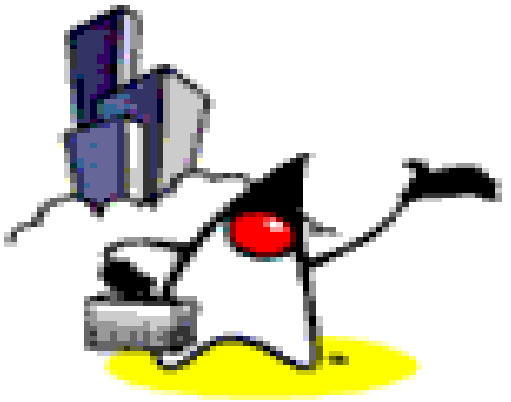
Java Persistence API (JPA): Basics

Sang Shin
Java Technology Evangelist
Sun Microsystems, Inc.
javapassion.com

Agenda

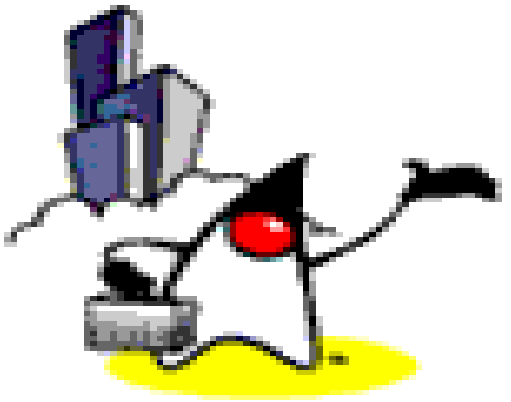
- Java Persistence Requirements
- O/R Mapping
- What is an entity?
- JPA Programming Model
- Entity Manager & life-cycle operations
- Detached Entities
- Entity life-cycle transition
- Persistence context and Entity Manager
- Transactions



Why Use ORM?

Why Object/Relational Mapping?

- A major part of any enterprise application development project is the persistence layer
  - Accessing and manipulate persistent data typically with relational database
- ORM handles Object-relational impedance mismatch
  - Data lives in the relational database, which is table driven (with rows and columns)
    - Relational database is designed for fast query operation of table-driven data
  - We want to work with objects, not rows and columns of table



What is & Why JPA?

What is JPA?

- Java EE standard O/R Mapping framework
- Object/relational mapping framework for enabling transparent POJO persistence
  - Let you work without being constrained by table-driven relational database model – handles Object-Relational impedance mismatch
- Lets you build persistent objects following common OO programing concepts

Java Persistence Requirements

Java Persistence Requirements

- Simplify the persistence model
  - Default over configuration
  - Elimination of deployment descriptor
- Provide light-weight persistence model
  - Programming and deployment model
  - Runtime performance
- Enable testability outside of the containers
  - Test-driven development
  - Test entities as part of nightly-build process

Java Persistence Requirements

- Support rich domain modelling
  - Inheritance and polymorphism
- Provide standardized and efficient Object/Relational (O/R) mapping
  - Optimized for relational database
  - Standardize annotations and XML configuration files
- Provide extensive querying capabilities
- Support for pluggable, third-party persistence providers
  - Through persistence unit - represented by *persistence.xml*

