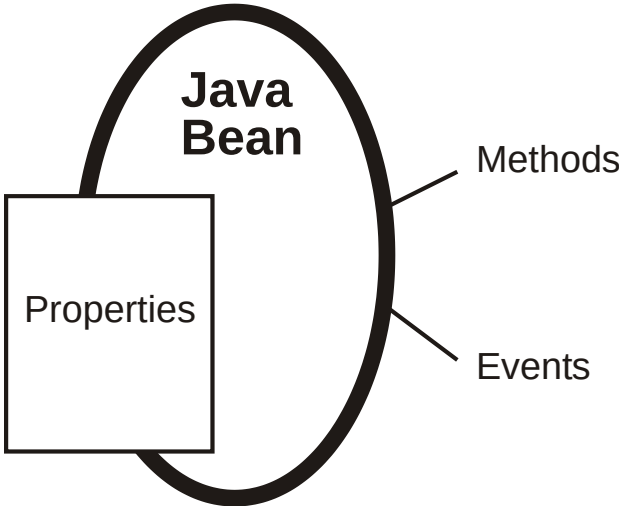
Network Computing Lecture 12

Professor Louise E. Moser

Spring 2012

Java Bean

- **Component -** A piece of software with a published interface that has properties and methods that control the way it works
- **Component Model -** Defines how different components interact with each other
- **Java Bean -** A component that has properties, methods, and events by which it communicates with other Beans



Java Bean

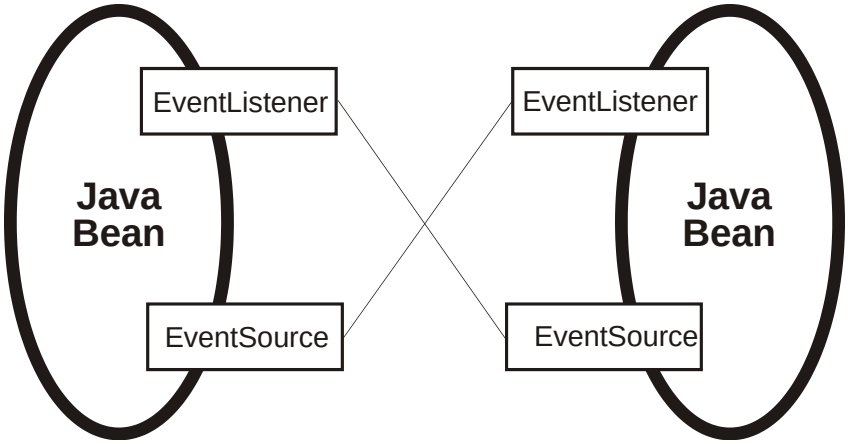
- Has a **well-published interface** that allows interactions with other Beans based on **events**
- Has **properties** that provide the Bean's internal representation, which can be saved between method invocations
- Has **methods** that follow a specific naming convention, i.e., *setXXX()* and *getXXX()*, that allow its properties to be assigned and retrieved
- Can handle its own **persistence** through **serialization** by writing to (reading from) a file, a database, or a directory using the *ObjectOutputStream (ObjectInputStream)*
- Is **packaged in a Java Archive (JAR) file**, so that it is easy to distribute and to use

Properties and Methods

```
public class myAccount
{
  private float checkingAmount;
  private float savingsAmount;

  public void setChecking(float amount)
  { checkingAmount = amount; }

  public float getChecking()
  { return checkingAmount; }

  public void setSavings(float amount)
  { savingsAmount = amount; }

  public float getSavings()
  { return savingsAmount; }
}
```

Events

- **Events** are exchanged between Java Beans through the *EventSource* and *EventListener* objects
  - The ***EventSource*** object creates and publishes events for other Beans
  - The ***EventListener*** object looks for certain kinds of events from other Beans
- Each Bean creates a listener for itself if it wants to receive events, i.e., if it subscribes for specific events



