

Transactions

Packages

We have seen how packages allow us to organize code to make the software architecture clear:

- one package for each layer: presentation, service, dao
- one package for domain classes
- one package for building the system at startup: config

We also need to know how packages actually work as part of Java.

Look at [Java tutorial track on packages](#).

At first, packages appear to be hierarchical, but they are not. In music1, we have `cs.music.service` and `cs.music.service.data`, but they are independent packages.

Packages, continued

Definition: A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types.

Name space management: `cs636.pizza.service.UserService.java` provides a unique name for `UserService`, avoiding conflicts if we bring in other code with its own `UserService.java`

Access protection: package protection, specified by lack of other keyword

`public class Foo` vs. `class Foo`: `class Foo` is package private (not in use in our projects)

This is our only package-private method: in `DbDAO`, provides `Connection` object to other DAOs

```
// package protection: no need to call this from service layer
Connection getConnection() {
    return connection;
}
```

Transactions

Currently we are using auto-commit, the JDBC default.

Auto-commit means Commit each statement.

The database itself needs real commits, so JDBC sends commit commands for us in auto-commit.

This works fine for single-user work, but concurrent user sessions can get into trouble because of interleaved actions that should have been separated.

If we need to bundle statements together for a certain action, we need to turn off auto-commit and start using `conn.commit()` and `conn.rollback()`.

```
conn.setAutoCommit(false);
```

Then we can design multi-statement transactions. For example, taking money out from one account and putting it in another.

What can go wrong?

Standard ex: lost update: 2 concurrent processes both adding to an account, only one increment shows (R = read, W=write)

User A

User B

R old amt

R old amt

Add delta1

Add delta2

W new amt

W new amt – overwrites other's update

Another Ex: Suppose in pizza we had code for new PizzaTopping id: find max(id) in toppings, use id+1 in insert

2 admins adding a Topping can get same new id causing failure of one (PK violation)

R old ids

R old ids

Add one

add one

end up with same id, one insert fails with PK violation

What could go wrong, continued

Our way:

Read pizza_sys_tab, see next topping id Read pizza_sys_tab, see same next topping id

Update row to set id = id+1

Update row to set id = id+1

Here the updates execute atomically, so the id advances by 2, but the insert fails on PK violation.

If we put this into a (ACID) transaction, the reads can proceed together, but the fact that a read has occurred will make the writes wait, causing a deadlock, and one will be aborted. It can be retried and should succeed.

DB transactions were invented to solve these problems. They bind together several actions into one.

ACID properties

Transactions, the deal:

- We programmers specify actions for the transaction by putting commits & rollback at right place
- The DB will execute these tx's as if they were executed one-by-one (serially)

Serializable transactions = textbook tx's

Famous ACID properties:

Atomic – happen in whole or not at all (undo wipes out uncommitted changes)

Consistent – (if each tx code maintains a constraint, so will concurrent tx's)

Isolation – each tx runs as if only one on system

Durable – once committed, data is held across system crash, disk crash

Isolation levels

In real life, “isolation levels” defined by SQL standard are in common use.

There are 4 isolation levels, 2 of which are common:

- Read committed (RC) – default level of most DB’s*, commonly used
- Serializable -- full ACID properties

Both are atomic and durable

The difference is in “isolation”.

- At RC, one tx’s available data can be affected by another concurrent tx: they are imperfectly isolated from one another.
 - A tx only sees committed data, so not total junk, but it could be very newly committed, since this tx started.

*mysql defaults to Repeatable Read, the next isolation level up

Serializable vs. Read Committed

Serializable: DB in effect freezes (until commit/rollback) all data accessed (R or W) by a transaction. This prevents other Tx's from changing it. The tx can reread data and see the same values.

Read committed: DB freezes data *changed* by this Tx, but not data merely read by this Tx. Data can be read by this transaction, then changed by another tx, thus changing the universe this tx runs in.

Note “freezes” is not a standard term, but I think it captures what’s happening.

Let’s look at the problems discussed above...

RC allows lost update unless special coding is used – “select for update” – reads with write locking

Or for SQL `update bal = bal + delta` in a single update statement

-this is guarded by locks on the updated record, so other tx’s can’t intervene

RC uses much less locking, runs faster, but can have real problems for apps with needs for coordinated changes of multiple entities in the DB.

