Music Project, JUnit

Last time: domain objects

We looked at relationships between domain objects expressed in refs from one to another and using set objects to ref out to multiple objects

We explored the resulting object graphs

We looked into creating them properly using DI

Now, a more technical detail…

Domain classes: equals/hashCode/compareTo.

In pizza1, look at end of PizzaTopping.java: See full set of these methods

```
// so we can use TreeSet<PizzaTopping> or HashSet<PizzaTopping> any time we want--
// "business key equality/comparison/hashCode" where business key is toppingName
// see comments in PizzaSize
    public int compareTo(PizzaTopping x)
    {
        return getToppingName().compareTo(x.getToppingName());
    }
    @Override
    public boolean equals(Object x)
    {
        if (x == null || x.getClass()!= getClass())
            return false;
        return getToppingName().equals(((PizzaTopping)x).getToppingName());
    }
    @Override
    public int hashCode()
    {
        return getToppingName().hashCode();
    }
```

Note top line: `PizzaTopping implements Comparable<PizzaTopping>`
Could put @Override here

Important to use @Override here to guide use of Object argument

But PizzaOrder doesn’t…

End of PizzaOrder.java: none of these, just a comment on them--

```
// Note: no compareTo, so can't use TreeSet<PizzaOrder>, just
HashSet<PizzaOrder>
// or List<PizzaOrder>.
// Here equals and hashCode are not overridden,
// so simple object equality holds, based on object memory
addresses.
// Thus two PizzaOrder objects o1 and o2 are equal, i.e.,
// o1.equals(o2) is true, if and only if o1 and o2
// are the same object, so o1 == o2.
```

So you see domain objects may usefully have these methods, but they don’t have to. Downside: here, a Set<PizzaOrder> may have two PizzaOrders of the same order id in it. The Set itself does not reject order-id duplicates, only identical-ref objects.

Music Domain classes equals/hashCode/compareTo

As provided, the music project domain classes have no equals/hashCode/compareTo methods, and thus rely on Object equals and hashCode.

So, like PizzaOrder, Set<Product> (implemented by HashSet) won't reject duplicate-product-id Product objects. We just have to make sure we don't add dups to the Set.

The default equals and hashCode are consistent, so that if a and b are equals, they have the same hashCodes. That's what HashSet and HashMap need for their proper working.

If you override equals or hashCode, you must override the other one consistently, or HashSet/HashMap may fail, and anything else that uses both equals and hashCode.

Moral : it's absolutely fine to leave equals and hashCode unimplemented in almost any class. If you feel the urge to override one, proceed cautiously! Guard against duplicates when filling Sets.

Using TreeSet<domainclass>

If you want to use TreeSet<domainclass>, you need to make the domain class implement Comparable<domainclass>, and thus override compareTo.

But you must also override equals consistently, and because of that, override hashCode consistently too.

See PizzaTopping.java for an example. In PizzaTopping, it's nice to use a TreeSet<PizzaTopping> with equals based on the string name so that toppings are listed in alphabetic order.

Another pointer : use @Override when you do override these methods, to make sure you're actually overriding what you think you are. It's really easy to miss the mark, esp. on equals.

What’s provided in music1-setup

- DAO: music1-setup has DbDAO, AdminDAO, DownloadDAO, InvoiceDAO, LineItemDAO: you need to add UserDAO, ProductDAO (and TrackDAO, or handle Tracks in ProductDAO)
- Service: has ServiceException, AdminService (incomplete), you need to add UserService
- Presentation: UserApp, SystemTest: mostly written, you add service calls (AdminApp should work once you have finished AdminService)
- Config: MusicSystemConfig: mostly written, uncomment as progress

First steps in music1: from the assignment, Register.java for pa1a

Write a program Register.java (in package cs636.music.presentation, and thus in directory src/main/java/cs636/music/presentation of your music1 project) to insert a new user, directly, no User object, no "DAO" yet.

This brute force starter program follows the idea of using all the pieces of the needed technology as soon as possible. Feel free to copy code from JdbcCheckup.java and Java files in the music1-setup project.

Register.java should accept database information the way SystemTest does, but it has no input file. Don’t take any user input for the information about the new user but rather just invent values in the program.

Note that you can run Register using the already existent scripts runOnH2, runOnOracle, and runOnMysql.

