Enterprise Application Integration

Java Persistence API & Enterprise JavaBeans

2014/2015

Filipe Araújo
Informatics Engineering Department
University of Coimbra
filipius@uc.pt

SQL Example – Creating & Writing Table

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;
...

public class SQLWriteStudents {
    final static String name = ...
    final static String password = ...
    final static String tablename = "STUDENTS";

    public static void main(String[] args) throws SQLException {
        String url = "jdbc:mysql://localhost:3306/StudentDB";
        Connection conn = (Connection)
              DriverManager.getConnection(url, name, password);
        Statement stmt = (Statement) conn.createStatement();

        String sql = "show tables like '" + tablename + "'";
        ResultSet rs;
        rs = stmt.executeQuery(sql);
        if (!rs.first()) { //create the table
                sql = "CREATE TABLE " + tablename +
                      "(Name VARCHAR(254), Phone VARCHAR(20))";
                stmt.executeUpdate(sql);
        }
```


SQL Example – Creating & Writing Table

```
        List<Student> mylist = GetStudentInfo.get();

        int rows = 0;
        for (Student st : mylist) {
            sql = "INSERT INTO " + tablename +
                  "(Name, Phone)" +
                  "VALUES" +
                  "('" + st.getName() + "','" + st.getPhone() + "')";

            rows += stmt.executeUpdate(sql);
        }

        System.out.println("Added " + rows + " students.");
    }
}
```

```
Output - JPA (run) #2
run:
Student's name (empty to finish):
Pedro
Student's phone:
555183774
Student's name (empty to finish):
Paulo
Student's phone:
555883347
Student's name (empty to finish):
Maria
Student's phone:
555888377
Student's name (empty to finish):

Added 3 students.
BUILD SUCCESSFUL (total time: 1 minute 4 seconds)
```

```
mysql> select * from students;
+-------+-----------+
| Name  | Phone     |
+-------+-----------+
| Pedro | 555183774 |
| Paulo | 555883347 |
| Maria | 555888377 |
+-------+-----------+
3 rows in set (0.00 sec)
```


SQL Example – Reading Table

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.Statement;

public class SQLReadStudents {
    ...
    public static void main(String[] args) {
        try {
            String url = "jdbc:mysql://localhost:3306/StudentDB";
            Connection conn = (Connection)
                 DriverManager.getConnection(url, name, password);
            Statement stmt = (Statement) conn.createStatement();
            ResultSet rs;

            rs = stmt.executeQuery("SELECT * FROM STUDENTS");
            while (rs.next()) {
                String nme = rs.getString("Name");
                String phone = rs.getString("Phone");
                System.out.println(nme + " " + phone);
            }
            conn.close();
        } catch (Exception e) {
            System.err.println("Got an exception! ");
            System.err.println(e.getMessage());
        }
    }
}
```

Persistence

- Programming SQL in Java is somewhat cumbersome
- Alternative: stick to the objects and use a Persistence API
  - Hibernate
  - Toplink
  - JDO
  - EclipseLink
  - …
- We shall illustrate Hibernate XML mapping between POJOs and database tables
  - Note that Hibernate also supports annotations

Hibernate XML Mapping – Writing to Database

```
public class Main {

    public static void main(String[] args) {

        List<Student> mylist = GetStudentInfo.get();

        Session session =
              HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();
        for (Student st : mylist) {
            session.save(st);
        }
        session.getTransaction().commit();
    }
}
```

```
Output - JPA_Hibernate (run)
run:
Student's name (empty to finish):
Paulo Jorge
Student's phone:
555113456
Student's name (empty to finish):
Luisa Reis
Student's phone:
555184444
Student's name (empty to finish):
Sara Peixoto
Student's phone:
555848888
Student's name (empty to finish):

Hibernate: insert into STUDENTS_HIBERNATE (name, phone) values (?, ?)
Hibernate: insert into STUDENTS_HIBERNATE (name, phone) values (?, ?)
Hibernate: insert into STUDENTS_HIBERNATE (name, phone) values (?, ?)
```

```
mysql> select * from students_hibernate;
+------------+--------------+-----------+
| STUDENT_ID | name         | phone     |
+------------+--------------+-----------+
|          1 | Paulo Jorge  | 555113456 |
|          2 | Luisa Reis   | 555184444 |
|          3 | Sara Peixoto | 555848888 |
+------------+--------------+-----------+
3 rows in set (0.00 sec)
```


