
    }
}
```

you need this package

You MUST implement one of the three bean type interfaces (Session, Entity, or MessageDriven)

The business method (getAdvice()) randomly picks one of these Strings to return.

These four methods are from the SessionBean interface, so you have to put them in here. For this simple bean, we don't need to do anything in the methods, but we've got print statements so you can see when (or if) they're called. For now, don't worry about what these are for!

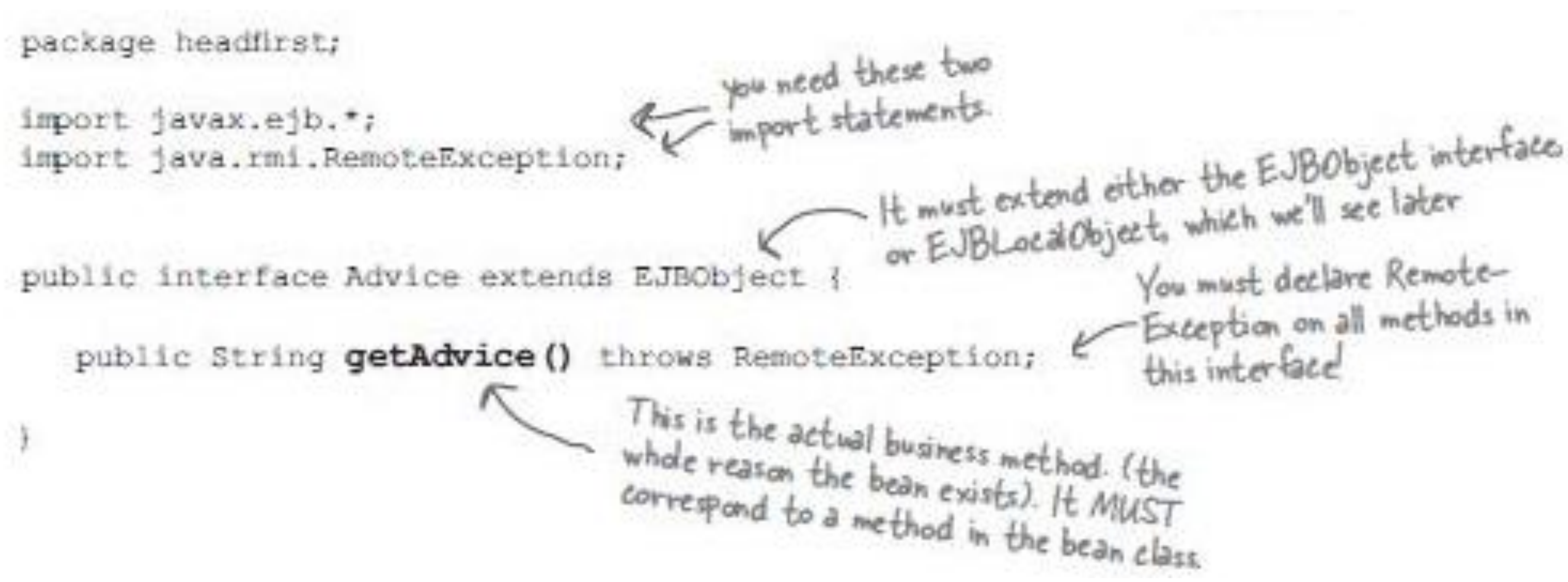
Finally! The actual business method from the component interface. It's the whole point of the bean... the thing the client wants to call.

You must have an ejbCreate() method. It's an EJB rule you'll learn about later. But it does not come from the SessionBean interface.

2. interfaces

- There are two interfaces that client sees.
- Component Interface - This is where all the business methods are declared. It's where you put the methods the client wants to call.
- Home Interface - The client uses the home interface to ask for a reference to the component interface. The home is the client's starting point for getting hold of a reference to a bean.

Component Interface Example



```java
package headfirst;

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


public interface Advice extends EJBObject {

   public String getAdvice() throws RemoteException;

