EJB 3.0

Agenda

- Introduction
- EJB 3.0
  - Session Beans
  - Message Driven Beans
  - Interceptors
  - Callback Listeners
  - Java Persistence API (JPA)

Enterprise Java Bean - Definition

An enterprise Java bean (EJB) is a server-side component that implements the business logic of an enterprise application and adheres to the rules of the Enterprise JavaBean architecture

Enterprise Java Bean

- An enterprise Java bean (EJB) is a server-side component that implements the business logic of an enterprise application and adheres to the rules of the Enterprise JavaBean architecture

- Enterprise beans live in an EJB container
  - a runtime environment within a J2EE server
  - the EJB container provides multiple services to support the enterprise beans
- Enterprise JavaBeans (EJBs) contain the application's business logic and live business data
- Although it is possible to use standard Java objects to contain your business logic and business data, using EJBs addresses many of the issues you would find by using simple Java objects, such as scalability, lifecycle management, and state management

Benefits of Enterprise Beans

- Enterprise beans simplify the development of large, distributed applications
  - First, because the EJB container provides system-level services to enterprise beans
  - The bean developer can concentrate on solving business problems
  - Second, because the beans--and not the clients--contain the application's business logic, the client developer can focus on the presentation of the client
  - The client developer does not have to code the routines that implement business rules or access databases
    - As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices

When to use EJB

If any of these requirements hold for your application

- **The application must be scalable and distributable**
- **Transactions will be required to ensure data integrity**
- **the application will have a variety of clients**

Types of Enterprise Beans

- **SessionBean**
  - Performs a task for a client
  - optionally may implement a web service
- **Message-Driven**
  - Acts as a listener for a particular messaging type, such as the Java Message Service API

What Is a Session Bean?

- A *session bean* represents a single client inside the Application Server
- To access an application that is deployed on the server, the client invokes the session bean's methods
- The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server
- A session bean is similar to an interactive session. A session bean is not shared
- it can have only one client, in the same way that an interactive session can have only one user
- Like an interactive session, a session bean is not persistent (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client

Types of Session Beans

- Stateless Session Bean
- Stateful Session Bean

Stateful Session Beans

- The state of an object consists of the values of its instance variables.
- In a ***stateful*** session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the ***conversational state***
- The state is retained for the duration of the client-bean session
- If the client removes the bean or terminates, the session ends and the state disappears
- This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state

Stateless Session Beans

- A *stateless* session bean does not maintain a conversational state with the client.
  - When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation.
  - When the method is finished, the state is no longer retained.
  - Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.
- Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients

Stateless Session Beans

- Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.
- Under certain circumstances, the EJB container may write a stateful session bean to secondary storage. However, stateless session beans are never written to secondary storage.
- stateless beans may offer better performance than stateful beans, as the act of retrieving a stateful session bean from secondary storage adds latency to bean activation.
- A stateless session bean can implement a web service, but other types of enterprise beans cannot

When to use Session Beans

- In general, you 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 (perhaps a few hours)
  - The bean implements a web service.

When to use Session Beans

- **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

When to use Session Beans

- Stateless Session beans can be used 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 an email that confirms an online order

What Is a Message-Driven Bean?

- A *message-driven bean* is an enterprise bean that allows Java EE applications to process messages asynchronously
- It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events
- The messages can be sent by any Java EE component
  - an application client, another enterprise bean, or a web component
  - a JMS application or system that does not use Java EE technology
- Message-driven beans can process JMS messages or other kinds of messages

Characteristics of a Message Driven Bean

- 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

Defining Client Access with Interfaces

- **Session Bean Clients**
  - A client can access a session bean only through the methods defined in the bean's business interface
  - The business interface defines the client's view of a bean
  - All other aspects of the bean--method implementations and deployment settings--are hidden from the client

Type of client access

- Type of client access allowed by the enterprise beans
  - Remote
  - Local
  - Web service

Remote Clients

- A remote client of an enterprise bean has the following traits
  - It can run on a different machine and a different Java virtual machine (JVM) than the enterprise bean it accesses (It is not required to run on a different JVM.)
  - It can be a web component, an application client, or another enterprise bean.
  - To a remote client, the location of the enterprise bean is transparent

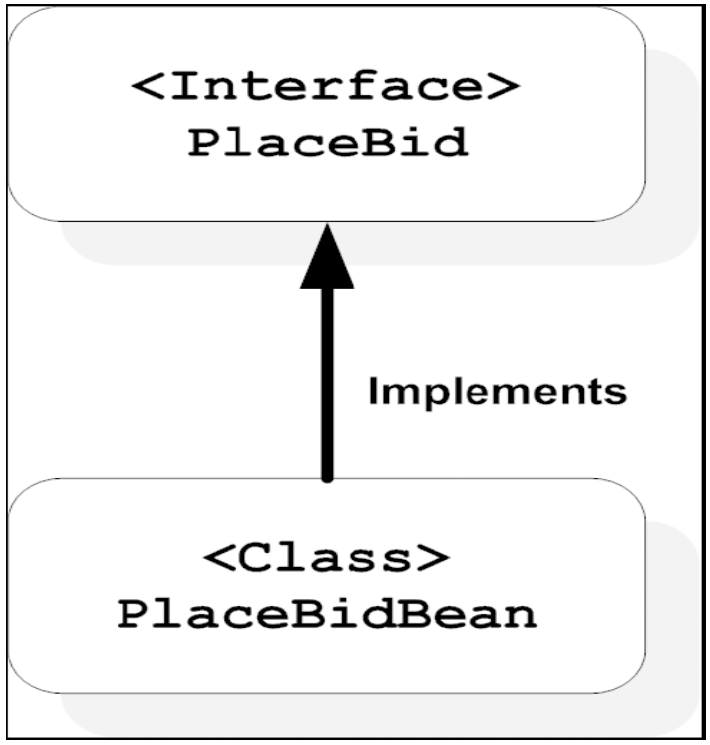
Session Beans (in detail)

Anatomy of a session bean

- Each Session bean contains
  - Interface
  - Implementation class
- All session beans must be divided into these two parts.
- This is because clients cannot have access to the bean implementation class directly.
- Instead, they must use session beans through a business interface.

Session Bean

- Image shows an example of
  - Business interface PlaceBid
  - Session bean PlaceBidBean



Business Interfaces

- An interface through which a client invokes the bean is called a *business interface*.
- This interface contains the methods and the client can access these methods only.

Business Interfaces

- Plain Java language interface
  - EJBObject , EJBHome interface requirements removed
- Either local or remote access
  - Local by default
  - Remote by annotation or deployment descriptor
  - Remote methods not required to throw RemoteExcepion
- Bean class can implement its interface
- Annotations: @Remote ,@Local ,@WebService
- Can specify on bean class or interface

Example Business Interface

```
@Local
 public interface PlaceBid
 {
      Bid addBid(Bid bid);
  }

                    or
@Remote
 public interface PlaceBid
 {
      Bid addBid(Bid bid);
  }
```

Bean Class

```
@Stateless
public class PlaceBidBean implements PlaceBid {
    ...
    public PlaceBidBean() {}
    public Bid addBid(Bid bid) {
            ……
            ……
        return save(bid);
     }
}
```

Bean Class

- Class will implement the interface
- @Stateless annotation indicates that it is a stateless session bean
- In place of @Stateless we can also write @Stateful to create stateful session bean
- We are implementing the methods which are declared in the interface

Callbacks

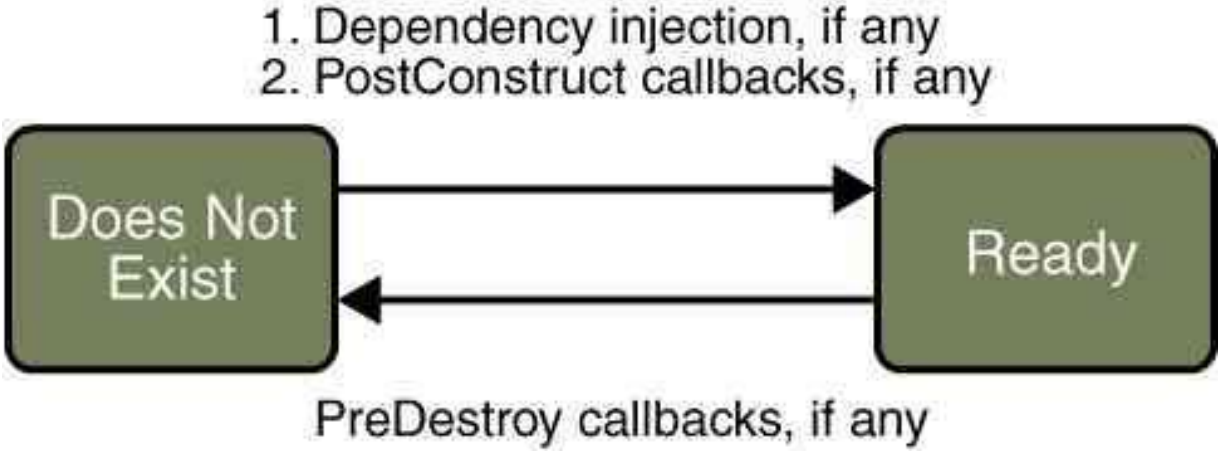
- Callbacks are bean methods that the container calls to notify the bean about a lifecycle transition, or event.
- When the event occurs, the container invokes the corresponding callback method
- You can use these methods to perform operations such as initialization and cleanup of resources .