Common Java Persistence Between J2SE and J2EE Environments

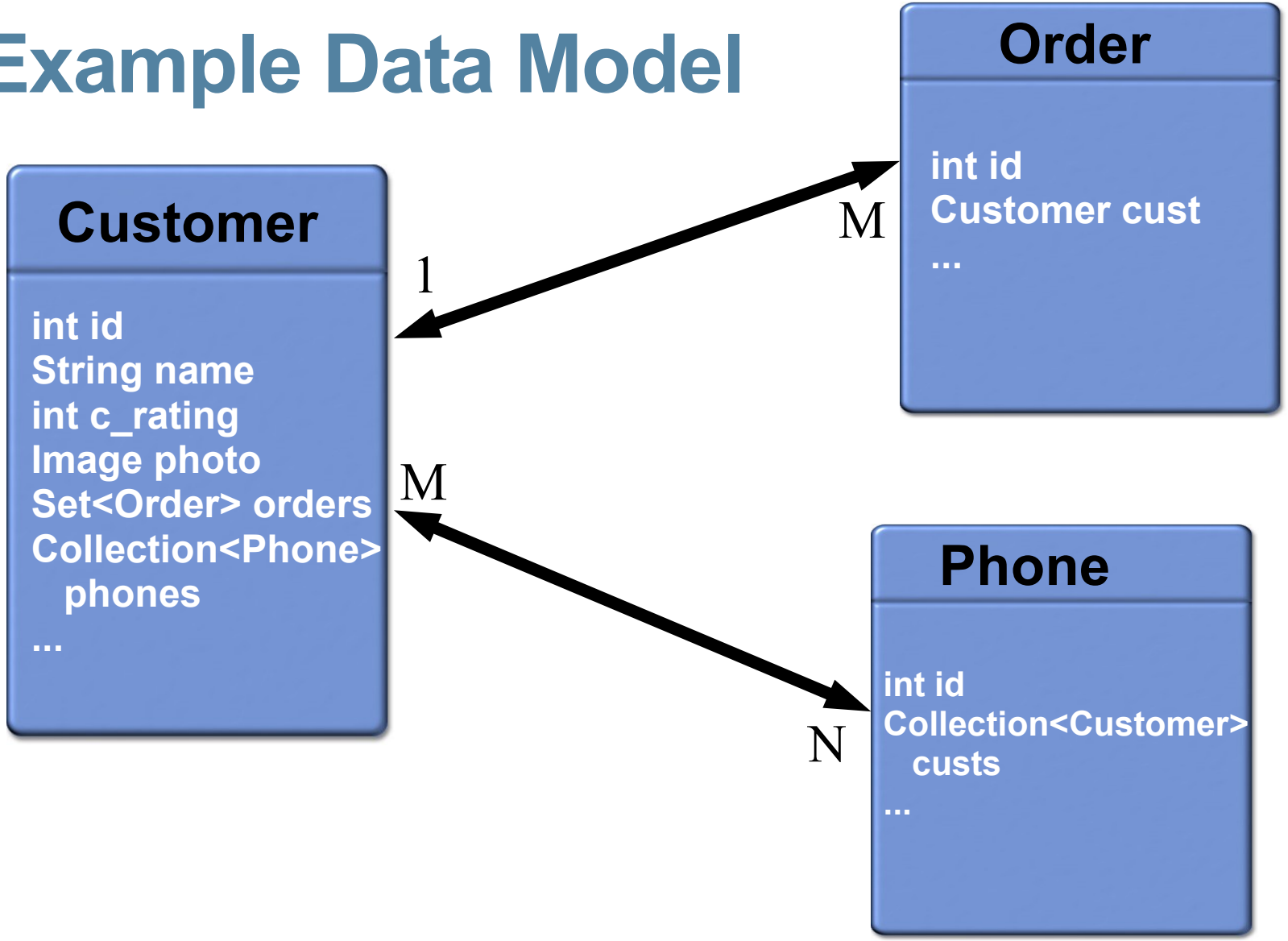
- Persistence API expanded to include use outside of EJB container
- Evolved into “common” Java persistence API between Java SE and Java EE apps
  - You can use new Java persistence API in Java SE, Web, and EJB applications