}
```

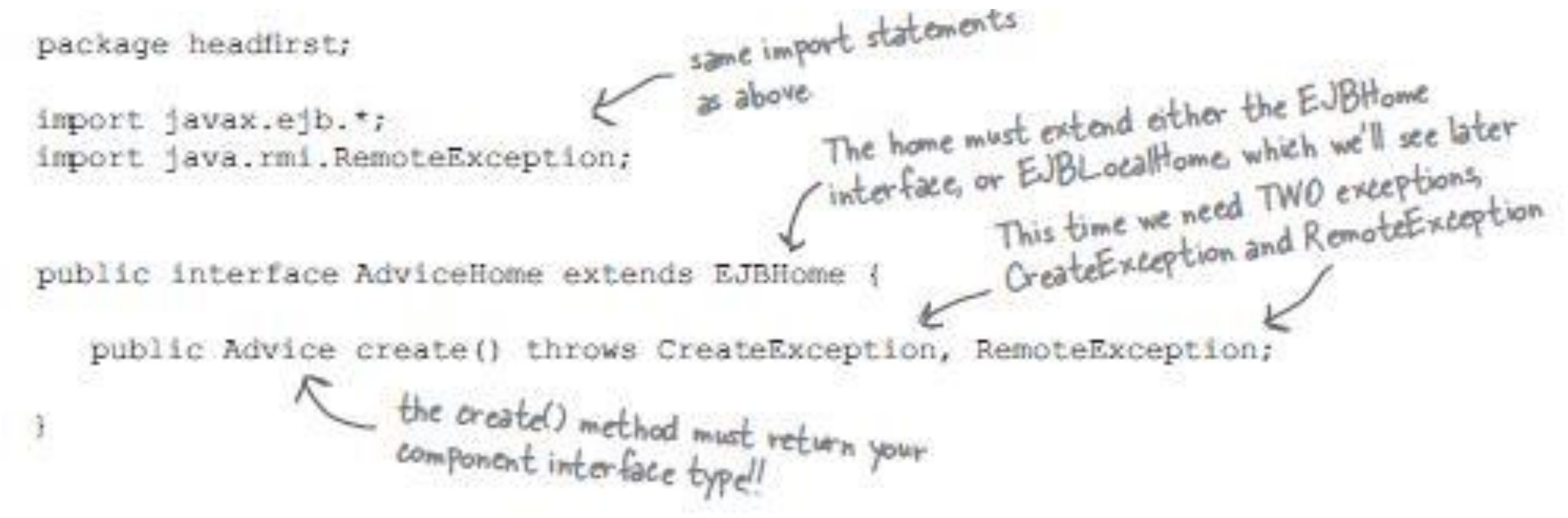
you need these two import statements.

It must extend either the EJBObject interface or EJBLocalObject, which we'll see later

You must declare Remote-Exception on all methods in this interface!

This is the actual business method. (the whole reason the bean exists). It MUST correspond to a method in the bean class.

Home Interface Example



```
package headfirst;

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


public interface AdviceHome extends EJBHome {

   public Advice create() throws CreateException, RemoteException;

