EJB 3.0

Objective

Learn how to use the new EJB™ 3.0 API and Java Persistence API to simplify your enterprise applications

EJB 3.0 Approach

EJB Container

- Managed environment for the execution of components
- Container interposes to provide services
- Container-provided services
  - Concurrency
  - Transactions
  - Environment
  - Security
  - > Distribution
  - > EIS integration
  - > Resource pooling
  - > Persistence

Example

```
// EJB 2.1 Stateless Session Bean: Bean Class

public class PayrollBean
             implements javax.ejb.SessionBean {
    SessionContext ctx;
    DataSource payrollDB;

    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
```

Example

```
// EJB 2.1 Stateless Session Bean: Bean Class (continued)

    public void ejbCreate() {
        ...
        Context initialCtx = new InitialContext();
        payrollDB = (DataSource)initialCtx.lookup
    ("java:com/env/jdbc/empDB");
        ...
    }

    public void setTaxDeductions(int empId,int deductions)
    {
        ...
        Connection conn = payrollDB.getConnection();
        Statement stmt = conn.createStatement();
        ...
    }
}
```

Example

```
// EJB 2.1 Stateless Session Bean: Interfaces

public interface PayrollHome
                extends javax.ejb.EJBLocalHome {

    public Payroll create() throws CreateException;
}

public interface Payroll
                extends javax.ejb.EJBLocalObject {

    public void setTaxDeductions(int empID, int
deductions);

}
```

Example

```
// EJB 2.1 Stateless Session Bean: Deployment Descriptor

<session>
    <ejb-name>PayrollBean</ejb-name>
    <local-home>com.example.PayrollHome</local-home>
    <local>com.example.Payroll</local>
    <ejb-class>com.example.PayrollBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <resource-ref>
        <res-ref-name>jdbc/empDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</session>
```

Example

```
// Deployment Descriptor(continued)
<assembly-descriptor>
    <method-permission>
        <unchecked/>
        <method>
                <ejb-name>PayrollBean</ejb-name>
                <method-name>*</method-name>
        </method>
    </method-permission>
    <container-transaction>
        <method>
                <ejb-name>PayrollBean</ejb-name>
                <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

EJB 3.0 Goal

FIX THIS!

How?
EJB 3.0 Approach

- More work is done by container, less by developer
- Inversion of contractual view
- Contracts benefit developer rather than container
  - Bean specifies what it needs through metadata
  - No longer written to unneeded container interfaces
  - Deployment descriptor no longer required
  - Configuration by exception
- Container provides requested services to bean

Compatibility Constraints

- Existing EJB applications work unchanged
- New EJB components interoperate with existing EJB components
- Components can be updated or replaced without change to existing clients
- Clients can be updated without change to existing components
- Compatibility with other Java EE 5 APIs

Simplified Bean Classes

Bean Classes

- In EJB 3.0, session beans, message-driven beans are ordinary Java classes
  - Container interface requirements removed
  - Bean type specified by annotation or XML
- Annotations
  - @Stateless, @Stateful, @MessageDriven
  - Specified on bean class
- EJB 2.x entity beans are unchanged
  - Java™ Persistence API entities provide new functionality
  - @Entity applies to Java Persistence API entities only

Example

```
// EJB 2.1 Stateless Session Bean: Bean Class

public class PayrollBean
                implements javax.ejb.SessionBean {
    SessionContext ctx;
    public void setSessionContext(SessionContext ctx) {
        this.ctx = ctx;
    }
    public void ejbCreate() {...}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public void setTaxDeductions(int empId, int
    deductions) {
        ...
    }
}
```

Example

```
// EJB 3.0 Stateless Session Bean: Bean Class

@Stateless public class PayrollBean implements
Payroll {

    public void setTaxDeductions(int empId,int
    deductions) {
        ...
    }
}
```

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 RemoteException
- Bean class can implement its interface
- Annotations: @Remote, @Local, @WebService
  - Can specify on bean class or interface

Example

```
// EJB 2.1 Stateless Session Bean: Interfaces

public interface PayrollHome
             extends javax.ejb.EJBLocalHome {

   public Payroll create() throws CreateException;
}

public interface Payroll
             extends javax.ejb.EJBLocalObject {

   public void setTaxDeductions(int empId, int
deductions);