Hibernate XML Mapping – Helper class

```
package util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static final SessionFactory sessionFactory =
                                buildSessionFactory();

    private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            return new
                  Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception
            System.err.println("Initial SessionFactory creation
failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
```


Hibernate XML Mapping – the Class & the XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/
Hibernate Mapping DTD 3.0//EN" "http://
hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="jpa_hibernate">
  <class name="Student" table="STUDENTS_HIBERNATE">
    <id column="STUDENT_ID" name="id">
      <generator class="native"/>
    </id>
    <property name="name"/>
    <property name="phone"/>
  </class>
</hibernate-mapping>
```

```
public class Student {
    private int id;
    private String name;
    private String phone;

    // constructor,
getters and setters
follow
```

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/
Hibernate Configuration DTD 3.0//EN" "http://
hibernate.sourceforge.net/hibernate-
configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <!-- Many things missing -->
    <mapping resource="jpa_hibernate/Mapping.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Hibernate.cfg.xml refers to the hibernate mapping file

JPA – Java Persistence API

- JPA provides developers with an object/relational mapping facility for managing relational data in Java applications
  - Simplifies development of Java EE and SE
- JPA Standardizes Persistence (part of JSR-318 – EJB 3.1)
  - Hibernate, TopLink,…, or JDO? None is really a standard
- Full object/relational mapping
  - Through Java metadata annotations
  - Or XML descriptors
- Java Persistence consists of four areas:
  - The Java Persistence API
  - The query language (similar to SQL)
  - The Java Persistence Criteria API
  - Object/relational mapping metadata

JPA – Entity & Main Class

```
public class Main {

    public static void main(String[] args) {
        List<Student> mylist = GetStudentInfo.get();

        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("TestPersistence");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();

        tx.begin();
        for (Student st : mylist) {
            em.persist(st);
        }
        tx.commit();

    }
}
```

The **Entity** Class (a lightweight persistence domain object) is usually a table.

Each instance is a row of the table in the DB

```
package common;
//… imports
@Entity
@Table(name = "STUDENT2")
public class Student implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "studentid", nullable = false)
    private Long id;
    private String name;
    //constructors, getters and setters follow
}
```

JPA persistence.xml (standalone app)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
     xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/
persistence/persistence_2_0.xsd">
     <persistence-unit name="TestPersistence" transaction-type="RESOURCE_LOCAL">
          <provider>org.hibernate.ejb.HibernatePersistence</provider>
          <class>common.Student</class>
          <properties>
               <property name="hibernate.dialect"
                         value="org.hibernate.dialect.PostgreSQLDialect" />
               <property name="hibernate.hbm2ddl.auto" value="update" />
               <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
               <property name="javax.persistence.jdbc.url"
                         value="jdbc:postgresql://localhost/postgres" />
               <property name="javax.persistence.jdbc.user" value="postgres" />
               <property name="javax.persistence.jdbc.password" value="postgres" />
          </properties>
     </persistence-unit>
</persistence>
```

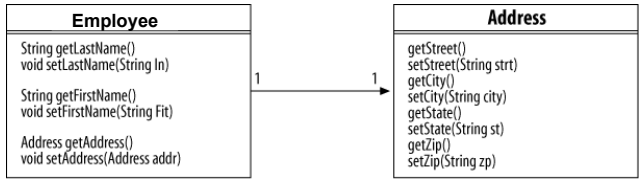
Referred to in the Java source code

JPA Annotations Define

- Entities
- Primary keys
- Multiplicity in Entity Relationships
  - One-to-one, one-to-many, many-to-many
- We can also define the names of the tables in the database, or the name of the columns

One-to-One Unidirectional Relationships

- Each **employee has exactly one address**, and each address has exactly one employee.



