SQL Programming

Lecture 5

Outline

1. General approaches
2. Typical programming sequence

General Approaches

- **SQL via API**
- Embedded SQL
  - SQLJ
- DB Programming Language
  - PL/SQL, T-SQL
- Hybrid
  - MS Access, Filemaker

SQL via API

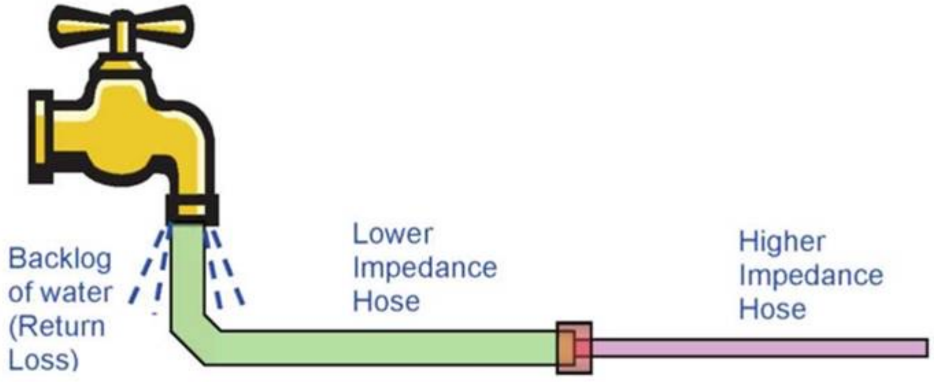
Most common approach:

```
PreparedStatement stmt = conn.prepareStatement(
   "SELECT LASTNAME" +
   " , FIRSTNAME" +
   " , SALARY" +
   " FROM EMPLOYEE" +
   " WHERE SALARY BETWEEN ? AND ?"
);
stmt.setBigDecimal( 1, min );
stmt.setBigDecimal( 2, max );
ResultSet rs = stmt.executeQuery();
while ( rs.next() ) {
  lastname = rs.getString( 1 );
  firstname = rs.getString( 2 );
  salary = rs.getBigDecimal( 3 );
  // Print row...
}
rs.close();
stmt.close();
```

Issues with Accessing SQL via API

- Impedance mismatch
  - Object-relational mapping
- DBMS abstraction layer
- Cursors
- Injection attacks

Impedance Mismatch



In this context, refers to several issues that arise when OO language interacts with RDBMS

- Differences in data types
- Query results as row/column
- Limited compile-time error detection w.r.t. SQL

Object-Relational Mapping (ORM)

Common technique to convert between incompatible systems (e.g. objects and RDBMS rows/columns)

```
part = new Part();
part.name = "Sample part";
part.price = 123.45;
part.save();
```

```
INSERT INTO parts (name, price) VALUES ('Sample part', 123.45);
```

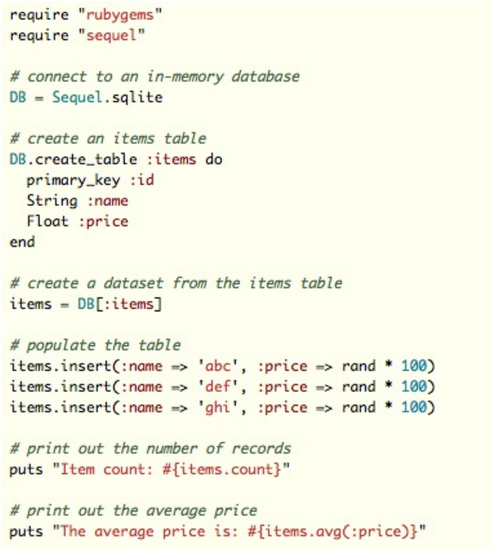
Database Abstraction Layer

- Most database systems have native APIs for several programming languages

- To ease software development, there are database abstraction efforts
  - Libraries: JDBC (Java), MDB2 (PHP), SQLAlchemy (Python)
  - Middleware: ODBC

- Varying degree of abstraction from DBMS/SQL

- Works well for many applications; efficiency and/or access to specific functionality



