The Object-Oriented Paradigm

- The world consists of objects
- So we use object-oriented languages to write applications
- We want to store some of the application objects (a.k.a. persistent objects)
- So we use a Object Database?

The Reality of DBMS

- Relational DBMS are still predominant
  - Best performance
  - Most reliable
  - Widest support
- Bridge between OO applications and relational databases
  - CLI and embedded SQL
  - Object-Relational Mapping (ORM) tools

Call-Level Interface (CLI)

- Application interacts with database through functions calls

```java
String sql = "select name from items where id = 1";
Connection c = DriverManager.getConnection( url );
Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery( sql );
if( rs.next() ) System.out.println( rs.getString("name") );
```

Employee – Application Object

```java
public class Employee {
    Integer id;
    String name;
    Employee supervisor;
}
```

Embedded SQL

- SQL statements are embedded in host language

```java
String name;
String sql = "select name into :name from items where id = 1";
System.out.println( name );
```
Employee – Database Table

```sql
create table employees ( 
    id integer primary key,
    name varchar(255),
    supervisor integer references employees(id)
);
```

From Database to Application

So how do we construct an Employee object based on the data from the database?

```java
public class Employee {
    Integer id;
    String name;
    Employee supervisor;

    public Employee( Integer id ) {
        // access database to get name and supervisor
        ... ...
    }
}
```

Problems with CLI and Embedded SQL ...

SQL statements are hard-coded in applications

```java
public Employee( Integer id ) { 
    PreparedStatment p;
    p = connection.prepareStatement( 
            "select * from employees where id = ?"
    );
    ... 
}
```

... Problems with CLI and Embedded SQL ...

Tedious translation between application objects and database tables

```java
public Employee( Integer id ) { 
    ResultSet rs = p.executeQuery();
    if( rs.next() ) 
    { 
        name = rs.getString("name");
        ... 
    }
}
```

... Problems with CLI and Embedded SQL

Application design has to work around the limitations of relational DBMS

```java
public Employee( Integer id ) {
    ResultSet rs = p.executeQuery();
    if( rs.next() ) 
    { 
        ... 
        supervisor = ??
    }
}
```

The ORM Approach

```
Application
  
  customer
  
   employee

ORM tool
  
  Oracle, MySQL, SQL Server ...

Persistent Data Store
  
  Flat files, XML ...
```
Hibernate and JPA

- Java Persistence API (JPA)
  - Annotations for object-relational mapping
  - Data access API
  - An object-oriented query language JPQL
- Hibernate
  - The most popular Java ORM library
  - An implementation of JPA

Hibernate Usage

- Hibernate without JPA
  - API: SessionFactory, Session, Query, Transaction
  - More features
- Hibernate with JPA
  - API: EntityManagerFactory, EntityManager, Query, Transaction
  - Better portability
  - Behaviors are better defined and documented

A Hibernate Example

- Java classes
- JPA configuration file
- Code to access the persistent objects
- (Optional) Logging configuration files

Java Classes

- Plain Java classes (POJOs); however, it is recommended that
  - Each persistent class has an identity field
  - Each persistent class implements the Serializable interface
  - Each persistent field has a pair of getter and setter, which don't have to be public

O/R Mapping Annotations

- Describe how Java classes are mapped to relational tables

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Entity</td>
<td>Persistent Java Class</td>
</tr>
<tr>
<td>@Id</td>
<td>Id field</td>
</tr>
<tr>
<td>@Basic (can be omitted)</td>
<td>Fields of simple types</td>
</tr>
<tr>
<td>@ManyToOne</td>
<td>Fields of class types</td>
</tr>
<tr>
<td>@OneToMany</td>
<td></td>
</tr>
<tr>
<td>@ManyToOne</td>
<td></td>
</tr>
<tr>
<td>@OneToMany</td>
<td></td>
</tr>
</tbody>
</table>

Persistence.xml

- `<persistence-unit>`
  - name
- `<properties>`
  - Database information
  - Provider-specific properties
- No need to specify persistent classes
Access Persistent Objects

- **EntityManagerFactory**
- **EntityManager**
- **Query** and **TypedQuery**
- **Transaction**
  - A transaction is required for updates

Some EntityManager Methods

- `find(entityClass, primaryKey)
- `createQuery(query)
- `createQuery(query, resultClass)
- `persist(entity)
- `merge(entity)
- `getTransaction()"

A Common Scenario That Needs Merge()

1. Load an object from database
   - Open EntityManager
   - Load object
   - Close EntityManager
2. Save the object in HTTP session
3. Change some fields of the object
4. Save the object back to database
   - Open EntityManager
   - Save object
   - Close EntityManager