Persistence

- Save and subsequently load the state of a Bean from a file or a database, on disk

```
public void saveState()
{
  FileOutputStream fos = null;
  ObjectOutputStream oos = null;
  try
  {
    fos = new FileOutputStream(State.ser);
    oos = new ObjectOutputStream(fos);
    oos.writeObject(currentState);
  }
  catch(IOException e)
  {
    System.out.println(e.toString);
  }
}
```

Persistence (cont)

```
public void loadState()
 {
   FileInputStream fis = null;
   ObjectInputStream ois = null;
   try
   {
     fis = new FileInputStream(State.ser);
     ois = new ObjectInputStream(fis);
     State currentState = (State)ois.readObject();
   }
   catch(ClassNotFoundException e)
   {
     System.out.println(e.toString);
   }
 }
```

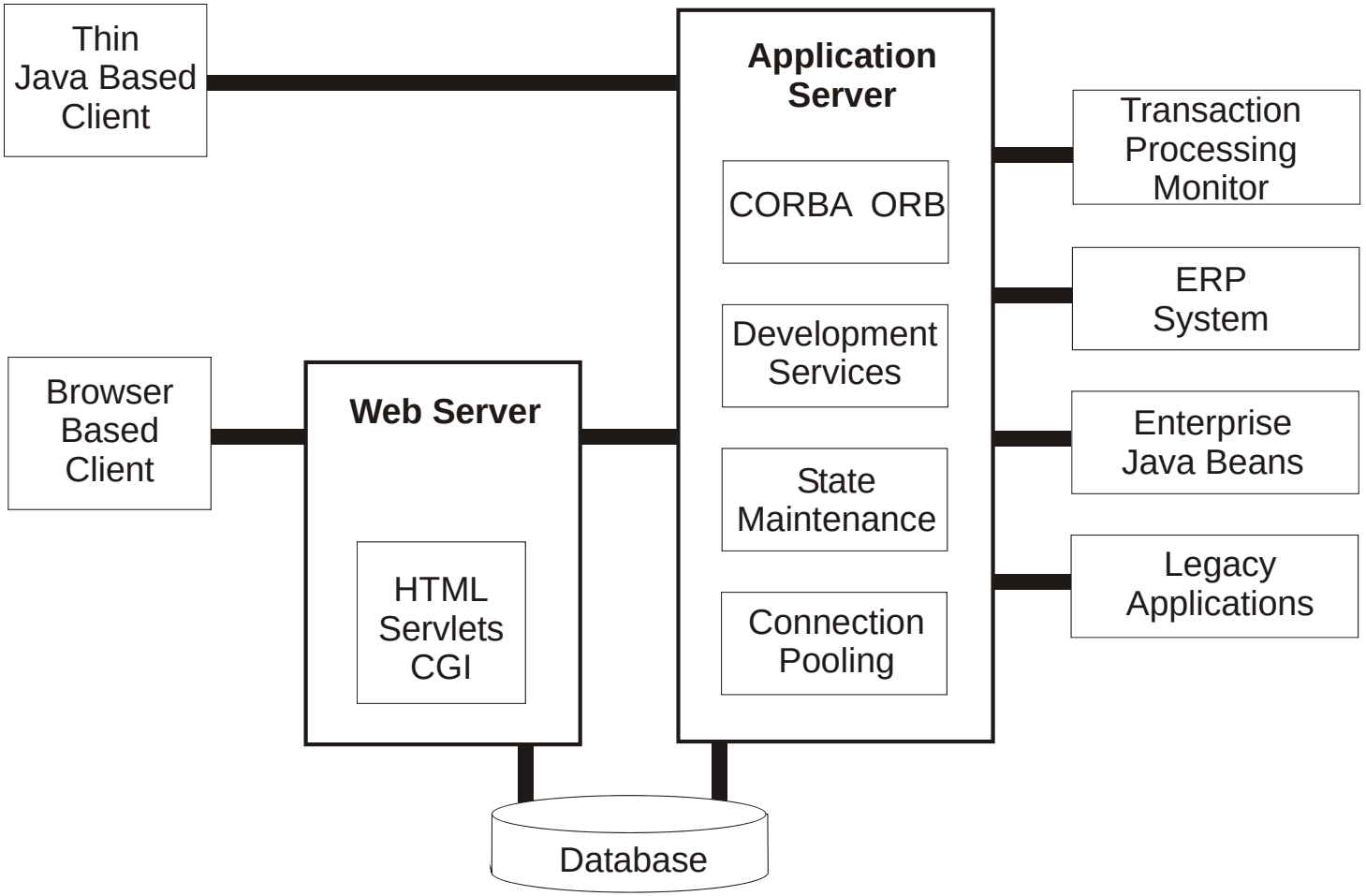
Enterprise Java Beans

- **Enterprise Java Beans (EJBs)** are Java server-side technology that allows the creation and deployment of Java components that run in an **EJB/J2EE application server**
- EJB/J2EE has the goal of **portability**, i.e., an EJB application can run on any platform and with any vendor's application server
- An EJB/J2EE application server provides **EJB containers**, which provide various services for the EJBs that they contain

EJB/J2EE Application Server