}
```

same import statements as above

The home must extend either the EJBHome interface, or EJBLocalHome, which we'll see later

This time we need TWO exceptions, CreateException and RemoteException

the create() method must return your component interface type!!

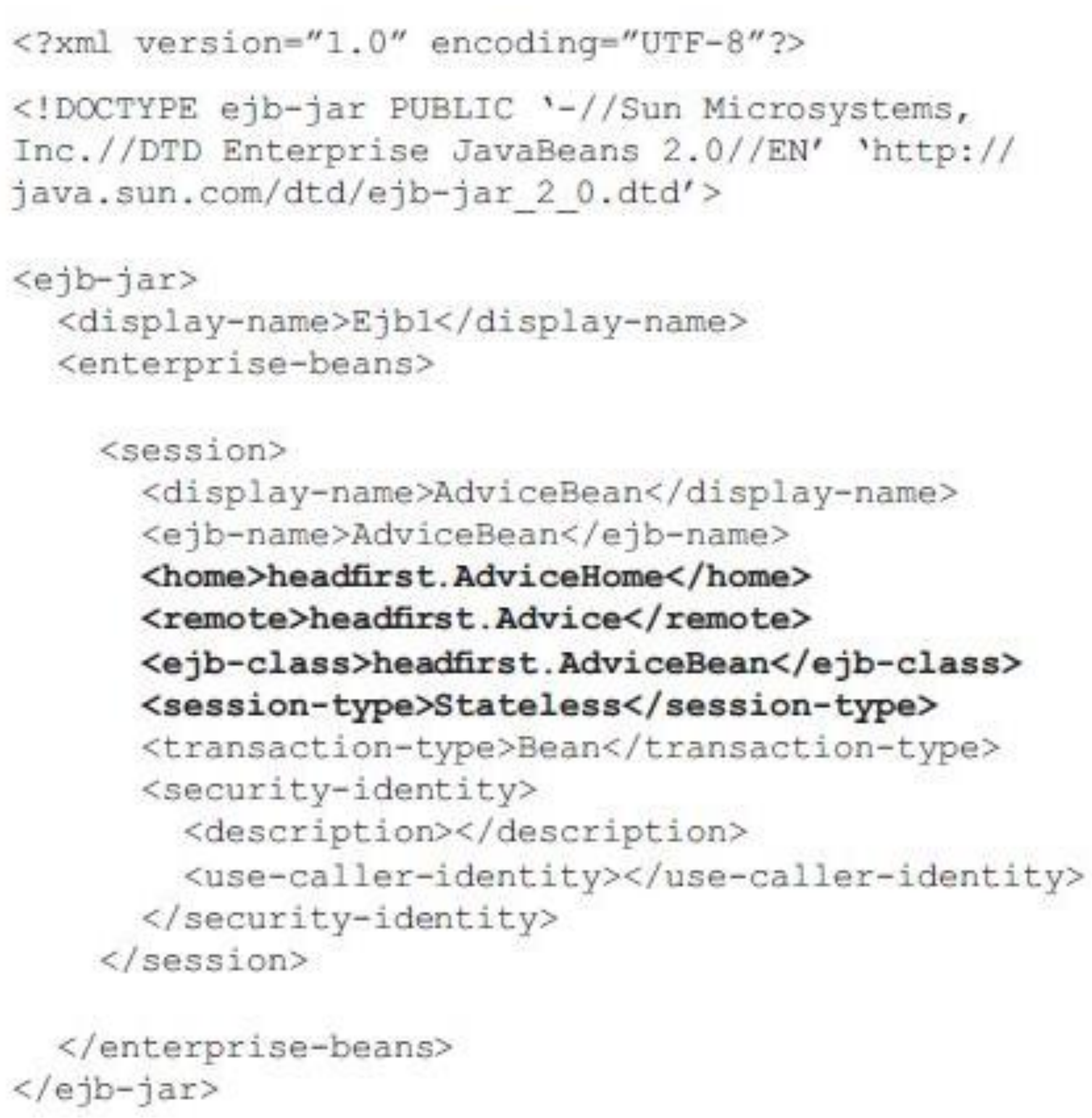
3. XML DD

- Create an XML deployment descriptor that tells the server what your bean is and how it should be managed.
- The deployment descriptor describes the structure of your bean including how the three files(component interface, home interface and bean class) are related to one another.