Java Persistence Query Language (JPQL)

- A query language that looks like SQL, but for accessing objects
- Automatically translated to DB-specific SQL statements
- `select e from Employee e` where e.id = :id
  - From all the Employee objects, find the one whose id matches the given value

See Chapter 4 of Java Persistence API, Version 2.0

Persist() vs. Merge()

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Persist</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object persisted never persisted</td>
<td>1. Object added to persistence context as new entity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. New entity inserted into database at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. New entity added to persistence context</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. New entity returned</td>
<td></td>
</tr>
<tr>
<td>Object was previously persisted, but not loaded in the persistence context</td>
<td>1. EntityExistsException thrown or a PersistenceException at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. EntityExistsException thrown or a PersistenceException at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. EntityExistsException thrown or a PersistenceException at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Loaded entity returned</td>
<td></td>
</tr>
<tr>
<td>Object was previously persisted and already loaded in the persistence context</td>
<td>1. EntityExistsException thrown or a PersistenceException at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. EntityExistsException thrown or a PersistenceException at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. EntityExistsException thrown or a PersistenceException at flush/commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Loaded entity returned</td>
<td></td>
</tr>
</tbody>
</table>

The Returned Value of Merge()

```java
Employee e = new Employee();
e.setName( "Joe" );
entityManager.persist( e );
e.getid() ➜ ??
```

```java
Employee e = new Employee();
e.setName( "Joe" );
entityManager.merge( e );
e.getid() ➜ ??
```
Hibernate Query Language (HQL)

- A superset of JPQL

Join in HQL ...

```java
class User {
    Integer id;
    String username;
}

class Section {
    Integer id;
    User instructor;
}
```

... Join in HQL ...

Query: find all the sections taught by the user “cysun”.
- **SQL**??
- **HQL**??

```sql
SELECT * FROM users WHERE username = 'cysun'
```

```hql
FROM User u INNER JOIN u.sections
WHERE u.username = 'cysun'
```

See [SectionDaoImpl in CSNS2 for more HQL join examples](#)

Advantages of ORM

- Make RDBMS look like ODBMS
- Data are accessed as objects, not rows and columns
- Simplify many common operations. E.g. `e.getSupervisor().getName()`
- Improve portability
  - Use an object-oriented query language
  - Separate DB specific SQL statements from application code
- Object caching
**SchemaExport**
- Part of the Hibernate Core library
- Generate DDL from Java classes and annotations
- In Hibernate Examples, run `Hbm2ddl <output_file>`

**Basic Object-Relational Mapping**
- Class-level annotations
  - `@Entity` and `@Table`
- `@Id` field
  - `@Id` and `@GeneratedValue`
- Fields of simple types
  - `@Basic` (can be omitted) and `@Column`
- Fields of class types
  - `@ManyToMany` and `@OneToOne`

**Advanced ORM**
- Embedded class
- Collections
- Inheritance

**Embedded Class**
```
public class Address {
    String street;
    String city;
    String state;
    String zip;
}
```
```
public class User {
    String username;
    Integer id;
    String password;
    Address address;
}
```
```
users
```
```
id | street | city | state | zip |
```

**Mapping Embedded Class**
```
@Embeddable
public class Address {
    String street;
    String city;
    String state;
    String zip;
}
```
```
@Entity
public class User {
    @Id
    Integer id;
    String username
    String password;
    @Embedded
    Address address;
}
```

**Collection of Simple Types**
```
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
}
```
Mapping Element Collection

```java
@ElementCollection
Set<String> phones;
```

Customize Collection Table

```java
@ElementCollection
@CollectionTable(name = "customer_phones",
        joinColumns=@JoinColumn(name = "customer_id")
)  
@Column(name="phone")
Set<String> phones;
```

List of Simple Types

- **Order by property**
  - `@OrderBy("<property_name> ASC|DESC")`
  - Simple types do not have properties
    ```java
    @ElementCollection
    @OrderBy("asc")
    List<String> phones;
    ```
- **Order by a separate column**
  ```java
  @ElementCollection
  @OrderColumn(name = "phone_order")
  List<String> phones;
  ```

Issues Related to Collections of Object Types

- **Relationships (a.k.a. associations)**
  - one-to-many
  - many-to-many
- **Unidirectional vs. Bidirectional**
- Set and List
- Cascading behaviors

Types of Relationships

- **Many-to-Many**
- **Many-to-One / One-to-Many**
- **One-to-One**

Many-to-Many Relationship

- Each entity in $E_1$ can be related to many entities in $E_2$
- Each entity in $E_2$ can be related to many entities in $E_1$
Many-to-One Relationship

- Each entity in E₁ can be related to one entity in E₂.
- Each entity in E₂ can be related to many entities in E₁.

