JPA Basics

Sang Shin
JPassion.com
"Code with Passion!"

Topics

- What is and Why O/R Mapper (ORM)?
- Why JPA?
- Java Persistence Requirements
- JPA O/R Mapping
- What is an entity?
- Entity Manager
- Persistence context
- Persistence unit

What is and Why use O/R Mapper (ORM)?

Why Object/Relational Mapping (ORM)?

- A major part of any enterprise application development project is the persistence layer
  - Accessing and manipulating 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 (Java developers) want to work with objects, not rows and columns of table, however

Why JPA?

What is JPA?

- Standard ORM framework for Java platform
- Enables transparent POJO persistence
  - Let you work without being constrained by table-driven relational database model – handles Object-Relational impedance mismatch
  - Like Hibernate
- Lets you build persistent objects with common OO programing concepts
  - Inheritance, Polymorphism

Java Persistence Requirements

Java Persistence Requirements (1)

- Simplify the persistence programming
  - Default over configuration (Convention over configuration)
  - Eliminate the need of the XML-based deployment descriptor
- Provide light-weight persistence model
  - In both programming model and deployment
  - Runtime performance
- Enable testability outside of the containers
  - Enables test-driven development
  - Test entities as part of nightly-build process

Java Persistence Requirements (2)

- Support rich domain modelling
  - Support inheritance and polymorphism among entities
- Provide standardized and efficient ORM
  - Optimized for relational database
  - Standardize annotations and XML configuration files
- Provide extensive querying capabilities
  - Comparable to Hibernate query capabilities
- Support for pluggable, third-party persistence providers
  - Through persistence unit - represented by *persistence.xml*

Common Java Persistence Between J2SE and J2EE Environments

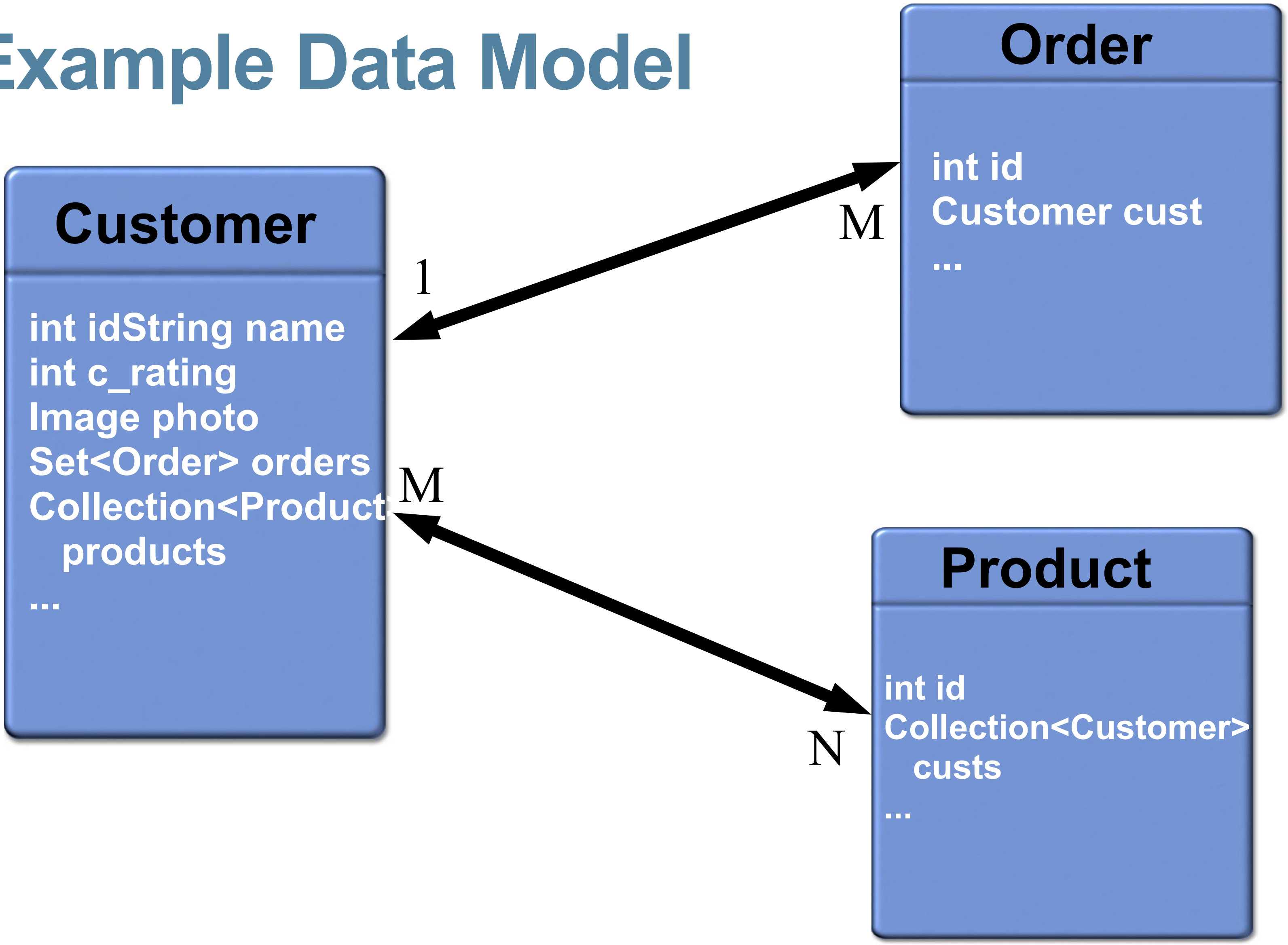
- Persistence API expanded to include use outside of EJB container
- Evolved into “common” Java persistence API between Java SE and Java EE apps
  - You can use Java persistence API (JPA) in Java SE, Web, and EJB applications