Stateless SessionBean Callbacks

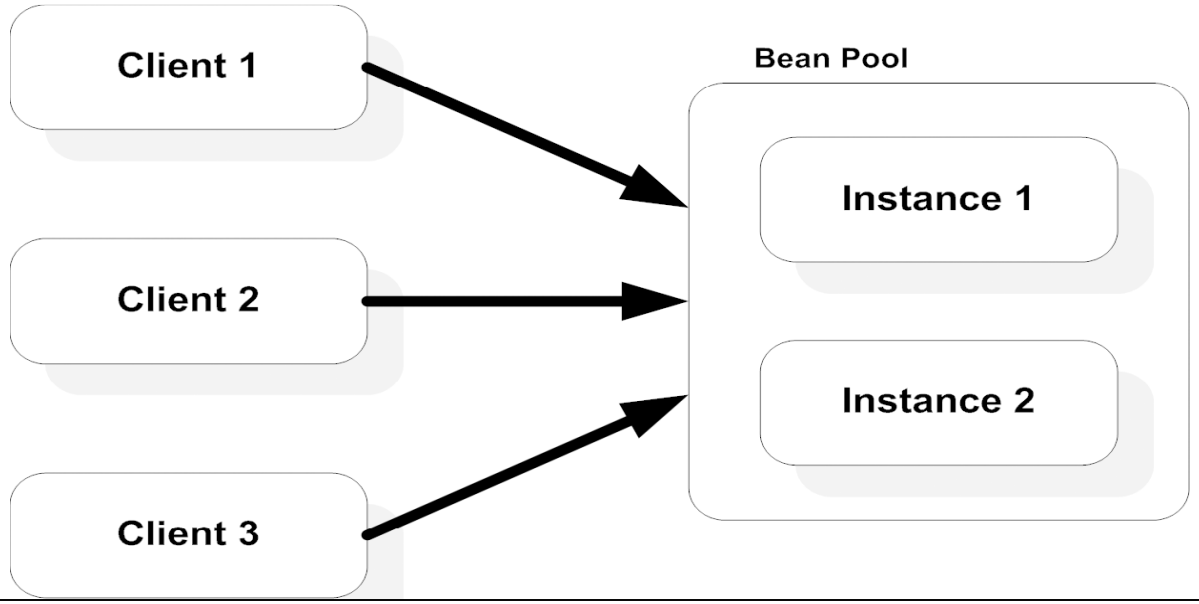
- @PostConstruct : This callback is invoked just after a bean instance is created and dependencies are injected

- @PreDestroycallback :This is invoked just before the bean is destroyed and is helpful for cleaning up resources used by the bean.

Life Cycle of Stateless SessionBean



Stateless bean pool



Stateless Session beans are pooled and are shared by the clients

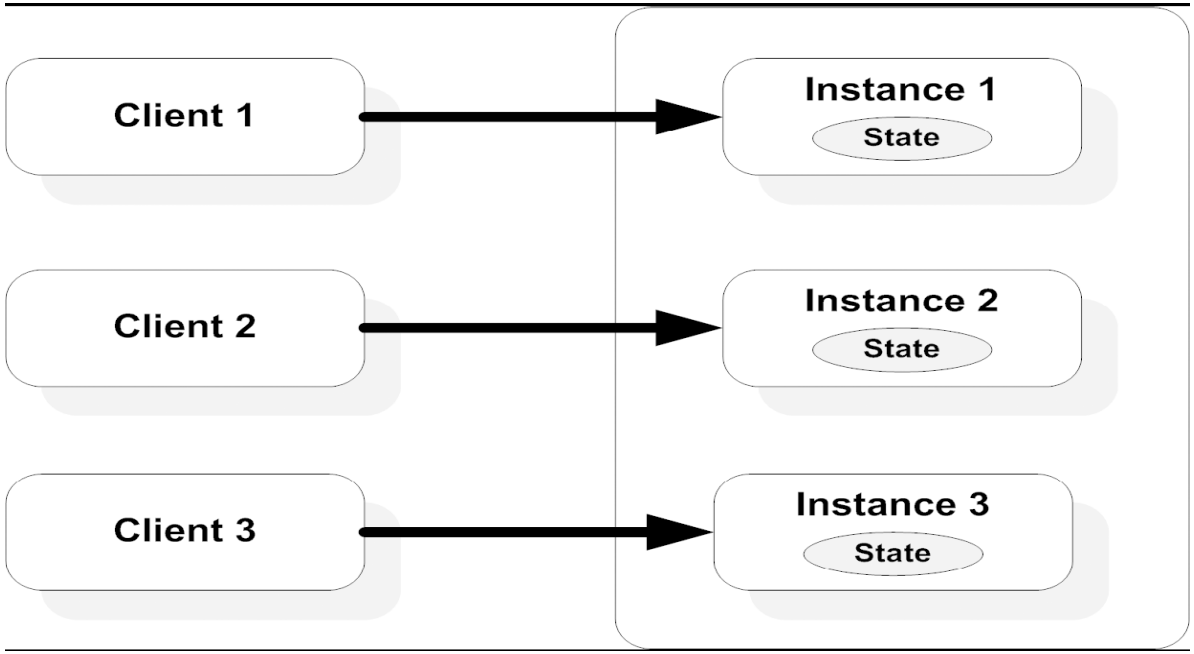
Stateful Session Bean

EJB 3.0

Stateful Session Bean

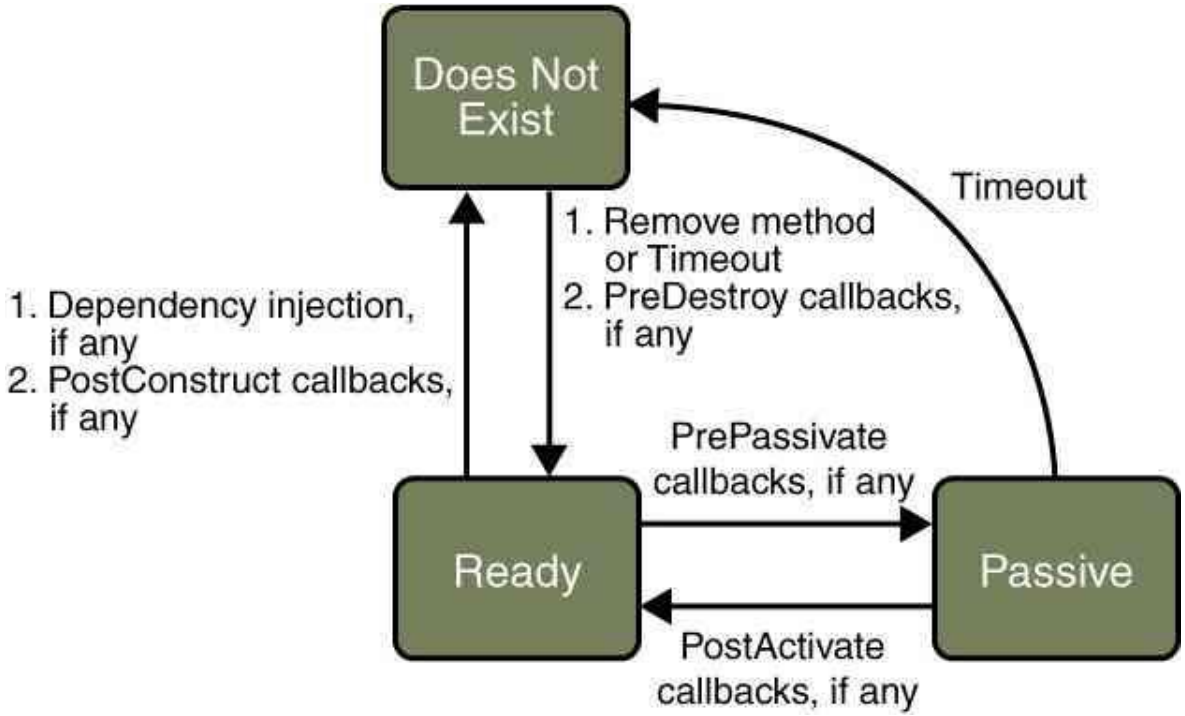
- Stateful session beans are guaranteed to maintain conversational state.
- Subsequent method invocations by the same client are handled by the same stateful bean instance.
- In Stateful session bean instance pool is not maintained.
- Activation & Passivation mechanism is used.