One-to-One Relationship

- Each entity in E₁ can be related to one entity in E₂.
- Each entity in E₂ can be related to one entity in E₁.

Relationship Type Examples

- Books and authors?
- Books and editors?

One-To-Many Example

- A customer may own multiple accounts.
- An account only has one owner.

Bidirectional Association – OO Design #1

```java
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    Customer owner;
}
```

```java
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    Set<Account> accounts;
}
```

Unidirectional Association – OO Design #2

```java
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    Set<Account> accounts;
}
```

```java
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    Set<Account> accounts;
}
```
Unidirectional Association – OO Design #3

```java
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    Customer owner;
}
```

```java
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
}
```

Unidirectional vs. Bidirectional

- Do the three OO designs result in different database schemas??
- Does it make any difference on the application side??
- Which one should we use??

Mapping Bidirectional One-To-Many

```java
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    @ManyToOne
    Customer owner;
}
```

```java
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    @OneToMany(mappedBy="owner")
    Set<Account> accounts;
}
```

Using List

```java
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    @OneToMany(mappedBy="owner")
    @OrderBy("createdOn asc")
    List<Account> accounts;
}
```

Many-To-Many Example

- A customer may own multiple accounts
- An account may have multiple owners

Mapping Many-To-Many

```java
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    @ManyToMany
    Set<Customer> owners;
}
```

```java
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    @ManyToMany(mappedBy="owners")
    Set<Account> accounts;
}
```
Customize Join Table

```java
@ManyToMany
@JoinTable(
    name = "account_owners",
    joinColumns=@JoinColumn(name = "account_id"),
    inverseJoinColumns=@JoinColumn(name="owner_id")
)
Set<Customer> owners;
```

Cascading Behavior

- Whether an operation on the parent object (e.g., Customer) should be applied to the children objects in a collection (e.g., List<Account>)

```java
Customer c = new Customer("cysun");
Account a1 = new Account();
Account a2 = new Account();
c.getAccounts().add(a1);
c.getAccounts().add(a2);
entityManager.persist(c); // will a1 and a2 be saved as well?
entityManager.remove(c); // will a1 and a2 be deleted from db??
```

Cascading Types in JPA

- [CascadeType Examples](http://sun.calstatela.edu/~cysun/documentation/jpa-2.0-api/javax/persistence/CascadeType.html)

```java
@OneToMany(mappedBy="owner",
cascade=CascadeType.PERSIST)
List<Account> accounts;
@OneToMany(mappedBy="owner",
cascade={CascadeType.PERSIST, CascadeType.MERGE})
List<Account> accounts;
@OneToMany(mappedBy="owner",
cascade=CascadeType.ALL)
List<Account> accounts;
```

Inheritance

```java
public class CDAccount extends Account {
    Integer term;
}
```

Everything in One Table

```sql
<table>
<thead>
<tr>
<th>id</th>
<th>account_type</th>
<th>balance</th>
<th>created_on</th>
<th>term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Discriminator column
Inheritance Type – SINGLE_TABLE

@Entity
@Table(name="accounts")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="account_type")
@DiscriminatorValue("CHECKING")
public class Account { ... }

@Entity
@DiscriminatorValue("CD")
public class CDAccount { ... }

Table Per Subclass

<table>
<thead>
<tr>
<th>accounts</th>
<th>id</th>
<th>balance</th>
<th>created_on</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd_accounts</td>
<td>account_id</td>
<td>term</td>
<td></td>
</tr>
</tbody>
</table>

Inheritance Type – JOINED

@Entity
@Table(name="accounts")
@Inheritance(strategy=InheritanceType.JOINED)
public class Account { ... }

@Entity
@Table(name="cd_accounts")
public class CDAccount { ... }

Table Per Concrete Class

<table>
<thead>
<tr>
<th>accounts</th>
<th>id</th>
<th>balance</th>
<th>created_on</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd_accounts</td>
<td>id</td>
<td>balance</td>
<td>created_on</td>
</tr>
</tbody>
</table>

Inheritance Type – TABLE_PER_CLASS

@Entity
@Table(name="accounts")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Account { ... }

@Entity
@Table(name="cd_accounts")
public class CDAccount { ... }

Choosing Inheritance Types

- Consider the following queries
  - List the information of all accounts (i.e. both checking and CD)
  - List the information of CD accounts
Tips for Hibernate Mapping

- Understand relational design
  - Know what the database schema should looks like before doing the mapping
- Understand OO design
  - Make sure the application design is object-oriented

Further Readings

- Pro JPA 2 by Mike Keith and Merrick Schincariol