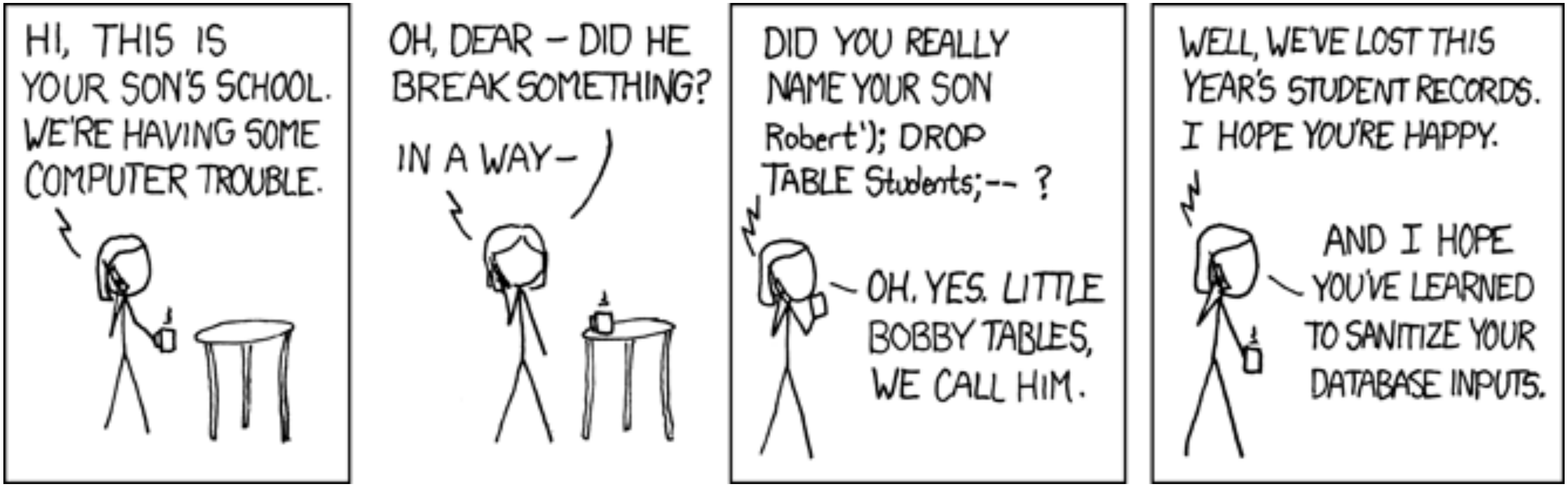
```
require "rubygems"
require "sequel"

# connect to an in-memory database
DB = Sequel.sqlite

# create an items table
DB.create_table :items do
  primary_key :id
  String :name
  Float :price
end

# create a dataset from the items table
items = DB[:items]

# populate the table
items.insert(:name => 'abc', :price => rand * 100)
items.insert(:name => 'def', :price => rand * 100)
items.insert(:name => 'ghi', :price => rand * 100)

# print out the number of records
puts "Item count: #{items.count}"

# print out the average price
puts "The average price is: #{items.avg(:price)}"
```

Cursors

- Libraries typically offer two types of access to query results (i.e. result set)
  - All at once (e.g. in an array/data structure)
  - Row-by-row

- The latter may be required for larger results, typically facilitated by a **cursor** data structure (can be thought of as a pointer to a single row within a larger set, similar to iterator)
  - Library may optimize for access patterns (e.g. read-only, forward-only, etc.)

SQL Injection Attacks ala XKCD

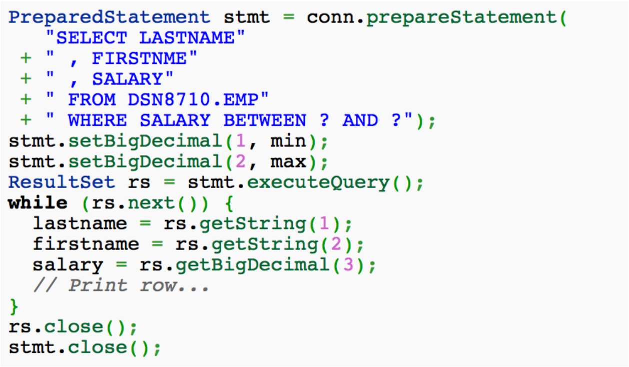

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Preventing SQL Injection

- Whenever user inputs interact with SQL, sanitizing is a vital security concern
  - Parameterization API
    - Use *prepared* statements (or stored queries); bind value via function call, API automatically escapes appropriate to DBMS
  - Value escaping API
    - Make sure string to be appended is properly quoted to prevent unintended leakage
- Principle of Least Privilege
  - Database user should only be allowed to access/change what is absolutely necessary; optionally use different users for different classes of operation

