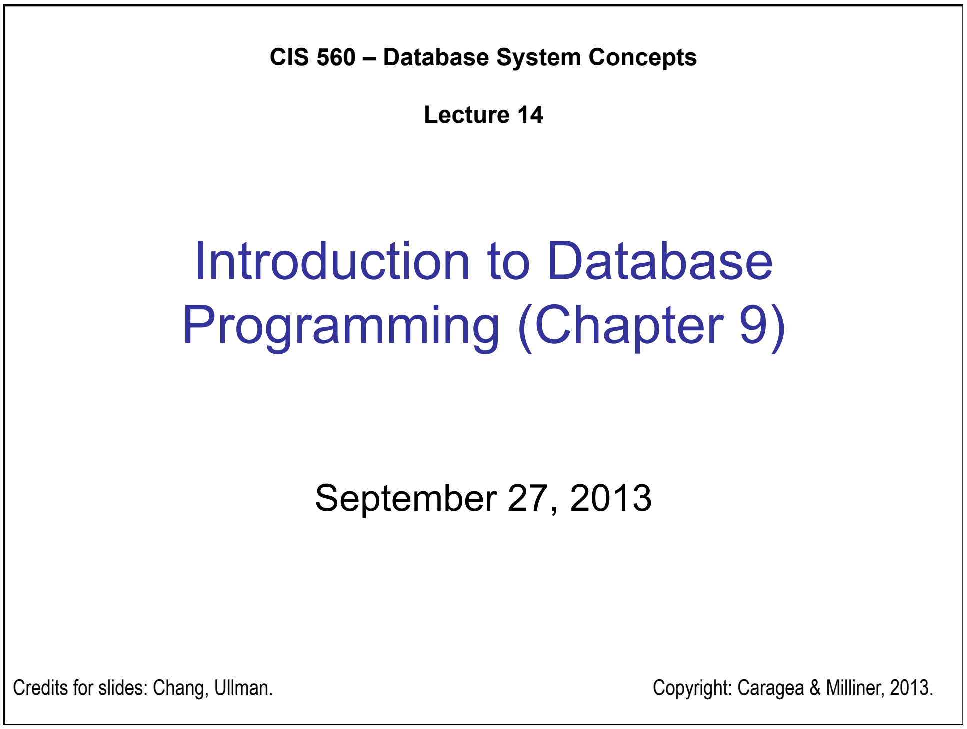


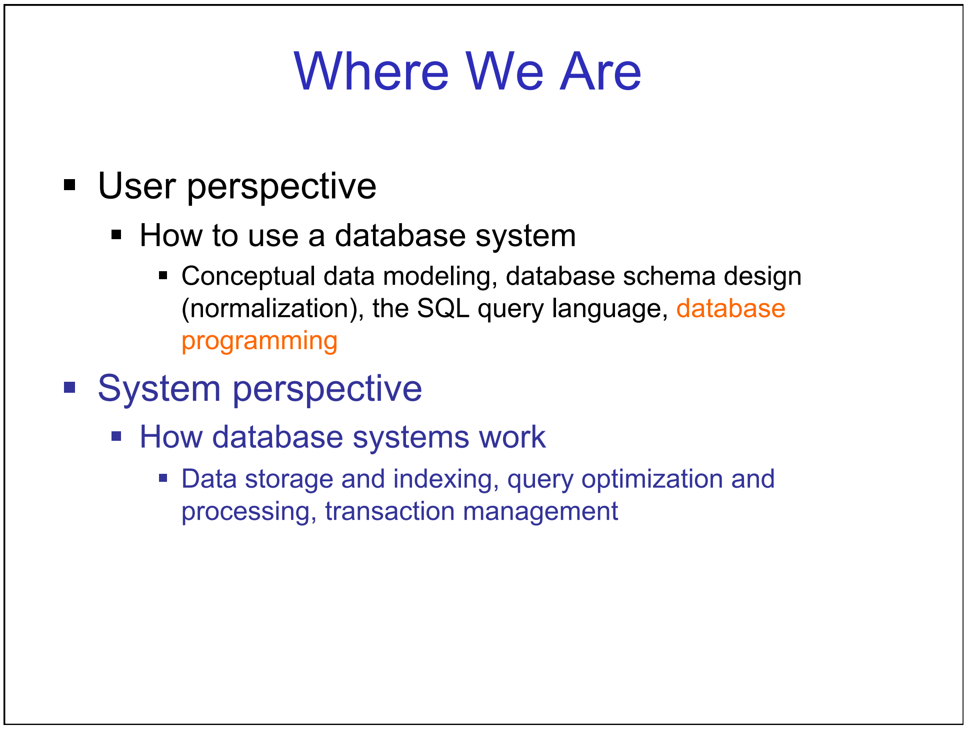
CIS 560 – Database System Concepts

Lecture 14

Introduction to Database Programming (Chapter 9)

September 27, 2013

Credits for slides: Chang, Ullman.



Where We Are

- User perspective
  - How to use a database system
    - Conceptual data modeling, database schema design (normalization), the SQL query language, database programming
- System perspective
  - How database systems work
    - Data storage and indexing, query optimization and processing, transaction management

SQL in Real Programs

- We have seen how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.
- SQL is a very high-level language - **not** intended for general-purpose computations.
- Most of the time, we need to write conventional programs that interact with SQL.

Three-Tier Architecture

A common environment for using a database has three tiers of processes:

1. *Web servers* - talk to the user.
2. *Application servers* - execute the business logic.
3. *Database servers* - get what the application servers need from the database.

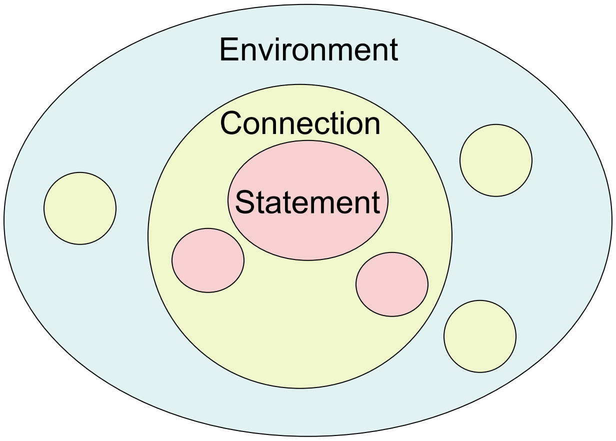
Example: Amazon

- Database holds the information about products, customers, etc.
- Business logic includes things like “what do I do after someone clicks ‘checkout’?”
  - Answer: Show the “how will you pay for this?” screen.

Environments, Connections, Statements

- The database is, in many DB-access languages, an *environment*.
- Database servers maintain some number of *connections*, so application servers can ask queries or perform modifications.
- The application server issues *statements*: queries and modifications, usually.

Diagram to Remember



DB Programming Options

- Outside DBMS: use SQL together with general-purpose programming languages.
  - SQL statements are embedded in a host language (e.g., C).
    - Directly embedded SQL, not widely used [70’s -]
  - Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB).
    - API approach - Call-Level Interface [90’s -]
- Inside DBMS: augment SQL with constructs from general-purpose programming languages.
  - Code in a specialized language is stored in the database itself (e.g., PSM, PL/SQL).
    - Stored procedures/functions [80’s -]

Basic Paradigm

- Connect to a DB server.
- Say what database you want to use.
- Assemble a string containing an SQL statement.
- Get the DBMS to prepare a plan for executing the statement.
- Execute the statement.
- Extract the results into variables in the local programming language.

