Pizza Project: the singletons for the APIs

The pizza1 APIs

- The most important API is the Service API. It defines what the application can do.

Doc on Service API: Has output of search for “public” in the service package, as done in class 3, notes on it, and sample service methods.

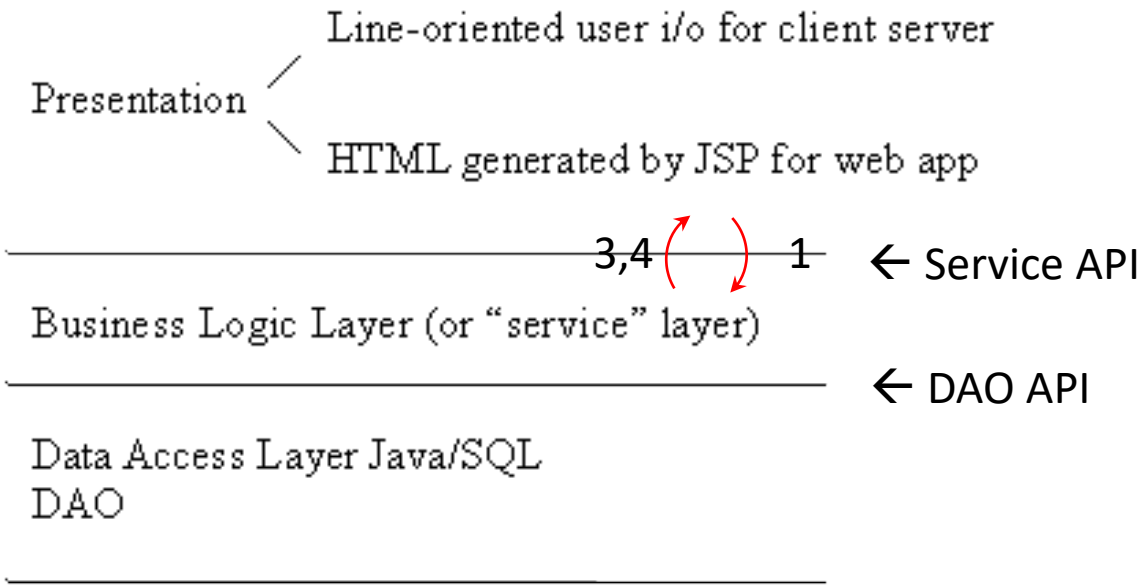
- The system also has an DAO API between the service layer and the DAO layer

Doc on DAO API: Has output of search for “public” in the dao package, notes on it, and sample DAO methods.

- The presentation layer doesn’t have an API, since it takes direction directly from the user, so we just have code snippets in Doc on Presentation Layer

Each API call has a story…

List<PizzaOrderData> getOrderStatus(int roomNumber) throws ServiceException



1. Presentation: wants order status for room 5,
Calls getOrderStatus(5)
2. Service layer gets orders for room 5 from DAO, packs them up in PizzaOrderData objects
3. Call Returns with List<PizzaOrderData> for presentation to use.
Or
4. Call throws ServiceException

Calling down through the layers

We can see in the code:

Presentation code such as SystemTest calls `studentService.makeOrder(...)` in the service layer
Then the service code calls `pizzaOrderDAO.insertOrder(...)` in a PizzaOrderDAO

Then the DAO code calls `menuDAO.findMenuSize(sizeName)` and
Statement stmt = `connection.createStatement();`

What are these variables `studentService` and `pizzaOrderDAO`?? Also `menuDAO` and `connection`?

Looking in the sources, we see they are instance variables (or fields) of the classes SystemTest, StudentService, and PizzaOrderDAO, i.e. the classes for the various bits of code.

studentService is a variable of type StudentService, etc.

Kinds of objects in pizza1

Our projects have two kinds of objects:

- Domain objects to carry data around, know how to modify it, etc.
- Infrastructure objects to hold the API methods for the call-down layers

Example domain object classes: PizzaOrder, PizzaSize, PizzaTopping, MenuSize, etc.