O/R Mapping

O/R Mapping Annotations

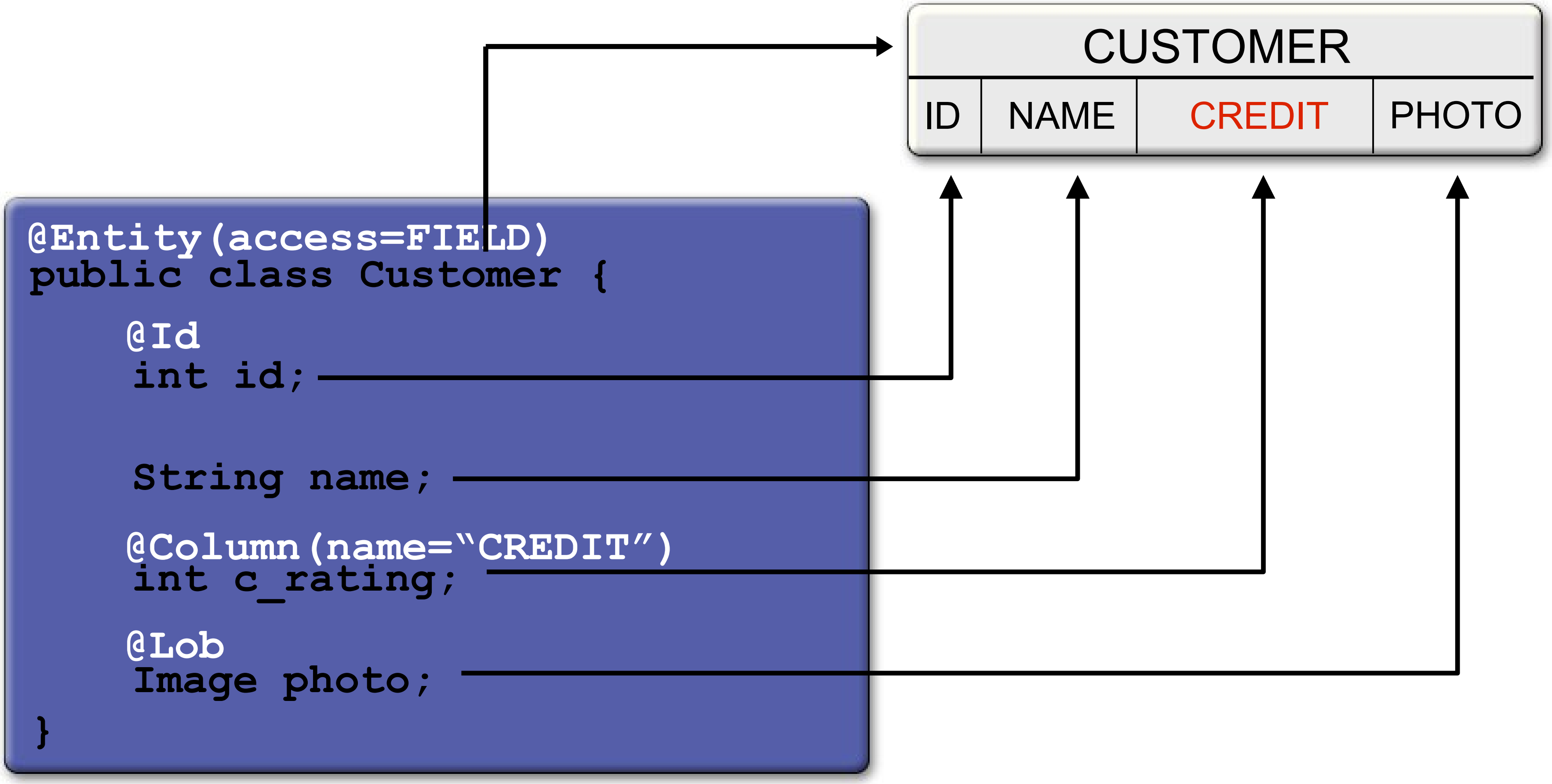
- Comprehensive set of annotations defined for mapping
  - Relationships
  - Joins
  - Database tables and columns
  - Much more