- Which entity references which determines the direction of navigation
- The Employee has a reference to the Address but the Address doesn't reference the Employee
  - An Address entity has no idea who owns it
- **DB Schema:** Employee table contains a foreign key to the Address table, but the Address table doesn't contain a foreign key to the Employee table

One-to-One Unidirect. Relationship – Java

class Employee

```
@Entity
public class Employee
{
  /**
  * The employee's address
  */
  @OneToOne
  @JoinColumn(name="ADDRESS_ID")
  // Unidirectional relationship
  private Address address;
}
```

Generated table

```
CREATE TABLE "PUBLIC"."EMPLOYEE"
(
ID bigint PRIMARY KEY NOT NULL,
ADDRESS_ID bigint
) ;
ALTER TABLE "PUBLIC"."EMPLOYEE"
ADD CONSTRAINT FK4AFD4ACEE5310533
FOREIGN KEY (ADDRESS_ID)
REFERENCES
"PUBLIC"."ADDRESS"(ADDRESS_ID) ;
```

- If your persistence provider supports auto schema generation, you do not need to specify metadata such as @JoinColumn.

One-to-One Bidirectional Relationship

- Besides knowing which computer an employee has, we may need to know the owner of a specific computer
- We can expand **Employee** to include a reference to a **Computer**
- The computer table will have a pointer to its owner
- In a relational database model, there is no notion of directionality!

DB Schema

```
CREATE TABLE "PUBLIC"."COMPUTER"
(
ID bigint PRIMARY KEY NOT NULL,
MAKE varchar,
MODEL varchar,
OWNER_ID bigint
)
;
ALTER TABLE "PUBLIC"."COMPUTER"
ADD CONSTRAINT FKE023E33B5EAFBFC
FOREIGN KEY (OWNER_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(OWNER_ID) ;
```


One-to-One Bidirectional Relationship – Java

class Computer

```
@Entity
public class Computer
{
...
@OneToOne
// Bidirectional relationship, mappedBy
// is declared on the non-owning side
private Employee owner;
... }
```

class Employee

```
/**
* The employee's computer
*/
@OneToOne(mappedBy = "owner")
// Bidirectional relationship
private Computer computer;
```

- **mappedBy:** Sets up the bidirectional relationship and tells the persistence manager that the information for mapping this relationship to tables is specified in the Computer class, specifically to the *owner* property of Computer
- In bidirectional relationship types there is always an *owning* side of the relationship

One-to-One Bidirect. Relationship – Example

```
// Create a new Computer
final Computer computer = new Computer();
computer.setMake("Computicorp");
computer.setModel("ZoomFast 100");

// Create a new Employee
final Employee carloDeWolf = new Employee("Carlo de Wolf");

// Persist; now we have managed objects
EntityManager em = null; // Assume we have this
em.persist(carloDeWolf);
em.persist(computer);

// Associate *both* sides of a bidirectional relationship
carloDeWolf.setComputer(computer);
computer.setOwner(carloDeWolf);
```

- Check
  - mappedBy
  - cascade=CascadeType.ALL

One-to-Many Unidirectional Relationship

- One entity can aggregate or contain many other entities
- An **employee** may have **many phones**, each of which represents a phone number
- One-to-many and many-to-many relationships require the developer to work with a collection of references instead of a single reference when accessing the relationship field.

class Phone

```
@Entity
public class Phone
{
...
/**
* Phone number
*/
private String number;
... }
```

class Employee

```
/**
* All {@link Phone}s for this {@link
Employee}
*/
@OneToMany
// Unidirectional relationship
private Collection<Phone> phones;
```