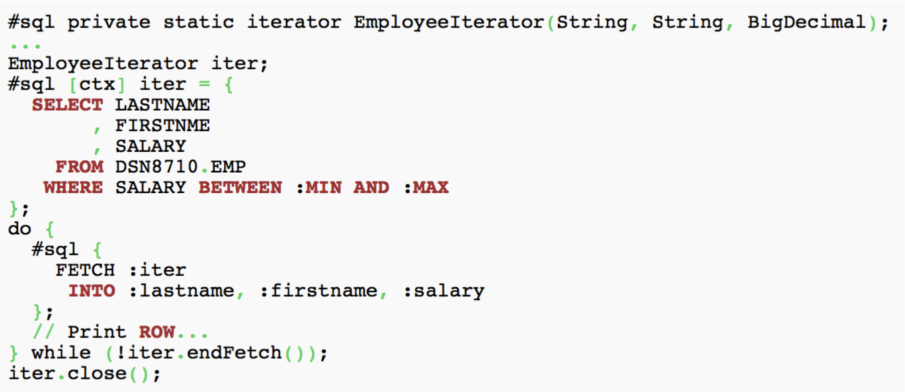
Embedded SQL

Insert [typically prefixed] code directly into source; compiler auto-generates DBMS-specific code



```
PreparedStatement stmt = conn.prepareStatement(
   "SELECT LASTNAME"
 + " , FIRSTNME"
 + " , SALARY"
 + " FROM DSN8710.EMP"
 + " WHERE SALARY BETWEEN ? AND ?");
stmt.setBigDecimal(1, min);
stmt.setBigDecimal(2, max);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
  lastname = rs.getString(1);
  firstname = rs.getString(2);
  salary = rs.getBigDecimal(3);
  // Print row...
}
rs.close();
stmt.close();
```

vs.



```
#sql private static iterator EmployeeIterator(String, String, BigDecimal);
...
EmployeeIterator iter;
#sql [ctx] iter = {
  SELECT LASTNAME
       , FIRSTNME
       , SALARY
    FROM DSN8710.EMP
   WHERE SALARY BETWEEN :MIN AND :MAX
};
do {
  #sql {
    FETCH :iter
     INTO :lastname, :firstname, :salary
  };
  // Print ROW...
} while (!iter.endFetch());
iter.close();
```

DB Language (SQL/PSM)

Store Procedures

```
//Function PSM1:
CREATE FUNCTION Dept_size(IN deptno INTEGER)
RETURNS VARCHAR [7]
DECLARE No_of_emps INTEGER ;
SELECT COUNT(*) INTO No_of_emps
FROM EMPLOYEE WHERE Dno = deptno ;
IF No_of_emps > 100 THEN RETURN "HUGE"
  ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
  ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
  ELSE RETURN "SMALL"
END IF ;
```

Typical Programming Sequence

1. Connect to DBMS
   - URL, database name, user/pw, driver
   - Sometimes *persistent* for performance

2. Arbitrary interactions
   - Transactions via SQL

3. Close the connection

Query Sequence

1. Generate SQL
   – Could be static or composed of algorithmic/user-contributed parts

2. Execute

3. Get results

Prepared Query Sequence

1. Generate parameterized SQL
   - Could be static or composed of algorithmic parts (typically nothing user-contributed)

2. Bind values to SQL parameters
   - Could be static or algorithmic/user-contributed

3. Execute

4. Get results

Summary

- You now have a general framework for writing a program that interacts with a database via an API
  - Connect, transactions, close
    - [Prepare] SQL, [bind values,] execute, get results

- Remember to be cautious from an efficiency and security perspective (more next time)
  - Database abstraction, ORM
  - SQL Injection attacks