Programming Outside the DBMS

Embedded SQL

- A standard for combining SQL with different languages.
- Key idea: Use a preprocessor to turn SQL statements into procedure calls that fit with the host-language code surrounding.
- All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.

```
EXEC SQL INSERT Studio(name, address)
          VALUES (:studioName, :studioAddr);
```

API Approach

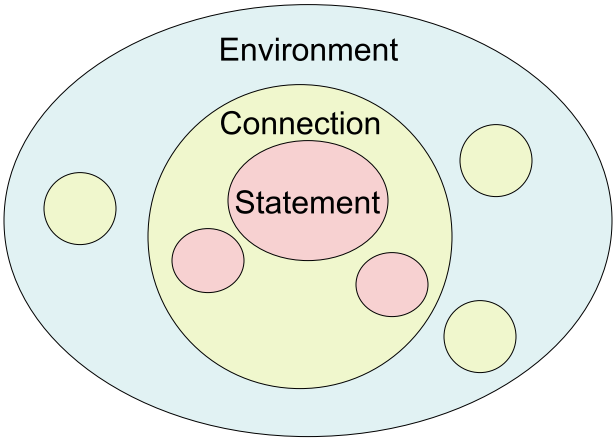
- Instead of making use of a preprocessor to translate EXEC SQL … statements into function calls (as in embedded SQL), the programmer uses a library of functions or classes and calls them as part of an ordinary C or Java program.
- SQL commands are sent to the DBMS at runtime.
- These API's are based on a standard called SQL/CLI = "Call-Level Interface."
  - C + CLI
  - Java + JDBC
  - PHP + PEAR/DB

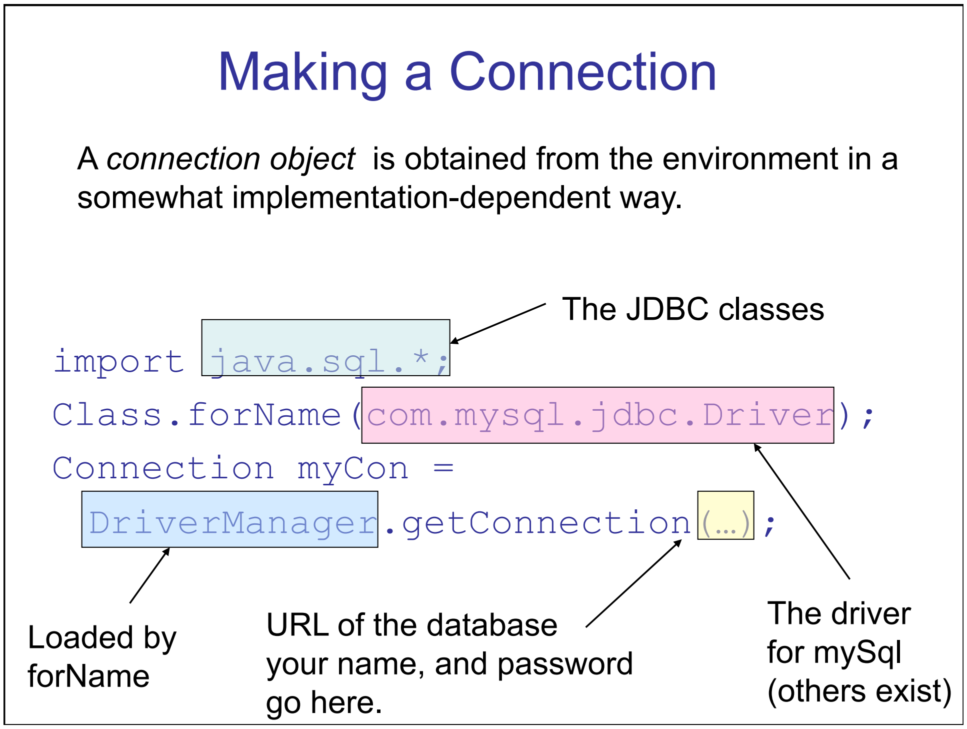
Open Interface: JDBC

- Database specific API makes the program dependent to one DBMS.
- Open interfaces solve this problem:
  - Designed by a third party, e.g. the creator of the language.
  - Implemented by DBMS vendors.
  - Used by DB programmers.
- Programmers do not have to change their code to work with another DBMS.

http://java.sun.com/products/jdbc/overview.html

Environment, Connection, Statements





Making a Connection

A *connection object* is obtained from the environment in a somewhat implementation-dependent way.