O/R Mapping

O/R Mapping

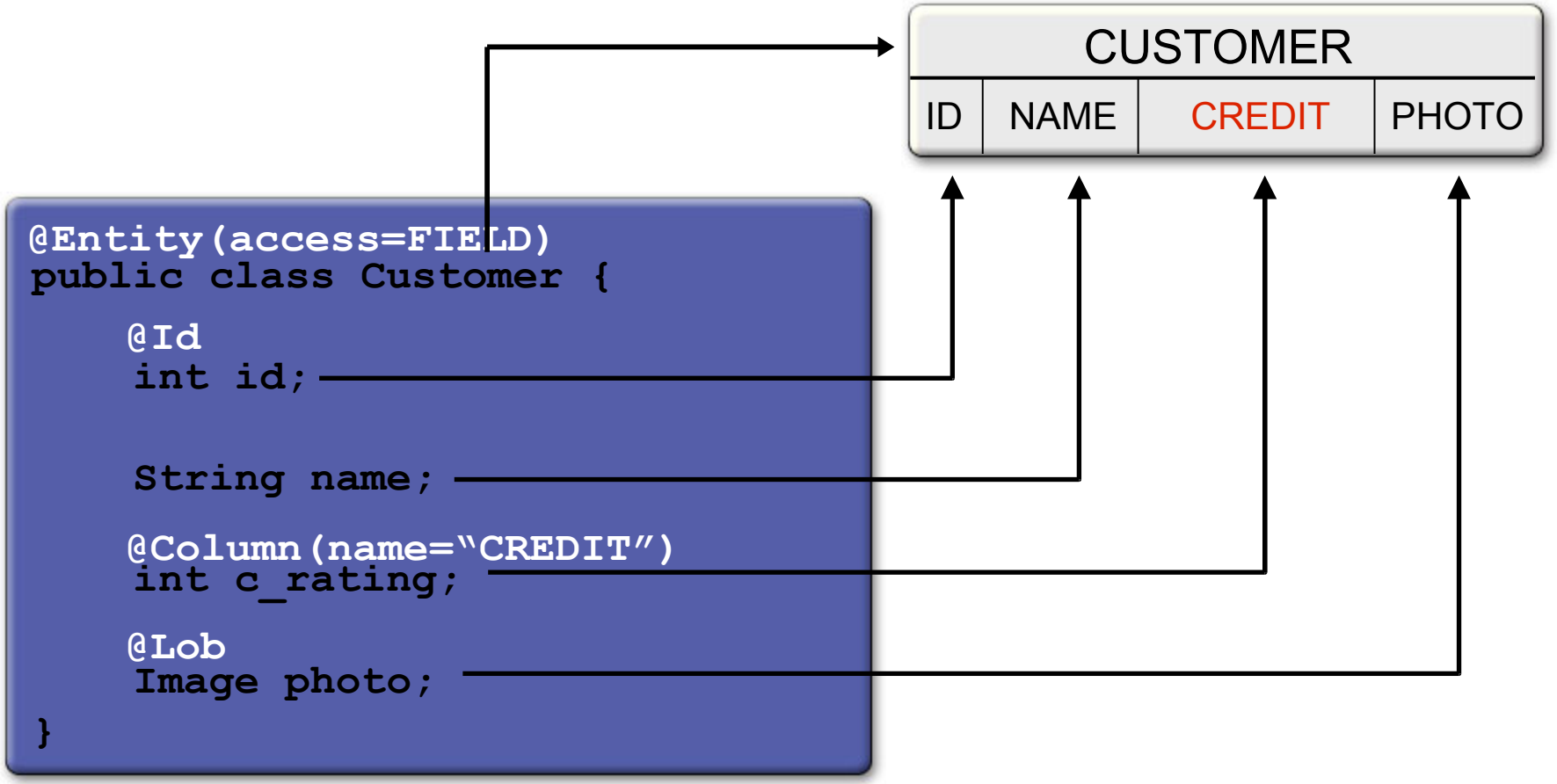
- Comprehensive set of annotations defined for mapping
  - Relationships
  - Joins
  - Database tables and columns
  - Database sequence generators
  - Much more
- Specified using standard description elements in a separate mapping file or within the code as annotations

An Example Data Model



Maps entity state to data store
Maps relationship to other entities

Simple Mapping



Mapping defaults to matching column name. Only configure if entity field and table column names are different.

What is an Entity?

What is an Entity?

- Plain Old Java Object (POJO)
  - Created by means of new keyword just like a normal Java class
  - No need to implement interfaces unlike EJB 2.1 entity beans
- May have both persistent and non-persistent state
  - 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

Entity 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
  - @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
  - @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

Programming Model

Java Persistence Programming Model

- Entity is a POJO (no need to implement EntityBean)
- Use of Annotation to denote a POJO as an entity (instead of deployment descriptor)

```
// @Entity is an annotation
// It annotates Employee POJO class to be Entity
@Entity
public class Employee {
   // Persistent/transient fields
   // Property accessor methods
   // Persistence logic methods
}
```

Another Persistence Entity Example

```
@Entity
public class Customer {
   private Long id;
   private String name;
   private Address address;
   private Collection<Order> orders = new HashSet();

   public Customer() {}

   @Id public Long getID() {
      return id;
   }

   protected void setID (Long id) {
      this.id = id;
   }

...
```

Annotated as “Entity”

@Id denotes primary key

Getters/setters to access state

Persistence Entity Example (Contd.)

```
    . . .

    // Relationship between Customer and Orders
    @OneToMany
    public Collection<Order> getOrders() {
      return orders;
    }

    public void setOrders(Collection<Order> orders) {
      this.orders = orders;
    }

    // Other business methods
    ...
}
```