One-to-Many Unidirectional Relationship

DB Schema (possible!)

```
CREATE TABLE "PUBLIC"."EMPLOYEE_PHONE"
(
EMPLOYEE_ID bigint NOT NULL,
PHONES_ID bigint NOT NULL
) ;
```

- The structure and relationships of the actual database can differ from the relationships as defined in the programming model (e.g., a **join-table**)
- When using legacy databases it's important to have the JPA mapping options so that the object model is not dictated by the schema

One-to-Many Bidirectional Relationship

- An **employee has a manager**, and likewise a **manager has many direct subordinates**
- One entity maintains a collection-based relationship property with another entity, and each entity in the collection holds a reference back to its aggregating entity
- The relationship is a bidirectional **one-to-many relationship from the perspective** of the manager (an Employee) and a **many-to-one from the perspective of the subordinate** (also an Employee)

One-to-Many Bidirectional Relationship

DB Schema

```
CREATE TABLE "PUBLIC"."EMPLOYEE"
(
ID bigint PRIMARY KEY NOT NULL,
NAME varchar,
MANAGER_ID bigint
)
;
ALTER TABLE "PUBLIC"."EMPLOYEE"
ADD CONSTRAINT
FK4AFD4ACE378204C2
FOREIGN KEY (MANAGER_ID)
REFERENCES
"PUBLIC"."EMPLOYEE"(MANAGER_ID) ;
```

class Employee

```
/**
* {@link Employee}s reporting to this {@link
* Employee}
*/
@OneToMany(mappedBy = "manager")
private Collection<Employee> peons;

/**
* Manager of the {@link Employee}
*/
@ManyToOne
private Employee manager;
```


Many-to-Many Bidirectional Relationship

- **One Employee** may belong to **many Teams**, and **one Team** may be composed of **many Employees**.
- Many beans maintain a collection-based relationship property with another bean, and each bean referenced in the collection maintains a collection-based relationship property back to the aggregating beans
- **DB Schema** has:
  - Team table
  - Employee table
  - A join table

Many-to-Many Bidirectional Relationship

```
@Entity
public class Employee
{
...
/**
* The {@link Team}s to which
* this {@link Employee} belongs
*/
@ManyToMany(mappedBy ="members")
private Collection<Team> teams;
 ...
}
```

```
@Entity
public class Team
{
...
/**
* {@link Employee}s on this {@link Task}.
*/
@ManyToMany
private Collection<Employee> members;
... }
```

- As with all bidirectional relationships, there has to be an owning side. In this case, it is the **Team** entity
- The mappedBy attribute identifies the property on the Team entity class that defines the relationship. This also identifies the Employee entity as the inverse side of the relationship.

Many-to-Many Unidirectional Relationship

- We may assign **many Tasks to many Employees**, and **Employees may be assigned to any number of Tasks**. We'll maintain a reference from Task to Employee, but not the other way around

```
@Entity
public class Task
{
...
/**
* {@link Employee} in charge of this {@link Task}
*/
@ManyToMany
private Collection<Employee> owners;
... }
```

- Because the relationship is unidirectional, there are no owning or inverse sides, and we may omit the mappedBy attribute of @ManyToMany.
- DB Schema has a join table.

Transient Fields

```
@Entity
public class EntityWithTransientFields {
   static int transient1; // not persistent because of static
   final int transient2 = 0;  // not persistent because of final
   transient int transient3; // not persistent because of transient
   @Transient int transient4; // not persistent because of @Transient
}
```


Querying

- Native Structured Query Language (SQL)
- JPQL (query language)
  - Declarative query language similar to the SQL, tailored for Java objects
  - To execute queries, you reference the properties and relationships of your entity beans rather than the underlying tables and columns these objects are mapped to.
  - Typically more concise and more readable than Criteria queries
- Criteria API
  - Interface for building queries using an object model
  - Can be checked for structural correctness by the Java compiler
- javax.persistence.Query interface
  - SQL & JPQL
- javax.persistence.criteria.CriteriaQuery interface
  - Criteria API