An Example Data Model



Maps entity state to data store
Maps relationship to other entities

Simple Mapping



```
@Entity(access=FIELD)
public class Customer {
    @Id
    int id;

    String name;

    @Column(name="CREDIT")
    int c_rating;

    @Lob
    Image photo;
}
```

Mapping defaults to matching column name. Only configure if entity field and table column names are different. (By the way, @Lob is for large object)

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
  - Supports OO programming model – inheritance, polymorphic relationship
- May be in either persistent (managed) or non-persistent state (non-managed)
  - Example of non-persistent state is “transient” state
- Have persistent identity
  - When it is in managed state
- Can extend other entity and non-entity classes
  - Inheritance
- Serializable; usable as detached objects in other tiers
  - No need for Data Transfer Objects (DTOs) anymore

Entity Class (@Entity)

- Annotated with *@Entity*
- Can extend another entity
- Programming requirement
  - Must have a primary key field
  - Must have a public no-arg constructor
  - Instance variables must not be public
  - Must not be final or have final methods

Default Mapping

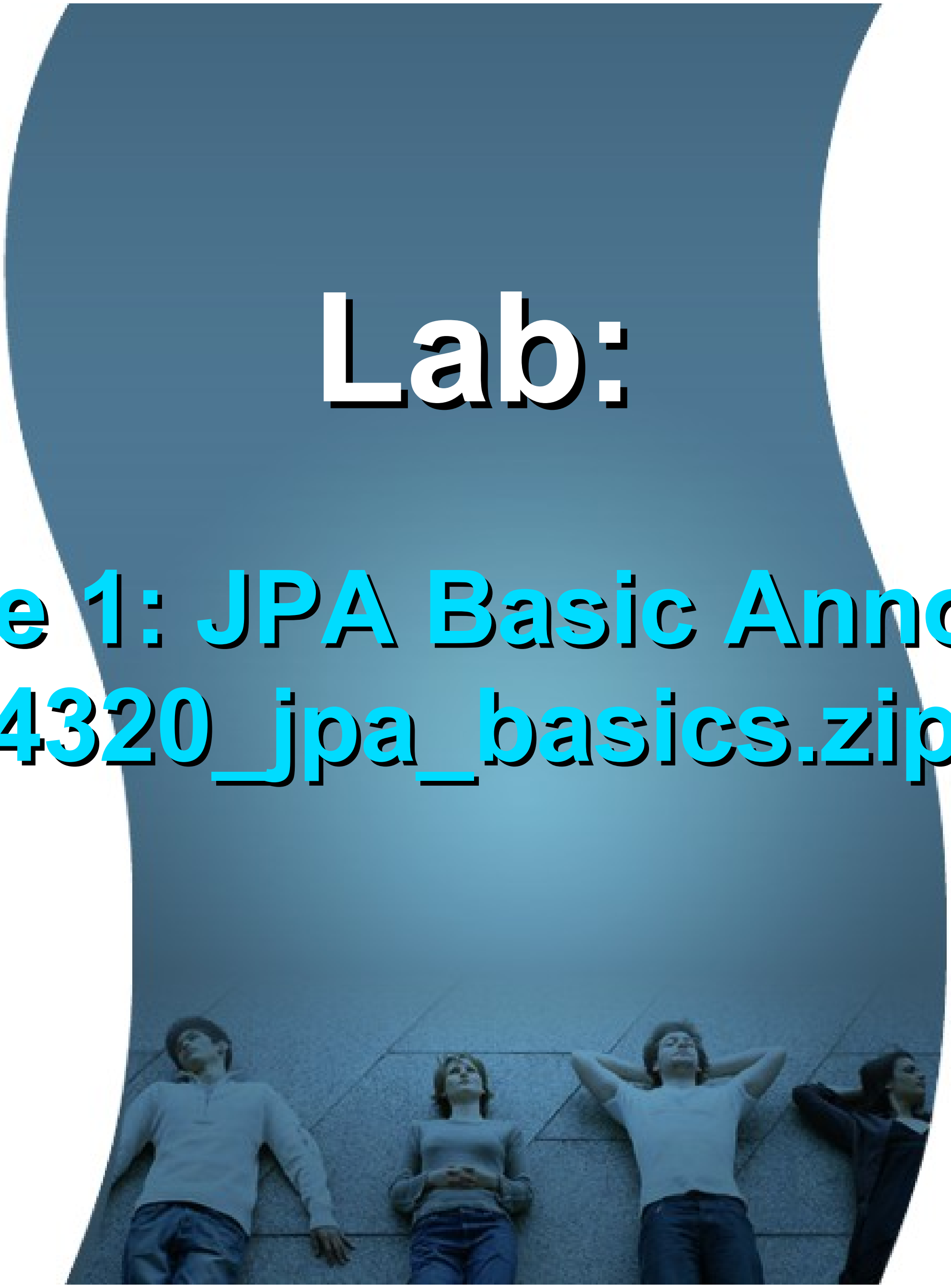
- Entity name → table name (customizable via @Table)
- Attribute name → column name (customizable via @Column)
- Data type mapping (some differences among databases)
  - String → VARCHAR(255)
  - Long, long → BIGINT
  - Double, double → DOUBLE
  - Boolean → SMALLINT

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 (@Id, @GeneratedValue)

- 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
    - AUTO - Choose type depending on database, e.g. IDENTITY for MySQL
    - IDENTITY - using a database identity column.
    - SEQUENCE - using a database sequence
    - TABLE - Use a sequence table for key generation (most portable)



Lab:

Exercise 1: JPA Basic Annotations
4320_jpa_basics.zip

@Transient

- Use it for any attribute that does not map to a column

```
@Entity
@Table(name="my_own_employee_table")
public class Employee {

    @Id
    private int id;
    @Column(name="my_name")
    private String name;
    @Column(name="my_bonus")
    private long salary;
    @Transient
    private Double bonus;
    @Transient
    private Boolean b;
```