}
```

Example

```
// EJB 3.0 Stateless Session Bean: Business
Interface

public interface Payroll {

   public void setTaxDeductions(int empId, int
deductions);

}
```

Example

```
// EJB 3.0 Stateless Session Bean: Remote Interface

@Remote public interface Payroll {

   public void setTaxDeductions(int empId, int
deductions);

}
```

Example

```
// EJB 3.0 Stateless Session Bean:
// Alternative: Remote Interface specified on bean
class

@Stateless @Remote public class PayrollBean
      implements Payroll {

   public void setTaxDeductions(int empId,int
   deductions) {
      ...
   }
}
```

Message-driven Beans

- Message listener interface is business interface
  - Bean class implements it or designates with @MessageListener
- No requirement to implement other interfaces
- Annotations
  - @MessageDriven

Example

```
// EJB 3.0 Message-driven bean: Bean Class

@MessageDriven public class PayrollMDB
      implements javax.jms.MessageListener {

   public void onMessage(Message msg) {
      ...
   }
}
```

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

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

Example

```
// EJB 3.0 Stateless Session Bean: Bean Class
// Data access using injection and Java Persistence API

@Stateless public class PayrollBean implements Payroll {

    @PersistenceContext EntityManager payrollMgr;

    public void setTaxDeductions(int empId,int deductions)
    {
       payrollMgr.find(Employee.class,
    empId).setTaxDeductions(deductions);
    }
}
```

Dynamic Environment Lookup

- Use EJBContext lookup method
- Dependencies declared using annotations on bean class

Example

```
// EJB 3.0 Stateless Session Bean
// Using dynamic lookup

@PersistenceContext(name="payrollMgr")
@Stateless public class PayrollBean implements Payroll {

    @Resource SessionContext ctx;

    public void setTaxDeductions(int empId,int deductions)
    {
      EntityManager payrollMgr = ctx.lookup("payrollMgr");
      payrollMgr.find(Employee.class,
    empId).setDeductions(deductions);
    }
}
```

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 2.1: Client View

...
Context initialContext = new InitialContext();
PayrollHome payrollHome = (PayrollHome)
initialContext.lookup("java:comp/env/ejb/payroll");

Payroll payroll = payrollHome.create();
...


// Use the bean

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

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 not 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 part of application semantics
    - Don't need a separate interface for it!
    - Supply init() method whenever there is a need to support older clients
- Both support use of legacy home interfaces

Using Container Services

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 to bean class (or superclass)

Container Managed Transactions
Transaction Attributes

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

Example

```
// EJB 3.0: Container-managed transactions

@Stateless public class PayrollBean implements
Payroll {

    @TransactionAttribute(MANDATORY)
    public void setTaxDeductions(int empId,int
    deductions) {
        ...
    }

    public int getTaxDeductions(int empId)
    {
        ...
    }
}
```

Example

```
// EJB 3.0: Container-managed transactions

@TransactionAttribute(MANDATORY)
@Stateless public class PayrollBean implements
Payroll {

   public void setTaxDeductions(int empId,int
   deductions) {
      ...
   }

   @TransactionAttribute(REQUIRED)
   public int getTaxDeductions(int empId)
   {
      ...
   }
}
```

Example

```
// EJB 3.0: Bean-managed transactions

@TransactionManagement(BEAN)
@Stateless public class PayrollBean implements Payroll {

    @Resource UserTransaction utx;
    @PersistenceContext EntityManager payrollMgr;

    public void setTaxDeductions(int empId, int deductions)
    {
        utx.begin();
        payrollMgr.find(Employee.class,
    empId).setDeductions(deductions);
        utx.commit();
    }
...
}
```

Security Concepts

- Method permissions
  - Security roles that are allowed to execute a given set 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 to bean class apply to all methods of bean class unless overridden at 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 at method-level only)

Example

```
// EJB 3.0: Security View

@RolesAllowed(HR_Manager)
@Stateless public class PayrollBean implements
Payroll {

   public void setSalary(int empId, double salary) {
      ...
   }

   @RolesAllowed({HR_Manager, HR_Admin})
   public int getSalary(int empId)
   {
      ...
   }
}
```

Event Notification

Bean Lifecycle Events

- EJB 2.1 specification required EnterpriseBean interfaces
- EJB 3.0 specification: only specify events you need
- Annotations:
  - @PostConstruct
  - @PreDestroy
  - @PostActivate
  - @PrePassivate
- Annotations applied to method of bean class or method of interceptor class
- Same method can serve for multiple events

Example

```
// EJB 3.0: Event Notification

@Stateful public class TravelBookingBean
    implements TravelBooking {

    @PostConstruct
    @PostActivate
    private void connectToBookingSystem() {...}

    @PreDestroy
    @PrePassivate
    private void disconnectFromBookingSystem() {...}

    ...