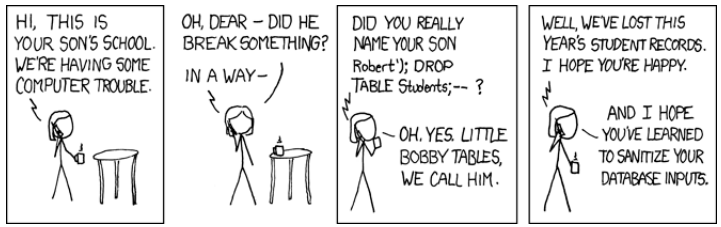
Criteria API example

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q.getResultList();
```

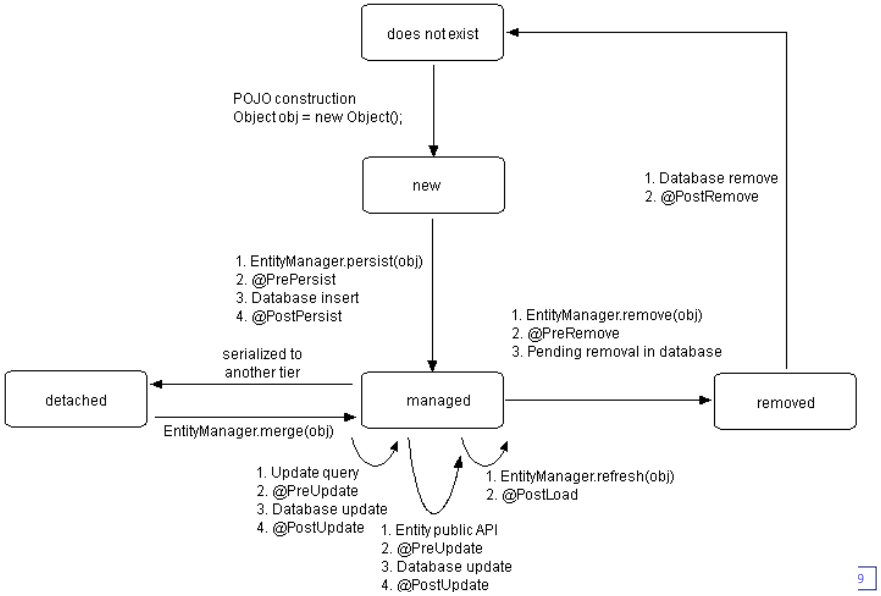

JPQL

- Using a named parameter…