Client View: From Stateless Session Bean

```
@Stateless
public class OrderEntry {

        // Dependency injection of Entity Manager for
        // the given persistence unit
        @PersistenceContext
        EntityManager em;

        public void enterOrder(int custID, Order newOrder){

                // Use find method to locate customer entity
                  Customer c = em.find(Customer.class, custID);
                // Add a new order to the Orders
                  c.getOrders().add(newOrder);
                  newOrder.setCustomer(c);
        }

        // other business methods
}
```

Client Code: From Java SE Client

```
public static void main(String[] args) {
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("EmployeeService");
    EntityManager em = emf.createEntityManager();

    Collection emps = em.createQuery("SELECT e FROM Employee e")
                        .getResultList();

    // More code
```

Entity Manager & Entity Life-cycle Operations

EntityManager

- Similar in functionality to Hibernate Session, JDO PersistenceManager, etc.
- Controls life-cycle of entities
  - persist() - insert an entity into the DB
  - remove() - remove an entity from the DB
  - merge() - synchronize the state of detached entities
  - refresh() - reloads state from the database

Persist Operation

```
public Order createNewOrder(Customer customer) {
    // Create new object instance – entity is in transient state
    Order order = new Order(customer);

    //  After persist() method is called upon the entity,
    //  the entity state is changed to managed. On the
    //  next flush or commit, the newly persisted
    //  instances will be inserted into the database table.
    entityManager.persist(order);

    return order;
}
```

Find and Remove Operations

```
public void removeOrder(Long orderId) {
    Order order =
            entityManager.find(Order.class, orderId);

    // The instances will be deleted from the table
    // on the next flush or commit. Accessing a
    // removed entity has undefined results.
    entityManager.remove(order);
}
```

Merge Operation

```
public OrderLine updateOrderLine(OrderLine
    orderLine) {

   // The merge method returns a managed copy of
   // the given detached entity. Changes made to the
   // persistent state of the detached entity are
   // applied to this managed instance.
    return entityManager.merge(orderLine);
}
```

Detached Entities

Detached Entities

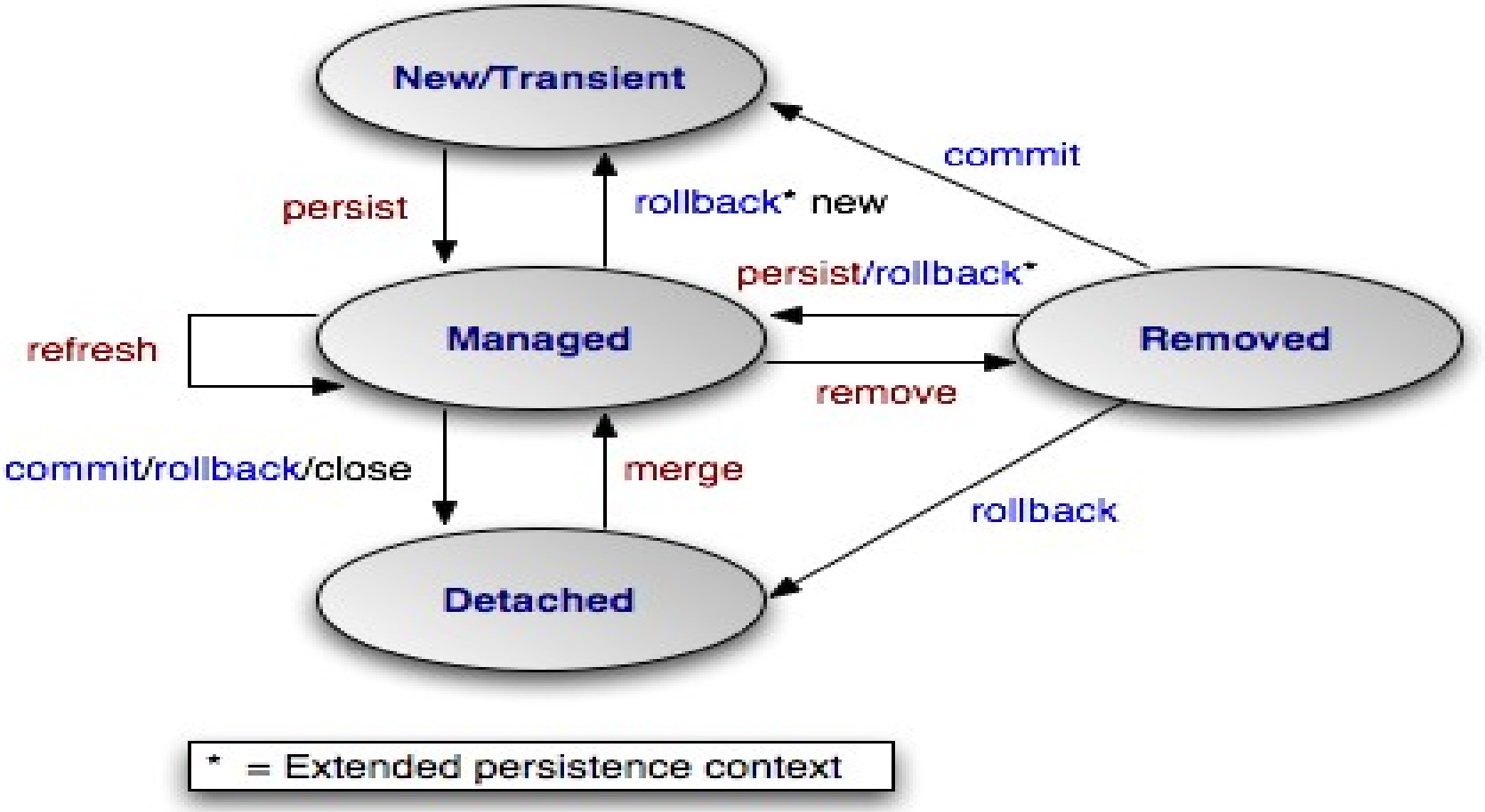
- Must implement Serializable interface if detached object has to be sent across the wire
- No need for DTO (Data Transfer Object) anti-design pattern
- Merge of detached objects can be cascaded