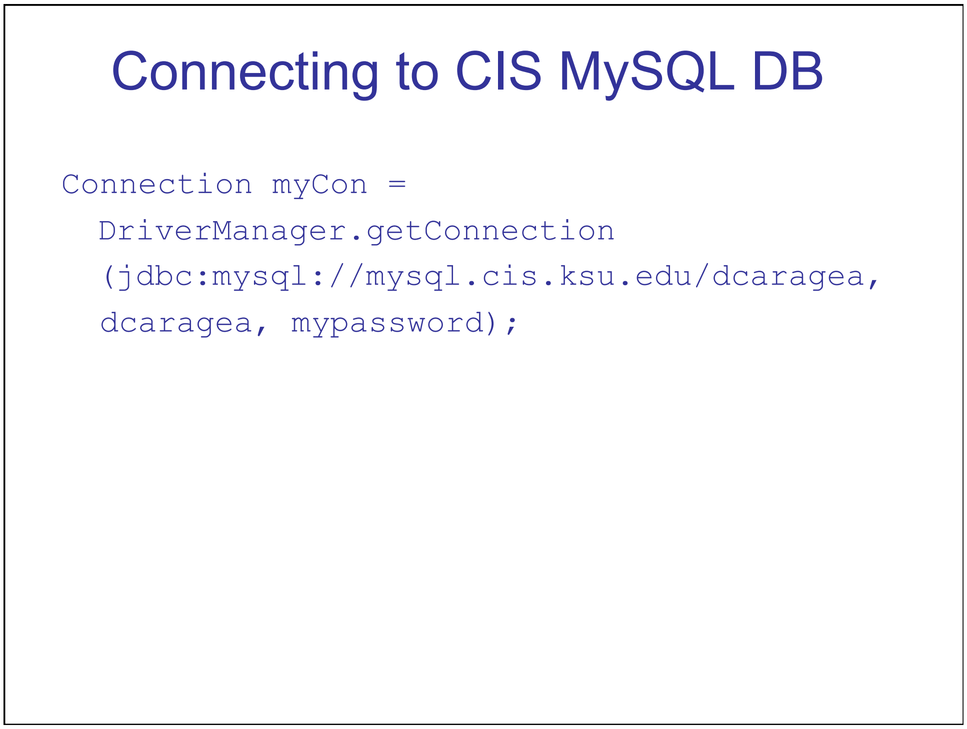
```
import java.sql.*;
Class.forName(com.mysql.jdbc.Driver);
Connection myCon =
  DriverManager.getConnection(…);
```

The JDBC classes

Loaded by forName

URL of the database your name, and password go here.

The driver for mySql (others exist)



Connecting to CIS MySQL DB

```
Connection myCon =
  DriverManager.getConnection
  (jdbc:mysql://mysql.cis.ksu.edu/dcaragea,
  dcaragea, mypassword);
```

JDBC Statements

- JDBC provides two classes:
  - `Statement` – an object that can accept a string that is an SQL statement and can execute such a string.
  - `PreparedStatement` – an object that has an associated SQL statement ready to execute.
- Why `PreparedStatement`? Performance!
  - The SQL command is stored in DBMS after the first call.

Creating Statements

The `Connection` class has methods to create `Statement` and `PreparedStatement`.

```
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
  myCon.prepareStatement(
      "SELECT Address FROM Supplier " +
      "WHERE Supplier_Name = 'John Smith'"
  );
```

String concatenation

Supplier(Supplier_ID, Supplier_Name, Address)

Executing SQL Statements

- JDBC distinguishes queries from modifications (which it calls “updates”).
- `Statement` and `PreparedStatement` have methods `executeQuery` and `executeUpdate`.
  - For `Statement`, these methods have one argument: the query or modification to be executed.
  - For `PreparedStatement`: no argument.
- Programmer handles errors using `SQLException` class.