Client & Bean relationship



For every client separate Bean instance is used

LifeCycle of Stateful Session Bean



Passivation & Activation

- If clients don’t invoke a bean for a long enough time, it is not a good idea to continue keeping it in memory.
- The container employs the technique of passivation to save memory when possible

Passivation & Activation

- Passivation essentially means saving a bean instance into disk instead of holding it in memory.
- The container accomplishes this task by serializing the entire bean instance and moving it into permanent storage
- Activation is done when the bean instance is needed again.
- The container activates a bean instance by retrieving it from permanent storage, deserializing it, and moving it back into memory.

Callbacks for Statefull bean

- @PostConstruct
- @PreDestroy
- @PrePassivate
- @PostActivate

@Remove

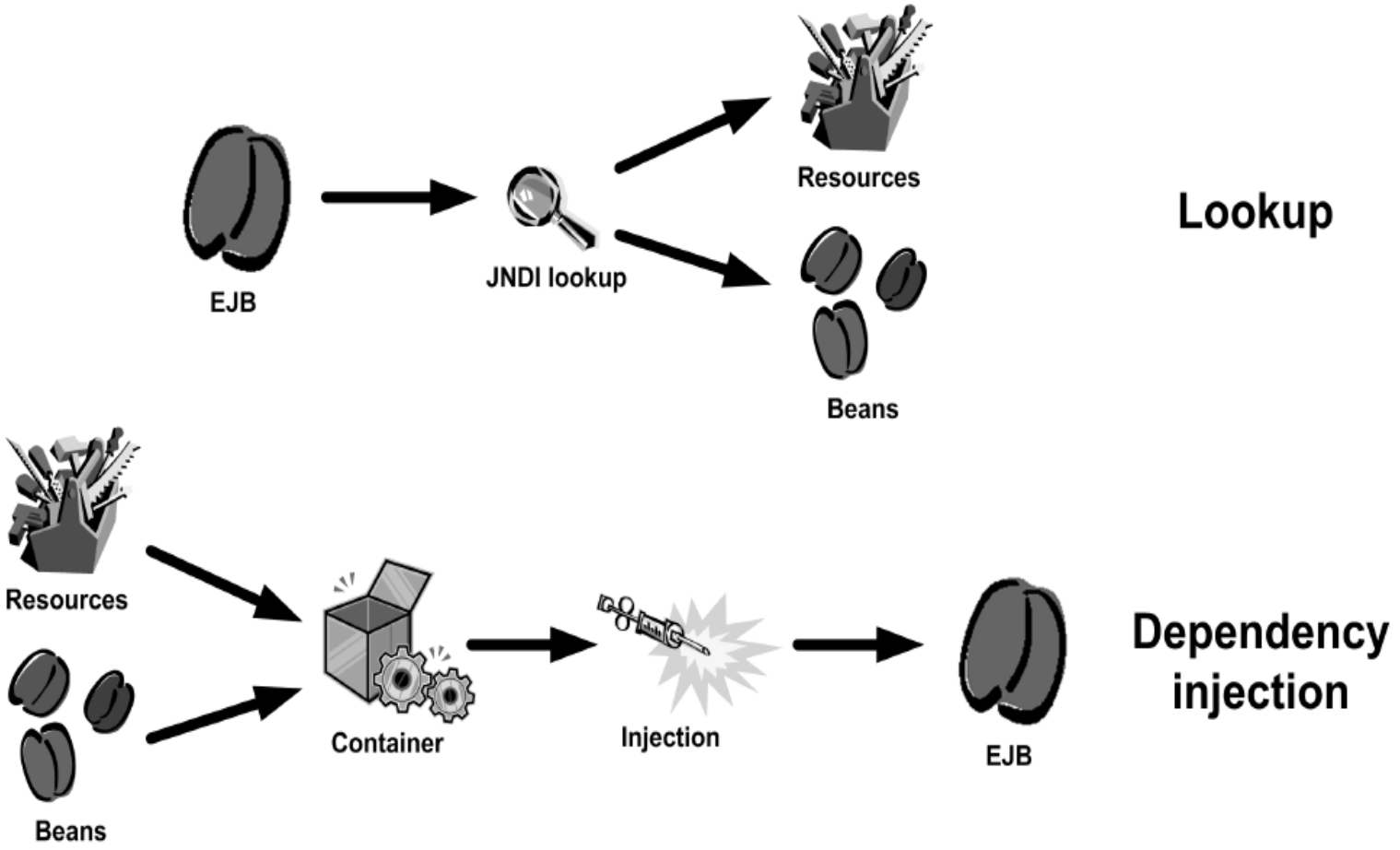
- Since stateful session beans cannot be pooled and reused like stateless beans, there is a real danger of accumulating too many of them
- Therefore, we have to define a business method for removing the bean instance by the client using the @Remove annotation.

Usage of @Remove

```
@Remove
public void cancelAccountCreation()
 {
      loginInfo = null;
      biographicalInfo = null;
       billingInfo = null;
}
```

Dependency Injection

Dependency Injection



Dependency Injection

- A traditional J2EE application uses JNDI as the mechanism that one component uses to access another.
  - For example an EJB uses JNDI to access the resources that it needs, such as a JDBC DataSource.
- Instead we can use DI which gets the ref of the resource with out we writing the code

Dependency Injection

- Bean instance is supplied with references to resources in environment
- Occurs when instance of bean class is created
- No assumptions as to order of injection
- Optional @PostConstruct method is called when injection is complete

DI example

```
@EJB
private HelloUser helloUser;
void hello()
{
    helloUser.sayHello("Curious George");
}
...
```

@EJB will automatically get the HelloUser ejb reference using JNDI and assigns to the helloUser variable and we can directly use the variable and call the methods on it

DI examples

```
@EJB AdminService bean;
public void privilegedTask()
{
bean.adminTask();
}
```

```
@Resource(name=”myDB”)
 public void setDataSource(DataSource myDB)
{
customerDB = myDB;
}
```

Environment Access

- By dependency injection or simple lookup
  - Use of JNDI interfaces no longer needed
- Specify dependencies by annotations or XML
- Annotations applied to:
  - Instance variable or setter property =>injection
  - Bean class =>dynamic lookup

Environment Access Annotations

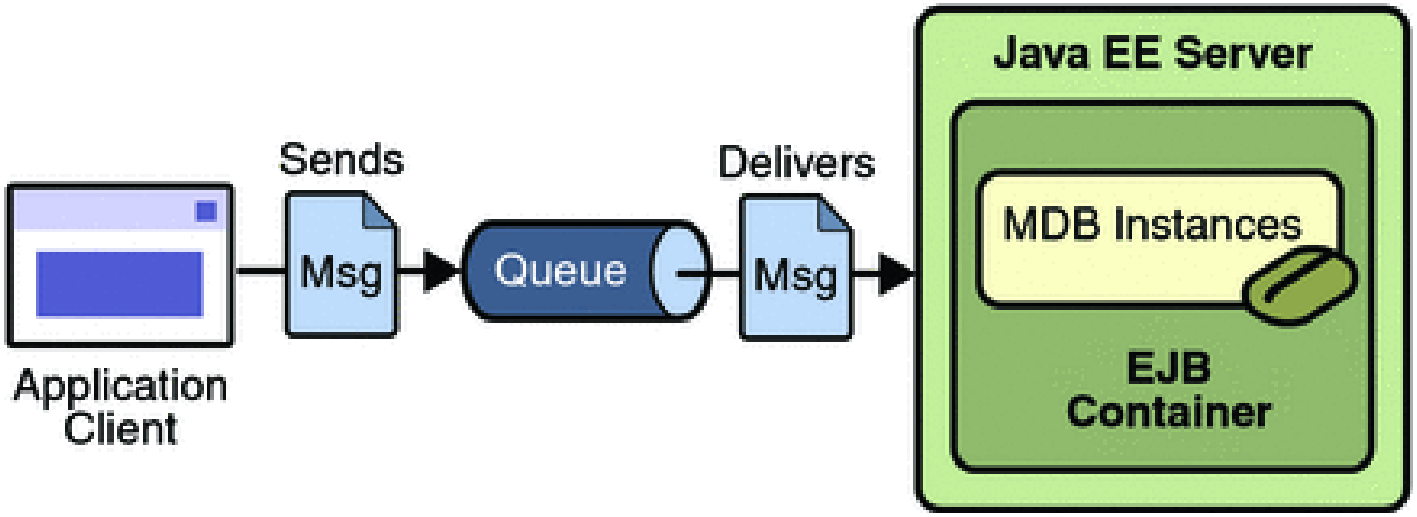
- @Resource
  - For connection factories, simple environment entries, topics/queues, EJBContext ,UserTransaction, etc.
- @EJB
  - For EJB business interfaces or EJB Home interfaces
- @PersistenceContext
  - For container-managed EntityManager
- @PersistenceUnit
  - For EntityManagerFactory