- You have to tell the server, through the DD, which class is which, and how they are connected.

Deployment descriptors and deploying a Java Bean

- To deploy a bean need to create a deployment descriptor for the bean and then wrapping the entire bean into a deployable unit.
- Deployment descriptors describes how the web application should be deployed.
- Deployment descriptor is written using the EXtensible Markup Language (XML) syntax.
- ***Refer to topic 3 and 4 for more details about Deployment Descriptor.***

Deployment Descriptor Example

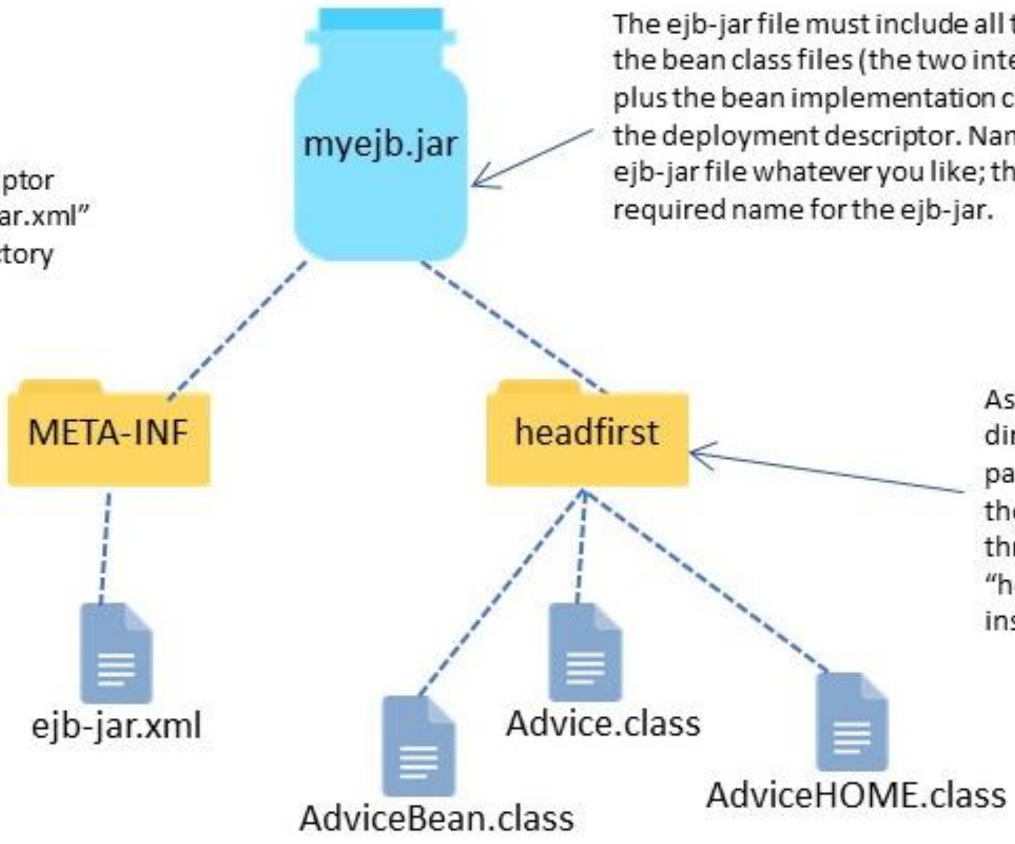