- An **EJB/J2EE application server** typically provides
  - High-performance **Web Server** or the ability to integrate with such a Web Server
  - Integration with a **Database Management System (DBMS)**
  - Ability to interface to a **Transaction Processing (TP) Monitor** (e.g., BEA's Tuxedo)
  - Ability to interface to an **Enterprise Resource Planning (ERP) System** (e.g., from SAP or QAD)
  - **Integrated Development Environment (IDE)** or the ability to integrate an IDE
- Commercial application servers include IBM's WebSphere, BEA's WebLogic, Oracle 9I, and JBoss (open source, free)

EJB/J2EE Architecture



EJB/J2EE Architecture

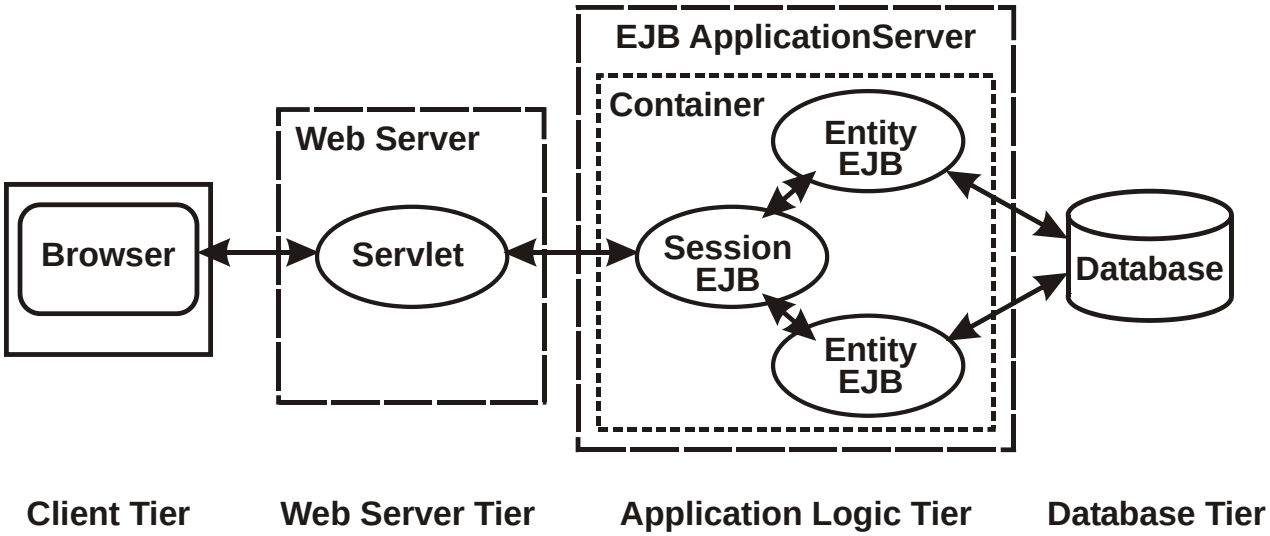
- A typical **EJB/J2EE application server** provides
  - **Execution environment**
  - **Concurrent processing**
  - **Load balancing**
  - **Device access**
- A typical **EJB/J2EE application server** uses
  - **Java Naming and Directory Interface (JNDI)**
  - **Java Transaction Service**
- An **EJB container** contains EJBs and provides an interface between the EJBs and the clients
- An **EJB client** never accesses a Bean directly, but does so by means of container methods which, in turn, invoke the Bean's methods
- An EJB client finds the EJB container using the JNDI