Transition to Detached Entities

- When a transaction is committed or rollback'ed
- When an entity is serialized

Entity Life-cycle Transition

Entity Life-cycle



Demo #1

1. Creating Entities from Existing Database tables

2. Performing CRUD (Create, Read, Update, Delete) operations against Entities

You can try this demo from www.javapassion.com/handsonlabs/jpabasics

Persistence Context & Entity Manager

Persistence Context & Entity Manager

- Persistence context
  - Represents a set of managed entity instances at runtime
  - "Entity instance is in managed state" means it is contained in a particular persistent context
  - Entity instances in a particular persistent context behaves in a consistent manner
- Entity manager
  - Performs life-cycle operations on entities – manages persistence context

Persistence Context & Entity Manager

- Persistence context is not directly accessible to developers
  - There is no programming API for accessing persistence context – there is no need
  - Persistence context is accessed indirectly through entity manager
- The type of entity manager determines how a persistence context is created and removed
- Why do you care as a developer?
  - Because inclusion or exclusion of an entity into/from the persistence context will affect the outcome of any persistence operation on it

Types of Entity Managers

Types of Entity Managers

- Container-Managed Entity Manager (Java EE environment)
  - Transaction scope entity manager - our focus of discussion
  - Extended scope entity manager
- Application-Managed Entity Manager (Java SE environment)

How Entity Manager Is Created

- Different type of Entity Manager is created and acquired by an application differently
  - Container-managed entity manager (for Java EE) is acquired by an application through *@PersistenceContext* annotation – the container creates an entity manager and injects it into the application
  - Application-managed entity manager (for Java SE) is created and closed by the application itself

Demo #2

Creating Entity Manager for Java SE Environment

You can try this demo from www.javapassion.com/handsonlabs/jpabasics

Entity Managers & Persistence Context

- Different type of Entity Manager creates and manages a persistence context differently
  - The lifetime of persistence context is determined by the type of Entity manager

Transaction-Scope Entity Manager

- Persistence context is created when a transaction gets started and is removed when the transaction is finished (committed or rolled-back)
  - The life-cycle of the persistence context is tied up with transactional scope
- Persistence context is propagated
  - The same persistence context is used for operations that are being performed in a same transaction
- The most common entity manager in Java EE environment

Extended-Scope Entity Manager

- Extended-scope Entity manager work with a single persistent context that is tied to the life-cycle of a stateful session bean