```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems,
Inc.//DTD Enterprise JavaBeans 2.0//EN' 'http://
java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>Ejb1</display-name>
  <enterprise-beans>

    <session>
      <display-name>AdviceBean</display-name>
      <ejb-name>AdviceBean</ejb-name>
      <home>headfirst.AdviceHome</home>
      <remote>headfirst.Advice</remote>
      <ejb-class>headfirst.AdviceBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <description></description>
        <use-caller-identity></use-caller-identity>
      </security-identity>
    </session>

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

4. ejb-jar

- Put the bean, the interfaces, and the deployment descriptor into an ejb-jar file.
- As a bean developer, you'll always put your beans in a JAR. An ejb-jar file holds the things the bean depends on.

ejb-jar



The ejb-jar file must include all three of the bean class files (the two interfaces plus the bean implementation class) and the deployment descriptor. Name the ejb-jar file whatever you like; there's no required name for the ejb-jar.

The deployment descriptor MUST be named "ejb-jar.xml" and MUST be in a directory named "META-INF"

As with any other JAR file, the directory structure matching the package must be immediately inside the JAR. In other words, since these three have been defined in the "hedfirst" package, the files must be inside the "headfirst" directory.

5. deploy

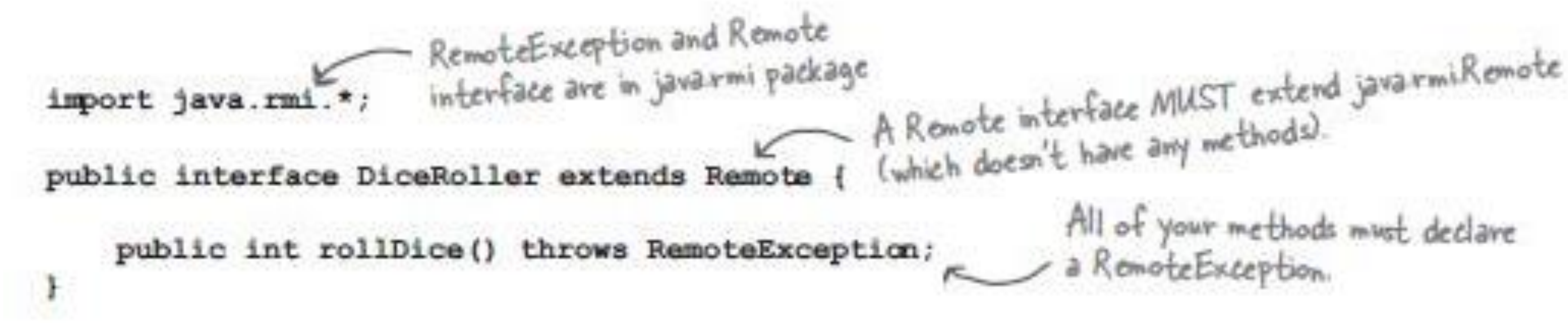
- Application Assembly - This means taking the bean from the reusable component stage to being part of an application. Simple beans, that might mean simply writing a client that can access the bean.
- Deployment - The two crucial parts of deployment are naming the bean and getting the bean into the container's control.

Remote Method Invocation

- EJB uses Java RMI(Remote Method Invocation) so that beans can be accessed by remote clients.
- A remote client, is an object running in a different JVM which means a different heap.
- A remote object stays in its own heap, while clients invoke methods on the Remote object's proxy, called a stub.
- The stub object handles all he low-level networking details in communicating with he Remote object.
- When the client wants to call a method on a Remote object, the client calls the same method on the stub.

- To be Remote, an interface must follow three rules;
  - It must extend java.rmi.Remote
  - each method must declare a java.rmi.RemoteException
  - arguments and return types must be shippable(Serializable, primitive etc.)

Code example



```
import java.rmi.*;

public interface DiceRoller extends Remote {

    public int rollDice() throws RemoteException;
}
```

RemoteException and Remote interface are in java.rmi package

A Remote interface MUST extend java.rmi.Remote (which doesn't have any methods).

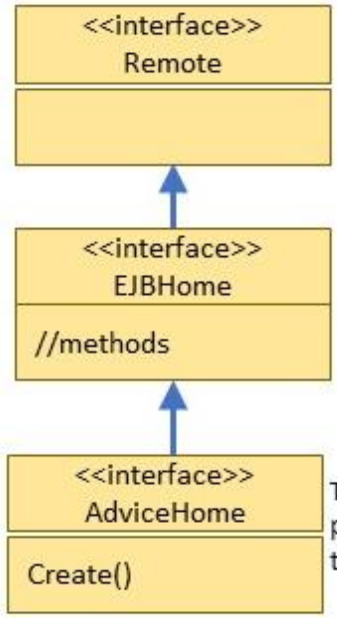
All of your methods must declare a RemoteException.

How a client uses a session bean:
create, use and remove

- Create - Client asks the home interface for a reference to the bean's component interface.