Example API object classes: StudentService, AdminService, PizzaOrderDAO, etc.

Domain classes can have multiple objects existing at once, for example, multiple PizzaTopping objects for one PizzaOrder object

API classes are singletons, having only one instance of the object in the system.

Domain objects are typically short-lived, representing scratch copies of database data for the current request, whereas API objects are long-lived, set up at the start of the program.

In the code inside makeOrder of StudentService we see both kinds of objects in one line of code:

```
pizzaOrderDAO.insertOrder(order);
```

- pizzaOrderDAO is a ref to a PizzaOrderDAO API object, holding the insertOrder(...) API call
- order is a ref to a PizzaOrder domain object (which has PizzaTopping objects and a PizzaSize hanging off of it). That order was set up in the service layer and is now being sent down to the DAO.
- Each API call is of that same form: api.action(relevant domain objects)
- or [returned domain object(s)] = api.action(relevant domain objects)

How do we create all these API objects?

What is needed:

Presentation (SystemTest, etc.): needs 2 refs: studentService and adminService, so it can do calls like studentService.makeOrder(...)

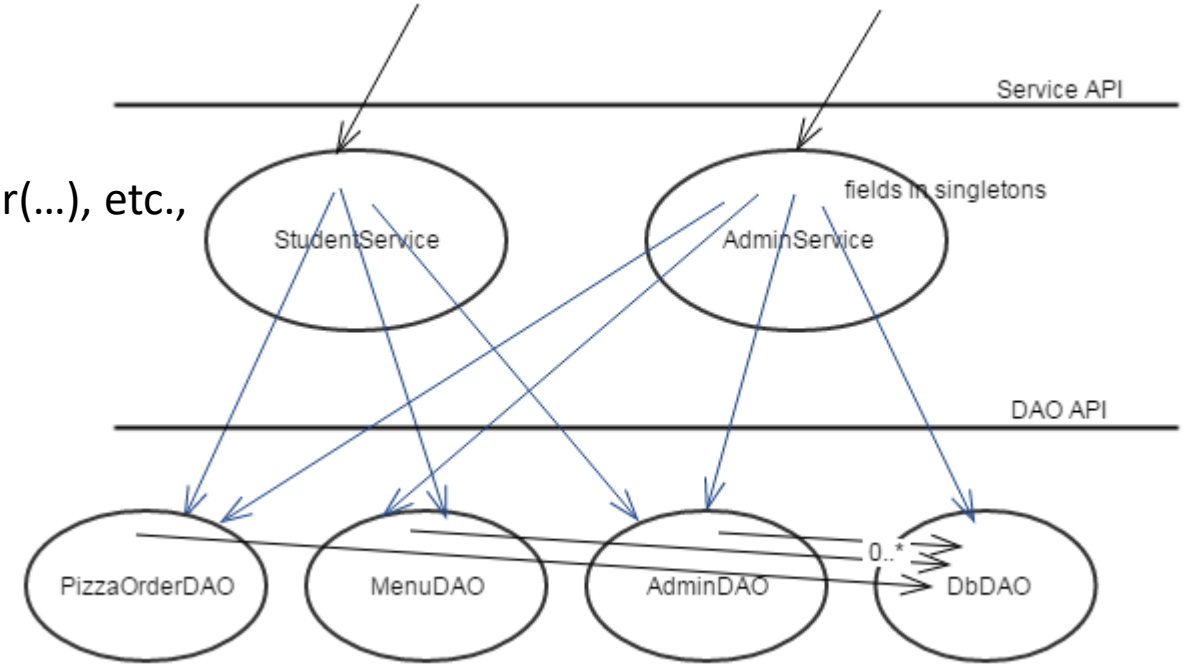
Service layer (StudentService, AdminService): needs refs to DAO objects, so it can do calls like pizzOrderDAO.insertOrder(...)