Transactions and UI

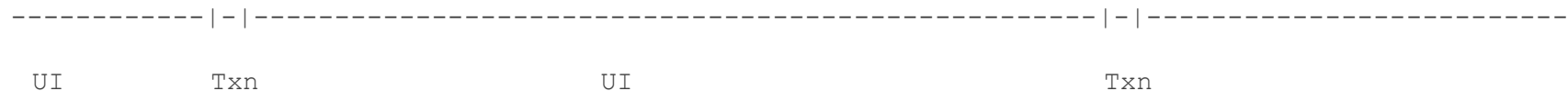
Tx's use locks, which hold off other Tx's progress, so Tx's need to run quickly, say ≤ 50 ms $<$ human response time

DB App Rule: no UI during a Tx. (in production)

Tx's should be short, say less than 50 ms (milliseconds, 1/1000's of seconds)

Transactions over time: UI takes much more time than properly short Tx's, looks like this:

Timeline for one user: mostly think time



We see that sometimes the short Tx's overlap in time (for different users), though not very commonly at low load. We say they are concurrent. That's when data problems can occur at read-committed isolation, and deadlocks can happen in Serializable.

Transactions in our layered System

Our setup has layers. The execution goes like this:

Presentation = UI, calls service API, calls DAO,

returns to Service, [calls DAO, returns],

returns to Presentation (more UI here)

So if the service layer method contains the Tx, we will be following the DB App Rule

(Note: if the DAO call contained the Tx, that would also satisfy the Rule, but would be overly restrictive. The service call may want to do two DAO calls in one Tx.)

The service layer is in charge, so it defines what actions are bundled together into Tx's

Conclusion: **put commit/rollback in service layer**

Put commit/rollback in service layer

Sounds funny – we are putting all the SQL into the DAO and commit is a DB action...

But start & end Tx are more abstract than “select pid from T” in DAO

Conclusion: **start and end Tx, in each single service layer call.** This is what we’ll do in pizza2 and (eventually) music3.

Pizza2: client-server but with transactions and object-relational (JPA) support.

Pizza: has several DB-changing service API calls: here are some

makeOrder

addTopping, etc.

markNextOrderReady

receiveOrders

These all will be turned into transactions in pizza2 (JPA client-server) and pizza3 (web app). Read-only actions also.

Summary on Transactions

- Need for transactions to prevent problems in shared data.
- DB is the expert on guarding shared data
- DB uses locking, causing waits (aborts sometimes)
- Multiple Transaction can read data simultaneously, but only one write at a time is allowed
- In an ACID transaction (i.e. using serializable), the DB data appears frozen (unchanged by other transactions), other transactions are waiting to access the data your transaction is using.
- Even a read-only transactions have this effect of freezing data, at serializable isolation level
- CS634 covers this in detail
- DB App Rule: No UI inside a Tx

Transactions and JDBC

Messy Exceptions, messy transactions

Reference: [Oracle JDBC tutorial](#), see its section on “Using Transactions”

JDBC default is “auto-commit”, commit on each statement

JDBC is still in use when we start using JPA: JPA uses it

// Connection class has transaction methods. We take over on Tx's by turning off auto-commit:

```
conn.setAutoCommit(false)
```

Now no commits until we call commit

We can also request serializable Tx's: we override the default isolation level (almost certainly READ_COMMITTED, unless using mysql)

```
conn.setTransactionIsolation( Connection.TRANSACTION_SERIALIZABLE )
```

JDBC Transactions

db actions for Tx, in JDBC:

```
conn.commit();    // or conn.rollback()
```

```
conn.setAutoCommit(true); // We won't do this, but other apps could
```

Note no startTransaction here. In JDBC, as in embedded SQL in C, a transaction starts at the first use of database data.

(In pizza2, we are using JPA over JDBC, so we use its EntityTransaction objects. For an EntityTransaction tx, tx.begin(), tx.commit(), tx.rollback().)

In pizza3, we'll be back to using JDBC directly, with transactions, so we will be using conn.commit(), etc.

Coding transactions using JDBC

The general setup: no start-tx, so just

- Do DB actions
- Commit or Rollback

Note we can rollback based on data we see, not just DB problems. For example, we figure out the user is not authorized to do something.

The Tricky Part with JDBC: Cleanly aborting a transaction

Some DB problem --> SQL exception

Ex. Insert fails. The DB changes we've made are now in an *indeterminate state*, and still locked up to avoid affecting other transactions (other than slowing them down).