```
// Define query String
String jpaQlQuery = "SELECT e FROM " + Employee.class.getSimpleName() +
" e WHERE e.name=:name";
// Set parameter
jpaQlQuery.setParameter("name", "Dave");
// Query and get result
final Employee roundtrip = (Employee)em.createQuery(jpaQlQuery).getSingleResult();
```



Entity Life Cycle



Enterprise Application Integration

Java Persistence API & Enterprise JavaBeans

2014/2015

Filipe Araújo
Informatics Engineering Department
University of Coimbra
filipius@uc.pt

Enterprise Java Beans

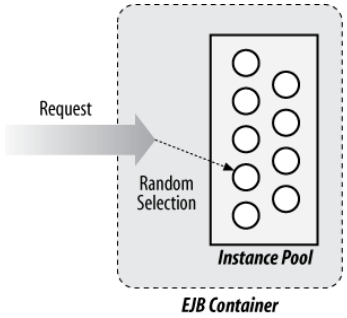
- Server-side component that encapsulates the business logic
  - i.e., the code that fulfills the purpose of the application
    - E.g., checkInventoryLevel() or orderProduct()
- EJBs simplify development of large, distributed applications
  - The EJB container provides system-level services, like transaction management and security authorization
  - The beans, not the clients, contain the logic. Thus, client developer can focus on the client
    - Clients can be thinner!
  - EJBs are portable: the application assembler can build new applications from existing beans.
- When to use EJBs
  - To make scalable applications → programmers can distribute them across several machines
  - Transactions must ensure data integrity → Enterprise beans support transactions
  - The application will have a variety of clients → Remote clients can easily locate enterprise beans: thin, various, numerous clients

Types of EJBs & When to Use them

- Session Beans
  - Perform a task for a client; optionally may implement a web service
  - Can be **Stateless** or **Stateful**
- Message-Driven Beans
  - Acts as a listener for a particular messaging type, such as JMS

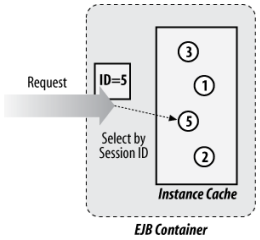
Types of Session Beans

- Stateless Session Beans
  - The bean's state has no data for a specific client.
  - A client cannot assume that subsequent requests will target any particular bean instance.
  - For performance reasons



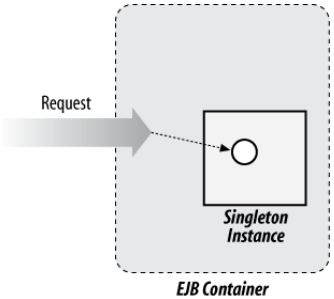
Types of Session Beans

- Stateful Session Beans
  - In a single method invocation, the bean performs a generic task for all clients. 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



Types of Session Beans

- Singleton Session Beans
  - Instantiated once per application and exists for the lifecycle of the application
  - Similar to Stateless… But only one exists!
  - May be used to perform initialization tasks for the application



Message-Driven Beans

- An enterprise bean that allows Java EE applications to process messages asynchronously
- It normally acts as a JMS message listener
- Clients do not access message-driven beans through interfaces
- Main characteristics:
  - They execute upon receipt of a single client message.
  - They are invoked asynchronously.
  - They are relatively short-lived.
  - They do not represent directly shared data in the database, but they can access and update this data.
  - They can be transaction-aware. They are **stateless**.

Example: StudentBean – The Bean

```
import javax.ejb.Stateless;

@Stateless
public class PlayStudentBean implements IPlayStudent {

    /**
     * Default constructor.
     */
    public PlayStudentBean() {}

    public int getNameLen(Student st) {
        return st.getName().length();
    }

}
```

Notice the annotations and the method name existing on both definitions

```
import javax.ejb.Remote;

@Remote
public interface IPlayStudent {
    public int getNameLen(Student st);
}
```


Example: StudentBean – The Client

```
public class BasicStudentClient {

    public static void main(String[] args) throws NamingException {
        Student st1 = new Student("Alex", "345234435");
        Student st2 = new Student("Paula", "345234435");

        InitialContext ctx = new InitialContext();
        PlayStudent ps = (IPlayStudent) ctx.lookup("StudentLen/
PlayStudentBean!namelen.IPlayStudent");

        System.out.println("len of name " +
        st1.getName() + ": " + ps.getNameLen(st1));
        System.out.println("len of name " +
        st2.getName() + ": " + ps.getNameLen(st2));
    }
}
```

File jndi.properties

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:8080
jboss.naming.client.ejb.context=true
#username
java.naming.security.principal=joao
#password
java.naming.security.credentials=pedro
```


A little Quiz

- Where does the bean live? Did you see any main method in the bean?
- How does the client discover the bean?
- What kind of object it gets in the ctx.lookup?
- What is the meaning of @Remote?
- Could we use the bean from another bean? Or from a jsp page, or from a Servlet?