We have special “config” code to create these major objects

The Big Picture of needed API refs

For studentService.makeOrder(…), etc.

For pizzaOrderDAO.insertOrder(…), etc., in StudentService



Each oval represents a singleton object.
Each arrow represents an object ref from a field in a singleton (or program variable, for presentation layer)

PizzaOrderDAO calls dbDAO example: int ordNo = dbDAO.findNextId("next_order_id");

Big picture, cont.

Blue arrows show object ref's from service objects to DAO objects, all singletons.

Example class:

```
public class StudentService {
    private PizzaOrderDAO pizzaOrderDAO;  <-- one blue arrow in the big picture
    private MenuDAO menuDAO;              <-- another blue arrow
    private AdminDAO adminDAO;            <-- another blue arrow
```

- These refs are used in calls down from the service layer to the DAO layer, so the arrows show the direction of the calls as well as the refs.
- We see that in the DAO, there are cross-calls from various DAOs to DbDAO, where some common code is held. This is OK. It's only upcalls we have disallowed.

Dependency Injection

We see that the service code needs refs to the DAO API objects to do its work.

One way: have the service code do “new PizzaOrderDAO(…)” and put the resulting ref in place.

But it is well known that this is a bad idea: it makes unit testing harder, and in this case we would end up with duplication of DAO objects.

Better idea: *provide* the DAO object refs to the service-layer object in the service-layer object’s constructor (or by using a setter). This important technique is called Dependency Injection, or DI for short. See https://en.wikipedia.org/wiki/Dependency_injection

This way, the service layer is not in charge of what the exact DAO object is—it has to take whatever it is given and use that. That’s also called inversion of control, or IoC.

Singletons

What is a singleton? The one-and-only object created from a class.

Normally a class is expected to be used to create multiple objects of that type.

But sometimes it is reasonable to create a class and then create only one object from it. In that case the object is a singleton object, or just “singleton”.

This is considered a pattern, the singleton pattern, but it is so simple it hardly deserves that title.

Why use a singleton for StudentService?

The StudentService class is set up to express part of the service API. It holds the code for the API calls. *All* its fields are refs to DAOs. These ref values never change, so two such objects would have the same values in all the fields, and just be redundant.

(Similar arguments hold for the others).

Using static methods for an API, instead

Another way to express an API like this is with no objects at all, just a static class with static methods expressing the API.

We'll see that Murach uses this kind of API implementation. On page 654 we see a call into his DAO class UserDB:

```
user = UserDB.selectUser(emailAddress);
```

The fact that UserDB is capitalized (so a class name, not a method name) means that this is calling a static method of UserDB.java. You can see ProductDB.java on page 668, with its static methods.

But this approach is not good for testability. The class is "wired in". You can't substitute one API implementation for another.

So objects are better, even when there is only one. Also, it's the "Spring way", and we want to start using Spring ASAP. These API objects will become "Spring beans".

How to set up an object graph like the big pic

Object graphs show object dependencies: here A depends on B

To create this graph using DI, we create B first and pass it to A in A's constructor (or use a setter).

So we see that following DI, we create the most dependent objects first, and work back to the ones depending on them.

The most dependent objects have no dependencies: here its only DbDAO's singleton

Then create AdminDAO, say, passing it the DbDAO ref, …

Where is this code?

Config Code

- So what code does the new for the singletons?
- Special code that sets up the system, typically in a static method, so that it can be called by Classname.config(...), i.e., before any objects are created. Note that a Java program always starts in static code, having no objects yet.
- In fact, PizzaSystemConfig.java has only static methods and static fields. No objects of this class are created. The static fields are housed in the **class object**, which every class has (once loaded) whether or not it is used to create ordinary objects.
- Look at Pizza1's configureServices, a static method, so called by PizzaSystemConfig.configureServices(...);
- It simply creates the DAOs, then the *Service singletons, passing them the DAO refs in their constructors, then offers up the resulting service API refs to anything interested in them, i.e., the apps. See the two getters at the end of the source.