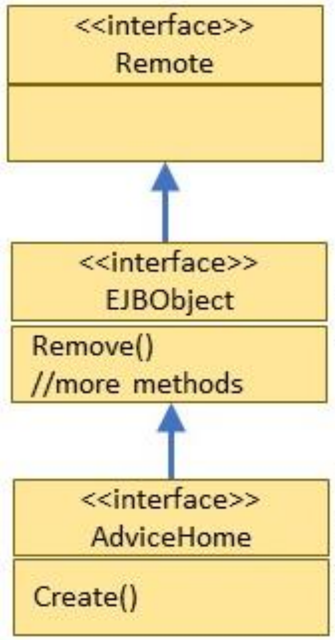
- Use - Client calls business methods declared in the component interface.

- Remove - Client tells the bean that he's done using it.



Note: over the next several pages, we're looking at only the REMOTE interface for a bean. Later we'll see the special-case "local" versions).

The bean developer must put a create() method in the bean's home



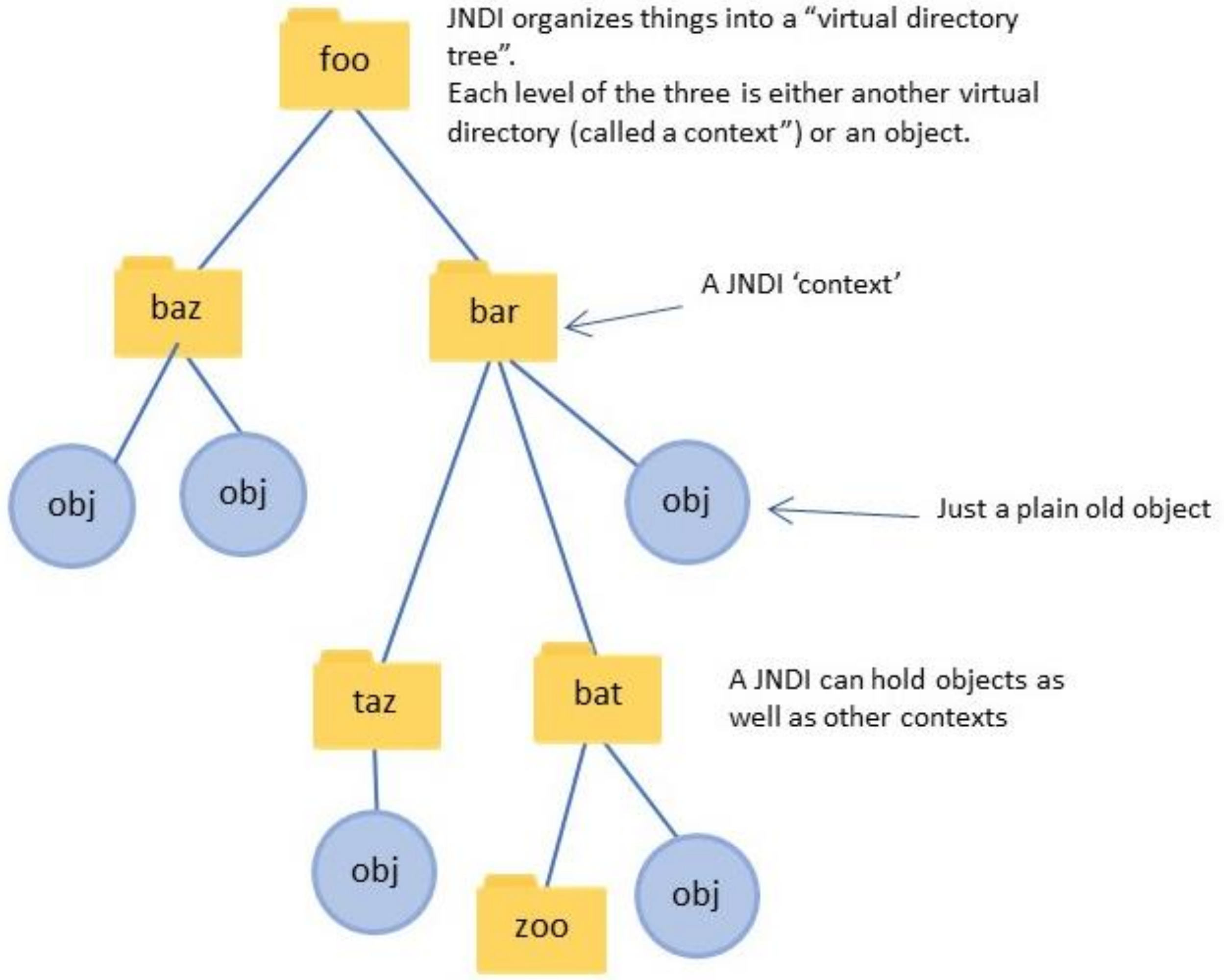
The javax.ejb.EJBObject interface has a remove() method the client can call.