Messing with a Stateless Bean – The Bean

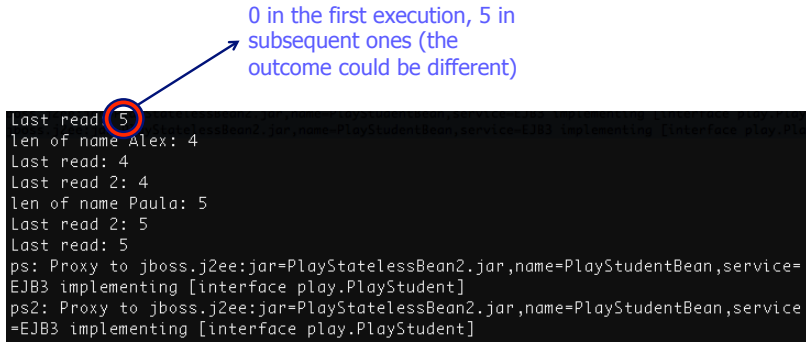
```
@Stateless
public class PlayStudentBean implements
IPlayStudent {

    int messwithme;

    /**
     * Default constructor.
     */
    public PlayStudentBean() {
        messwithme = 0;
    }

    public int getNameLen(Student st) {
        messwithme = st.getName().length();
        return messwithme;
    }

    public int getLastRead() {
        return messwithme;
    }

}
```

Methods in the @Remote interface

Messing with a Stateless Bean – The Client

```
Student st1 = new Student("Alex", "345234435");
Student st2 = new Student("Paula", "345234435");

InitialContext ctx = new InitialContext();
PlayStudent ps = (IPlayStudent)
      ctx.lookup("StudentLen/PlayStudentBean!namelen.PlayStudent");

System.out.println("Last read: " + ps.getLastRead());
System.out.println("len of name " +
      st1.getName() + ": " + ps.getNameLen(st1));
System.out.println("Last read: " + ps.getLastRead());

PlayStudent ps2 = (IPlayStudent)
      ctx.lookup("StudentLen/PlayStudentBean!namelen.PlayStudent");

System.out.println("Last read 2: " + ps2.getLastRead());
System.out.println("len of name " +
      st2.getName() + ": " + ps.getNameLen(st2));
System.out.println("Last read 2: " + ps2.getLastRead());
System.out.println("Last read: " + ps.getLastRead());

System.out.println("ps: " + ps);
System.out.println("ps2: " + ps2);
```


Messing with a Stateless Bean – **Possible** Result



Some Annotations (1)

- Access type
  - **@Local** – Local access only (i.e., same JVM)
  - **@Remote** – Allows remote access
- Session Bean type
  - **@Stateless**
  - **@Stateful**
  - **@Singleton**
- Message Driven Bean
  - **@MessageDriven**

Some Annotations (2)

- **@EJB**
  - For field level injection of EJB references
- **@Resource**
  - Used to declare a reference to a resource such as a data source, an enterprise bean, an environment entry, or a JMS Destination.
- **@PersistenceContext**
  - Dependency injection of an EntityManager

```
@PersistenceContext(name = "TestPersistence")
private EntityManager em;
```

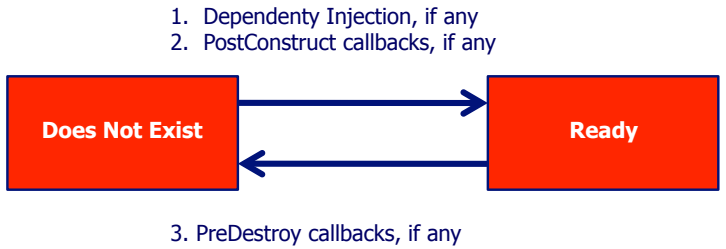
- **@Remove**
  - Indicates that the stateful session bean is to be removed by the container after completion of the method. The EJB container calls the method annotated @PreDestroy, if any.
- EJB Life-cycle method annotations:
  - **@PostConstruct**, **@PreDestroy**, **@PostActivate**, **@PrePassivate**