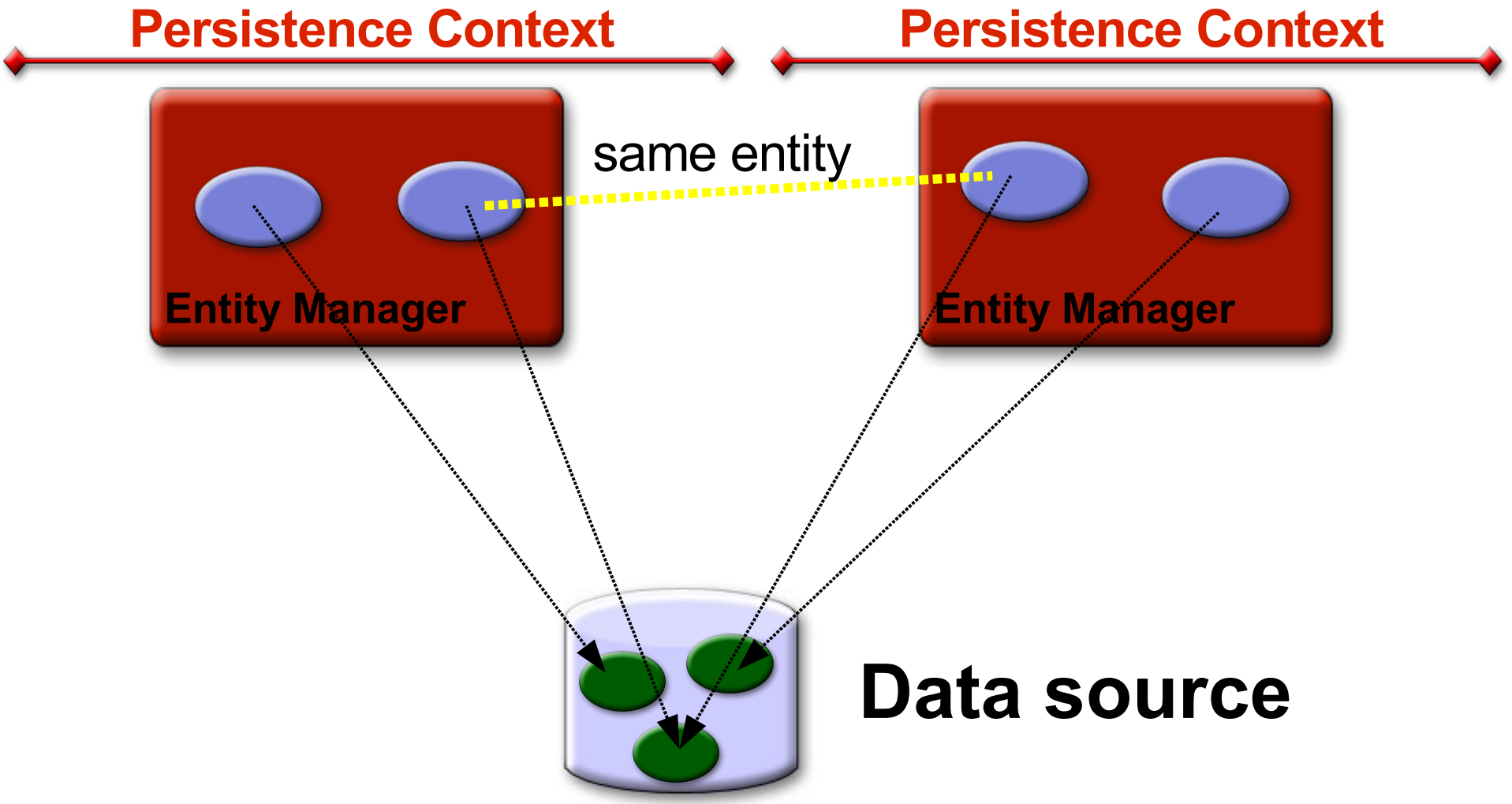
Persistence Unit

Persistence Unit

- All entities managed by a single EntityManager is defined by a persistence unit
- A persistence unit defines
  - all entities that are co-located
  - mapped onto a single database
- persistence.xml defines one or more persistence unit

```
<persistence-unit name="OrderManagement">
   <mapping-file>mappings.xml</mapping-file>
   <jar-file>order.jar</jar-file>
   <transaction-type>JTA</transaction-type>
</persistence-unit>
```