EJB Container

- An **EJB container** provides various services for the Beans that it contains, including the following services:
  - **Lifecycle** - Manages process, thread, object, and Bean creation, destruction and cleanup
  - **Security** - Performs security checking for Beans
  - **Transactions** - Starts, commits, and rolls back transactions on behalf of Beans
  - **Persistence** - Manages the storage and restoration of persistent state for Beans

Four-Tier EJB Application

- A **four-tier EJB application** consists of
  - **Browser-Based Client**
  - **Web Server and Servlets/JSPs**
  - **EJBs in Containers provided by the EJB Application Server**
  - **Database Management System**



When To Use EJB/J2EE

- **Use EJB/J2EE**
  - If your business logic must be protected by a firewall, where the Web server and the EJB/J2EE application server are on separate machines with the firewall between them
  - If you need additional services, such as persistence, transactions, failover, and load balancing
  - If you need to build your system faster and more reliably, and can use industry-standard tools
- **Do not use EJB/J2EE**
  - If your application can't be easily built, using EJB's limitations related to single threading, static variables, and native code
  - Your application is relatively simple, or does not involve much business logic, and can be built as a two- or three-tier system using other Java technology

When To Use EJB/J2EE with Servlets/JSPs

- It is not necessary to use EJBs with Servlets/JSPs
- It is appropriate to use EJBs with Servlets/JSPs
  - If you want to keep the presentation code separate from the business logic
  - If your users are outside firewalls, such as anonymous Internet clients, business partners, or other departments within an organization
  - If your clients are at a distant location across the Internet
- In this lecture, we will consider a simple HelloWorld application, which does not use EJBs with Servlets
- In the handout, you will find a Banking application, which does use EJBs with Servlets

Session Beans and Entity Beans

- **EJBs** are categorized as
  - **Session Beans**, which are further categorized as
    - **Stateless Session Beans**, which can be pooled to serve multiple clients
    - **Stateful Session Beans**, which serve a single client
  - **Entity Beans**, which have state and can be shared by multiple clients and are typically associated with databases
- The states of Stateful Session Beans and Entity Beans can be persisted to, and retrieved from, disk using
  - **Passivation** - Swapping out the Bean and storing its state on disk
  - **Activation** - Swapping in the Bean and restoring its state from disk

Session Beans and Entity Beans (cont)

- For **Entity Beans**, **persistence** is either
  - **Container Managed -** Container is responsible for saving the Bean's state; container managed fields are specified in a Deployment Descriptor
  - **Bean Managed -** Bean is responsible for saving its own state by invoking database methods
- An EJB is deployed as a serialized instance (****.ser*** file) and is listed in a "manifest" file
- An EJB has a Deployment Descriptor (i.e., ***SessionDescriptor*** or ***EntityDescriptor***)

When To Use Session Beans and Entity Beans

- **Session Beans**
  - Use Session Beans for the business application logic
  - Use Session Beans as the interface to the client
  - Use Session Beans to control the workflow of a set of Entity Beans
  - Do not expect to reuse Session Beans
- **Entity Beans**
  - Use Entity Beans to wrap your JDBC code
  - Use Entity Beans to enforce accuracy and integrity of the database
  - Use Entity Beans for objects that are shared by multiple clients and that need to be persisted
  - Plan to reuse Entity Beans