Example: Update

- `stat1` is a `Statement`.
- We can use it to insert a tuple as:

```
stat1.executeUpdate(
   “INSERT INTO Supplier ” +
   “VALUES(‘S4’, ‘Mary’, ‘12 Goodwin St.’)”
);
```

Example: Query

- stat2 is a `PreparedStatement` holding the query

```
“SELECT Address FROM Supplier WHERE
Supplier_Name = ‘John Smith’“
```

- `executeQuery` returns an object of class `ResultSet`.

```
ResultSet List = stat2.executeQuery();
```

Accessing the ResultSet

- An object of type `ResultSet` is a cursor.
- Method `next`() advances the “cursor” to the next tuple.
  - The first time `next`() is applied, it gets the first tuple.
  - If there are no more tuples, `next`() returns the value FALSE.

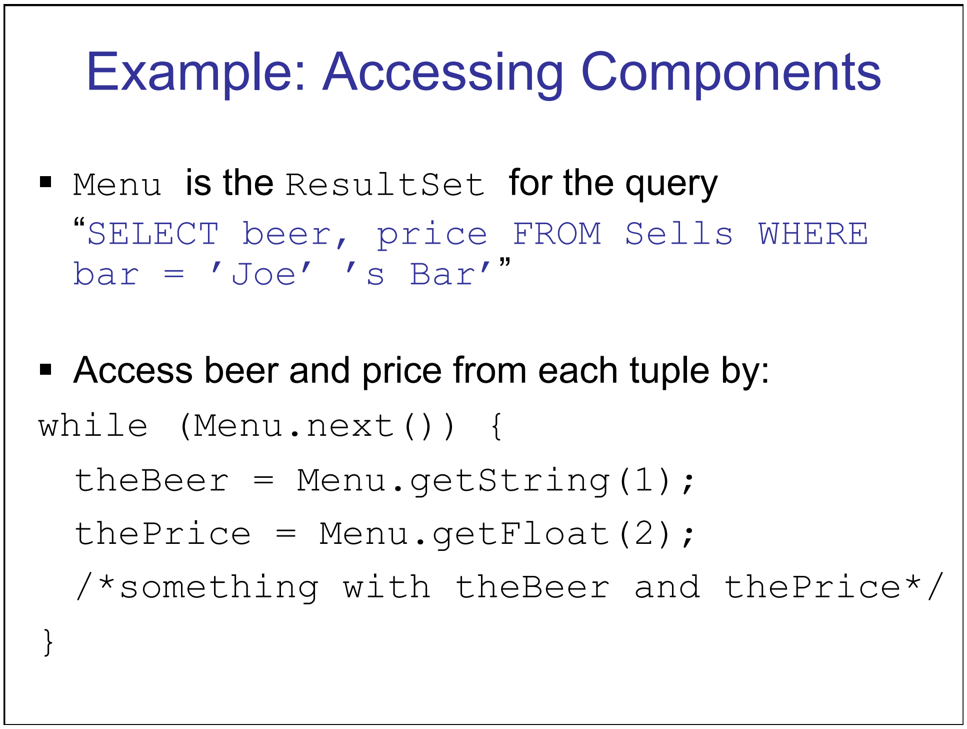
Accessing Components of Tuples

- When a ResultSet is referring to a single tuple, we can get the components of that tuple by applying certain methods to the ResultSet.
- Method getX(i), where X is some type, and i is the component number, returns the value of that component.
  - The value must have type X.