Look at class PizzaSystemConfig

```
public class PizzaSystemConfig {
// for ease of testing, handle only a few rooms--
public static final int NUM_OF_ROOMS = 10;

// the service objects in use, representing all lower layers to the app
private static AdminService adminService;
private static StudentService studentService;
// the lower-level service objects--
private static AdminDAO adminDAO;
…other DAOs
// set up service API, data access objects
public static void configureServices(String dbUrl, String usr, String psswd)
                                        throws Exception {
  try {
        dbDAO = new DbDAO(dbUrl, usr, psswd);
        adminDAO = new AdminDAO(dbDAO);
        menuDAO = new MenuDAO(dbDAO);
        pizzaOrderDAO = new PizzaOrderDAO(dbDAO, menuDAO);
        adminService = new AdminService(dbDAO, adminDAO, pizzaOrderDAO, menuDAO);
        studentService = new StudentService(pizzaOrderDAO, menuDAO, adminDAO);
…
```

We see only static fields and methods here. The app calls PizzaSystemConfig.configureServices(…) to get the system up and running.

←DbDAO constructor makes the JDBC Connection

Later in PizzaSystemConfig...

```
// Let the apps get the business logic layer services
public static AdminService getAdminService() {
   return adminService;
}

public static StudentService getStudentService() {
   return studentService;
}
```

After calling configureServices, an app can call these getters to get the needed refs to the service objects

Eclipse italicizes adminService and studentService because they are static fields.

Look at app (presentation) code

```
public class TakeOrder {
private StudentService studentService;
private Scanner in; // input from the user

public TakeOrder(String dbUrl, String usr, String psswd) throws Exception {
  PizzaSystemConfig.configureServices(dbUrl, usr, psswd);
  studentService = PizzaSystemConfig.getStudentService();
  in = new Scanner(System.in);
}
// later in the file:
studentService.makeOrder(roomNum, chosenSizeName, chosenToppings);
```

← Ref to service object, in object variable, not static

←constructor, so is an object class

See expected calls to PizzaSystemConfig to get started

StudentService (again)

```
public class StudentService {
        private PizzaOrderDAO pizzaOrderDAO;      ←refs to DAOs
        private MenuDAO menuDAO;
        private AdminDAO adminDAO;

        public StudentService(PizzaOrderDAO pizzaDAO, MenuDAO mDAO, AdminDAO admDAO) {
                pizzaOrderDAO = pizzaDAO;   ←accepting values for refs to DAOs at constructor time:
                                              (we now know this is called DI, dependency injection)
                menuDAO = mDAO;
                adminDAO = admDAO;
        }
// later in file:
      pizzaOrderDAO.insertOrder(order);     ← calling DAO using ref
```

DAO layer

```
public class PizzaOrderDAO {
   private Connection connection;          ←managed by DbDAO
   DbDAO dbDAO;  // for common DB methods
   private MenuDAO menuDAO;

   public PizzaOrderDAO(DbDAO db, MenuDAO menuDb) throws SQLException {
         dbDAO = db;                  ← accepting values for refs to other DAOs at constructor time
                                          (more DI)
         menuDAO = menuDb;
         connection = db.getConnection();  ←calling DbDAO for Connection ref
    }
// later in file:
Statement stmt = connection.createStatement();
```