EJB Interfaces and Implementation

- When you develop a **(Server) EJB**, you need to implement the
  - **Home interface**
  - **Remote interface**
  - **EJB implementation**
  - **XML deployment descriptor**
- You also need to implement the **Client**
- The Client may interact directly with the EJB, or it may be a browser-based Client that interacts with a Web Server and Servlets/JSPs
  - The Servets/JSPs in turn interact with the EJBs and produce dynamic Web pages for the Client
- Next, we consider the HelloWorld application, which does not use Servlets with EJBs

Home Interface

- The **home interface** acts as a **factory** for EJBs, and enables the container to create and destroy EJBs

HelloWorldHome.java

```
package ie.tcd.cs.ejb_example;

public interface HelloWorldHome extends javax.ejb.EJBHome
    {
    HelloWorld create() throws java.rmi.RemoteException,
                               javax.ejb.CreateException;
    }
```

Remote Interface

- The **remote interface** provides the business methods that the client calls to have the EJB do the work

HelloWorld.java

```
package ie.tcd.cs.ejb_example;
public interface HelloWorld extends javax.ejb.EJBObject
{
 public String hello() throws java.rmi.RemoteException;
}
```

EJB Implementation

- The **EJB class** implements the application business logic using a **Session Bean**

HelloWorldBean.java

```
package ie.tcd.cs.ejb_example;
import javax.ejb.SessionContext;

public class HelloWorldBean implements javax.ejb.SessionBean
  private SessionContext  ctx;
  public void setSessionContext (SessionContext ctx) {
    this.ctx = ctx;
  }
public String hello() {
    System.out.println("hello()");
    return "Hello World!";
}
```

EJB Implementation (cont)

```
public void ejbCreate() {
    System.out.println("ejbCreate()");
}
public void ejbRemove() {
    System.out.println("ejbRemove()");
  }
public void ejbActivate() {
    System.out.println("ejbActivate()");
}
public void ejbPassivate() {
    System.out.println("ejbPassivate()");
}
```

XML Deployment Descriptor

- The **XML Deployment Descriptor** is used to specify attributes of Beans, such as
  - **Security**
  - **Transactions**
  - **Resource usage**
  - **References to other Beans**
- You write the Deployment Descriptor in XML, and edit it by hand or using a tool such as XDocLet
- The Deployment Descriptor provides information that the container uses for the EJBs it contains

XML Deployment Descriptor (cont)

ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar>
  <descriptor>JBoss Hello World Application </descriptor>
  <display-name> </display-name>
  <enterprise-beans>
    <session>
      <ejb-name>HelloWorld</ejb-name>
      <home>ie.tcd.cs.ejb_example.HelloWorldHome</home>
      <remote>ie.tcd.cs.ejb_example.HelloWorld</remote>
      <ejb-class> ie.tcd.cs.ejb_example.HelloWorldBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Client

HelloWorldClient.java

```
package ie.tcd.cs.ejb_example;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;

public class HelloWorldClient {
    public static void main(String [] args) {
        Hashtable env = new Hashtable();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
        env.put(Context.PROVIDER_URL, "localhost:1099");
        env.put("java.naming.factory.url.pkgs",
                "org.jboss.naming:org.jnp.interfaces");
```

Client (cont)

```
try {
    Context ctx = new InitialContext(env);
    Object obj = ctx.lookup("HelloWorld");
    HelloWorldHome home =
      (HelloWorldHome)javax.rmi.PortableRemoteObject.
                              narrow(obj, HelloWorldHome.class);
    HelloWorld helloWorld = home.create();
    System.out.println(helloWorld.hello());
    helloWorld.remove();
}
catch (Exception e) {
    e.printStackTrace();
    System.out.println("Exception: " + e.getMessage());
}
```