CSE 530A
DAOs and CRUD
Washington University
Fall 2013
Application Integration

• How to integrate persistent storage into applications?

• Separation of Concerns
  – The rest of the application (logic and view layers) should be independent of the details of the data storage
Data Access Objects

• Data Access Objects (DAOs) are a object-oriented design pattern that provides an abstract interface to persistent storage
  – A DAO provides an interface for accessing persistent data while hiding details of the underlying persistence mechanism
CRUD

• DAOs typically provide the ability to
  – **CREATE**
    • dao.createUser(username, password, name, ...);
  – **RETRIEVE**
    • dao.getUser(username, password);
  – **UPDATE**
    • dao.updateUser(id, username, password, name, ...);
  – **DELETE**
    • dao.deleteUser(id);
CRUD

• Maps fairly straightforwardly to DBMS functionality
  – CREATE
    • INSERT INTO...
  – RETRIEVE
    • SELECT...
  – UPDATE
    • UPDATE...
  – DELETE
    • DELETE...

• But what should operations like CREATE or RETRIEVE return?
ResultSets

• Having the DAOs return ResultSets breaks the independence of the persistence layer from the rest of the code
  – Ties the code to a particular type of storage

• ResultSets are cumbersome
  – Access to data values is error prone and dependent on underlying design

```java
ResultSet rs = dao.getUser(username, password);
if (rs.next()) {
    String name = rs.getString(3);
}
```
HashMaps

• HashMaps are more generic than ResultSets
  – Persistent storage layer can be replaced without affecting the rest of the code

• Still error prone
  – Compare
    • HashMap<Object> userMap = dao.getUser(username, password);
    • String name = (String) userMap.get("name");
  – to
    • User user = dao.getUser(username, password);
    • String name = user.getName();
Model Objects

- Model Objects are data-centric objects/classes that form the data model of an application
  - In Java, usually POJOs (Plain Old Java Objects) with accessor (getter/setter) methods
- Safer than HashMaps as data objects
- Still (mostly) independent of underlying storage mechanism
- Can be expensive to instantiate
Model Objects

public class User {
    private long userId;
    private String username;
    private String password;

    public long getUserId() { return userId; }
    public void setId(long id) { userId = id; }

    ...
}

- Generally, there is a class per table
  - Exception: pure mapping tables
Model Objects

```java
public class ShoppingCart {
    private long userId;
    private DateTime lastModified;

    public long getUserId() { return userId; }
    public void setUserId(long id) { userId = id; }

    ...
}

Note: Java's Date class lacks useful functionality. Joda Time is a popular third-party date/time library for Java.

• Looks weird for ShoppingCart to have a userId but not a cartId
• Should ShoppingCart have a User field (or User have a ShoppingCart field) instead of userId?
How do we handle the many-to-many relationship between books and authors?

- Should Book have a list or set of Author (or Author a list or set of Book)?
• What about transactions? Is each DAO method its own transaction?
  – Too low level. We need transactions that span multiple DAO method calls