}
```

Interceptors

- Ease-of-use facility for more advanced cases
- Container interposes on all business method invocations
- Interceptors interpose 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

- Default Interceptors
  - Apply to all business methods of components in ejb-jar
  - Specified in deployment descriptor
    - Due to lack of application-level metadata annotations
- Class-level interceptors
  - Apply to business methods of bean class
- Method-level interceptors
  - Apply to specific business method
- Very flexible customization
  - Ability to exclude interceptors, reorder interceptors for class or method

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 application-level metadata
  - 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

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
- Scope expanded at request of community to support general use in Java™ EE and Java SE environments
- Reference implementation under Project GlassFish
  - Oracle TopLink Essentials

Primary Features

- POJO-based persistence model
  - Simple Java classes—not components
- Support for enriched domain modelling
  - 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 detached objects 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/property in entity class
    - @GeneratedValue—value can be generated automatically using various strategies (SEQUENCE, TABLE, IDENTITY, AUTO)
- Can correspond to 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 at runtime
- 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 the same persistence context
  - Hence, the persistence context has to be propagated with the 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 the EntityManager

Two types of container-managed entity manager

- Transaction scope entity manager
  - Transaction scoped persistence context begins when entity manager is invoked within the scope of a transaction, and ends when the transaction is committed or rolled-back
  - If entity manager is invoked outside a transaction, any entities loaded from the database immediately become detached at the end of the 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 the entity manager
  - Extended scope persistence context maintains references to entities after the transaction has committed 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 deleted from database on transaction commit
- refresh
  - Entity's state is reloaded from database
- merge
  - State of detached entity is merged back into 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 the 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 table 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> lineItems = new
HashSet();
}
```

```
…
@PersistenceContext EntityManager em;
…
public void deleteOrder(Long orderId) {
   Order order = em.find(Order.class, orderId);
   em.remove(order);
}
```

Remove all 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> lineItems = 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 operations
  - 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 l JOIN
l.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 metadata or XML
    - Annotations: @NamedQuery, @NamedNativeQuery
- Dynamic queries
  - Query string is specified at runtime
- Use Java Persistence query language or SQL
- Named or positional parameters
- EntityManager is factory for Query objects
  - createNamedQuery, createQuery, createNativeQuery
- Query methods for controlling 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 = :zipcode")
   .setParameter("zipcode", zip)
   .setMaxResults(20)
   .getResultList();
}
```

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)
   .setMaxResults(20)
   .getResultList();
}
```

…

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 both)
- Annotations
  - Logical—object model (e.g., @OneToMany, @Id, @Transient)
  - Physical—DB tables and columns (e.g., @Table, @Column)
- XML
  - Elements for mapping entities and their fields or properties
  - Can specify metadata for different scopes
- Rules for defaulting of database table and column names

Object/Relational Mapping

- State or relationships may be loaded or "fetched" as EAGER or LAZY
  - LAZY is a hint to the Container to defer loading until the field or property 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—optional 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 the defaults

Simple Mappings

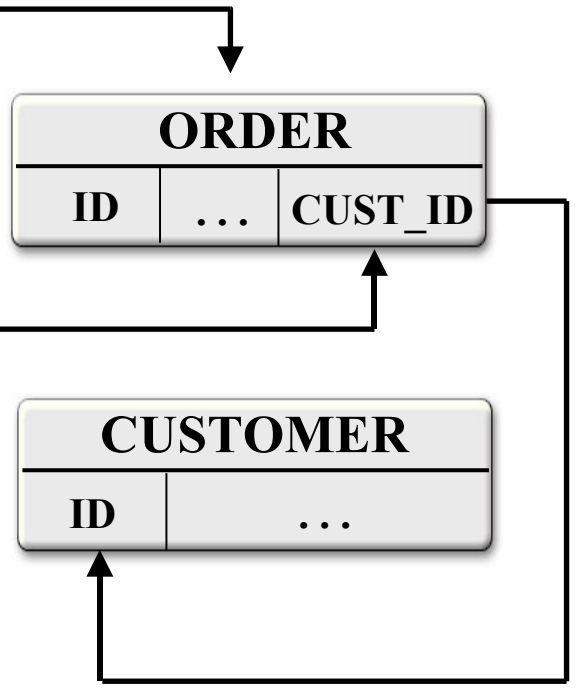
```
@Entity
public class Customer {
    @Id
    int id;
    @Column(name="CREDIT")
    String name;

    int c_rating;

    @Lob
    Image photo;
}
```