Message Driven Bean (MDB)

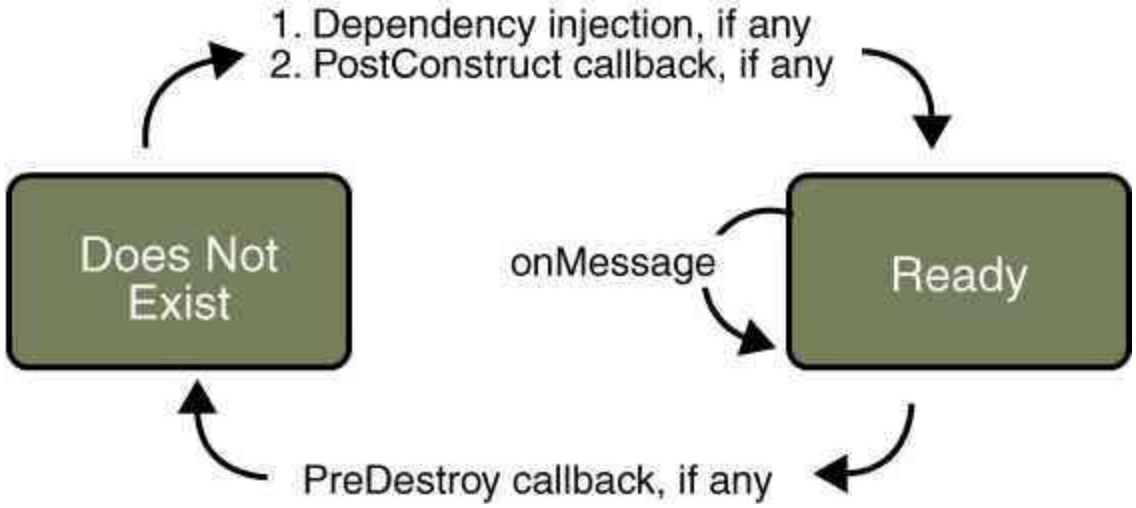
EJB 3.0

Message Driven Bean(MDB)

- MDB is a special EJB that can receive JMS message.
- MDB consumes messages from queue or topic that are sent by any valid JMS client.
- Client cannot access MDB directly
- It is used for asynchronous communications



Life cycle of MDB



MessageListener inerface

- MessageListener interface contains a method onMessage()
- MDB should implement this interface
- Once a message is received, container calls the method onMessage

MDB example

```
@MessageDriven(activationConfig =
        {
        @ActivationConfigProperty(propertyName="destinationType",
              propertyValue="javax.jms.Topic"),
        @ActivationConfigProperty(propertyName="destination",
              propertyValue="topic/mypTopic")
        })
public class PavMdbBean implements MessageListener {
    public void onMessage(Message message) {
    TextMessage tm=(TextMessage)message;
        try {
            System.out.println("message received "+tm.getText());
        } catch (JMSException e) {
            System.out.println(e);       }   }}
```

MDB

- @MessageDriven annotation tells that it is a MDB
- Other resource info like destination name,type are supplied by the @ActivationConfig

Using Container Services

Transactions

Transactions

Transaction Demarcation Types

- Container-managed Transactions
  - Specify declaratively
- Bean-managed transactions
  - UserTransaction API
- Container-managed transaction demarcation is default
- Annotation: @TransactionManagement
  - Values: CONTAINER (default)or BEAN
  - Annotation is applied o bean class (or superclass)

Transactions

- Declaring a container managed transaction is very easy in EJB3
- We just need to add the @TransactionAttribute annotation to the method
- Specify the transaction attribute type.
- There are different transaction atrribute types

Container Managed Transactions

Transaction Attributes

- Annotations are applied o bean class and/or methods of bean class
  - Annotations applied o bean class apply o all methods of bean class unless overridden a me hod-level
  - Annotations applied o method apply o method only
- Annotation: @TransactionAttribute
  - Values: **REQUIRED (default), REQUIRES_NEW, MANDATORY, NEVER,NOT_SUPPORTED, SUPPORTS**

@TransactionAttribute

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public int add(int x, int y)
 {
 return x + y;
}
```

Different Transaction Attributes and their effect

Transaction Attribute	Caller Transaction Exists?	Effect
REQUIRED	No	Container creates a new transaction.
	Yes	Method joins the caller's transaction.
REQUIRES_NEW	No	Container creates a new transaction.
	Yes	Container creates a new transaction and the caller's transaction is suspended.
SUPPORTS	No	No transaction is used.
	Yes	Method joins the caller's transaction.
MANDATORY	No	`javax.ejb.EJBTransactionRequired-Exception` is thrown.
	Yes	Method joins the caller's transaction.
NOT_SUPPORTED	No	No transaction is used.
	Yes	The caller's transaction is suspended and the metho called without a transaction.
NEVER	No	No transaction is used.
	Yes	`javax.ejb.EJBException` is thrown.

Security

Security concepts

- Method permissions
  - Security roles ha are allowed to execute a given se of methods
- Caller principal
  - Security principal under which a method is executed
    - @RunAs for run-as principal
- Runtime security role determination
  - isCallerInRole, getCallerPrincipal
    - **@DeclareRoles**

Method Permissions

- Annotations are applied to bean class and/or methods of bean class
  - Annotations applied o bean class apply o all methods of bean class unless overridden a method-level
  - Annotations applied to method apply to method only
  - No defaults
- Annotations
  - **@RolesAllowed**
    - Value is a list of security role names
  - **@PermitAll**
  - **@DenyAll** (applicable a method-level only)

Security annotations examples

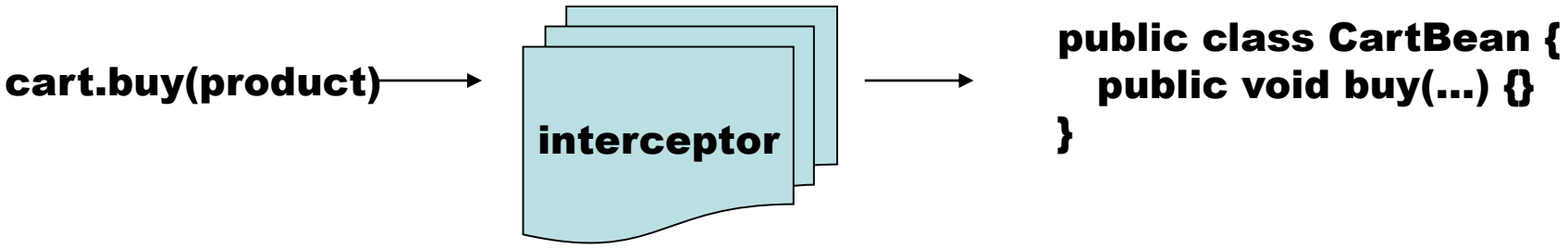
```
public class CalculatorBean implements Calculator
 {
 @PermitAll
public int add(int x, int y)
            { return x + y; }
 @RolesAllowed({"student"})
public int subtract(int x, int y)
        { return x - y; }
 @RolesAllowed({"teacher"})
public int divide(int x, int y)
          {return x / y; }
```

Interceptors

Interceptors

- Ease-of-use facility for more advanced cases
- Container interposes on all business method invocations
- Interceptors interposes after Container
- Invocation model: “around ” methods
  - Wrappered around business method invocations
  - Control invocation of next method (interceptor or business method)
  - Can manipulate arguments and results
  - Context data can be maintained by interceptor chain

Interceptors



- Interceptors intercept calls
- Interceptors sit between caller and a session bean
- Analogous to servlet filters
- Can only be used with session and message driven beans
- Precursor to full aspect-oriented programming

Interceptors

- Using interceptors we can add Cross cutting concerns to our EJB3 code.
- It is one way of working with Aspect Oriented Programming
- Using Interceptors we can define pre and post processing tasks
- By using @AroundInvoke annotation we can declare a method as a interceptor method

Interceptor mehtod

```
@AroundInvoke
 public Object mdbInterceptor(InvocationContext ctx) throws Exception
{
 System.out.println (“this is pre processing task");
        //executed before actual task

 Object o= ctx.proceed();
          //call to actual method or another interceptor

 System.out.println(“this is post processing task”);
           //executed after the actual task

   return(o);
 }
```

External interceptors

- These are the interceptors which are written outside the bean class
- Types of external interceptors
  - Default
  - Class-level
  - Method-level

Default interceptors

- A Default interceptor is an interceptor that is invoked whenever a business method is invoked on any bean within the deployment.
- Default interceptors can only be bound via xml

Default interceptors