@Temporal

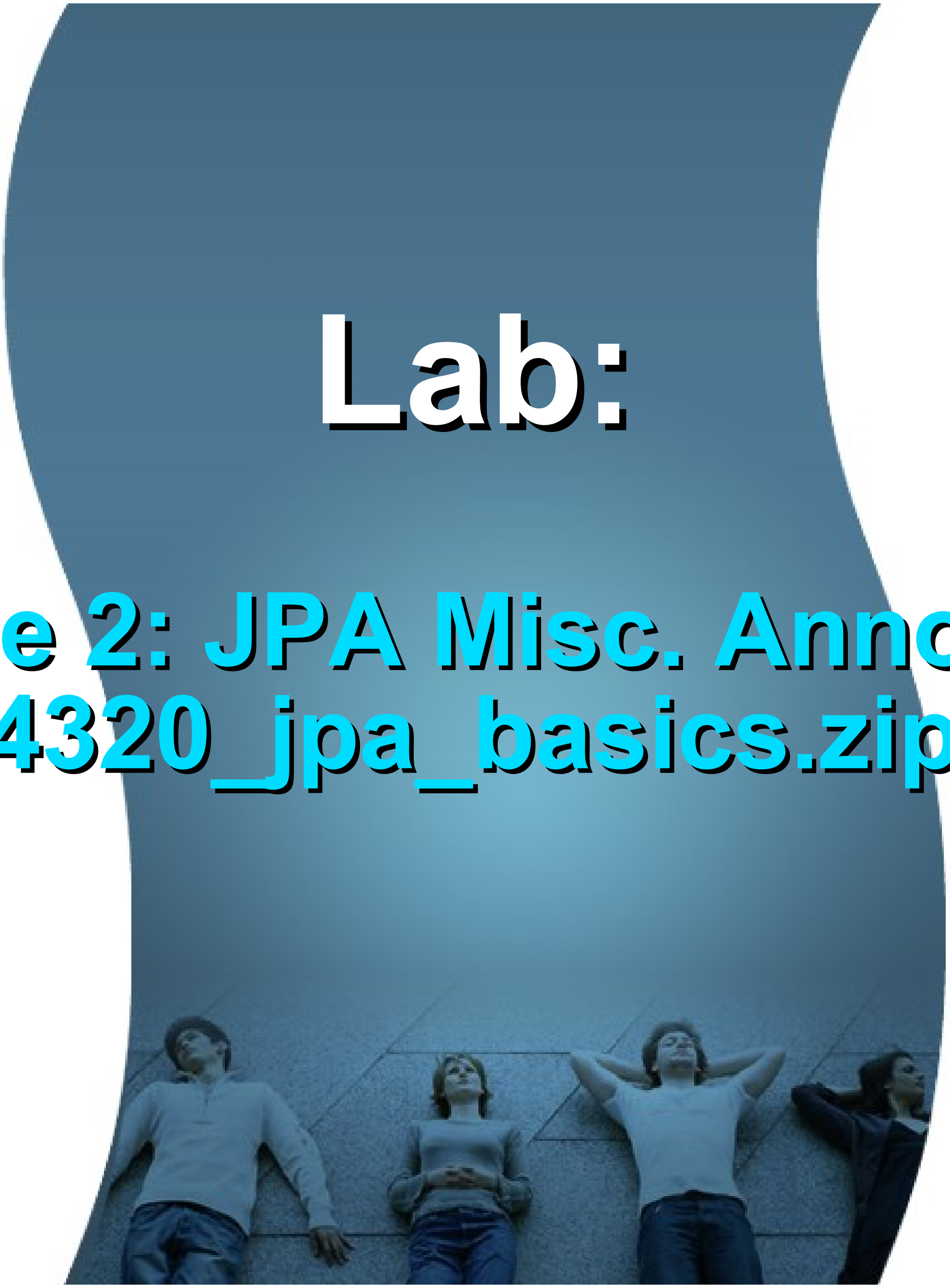
- Use it to map *Date* or *Calendar* attribute

```
@Entity
public class Employee {

    @Id
    private int id;

    @Temporal(TemporalType.DATE)
    @Column(name = "my_birthday")
    private Date dateOfBirth;

    @Temporal(TemporalType.TIME)
    private Date currentTime;

    @Temporal(TemporalType.TIMESTAMP)
    private Calendar dateOfHiring;
```

@Enumerated

- Use it to map enum

```
public enum EmployeeType {
    ADMIN,
    MANAGER,
    OFFICER
}

@Entity
@Table(name="my_own_employee_table")
public class Employee {

    //@Enumerated(EnumType.STRING)
    @Enumerated(EnumType.ORDINAL)
    private EmployeeType employeeType;
```



Lab:

Exercise 2: JPA Misc. Annotations
4320_jpa_basics.zip

Entity Manager & Persistence Context & Persistence Unit

Key Concepts of JPA Operations

- Entity manager
- Persistence context
- Persistence unit

What is EntityManager?