Relationship Mappings

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

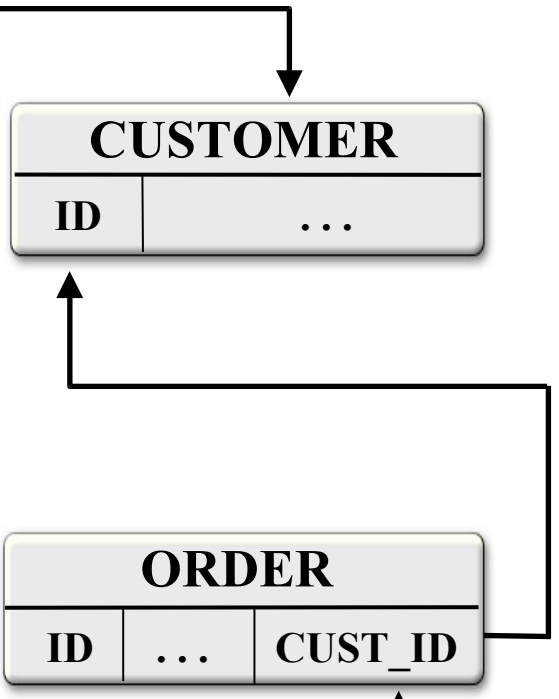
Many-to-One Mapping

```
@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;
}
```