```
public class DefaultInterceptor
{
   @AroundInvoke
    public Object intercept(InvocationContext ctx) throws Exception
    {
       System.out.println("DefaultInterceptor " + ctx.getMethod().getName());
       try
        {
            return ctx.proceed();
          }
        finally
       {
            System.out.println("*** DefaultInterceptor exiting");
         }
   }
}
```

ejb-jar.xml (interceptor binding)

```
<assembly-descriptor>
<!-- Default interceptor that will apply to all methods for all beans in deployment -->
 <interceptor-binding>
 <ejb-name>*</ejb-name>
 <interceptor-class>
      com.jp.DefaultInterceptor
</interceptor-class>
 </interceptor-binding>

... </assembly-descriptor>
```

Class-level interceptors

```
@Stateless
@Interceptors ({TracingInterceptor.class})
 public class EmailSystemBean
{
……..
}
```

This says that the @AroundInvoke annotated method of TracingInterceptor should wrap all calls to EmailSystemBean's business methods.

Class level interceptors using xml

```
<assembly-descriptor>
……
……
 <interceptor-binding>
        <ejb-name>  com.jp.EmailSystemBean</ejb-name>
        <interceptor-class>com.jp.OtherInterceptor</interceptor-class>
  </interceptor-binding>
…..
…..
</assembly-descriptor>
```

@AroundInvoke annotated method of OtherInterceptor will wrap all calls to EmailSystemBean's business methods.

Method-level interceptors

- Just as we can define default and class-level interceptors, we can also bind an interceptor to a particular business method within a bean

Method-level interceptors(annotations)

```
@Interceptors({AccountsConfirmInterceptor.class})
 public void sendBookingConfirmationMessage(long orderId)
 {
……
... ……
}
```

sendBookingConfirmationMessage() method has been annotated with the @Interceptors annotation specifying interceptors that will intercept when this perticular method is invoked.

Method level interceptor using xml

```
<interceptors>
 <interceptor>
<interceptor-class>com.jp.AccountsCancelInterceptor</interceptor-class>
 <around-invoke-method>
        <method-name>sendCancelMessage</method-name>
</around-invoke-method>
</interceptor>
... </interceptors>
```

Exceptions

System Exceptions

- In EJB 2.1 specification
  - Remote system exceptions were checked exceptions
    - Subtypes of java.rmi.RemoteException
  - Local system exceptions were unchecked exceptions
    - Subtypes of EJBException

- In EJB 3.0,system exceptions are unchecked
  - Extend EJBException
  - Same set of exceptions independent of whether interface is local or remote
  - ConcurrentAccessException; NoSuchEJBException; EJBTransactionRequiredException; EJBTransactionRolledbackException; EJBAccessException

Exceptions

Application Exceptions

- Business logic exceptions
- Can be checked or unchecked
- Annotation: @ApplicationException
  - Applied to exception class (for unchecked exceptions)
  - Can specify whether Container should mark transaction for rollback
    - Use rollback element
      - @ApplicationException(rollback=true)
    - Defaults to false

Deployment Descriptors

- Available as alternative to annotations
  - Some developers prefer them
- Needed for applica ion-level me ada a
  - Default interceptors
- Can be used to override (some) Annotations
- Useful for deferred configuration
  - Security attributes
- Useful for multiple configurations
  - Java Persistence API O/R mapping
- Can be sparse, full,and/or metadata-complete

Client View

Simplification of client View

- Use of dependency injection
- Simple business interface view
- Removal of need for Home interface
- Removal of need for RemoteExceptions
- Removal of need for handling of other checked exceptions

Example

- **//EJB 3.0:Client View**

```
@EJB payroll payroll;
```

- **//Use the bean**

```
payroll.setTaxDeductions(1234,3);
```

Removal of Home Interface

- Stateless Session Beans
  - Home interface no needed anyway
  - Container creates or reuses bean instance when business method is invoked
    - EJB 2.1 Home.create() method didn’t really create
- Stateful Session Beans
  - Container creates bean instance when business method is invoked
  - Initialization is par of application semantics
    - Don’t need a separate interface for i !
    - Supply init ()method whenever there is a need to support older clients
- •Both support use of legacy home interfaces

summary

- EJB3 is flexible
- EJB3 is less complex when compared with EJB2
- EJB3 uses dependency injection and AOP (using interceptors) which makes programming flexible
- EJB3 uses JPA which allows us to work with relation database with out directly giving access to client

Introduction to Java Persistence API

Java Persistence API

- Part of JSR-220 (Enterprise JavaBeans ™ 3.0)
- Began as simplification of entity beans
  - Evolved into POJO persistence technology
- JPA was defined as part of EJB3.0 as a replacement to EJB 2 CMP Entity beans.
- It is now considered the standard industry approach for ORM
- Scope expanded a request of community to support general use in Java ™ EE and Java SE environments
- Reference implementaion under Project GlassFish
  - Oracle TopLink Essentials

Primary Features

- POJO-based persistence model
  - Simple Java classes —no components
- Support for enriched domain modeling
  - Inheritance, polymorphism,etc.
- Expanded query language
- Standardized object /relational mapping
  - Using Annotations and/or XML
- Usable in Java EE and Java SE environments
- Support for pluggable persistence providers

Entities

- Plain old Java objects
  - Created by means of new
  - No required interfaces
  - Have persistent identity
  - May have both persistent and non-persistent state
    - Simple types (e.g., primitives, wrappers, enums)
    - Composite dependent object types (e.g., Address)
    - Non-persistent state (transient or @Transient)
  - Can extend other entity and non-entity classes
  - Serializable; usable as de ached object s in other tiers
    - No need for data transfer objects

Example

```
@Entity public class Customer implements Serializable {
@Id protected Long id;
 protected String name;
 @Embedded
protected Address address;
 protected PreferredStatus status;
 @Transient
 protected int orderCount;
     public Customer(){}
     public Long getId(){return id;} protected void setId(Long id){this.id =id;}
     public String getName(){return name;}
     public void setName(String name){
     this.name =name;
   }
     …
}
```

Entity Identity

- Every entity has a persistence identity
  - Maps to primary key in database
- Can correspond to simple type
  - Annotations
    - @Id —single field/proper y in en i y class
    - @GeneratedValue —value can be genera ed automatically using various strategies (SEQUENCE,TABLE,IDENTITY,AUTO)
- Can correspond o user-defined class
  - Annotations
    - @EmbeddedId —single field/property in entity class
    - @IdClass —corresponds to multiple Id fields in entity class
- Must be defined on root of entity hierarchy or mapped superclass

Persistence context

- Represent a set of managed entity instances a run time
- Entity instances all belong to same persistence unit ;all mapped to same database
- **Persistence unit is a unit of packaging and deployment**
- EntityManager API is used to manage persistence context ,control lifecycle of entities, find entities by id, create queries

Types of Entity Managers

- Container-managed
  - A typical JTA transaction involves calls across multiple components, which in turn, may access he same persistence context
  - Hence, the persistence context has to be propagated with he JTA transaction to avoid the need for the application to pass references to EntityManager instances from one component to another
- Application-managed
  - Application manages the life time of he EnityManager

Two types of container-managed entity manager

- Transaction scope entity manager
  - Transaction scoped persistence con ex begins when en i y manager is invoked wi hin he scope of a Transaction,and ends when he ransaction is commi ed or rolled-back
  - If en i y manager is invoked ou side a ransac ion,any en i ies loaded from he da abase immediately become de ached a the end of he method call
- Extended scope entity manager
  - The persistence context exists from the time the entity manager instance is created until it is closed
  - Extended scope persistence context could span multiple transactional and non-transactional invocations of he entity manager
  - Extended scope persistence context maintains references to entities after the Transaction has commited i.e. Entities remain managed

Entity Lifecycle

- new
  - New entity instance is created
  - Entity is not yet managed or persistent
- persist
  - Entity becomes managed
  - Entity becomes persistent in database on Transaction commit
- remove
  - Entity is removed
  - Entity is dele ed from database on transaction commit
- refresh
  - Entity ’s state is reloaded from database
- merge
  - State of detached entity is merged back in o managed entity

Entity Relationships

- One-to-one, one-to-many, many-to-many, many-to-one relationships among entities
  - Support for Collection, Set ,List ,Map types
- May be unidirectional or bidirectional
  - Bidirectional relationships are managed by application, not Container
  - Bidirectional relationships have owning side and inverse side
  - Unidirectional relationships only have an owning side
  - Owning side determines he updates to the relationship in the database
    - When to delete related data?

Example