- Manages the state and life-cycle of entities
  - Creates and removes entity instances within the persistence context
- Handles querying entities within a persistence context
  - Performs finding entities via their primary keys
- Lock entities
- Accessible through *EntityManager* Java interface
  - The life-cycle operations are defined in the EntityManager interface
- Similar in functionality to Hibernate *Session*

Types of Entity Managers

- #1: Application-Created Entity Manager (Java SE environment)
  - > Entity manager is created and managed by the application
- #2: Container-Created Entity Manager (Java EE environment)
  - > Entity manager is created and managed by the Container
  - > Entity manager will be provided to the application via dependency injection

#1: Application Created EM (Java SE)

```
public static void main(String[] args) {
    // Application is responsible for explicitly obtaining Entity Manager
    // and life-cycle of it
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("EmployeeService");
    EntityManager em = emf.createEntityManager();

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

    // Some code

    em.close();
    emf.close();
```

#2: Container-Created EM (Java EE)

```
@Stateless
public class OrderEntry {

        // Entity Manager is created & injected by the container.
        @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);
        }

        // No need to close EntityManager
}
```

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
  - All entity instances in a particular persistent context behaves in a consistent manner – for example, all changed entity instances will be persisted to the database table next commit or flush
- Entity manager
  - Manages persistence context
  - Performs life-cycle operations on entities maintained in the persistence context

Manipulation of Entities (in the PersistenceContext) via EntityManager

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.  In other
    //  words, the entity is added to the persistence context.
    //  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

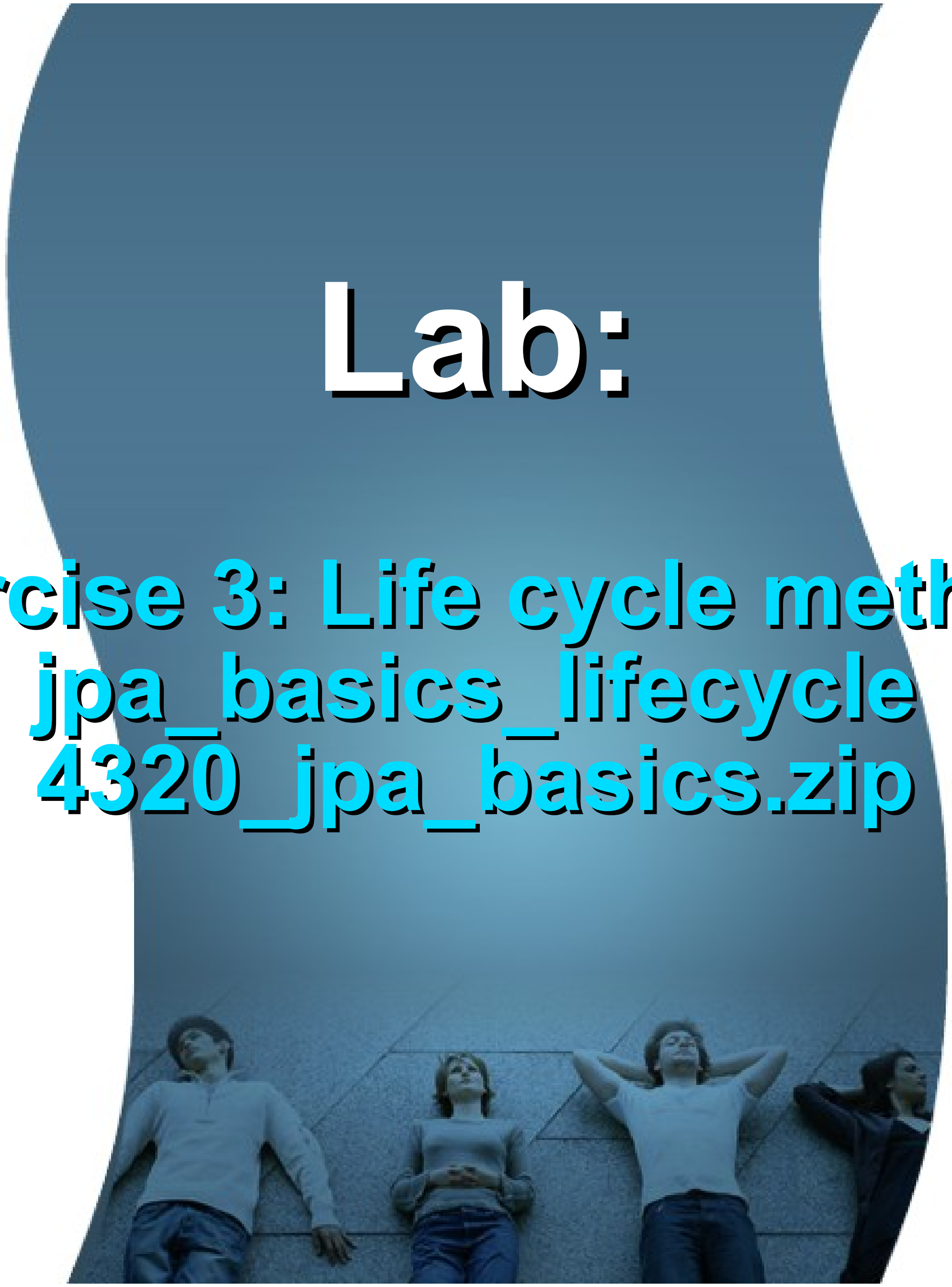
```
public void removeOrder(Long orderId) {
    Order order =
          // Try to find an entity in the persistence context
          entityManager.find(Order.class, orderId);

   // The instances will be deleted from the the
   // persistence context first.
   // And on the next flush or commit, corresponding
   // row will be deleted from the database table.
    entityManager.remove(order);
}
```