So we have to do a **rollback** to get to a determined DB state for our data.

Handling DB errors in JDBC

Ex. Insert fails. The DB changes we've made are now in an *indeterminate state*, and still locked up to avoid affecting other transactions (other than slowing them down).

So we have to do a **rollback** to get to a determined DB state for our data.

Further complication: JDBC rollback() can throw SQLException

Suppose something (like an insert) has thrown a SQLException e1. We try a rollback and it throws SQLException e2.

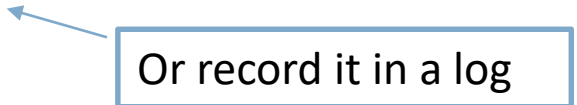
What to do with e2??? It's usually a lost connection. So we'll discard e2 and preserve e1, hopefully a more useful indication of the real problem.

A rare case of justifiable exception-squelching. (If we had a log, we could print its message there.)

Rollback after Exception code

Rollbacks are handled with `rollbackAfterException(Connection conn)` which does the "extra" try-catch needed to handle possible throws by rollback:

```
// The caller should issue its own exception based on the
// original exception (or do retry)
public static void rollbackAfterException(Connection conn) {
    try {
        conn.rollback();
    } catch (Exception e) {
        // discard secondary exception--probably server can't be reached
    }
}
```



A similar method is used in `pizza2` and `pizza3`, in their DAOs.

Transfer.java in the jdbc directory

We looked at [Transfer.java](#), available in \$cs636/jdbc along with JdbcCheckup.java.

This program turns off auto-commit, sets serializable isolation, and handles transfers between accounts.

Note that the method doCustomerTransfers has presentation code.

The method doTransfer works like a service-layer method, defining when transactions begin and end.

doTransfer calls transfer to do the actual database actions (other than commit and rollback), so transfer is like a DAO method.

Simplified code

If retries are not wanted, doTransfer can be simplified to just:

```
public static void doTransfer(Connection conn, String fromAccount,
    String toAccount, double dollars) throws SQLException {
    try {
        transfer(conn, fromAccount, toAccount, dollars);
        conn.commit();
        return; // success
    } catch (SQLException e) {
        rollbackAfterException(conn); // rollback, ignore any rollback exception
        throw e; // notify caller of problem
    }
}
```

Notes on Simplified Code

- In this simplified code the customer transfers would be cut short and fail by a deadlock or serialization abort.
- Note that these aborts are usually caused by transient conditions, so a retry can be successful and cause no problem to the presentation environment.
- Also note that money is represented by double here, not a good practice. Should use BigDecimal as we do in music's Product.

Service layer code pattern

Here is another snippet, showing what we will be doing with a DAO-generated Exception: create and throw a service-layer exception object whenever the DAO throws up to this code.

```
// starting with good Connection conn, in service-layer code
try {
    dao.doDBWork(conn); // this DAO method can throw SQLException
    dao.commit(conn);   // success
} catch (SQLException e) {
    dao.rollbackAfterException(conn);
    throw new ServiceException("DBWork failed", e);
} finally {
    conn.close();
}
```

Of course there are no explicit transactions in `pizza1` or `music1`, so we don't see throws from rollback there

DAO SQLException → ServiceException

Here Connection should be a local variable, since we don't want state in the fields of the Service layer. Let's defer the details of getting this to have a good value. Clearly the DAO is involved.

We see the pattern there: catch Exception e1, throw ServiceException e

Here, as in pizza1, e is created by new ServiceException(" useful message", e1)

The "cause"

End up with a Service Exception e, with saved cause e1, sent up to the presentation layer code.

This means we can get e1 from e by e1 = e.getCause();

Note that getCause() is a method of Exception, so this is a general mechanism to retrieve the exception provided in the constructor of this Exception.

We can use this in debugging: it's important to know the real reason something failed.

See method exceptionReport(Exception e) in PizzaSystemConfig

Object graph for SQLException saved in ServiceException

SQLException as cause of ServiceException, in pizza1/pizza2: two objects with a ref between:

```
<ServiceException>    e
    |
    |    ref in ServiceException
    v
<SQLException>       e1 = e.getCause()
```

Both these exceptions have toString()s that provide their messages, so System.out.println(e) gives the message provided in the constructor like "Order can not be placed ", while System.out.println(e.getCause()) could be "No such table pizza_orders".

Next time: We need to know more about HTTP.

HTTP: read Chap. 18 to pg. 555.