```
@Entity public class Customer {
@Id protected Long id;
 …
@OneToMany protected Set<Order>orders =new HashSet();
 @ManyToOne protected SalesRep rep;
 …
public Set<Order>getOrders(){return orders;}
 public SalesRep getSalesRep(){return rep;}
 public void setSalesRep(SalesRep rep){this.rep =rep;}
 }//
@Entity public class SalesRep {
 @Id protected Long id;
 …@OneToMany(mappedBy=“rep ”)
protected Set<Customer>customers =new HashSet();
 …
public Set<Customer>getCustomers(){return customers;}
 public void addCustomer(Customer customer){ getCustomers().add(customer);
 customer.setSalesRep(this);
 } }
```

Inheritance

- Entities can extend
  - Other entities
    - Either concrete or abstract
  - Mapped superclasses
    - Supply common entity state
  - Ordinary (non-entity) Java classes
    - Supply behavior and/or non-persistent state

Example

MappedSuperclass

```
@MappedSuperclass public class Person {
 @Id protected Long id;
 protected String name;
 @Embedded protected Address address;
 }
@Entity public class Customer extends Person {
 @Transient protected int orderCount;
@OneToMany protected Set<Order>orders =new HashSet();
 }
@Entity public class Employee extends Person {
 @ManyToOne protected Department dept;
 }
```

A mapped superclass cannot be a target of queries, and cannot be passed to methods on EntityManager interface. It cannot be target of persistent relationships.

Example

Abstract Entity

```
@Entity public abstract class Person {
@Id protected Long id;
protected String name;
@Embedded protected Address address;
 }
@Entity public class Customer extends Person { @Transient protected
    int orderCount;
@OneToMany protected Set<Order>orders =new HashSet();
 }
@Entity public class Employee extends Person {
 @ManyToOne protected Department dept;
 }
```

An abstract entity can be a target of queries, and can be passed to methods on EntityManager interface. It cannot be instantiated.

Persist

```
@Stateless public class OrderManagementBean implements
    OrderManagement {
 …
@PersistenceContext EntityManager em;
 …
public Order addNewOrder(Customer customer, Product product){
Order order =new Order(product);
 customer.addOrder(order);
 em.persist(order);
 return order;
} }
```

Should be able to get rid of this When we add an order to customer, an order should automatically be inserted in the underlying orders table.

Cascading Persist

```
@Entity public class Customer {
 @Id protected Long id; …
@OneToMany(cascade=PERSIST)
protected Set<Order>orders =new HashSet();
 }
…
public Order addNewOrder(Customer customer, Product product){
Order order =new Order(product); customer.addOrder(order);
 return order; }
```

Add Order into the underlying tab e at the time of adding Order to the Customer entity's state.

Remove

```
@Entity public class Order {
 @Id protected Long orderId;
 …
@OneToMany(cascade={PERSIST,REMOVE}) protected
    Set<LineItem>ineItems =new HashSet();
 }
…
@PersistenceContext EntityManager em; …
public void de eteOrder(Long orderId){ Order order =em.find(Order.c
    ass,orderId); em.remove(order);
 }
```

Remove a the associated LineItem entities when we remove Order entity.

Merge

```
@Entity public class Order {
 @Id protected Long orderId; …
@OneToMany(cascade={PERSIST,REMOVE,MERGE})
 protected Set<LineItem>ineItems =new HashSet();
 }
…
@PersistenceContext EntityManager em;
 …
public Order updateOrder(Order changedOrder){
 return em.merge(changedOrder);
 }
```

Propagate changes (if any)to LineItem entity upon merging the Order entity.

Queries

Java Persistence Query Language

- An extension of EJB ™ QL
  - Like EJB QL, a SQL-like language
- Added functionality
  - Projection list (SELECT clause)
  - Explicit JOINS
  - Subqueries
  - GROUP BY,HAVING
  - EXISTS,ALL,SOME/ANY
  - UPDATE,DELETE opera ions
  - Additional functions

Projection

```
SELECT e.name, d.name FROM Employee e JOIN
    e.department d WHERE e.status =‘FULLTIME ’

SELECT new com.example.EmployeeInfo(e.id, e.name,e.salary,
    e.status,d.name) FROM Employee e JOIN e.department d
    WHERE e.address.state =‘CA ’
```

Subqueries

```
SELECT DISTINCT emp FROM Employee emp WHERE EXISTS (
SELECT mgr FROM Manager mgr WHERE emp.manager =mgr
AND emp.salary >mgr.salary)
```

Joins

- SELECT DISTINCT o FROM Order o JOIN o.lineItems JOIN .product p WHERE p.productType =‘shoes ’
- SELECT DISTINCT c FROM Customer c LEFT JOIN FETCH c.orders WHERE c.address.city =‘San Francisco ’

Update,Delete

- UPDATE Employee e SET e.salary =e.salary *1.1 WHERE e.department.name =‘Engineering ’
- DELETE FROM Customer c WHERE c.status =‘inactive ’AND c.orders IS EMPTY AND c.balance =0

Queries

- Static queries
  - Defined with Java language metada a or XML
    - Annotations: @NamedQuery, @NamedNativeQuery
- Dynamic queries
  - Query string is specified a run time
- Use Java Persistence query language or SQL
- Named or positional parameters
- EntityManager is factory for Query objects
  - createNamedQuery,createQuery,createNativeQuery
- Query methods for con rolling max results, pagination,flush mode

Dynamic Queries

```
@PersistenceContext EntityManager em;
…
public List findByZipcode(String personType,
  int zip){
 return em.createQuery ( “SELECT p FROM ”
  +personType +“ p WHERE p.address.zip =:zip ”)
  .setParameter(“zipcode ”,zip)
  .setMaxResults(20));
}
```

Static Query