Merge Operation

```
public OrderLine updateOrderLine(OrderLine orderLine) {

   // The merge method returns a managed copy of
   // the given detached entity.  In other words, the
   // entity is now in the persistence context.
    return entityManager.merge(orderLine);
}
```

EntityManager Methods

- *void persist(Object entity)* – makes an instance managed (i.e. persistent)
- *void remove(Object entity)* - removes the entity from the persistence context (when the transaction is committed or persistence context is flushed, the corresponding row in the table is also removed – difference from “detach”)
- *void detach(Object entity)* - detaches the entity from the persistence context
- *entity = merge(Object entity)* - synchronize the state of detached entity, making it managed again, returns it
- *void refresh(Object entity)* - reloads state from the database
- *find(Class<T> entityClass, Object primaryKey)* - find an entity
- *void flush()* - synchronize the persistence context to the underlying database
- *void clear()* - clears the persistence context, causing all managed entities to become detached
- *boolean contains(Object entity)* - checks if the instance belongs to the current persistence context



Lab:

Exercise 3: Life cycle methods
jpa_basics_lifecycle
4320_jpa_basics.zip

Persistence Unit

What is Persistence Unit?

- Entity manager handles the communication to the database through a persistence provider
- When Entity manager is created either by Application or Container, configuration information is needed for configuring the persistence manager
  - > In the same way, your JDBC application need to have configuration information
- The configuration data is called “Persistence Unit”
- Represented by *persistence.xml*
  - > Every JPA application has to have a *persistence.xml* file
  - > Located under /META-INF

What Info. Does Persistence Unit Define?

- Name of the persistence unit
- Transaction type
  - “RESOURCE_LOCAL” for application-managed environment
  - “JTA” for container-managed environment
- Persistence provider class
- Entity classes (only for Java SE application)
- Table generation strategy
- Database related
  - “Database connection properties” for application-managed environment
  - “Datasource” for container-managed environment (since Datasource already captures the database connection information) – datasources are created and managed separately

persistence.xml (JPA 1.0) - for Application Managed environment

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
 <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
  <provider>oracle.toplink.essentials.PersistenceProvider</provider>
  <class>entities.Employee</class>
  <properties>
   <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
   <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/testdb"/>
   <property name="toplink.jdbc.user" value="app"/>
   <property name="toplink.jdbc.password" value="app"/>
   <!-- enable this property to see SQL and other logging -->
   <!-- property name="toplink.logging.level" value="FINE"/ -->
   <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
  </properties>
 </persistence-unit>
</persistence>
```

Persistence Provider

persistence.xml (JPA 1.0) - for Container Managed environment

```
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="jsf-jpa-war" transaction-type="JTA">
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <!-- use this property if target server is GlassFish V3 with EclipseLink -->
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
      <!-- use this property if toplink is a target server persistence provider
      <property name="toplink.ddl-generation" value="drop-and-create-tables"/> -->
    </properties>
  </persistence-unit>
</persistence>
```

Code with Passion!
JPassion.com