Object-Relational Mapping

• An object-relational mapping tool attempts to bridge the divide between OO languages and relational databases
  – We did this by hand in lab 2. There are tools and libraries that can do it for us
• ORM tools exist for many different languages
  – ActiveRecord (Ruby on Rails)
  – ADO.NET Entity Framework (Microsoft)
  – Django ORM (Python)
  – JPA, Hibernate (Java)
Entity Classes

- Annotations or XML configuration files are used to map classes and fields to tables and columns
  - `@Entity` marks the class as a model object
  - `@Table` specified the corresponding database table
  - `@SequenceGenerator` defines a database sequence to use for auto-generated ID fields

```java
@Entity
@Table(name = "employees")
@SequenceGenerator(name = "EMPLOYEE_SEQ", sequenceName = "employees_employee_id_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 6803370824626024108L;
    ...
}
```
Entity Classes

- Must have a default (no argument) constructor
- Must be a top-level class (not an inner class or an interface)
- Must not be final
- Should implement Serializable

```java
@Entity
@Table(name = "employees")
@SequenceGenerator(name = "EMPLOYEE_SEQ", sequenceName = "employees_employee_id_seq")
public class Employee implements Serializable {
    private static final long serialVersionUID = 6803370824626024108L;

    ...}
```
Entity Classes

• Annotations can be placed on either the field declaration or the getter method (preferred) to map fields to columns

```java
private Long id;

@Id
@GeneratedValue(strategy = GenerationType.AUTO, generator = "EMPLOYEE_SEQ")
@Column(name = "employee_id")
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}
...
```
Entity Classes

• @Id indicates primary key field
• @GeneratedValue indicates how to create database generated values
  – GenerationType.AUTO is the most DBMS agnostic
• @Column maps the field to a database column

```java
private Long id;

@Id
@GeneratedValue(strategy = GenerationType.AUTO, generator = "EMPLOYEE_SEQ")
@Column(name = "employee_id")
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}
...
```
Entity Classes

• Basic Java types can be mapped with a simple @Column
  – including: primitive types, primitive wrapper types, String, Date, byte[], other entity types

• Mapped fields must have standard getter and setter methods

```java
private String name;

@Column(name = "name")
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

...
Entity Classes

• Relationships between entities can be
  – one-to-one
  – one-to-many
  – many-to-one
  – many-to-many

• Relationships can be
  – unidirectional
    • One entity has a reference to another, but not vice-versa
  – bidirectional
    • Each entity has a reference to the other
Entity Classes

- Collection-valued fields must use generic collection interfaces
  - Collection, Set, List, Map

```java
public class Department {

    ...

    public List<Employee> getEmployees() { ... }
    public void setEmployees(List<Employee> employees) { ... }

    ...
}
```
Entity Classes

- **@OneToOne**
  - Defines a one-to-one mapping between entities
  - Uses @JoinColumn to specify foreign key column

```java
public class Employee {
    ...

    private Address address;

    @OneToOne
    @JoinColumn(name = "address_id")
    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    ...
}
```
Entity Classes

- **@ManyToOne**
  - Defines a many-to-one mapping between entities
  - Uses @JoinColumn to specify foreign key column

```java
public class Employee {
    ...

    private Department department;

    @ManyToOne @JoinColumn(name = "department_id")
    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }

    ...
}
```
Entity Classes

- **@OneToMany**
  - Defines a one-to-many mapping between entities
  - mappedBy indicates field (not column) in target class

```java
public class Department {
    ...

    private List<Employee> employees;

    @OneToMany(mappedBy = "department", targetEntity = Employee.class)
    public List<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(List<Employee> employees) {
        this.employees = employees;
    }

    ...
}
```
Entity Classes

- **@ManyToMany**
  - `@JoinTable` declares the mapping table for the many-to-many relationship
  - `@JoinTable` can also be used for one-to-many/many-to-one relationships with a separate mapping table

```java
public class Employee {
    ...

    private List<Project> projects;

    @ManyToMany
    @JoinTable(name = "employee_project_map",
                joinColumns = { @JoinColumn(name = "employee_id") },
                inverseJoinColumns = { @JoinColumn(name = "project_id") })
    public List<Project> getProjects() {
        return projects;
    }

    public void setProjects(List<Project> projects) {
        this.projects = projects;
    }

    ...
}
```
Entity Classes

• A bidirectional relationship is simply one where both entity types have a reference to the other
  – Employee and Department in the previous slides defined a bidirectional relationship

• A unidirectional relationship is simply where one entity type has a reference to another but the reverse mapping is not defined
EntityManager

- Access to entities is done through an instance of EntityManager
- EntityManager objects are obtained via an EntityManagerFactory
- EntityManager is not thread safe
  - Should use a separate EntityManager for every thread
    - This means a separate EntityManager for every request in a Servlet
- EntityManagerFactory is thread safe
  - Generally create only one EntityManagerFactory in an application
EntityManager

• Can think of EntityManagerFactory as wrapping the DataSource and EntityManager as wrapping a Connection
  – Same logic applies
    • Singleton EntityManagerFactory
    • EntityManager per thread/connection
**EntityManager**

- EntityManagerFactory usually created via configuration
  - persistence.xml file
- EntityManagerFactory is obtained through Persistence static method
- EntityManager is obtained through EntityManagerFactory
  - Should always close the EntityManager
    - Analogous to closing database connection

```java
class Private void process(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    ...
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
    EntityManager em = emf.createEntityManager();

    try {
        ...
    } finally {
        em.close();
    }
    ...
}
```
Accessing Entities

• There are several ways to access entities
  – find() by primary key
  – SQL-like queries
  – Criteria queries
Accessing Entities

• **find() by primary key**
  - Takes an entity class and a key value
  - Entity object is automatically created and populated

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
EntityManager em = emf.createEntityManager();

try {
    Employee employee = em.find(Employee.class, employeeId);

    String name = employee.getName();

    ...
} finally {
    em.close();
}
```

Accessing Entities

- Select by query
  - Similar to SQL but uses entity class and field names instead of table and columns
  - Can use dot-notation to reference fields

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
EntityManager em = emf.createEntityManager();

try {
    Query query = em.createQuery("SELECT x FROM Employee AS x WHERE x.name = '" + name + "'");
    List<Employee> employees = query.getResultList();
    if (employees != null && employees.size() > 0) {
        ...
    }
} finally {
    em.close();
}
```
Transactions

• An EntityTransaction can be obtained from an EntityManager
  – Standard begin/commit/rollback logic

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("emf");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;

try {
    tx = em.getTransaction();
    tx.begin();

    ...

    tx.commit();
} catch (Exception e) {
    if (tx != null && tx.isActive()) {
        tx.rollback();
    }
    ...
} finally {
    em.close();
}
```
Entity Lifetime

- Entity objects are invalid after transaction or manager is closed
  - Accessing the properties of an entity object after close can cause error or a non-valid null value

```java
try {
    tx = em.getTransaction();
    tx.begin();

    ...

    // Processing of JSPs must now be done in the transaction
    request.getRequestDispatcher("WEB-INF/jsp/Welcome.jsp").forward(request, response);

    tx.commit();
} catch (Exception e) {
    if (tx != null && tx.isActive()) {
        tx.rollback();
    }

    ...
} finally {
    em.close();
}
```