```
@NamedQuery(name="customerFindByZipcode ", query = "SELECT c
    FROM Customer c WHERE c.address.zipcode =:zip ")
@Entity public class Customer {…}
…
public List findCustomerByZipcode(int zipcode){
 return em.createNamedQuery ("customerFindByZipcode ")
    .setParameter("zip ",zipcode).setMaxResu ts(20).getResutList();
 }
```

Object/Relational Mapping

Object/Relational Mapping

- Map persistent object state to relational database
- Map relationships to other entities
- Mapping metadata may be Annotations or XML (or bo h)
- Annotations
  - Logical —object model (e.g., @OneToMany, @Id, @Transient)
  - Physical —DB tables and columns (e.g., @Table, @Column)
- XML
  - Elements for mapping entities and heir fields or properties
  - Can specify metadata for different scopes
- Rules for defaulting of database able and column names

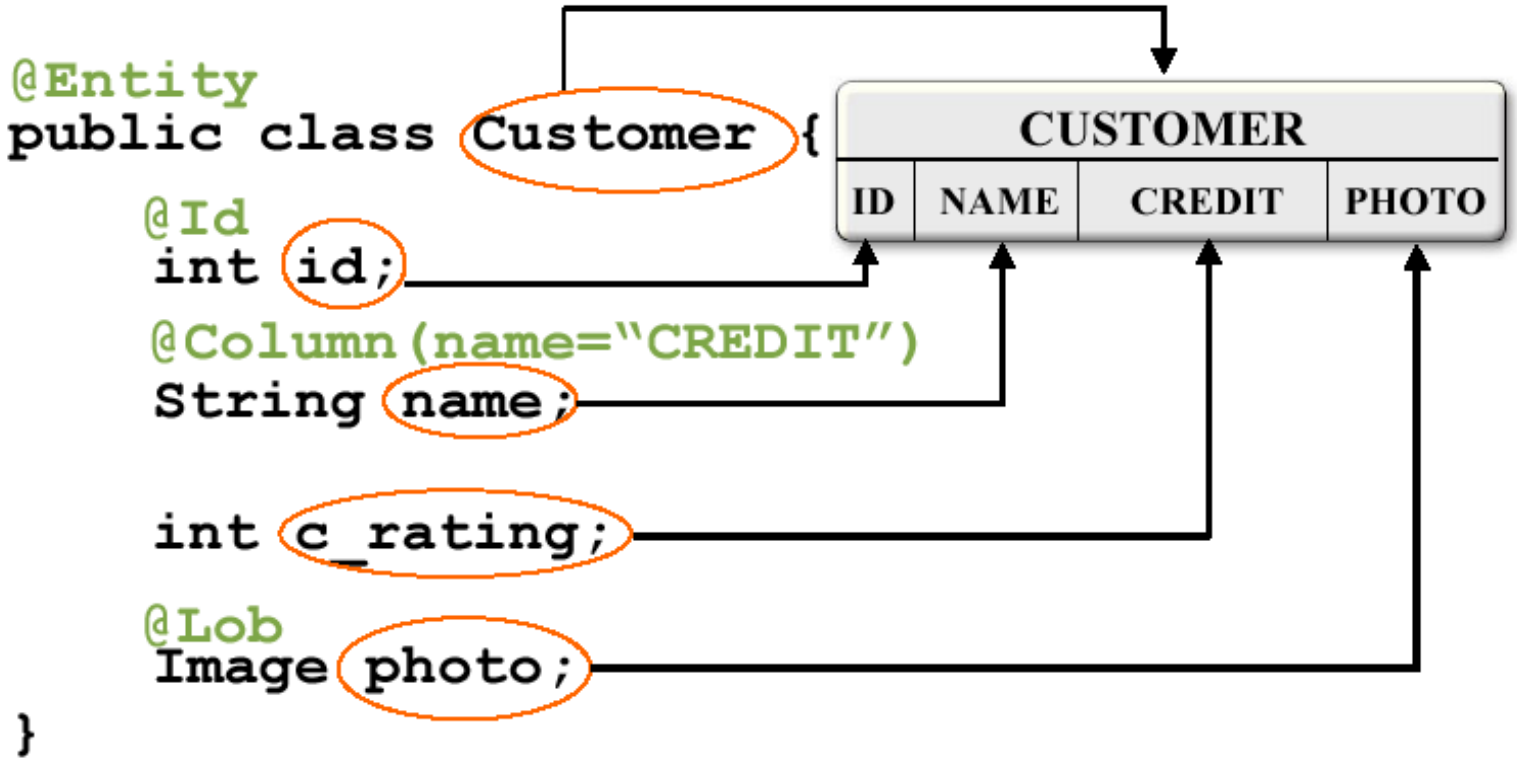
Object/Relational Mapping

- State or relationships may be loaded or “fetched ” as EAGER or LAZY
  - LAZY is a thin to the Container to defer loading until the field or proper y is accessed
- EAGER requires that the field or relationship be loaded when the referencing entity is loaded
- Cascading of entity operations to related entities
  - Setting may be defined per relationship
  - Configurable globally in mapping file for persistence-by-reachability

Simple Mappings

- Direct mappings of fields/properties to columns
  - @Basic —op ional Annotation to indicate simple mapped attribute
- Maps any of the common simple Java types
- Primitives, wrapper types, Date,Serializable,byte[ ],...
- Used in conjunction with @Column
- Defaults to the type deemed most appropriate if no mapping Annotation is present
- Can override any of he defaults

Simple Mappings

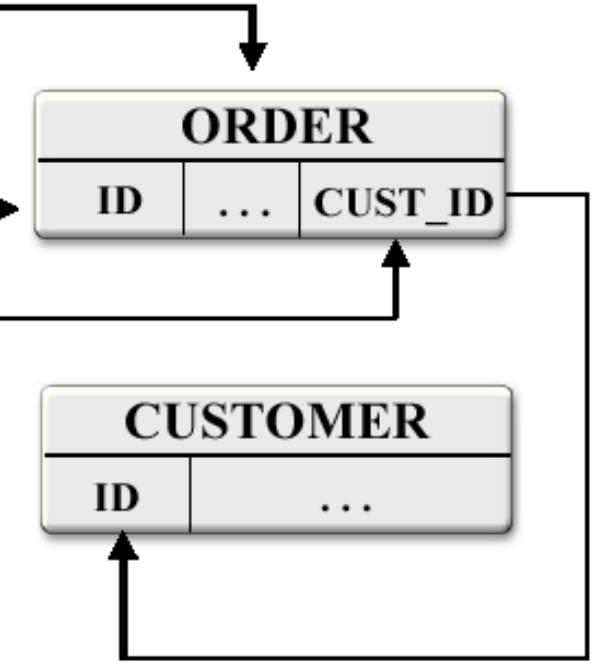


Relationship Mappings

- Common relationship mappings suppor ed
  - @ManyToOne, @OneToOne —single entity
  - @OneToMany, @ManyToMany —collection of entities
- Unidirectional or bidirectional
- Owning and inverse sides of every bidirectional relationship
- Owning side specifies he physical mapping
  - @JoinColumn to specify foreign key column
  - @JoinTable decouples physical relationship mappings from entity tables

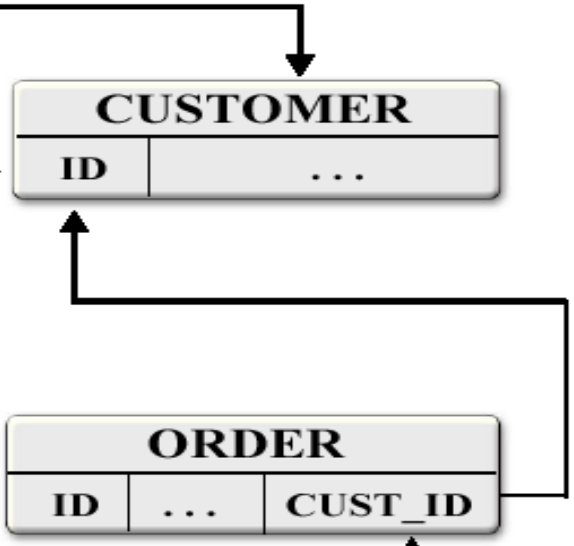
Many-to-One Mapping

```
@Entity
public class Order {

    @Id
    int id;

    ...

    @ManyToOne
    Customer cust;
}
```



One-to-Many Mapping

```
@Entity
public class Customer
{
   @Id
   int id;
   ...
   @OneToMany
   Set<Order> orders;
}

@Entity
public class Order {
   @Id
   int id;
   ...
   @OneToMany(mappedBy="orders")
   Customer cust;
}
```



Many-to-Many Mapping

```
@Entity
public class Customer
{
  @Id
   int id;
   ...
  @ManyToMany
   Collection<Phone>
phones;
}
```

```
@Entity
public class Phone {
   @Id
   int id;
   ...
   @ManyToMany(mappedBy="
   phones")
   Collection<Customer>
custs;
}
```

Many-to-Many Mapping

```
@Entity
public class Customer {
   ...
   @ManyToMany
   @JoinTable(table="CUST_PHONE"),
      joinColumns=@JoinColumn(name="CUST_ID"),
      inverseJoinColumns=@JoinColumn(name="PHONE_ID"))
   Collection<Phone> phones;
}
```

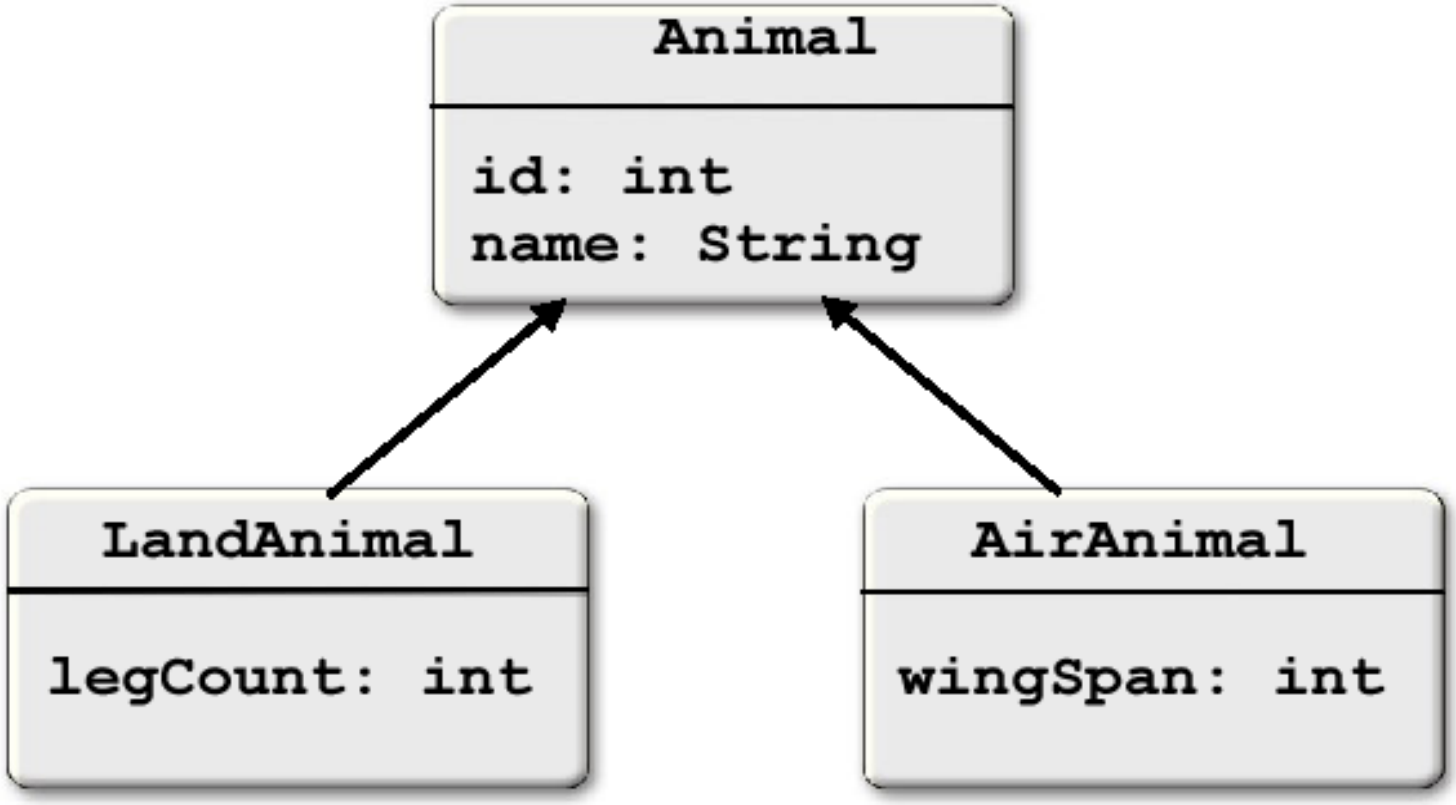
Embedded Objects