In the component interface, the developer puts in the business methods, but since the component interface EXTENDS the EJBObject interface, the client sees the methods from BOTH interfaces.

What is JNDI

- JANDI stands for java Naming and Directory Interface, and it's an API for accessing naming and directory services.
- The JNDI driver translates the method calls you make on the JNDI API into something the underlying naming/directory service understands.
- JNDI organizes things into a "virtual directory tree".
- Each level of the tree is either another virtual directory or an object.
- A JNDI context can hold objects as well as other contexts.

JNDI virtual directory structure



JNDI organizes things into a "virtual directory tree".
Each level of the three is either another virtual directory (called a context") or an object.

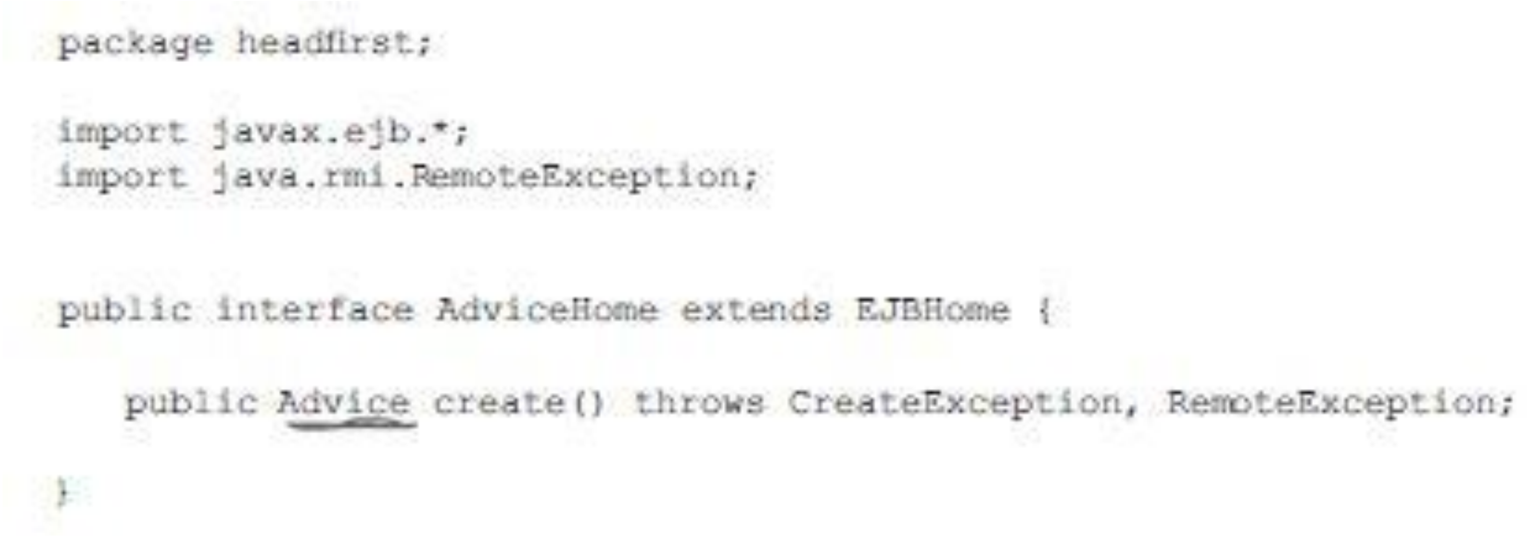
A JNDI can hold objects as well as other contexts