First steps in pa1, after Register.java

In DAO, needed next: **UserDAO.java,** with needed support for registering a user**.**

For UserDAO class setup, look at InvoiceDAO: package, import, private Connection connection, constructor

To start with, can set UserDAO up with no methods, similarly UserService, build little object graph

Goal 1. Write class-level code for UserService and UserDAO, uncomment them in MusicSystemConfig, run SystemTest without crash.

Problem: are you sure they are really up and working?

How to see calls down the layers

Write stubs for methods that just print out when they are called:

UserDAO

```
void insertUser(String email) throws SQLException
{
    System.out.println("insertUser called, email = "+email);
}
```

UserService

```
void registerUser(String email) throws ServiceException
{
     System.out.println("registerUser called, email = " +email);
     userDAO.insertUser(email);
}
```

SystemTest: call registerUser for ureg command

Result: run SystemTest, see these calls **Now we know we're in business! That's Goal 2**

Real Code for DAO, Service

What we need: insert a new user. DAO should be generic support as far as possible, so suggest insertUser or createUser

But note that somewhere we need to check if a certain user is already there.

The emails are supposed to be unique, so that's a good thing to check.

So need findUser, given email, plus insertUser and let caller check

You need to do an insert, as you must have done in Register.java, but now we need to use real values and a good PK.

Inserting a new user with a good PK

New PK: use col in music_sys_tab:

```
create table music_sys_tab (
    invoice_id integer not null,    <--see code for using this in InvoiceDAO.java
    user_id integer not null,       <---next user id
    download_id integer not null,
    lineitem_id integer not null);
```

Want portable id generation (can't use auto-increment!)

What we want to happen in the DB:

```
select user_id from music_sys_tab;

update music_sys_tab set user_id = user_id + 1;
```

Inserting a new user, continued

Looking at InvoiceDAO, see separate method for finding the next id, so here could have (though it's up to you) a helper method:

private int findNextUserId () {...}

insertUser, two ways

public void insertUser(User user) throws SQLException; ←more OO
or public void insertUser(String firstName, String lastName, ...) throws SQLException;

public void insertUser(User user)throws SQLException;
--look at insertInvoice in InvoiceDAO

Service layer for adding a new user

void registerUser(String firstName, String lastName, ...) throws ServiceException
call finder for User, then insertUser if needed

Goal 3: SystemTest shows expected output for ureg command, and runH2Script showdb.sql (etc.) shows new users

Goal for pa1: nice service API

- So write stub methods of service API if no time to implement them.
- If not enough time, skip AdminApp
- But get SystemTest working in full if you can, and UserApp.

Junit and Java Annotations

- We’re using JUnit 5, which uses Java annotations, as did the older JUnit 4, but not the even older JUnit3, still in use in some places.
- We’ll see that Spring Boot uses annotation, as well as JPA, the object-relational Java API.
- So we need some expertise on annotation…
- Refs: see Wikipedia Java annotation, Java tutorial
- Useful tutorial

Annotations in Java: built-in ones

Java built-in annotations: notes to the Java compiler.

@Override - Checks that the method is an override. Causes a compile error if the method is not found in one of the parent classes or implemented interfaces. No effect on the execution if the code compiles

Example: PizzaTopping.java:

```
@Override
public int hashCode()
{
  return getToppingName().hashCode();
}
```

Note how @Override occurs just before the method header. The newline in between is just whitespace to the compiler, so it could be on the same line.
This is a **method annotation.**

More built-in Java annotations

Others for compiler:

@Deprecated - Marks the method as obsolete. Causes a compile warning if the method is used.

@SuppressWarnings - Instructs the compiler to suppress the compile time warnings specified in the annotation parameters, for a method or a statement or a whole class.

Example: @SuppressWarnings({"unchecked"}) when casting from raw to generic type

Others for Javadoc, another code processor:

@author - for pointing up a class's header comment on who wrote the code
@version – version no.

Note: once the code compiles, then the annotation has no effect on the *execution* of the code once its executing. It might be *selected* for execution based on an annotation being read by some tool such as JUnit.