```
@Entity
public class Customer
{
   @Id
   int id;
   @Embedded
   CustomerInfo info;
}

@Embeddable
public class CustomerInfo {
   String name;
   int credit;
   @Lob
   Image photo;
}
```

Inheritance

- Entities can extend
  - Other entities – concrete or abstract
  - Non-entity classes – concrete or abstract
- Map inheritance hierarchies in three ways
  - SINGLE_TABLE
  - JOINED
  - TABLE_PER_CLASS

Object Model



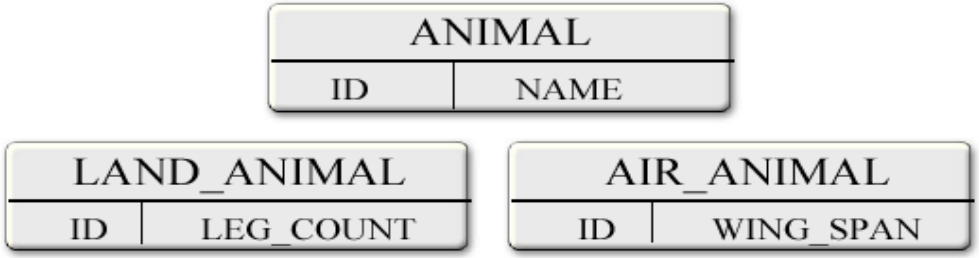
Data Models

Good polymorphic support; Requires columns corresponding to state specific to subclasses to be nullable.

Single table:

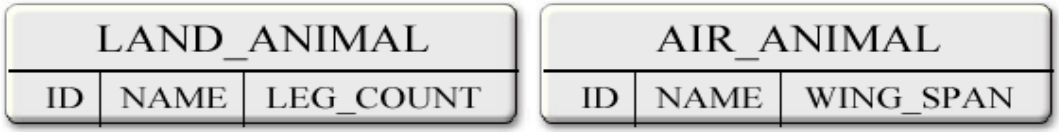
Decent polymorphic support; Requires JOIN to be performed for queries ranging over class hierarchies. Could perform badly in deep hierarchies.

Joined:



Poor support for polymorphic queries; Requires UNION queries for queries that range over class hierarchy.

Table per Class:



Persistence in Java SE

- No deployment phase
  - Application must use a “Boot strap API ” to obtain an EntityManagerFactory
- Typically use resource-local EntityManagers
  - Application uses a local EntityTransaction obtained from the EntityManager
- New persistence context for each and every EntityManager that is created
- No propagation of persistence contexts

Entity Transactions

- Resource-level ransac ion akin o a JDBC ransac ion
  - Isolated from Transactions in other EntityManagers
- Transaction demarcation under explicit application control using EntityTransaction API
- begin(), commit(), setRollbackOnly(), rollback(), isActive()
- Underlying (JDBC ™)resources allocated by EntityManager as required

Bootstrap classes

javax.persistence.Persistence

- Root class for bootstrapping an EntityManager
- Locates a provider service for a named persistence unit
- Invocations on the provider to obtain an EntityManagerFactory

javax.persistence.EntityManagerFactory

- creates EntityManagers for a named persistence unit or configuration

Example

```
public class SalaryChanger {
 public static void main(String [] args){
 EntityManagerFactory emf =
Persistence.createEntityManagerFactory(“HRSyste
  m ”); EntityManager
  em=emf.createEntityManager();
  em.getTransaction().begin();
 Employee emp =em.find( Employee.class.new
  Integer(args [0 ]));
 emp.setSa ary(new Integer(args [1 ]));
  em.getTransaction().commit();
 em.close();
 emf.close(); } }
```

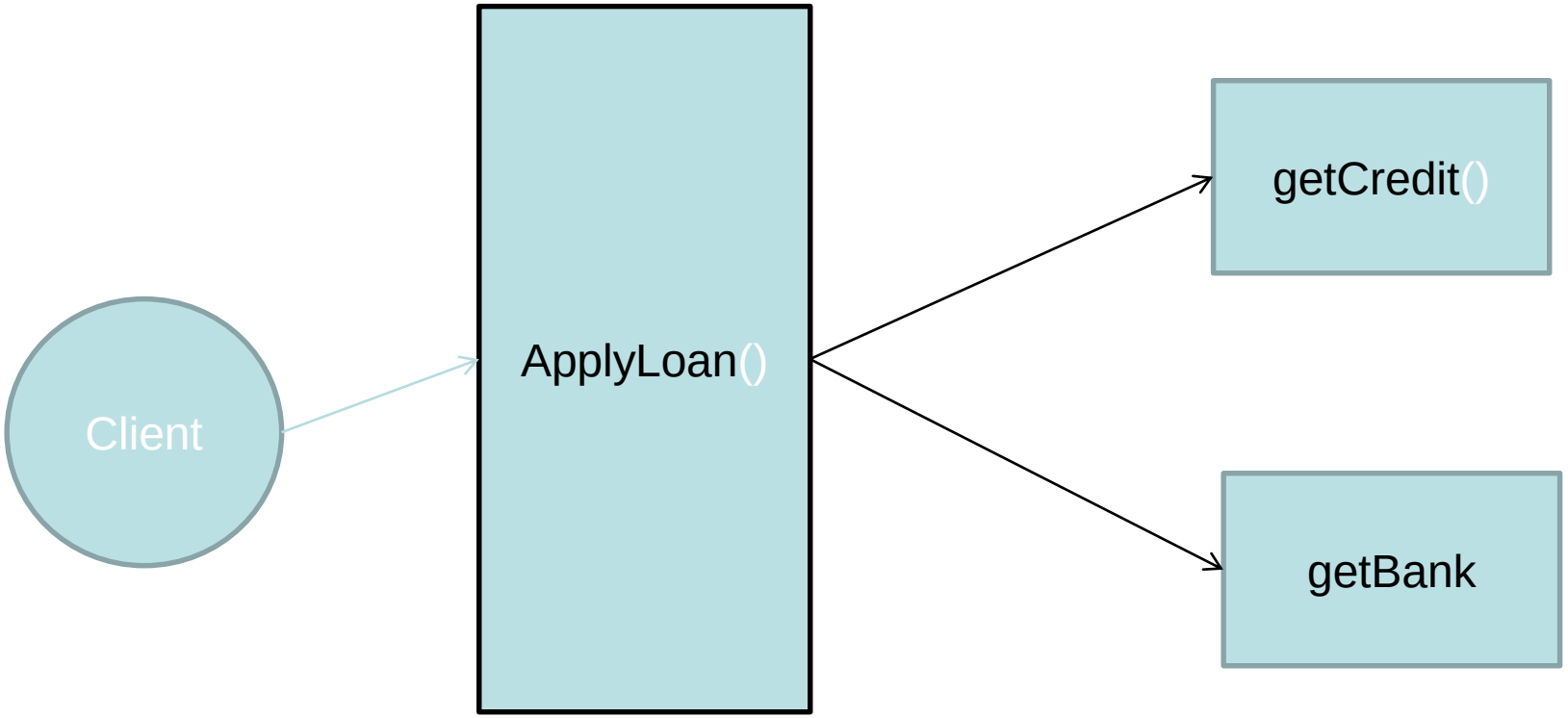
Summary and Resourses

Summary of EJB 3.0

- Major simplification of EJB technology for developers
  - Beans are plain Java classes wi h plain Java interfaces
  - APIs refocused on ease of use for developer
  - Easy access to Container services and environment
  - Deployment descriptors available,but generally unneeded
- EJB 3.0 components interoperate with existing components/applications
- Gives developer powerful *and* easy-o-use functionality

Summary of JPA

- Entities are simple Java classes
  - Easy o develop and intuitive o use
  - Can be moved to other server and client tiers
- EntityManager
  - Simple API for operating on entities
  - Supports use inside and out side Java EE Containers
- Standardization
  - O/R mapping using annotations or XML
  - Named and dynamic query definition
  - SPI for pluggable persistence providers


Client
ApplyLoan()
getCredit()
getBank