Persistence Unit



Transactions

Transaction Types

- Two different transaction types
  - Resource-local transactions
  - JTA (Java Transaction API)
    - Multiple participating resources
    - Distributed XA transactions
- Transaction type is defined in persistence unit (*persistence.xml* file)
  - Default to JTA in a Java EE environment
  - Default to RESOURCE_LOCAL in a Java SE environment

@TransactionAttribute Annotation

- TransactionAttributeType.REQUIRED
- TransactionAttributeType.REQUIRES_NEW
- TransactionAttributeType.MANDATORY
- TransactionAttributeType.NOT_SUPPORTED
- TransactionAttributeType.NEVER
- TransactionAttributeType.SUPPORTS

Entity Manager and Transaction Type

- Container managed entity manager use JTA transactions
- Propagation of persistence context with a JTA transaction is supported by the container
  - Sharing same persistence context among multiple entity managers

Transactions & Persistence Context

- Transactions define when new, modified, or removed entities are synchronized with the database
- How persistence context is created and used is determined by
  - Transaction type (JTA or Resource-local) and
  - Transaction attribute (REQUIRED or ..)

Demo #3

Use two different transaction attributes and see how persistence context is propagated.

You can try this demo from www.javapassion.com/handsonlabs/jpabasics

Demo Scenarios

- There are two stateless beans
  - EmployeeServiceBean (Calling bean)
  - AuditServiceBean (Callee bean)
- #1: The createEmployee() method of the EmployeeServiceBean invokes logTransaction() method of the AuditServiceBean
  - logTransaction() is set with TransactionAttributeType.REQUIRED annotation
- #2: The createEmployee2() method of the EmployeeServiceBean invokes logTransaction2() method of the AuditServiceBean
  - logTransaction2() is set with TransactionAttributeType.REQUIRES_NEW annotation

EmployeeServiceBean (Calling Bean)

```
@Stateless
public class EmployeeServiceBean implements EmployeeService {
    @PersistenceContext(unitName="EmployeeService")
    private EntityManager em;
    @EJB
    AuditService audit;

    public void createEmployee(Employee emp) {
        em.persist(emp);
        audit.logTransaction(emp.getId(), "created employee");
    }

    public void createEmployee2(Employee emp) {
        em.persist(emp);
        audit.logTransaction2(emp.getId(), "created employee");
    }
```

AuditServiceBean (Callee Bean)

```
@Stateless
public class AuditServiceBean implements AuditService {
    @PersistenceContext(unitName="EmployeeService")
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED) //Default
    public void logTransaction(int empId, String action) {
        // verify employee number is valid
        if (em.find(Employee.class, empId) == null) {
            throw new IllegalArgumentException("Unknown employee id");
        }
        LogRecord lr = new LogRecord(empId, action);
        em.persist(lr);
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void logTransaction2(int empId, String action) {
        // ... same code as logTransaction() ...
    }
```

Behind the Scene: Scenario #1

- The createEmployee() method of the EmployeeServiceBean (Calling bean) invokes logTransaction() method of the AuditServiceBean (Callee bean)
  - logTransaction() of AuditServiceBean (Bean #2) is set with TransactionAttributeType.REQUIRED annotation
- The createEmployee() method of Bean #1 starts a new transaction A as default, thus creating a persistence context A
  - The newly created Employee object A belongs to persistence context A
- The transaction A and persistence context A of createEmploy() method of Calling bean is propagated to the logTransaction() method of Callee bean
  - The logTransaction() method has access to Employee object A

Behind the Scene: Scenario #2

- The createEmployee2() method of the EmployeeServiceBean (Calling bean) invokes logTransaction2() method of the AuditServiceBean (Callee bean)
  - logTransaction2() of AuditServiceBean (Callee bean) is set with TransactionAttributeType.REQUIRES_NEW annotation
- The createEmployee2() method of Calling bean starts a new transaction A as default, thus creating a persistence context A
  - The newly created Employee object A belongs to persistence context A
- The logTransaction2() method of Callee bean creates a new transaction B, thus a new persistence context B
  - The persistence context B does not have Employee object A
  - Employee object A still belongs to persistence context A
  - em.find() will fail

Sun microsystems

Thank You!