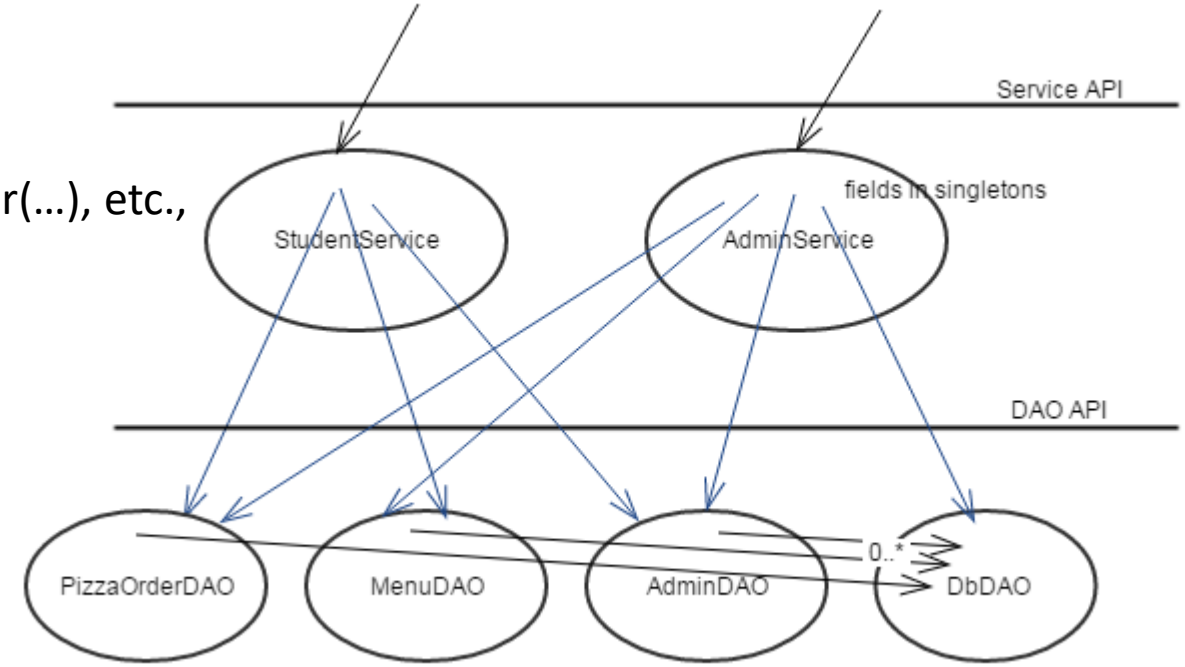
DbDAO: gets Connection

```
public class DbDAO {
    private Connection connection;

    public DbDAO(String dbUrl, String usr, String passwd) throws SQLException {  ←This is not actually DI, since these argument strings
                                                                                        are not specific  objects of this system
        if (dbUrl == null) {
                System.out.println("DbDAO constructor: replacing null dbUrl with "+H2_URL);
                dbUrl = H2_URL; // default to H2, an embedded DB
                usr = "test";
                passwd = "";
         } else {
                System.out.println("DbDAO constructor called with "+dbUrl);
        }
     }

  // package protection: no need to call this from service layer
  Connection getConnection() {          ← This is called from the other DAOs to get the Connection
      return connection;
  }
```

The Big Picture again…

For studentService.makeOrder(…), etc.

For pizzaOrderDAO.insertOrder(…), etc., in StudentService



The singletons we see here will become Spring beans in future projects, and will still be singleton POJOs

PizzaOrderDAO calls dbDAO example: int ordNo = dbDAO.findNextId("next_order_id");

Next: the Music Project and DB

Look at MusicProjectSchema.html, a portable schema.

Also, look at Murach's MySQL-specific script, with its mysql-specific constructs.

In Murach's script, skip over the database creation of the DB named "murach" and look at the music database created second in this script.

See mysql extensions of SQL (FYI, not officially covered in this class):

- Auto_increment (a nice feature, but not standardized in SQL92, or supported by Oracle)
- Case-sensitive database ids: table name LineItem, column UserID
- Datatypes DATETIME, bigint, tinyint, (SQL does have smallint), int(11)
- Enum, multiple inserts in one statement
- KEY for suggesting index on column