Timers

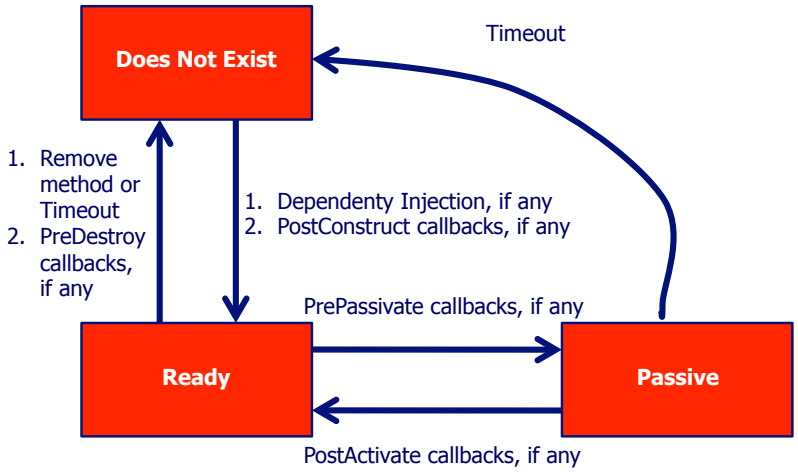
```
@Timeout
public void timeout(Timer timer)
{
   System.out.println("TimerBean: timeout occurred");
}
```

- Timer interface contains
  - void cancel();
  - TimerHandle getHandle();
  - public long getTimeRemaining();
  - public java.util.Date getNextTimeout();
  - public java.io.Serializable getInfo();

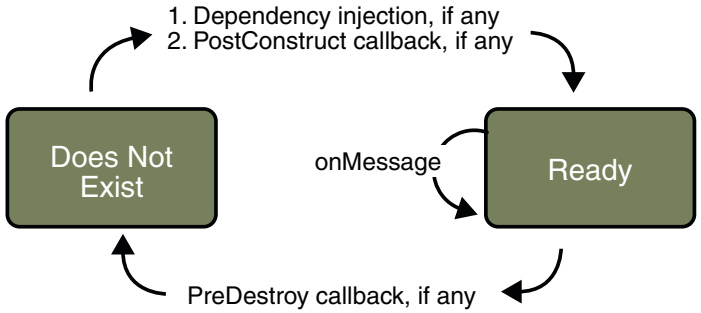
Life Cycle of a Stateless Session Bean



Life Cycle of a Stateful Session Bean



Life Cycle of a Message Driven Bean

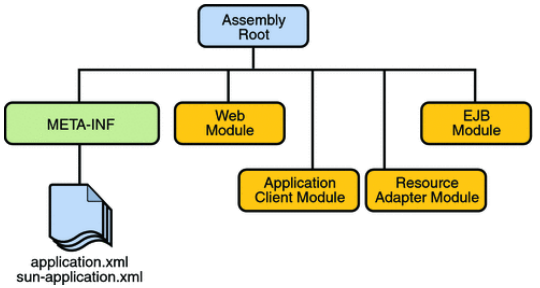


Parameter Passing in EJBs

- Similar to RMI
- Primitive types → By Value
- Composite types → By Value (only if they are serializable)
- EJBs →By reference

Well… but what if we need more? ☺

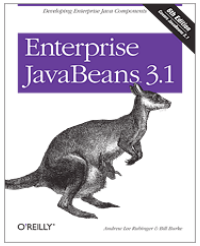
- EJBs + JPA + Web Services + JMS + JSPs + Servlets … **?**
- **How do I manage and deploy all of this?**
- A Java EE application can be delivered in an Enterprise Archive (EAR) file – a .zip file with an .ear extension.



- Let's see it in practice!

Essential reading

- **Enterprise JavaBeans 3.1 6th Edition**
  by Andrew Lee Rubinger and Bill Burke
  O'Reilly, ISBN 0596158025, 2010



- **The Java EE 6 Tutorial**, Oracle,
  - http://docs.oracle.com/javaee/6/tutorial/doc/