Non-built-in annotations: notes to other code processors

- JUnit: @Test marks a method for JUnit to execute.
  - It doesn't affect how that code is executed.
  - JUnit finds what tests to execute by using the Reflection API and finding the compiled annotations in the .class files.
  - JUnit5 User Guide
- Coming up: @Entity, etc. for JPA
- You can write your own annotations for your own code processor
- Each annotation has a source file. See example in tutorial

How a JUnit test Executes

The Junit runtime is itself a Java object, with ref to the Junit test, which itself ref's the Calculator object under test:

<Junit runtime> → <CalculatorTest> → <Calculator>

The Junit runtime finds the @Tests in CalculatorTest and calls them, and they call the methods in Calculator, and JUnit tracks the results

How Junit (JUnit5 in our case) runs the JUnit test:

For each @Test method in the class:

@BeforeEach method executes: to bring object(s) to known starting state

The @Test method executes causing an experiment from that state

@AfterEach method executes

Look at Doc on Junit code for Calculator and CalculatorTest

Notes on CalculatorTest

Important idea: tests are independent, and all start from a known program state.
That's why above @BeforeTest clears the calculator, so nothing is left over from the last test.

Needed Libraries

All-dependencies Jar for non-Maven projects: junit-platform-console-standalone-1.8.0-M1.jar, available in the Maven repository. It also can be used directly from the command line, but it's not that convenient.

Maven setup: See pizza1's or music1-setup's pom.xml for two dependencies needed for JUnit5.

Eclipse integration To execute them, once the right libraries are in the project, all you need to do is right-click on the project and select Run as JUnit test, and all tests of the project will be run, showing green bar for success and red bar for failure.

Maven and CalculatorTest

- That simple project for Calculator and its CalculatorTest doesn't qualify for Maven use.
- Build Maven setup with `mvn archetype:generate -DgroupId=cs636 -DartifactId=junit-maven …`
- Add package cs636 to Calculator and CalculatorTest: same package, the Maven way, but housed in different directory systems:
  - Put Calculator.java in src/main/java/cs636/Calculator.java
  - Put CalculatorTest.java in src/test/java/cs636/CalculatorTest.java
- Modify pom.xml to look like pizza1's, with its Junit 5 entries.
- Run Junit tests by "mvn test"
- Create an eclipse project with "Open Projects from File System" and use Maven>update project
  - See 6 jars in Maven Dependencies
- Now can run tests by right-clicking project or package or file and "Run As> Junit Test"

JUnit and expected exceptions

Note how JUnit is *smart about exceptions*. A unit test should not need a try/catch to test code that can throw exceptions.

This handling has changed quite a bit between JUnit 4 and 5. JUnit5 requires Java 8 and here we are using a Java 8 lambda function with it, as expected:

Junit 4:

```
@Test(expected = ArithmeticException.class)
public void divideByZero() {
    calculator.divide(0);
}
```

JUnit5:

```
@Test
 public void divideByZero() {
   Assertions.assertThrows(ArithmeticException.class,
   () -> {
            calculator.divide(0);
         });
}
```

[GeeksForGeeks Tutorial on Lambda functions]

Look at badMakeOrder in PizzaOrderDAOTest1.java as another example that *expects* an exception.

Handling Dependent Objects in Tests

- Recall earlier discussion in class 7 of using dependency injection with dependent objects. The hard part of unit testing is dealing with dependent objects.
- If one object does a new B() to create the dependent object, we're stuck with using a real B in testing, so it's hard to test A alone.
- But if DI is in use, we have more options. One way is to use “mock objects”, objects of a class that implements the same interface as the actual dependent, or is a subclass of it. The implementation can be fake, that is, it can know what the test will ask of it, and just cook up a return for that.
- We have H2 as a wonderful mock DB, so we can test our DAOs with it, and once they are tested, test our service objects. We see that layering is helpful for testability.

Pizza1's PizzaOrderDAOTest1

-- a JUnit 5 test class using H2 as a mock

- The @BeforeEach method creates db (the DbDAO object) and pizzaOrderDAO objects and menuDAO objects (the "fixture") separately for each test
- The @AfterEach method closes down the whole database
- Remember this happens for every test!
- It's really fast for H2, so no problem.
- If we used Oracle or mysql for this, we would start using @BeforeAll and @AfterAll to set up one connection and close it after all the tests. Creating a Connection for a real database takes some time (unless Connection pooling is in use, to be covered laterr).