Writing a remote home interface for a session bean

Rules for the home Interface

- Import javax.ejb.* and java.rmi.RemoteException.
- Extend EJBHome.
- Declare create() method that returns the component interface and declares a CreateException and RemoteException.
    - For stateless session beans, there can be only one create(), and it must NOT have arguments.
    - Stateful session beans can have multiple, overloaded create() methods, and do NOT need to have a no-arg create().

There are few more rules you can find in the reference.



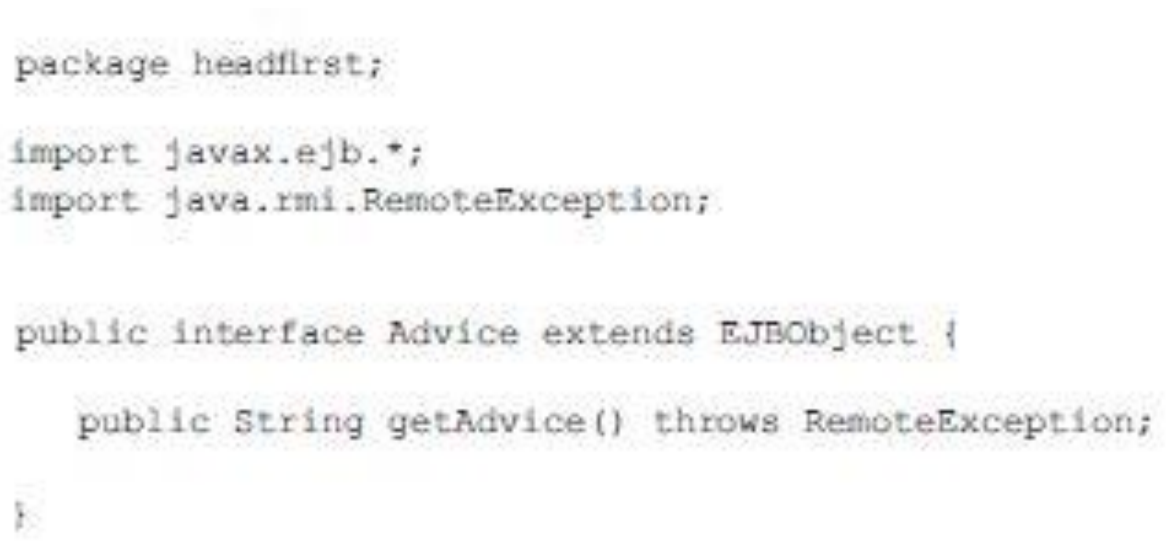
```
package headfirst;

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


public interface AdviceHome extends EJBHome {

   public Advice create() throws CreateException, RemoteException;

}
```

Writing a component interface for a session bean

Rules for the home Interface

- Import javax.ejb.* and java.rmi.RemoteException.
- Extend EJBObject.
- Declare one or more business methods, that throw a RemoteException.

There are few more rules you can find in the reference.



```
package headfirst;

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

public interface Advice extends EJBObject {

   public String getAdvice() throws RemoteException;

}
```

1.4 Generate an EJB

- JBOSS Developer Studio provides different templates to generate the shell of an EJB automatically

1.5 Converting a Plain Old Java Object(POJO) to an EJB

- POJO is an ordinary Java object, not bound by any special restriction other than those forced by the Java Language Specification and not requiring any classpath.
- To convert a POJO to an EJB;
  - annotating the POJO with one or more annotations defined in the Java EE standard.
  - running the resultant EJB in the context of an application server.
- Annotations are as follows;
  - **@Stateless**
  - **@Stateful**
  - **@Singleton**
  - **@Startup**
  - **@PostConstruct**