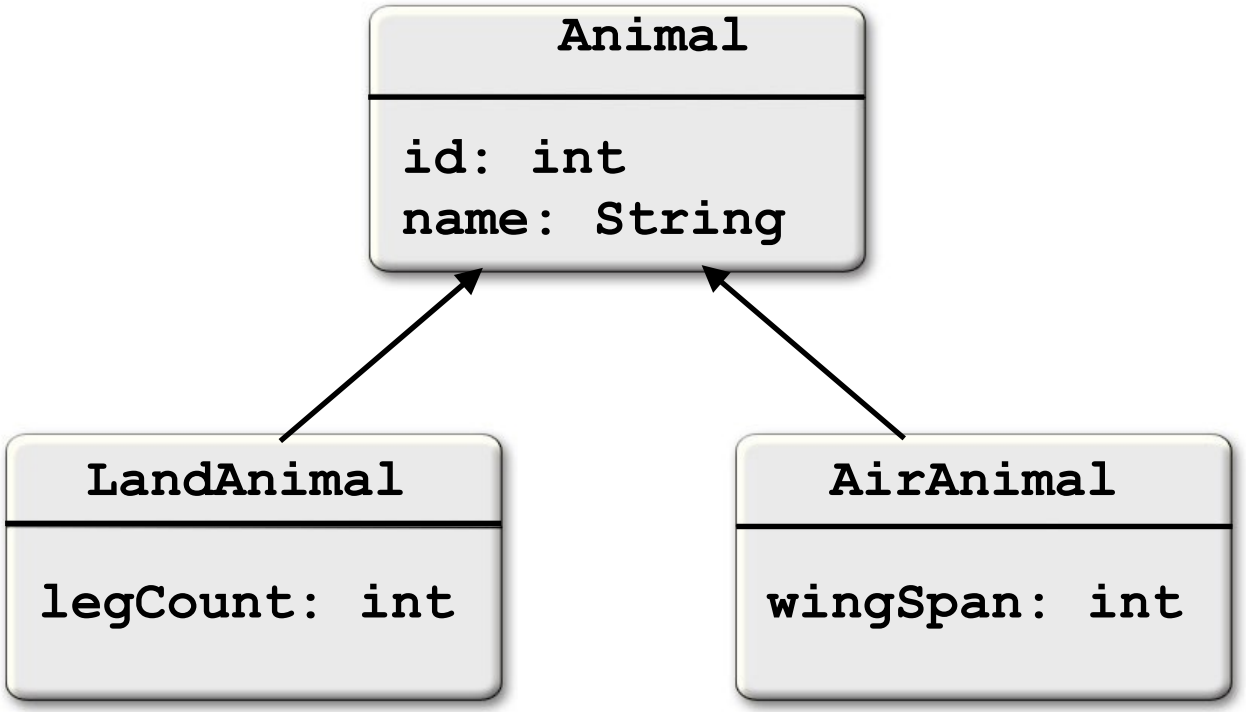
Embedded Objects

```
@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



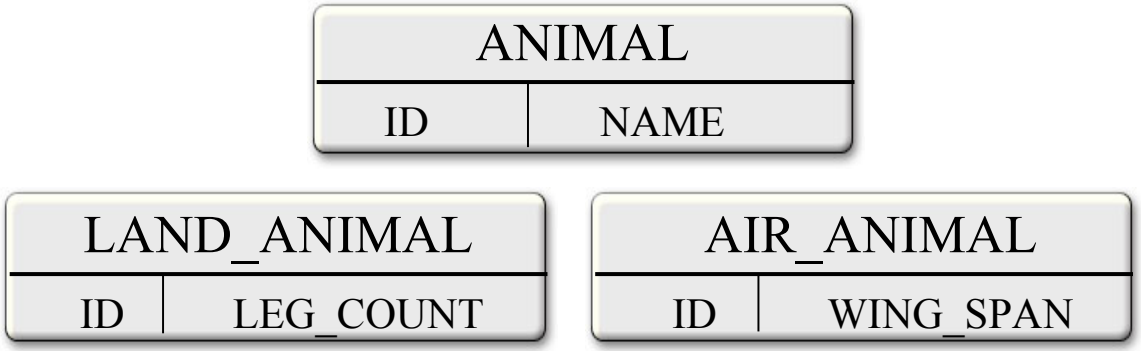
Data Models

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

Single table:

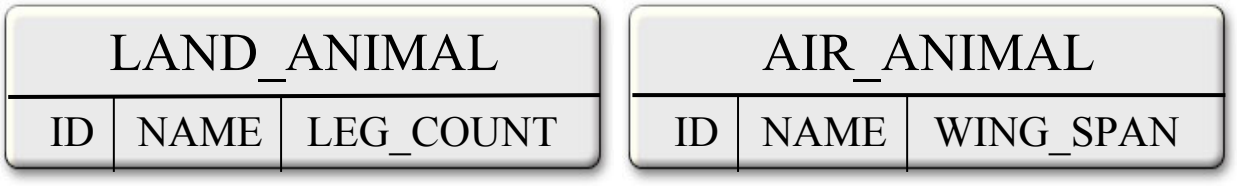
Decent polymorphic support; Requires JOIN to be performed for queries ranging over class hierarchies. Could perform badly in deep hierarchies.

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 “Bootstrap 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 transaction akin to a JDBC transaction
  - 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("HRSystem");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    Employee emp = em.find(
        Employee.class, new Integer(args[0]));
    emp.setSalary(new Integer(args[1]));
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

Summary and Resources

Summary of EJB 3.0

- Major simplification of EJB technology for developers
  - Beans are plain Java classes with 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-to-use functionality

Summary of JPA

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

When to use which persistence technology?

- Entity Beans
- JDO
- JPA
  - Hibernate, Kodo, Toplink, etc. implement JPA
- JDBC

Resources

- Glassfish persistence homepage
  - https://glassfish.dev.java.net/javaee5/persistence
- Persistence support page
  - https://glassfish.dev.java.net/javaee5/persistence/entity-persistence-support.html
- Blog on using persistence in Web applications
  - http://weblogs.java.net/blog/ss141213/archive/2005/12/using_java_pers.html
- Blog on schema generation
  - http://blogs.sun.com/roller/page/java2dbInGlassFish#automatic_table_generation_feature_in

GlassFish - glassfish.dev.java.net

- Sun's Open Source Application Server Platform Edition 9
  - CDDL license
  - Open processes
- Open access to code and binaries
  - CVS access to source code
  - Nightly builds, weekly promoted builds
- Must support Java EE compatibility
- Renewed partnership between Sun and the larger enterprise Java community

EJB 3.0