Example: Accessing Components

- List is the ResultSet for the query “SELECT Address FROM Supplier WHERE Supplier_Name = ‘John Smith’”.
- Access the address from each tuple by:

```
while(List.Next()){
  theAddress = List.getString(1);
     /*do something with the address */
}
```

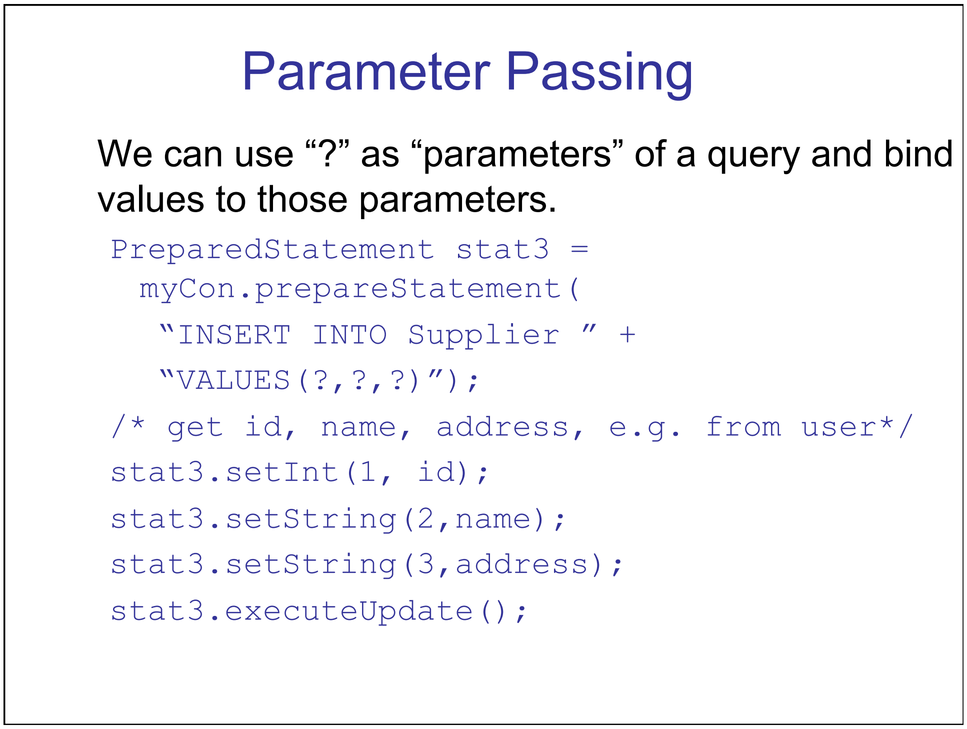


Example: Accessing Components

- `Menu` is the `ResultSet` for the query
  “SELECT beer, price FROM Sells WHERE bar = ’Joe’ ’s Bar’”

- Access beer and price from each tuple by:

```
while (Menu.next()) {
  theBeer = Menu.getString(1);
  thePrice = Menu.getFloat(2);
  /*something with theBeer and thePrice*/
}
```



Parameter Passing

We can use “?” as “parameters” of a query and bind values to those parameters.

```
PreparedStatement stat3 =
  myCon.prepareStatement(
   “INSERT INTO Supplier ” +
   “VALUES(?,?,?)”);
/* get id, name, address, e.g. from user*/
stat3.setInt(1, id);
stat3.setString(2,name);
stat3.setString(3,address);
stat3.executeUpdate();
```

SQL Injection

SQL Injection Attacks by Example
http://www.unixwiz.net/techtips/sql-injection.html

What is an SQL Injection Attack?

- Many web applications take user input from a form
- Often this user input is used literally in the construction of an SQL query submitted to a database. For example:
  - SELECT productdata FROM table WHERE productname = ‘***user input product name***’;
- An SQL injection attack involves placing SQL statements in the user input

An Example SQL Injection Attack

Product Search: **blah‘ OR ‘x’ = ‘x**

- The application
  - Reads the input into a variable `prodname`
  - The input is put directly into the SQL statement

```
String query = “SELECT prodinfo FROM prodtable WHERE prodname = ‘”
  + prodname + “‘”;
```

- Creates the following SQL query:

```
SELECT prodinfo FROM prodtable WHERE prodname = ‘blah‘ OR ‘x’ = ‘x’;
```

  - Attacker has now successfully caused the entire database to be returned.

A More Malicious Example

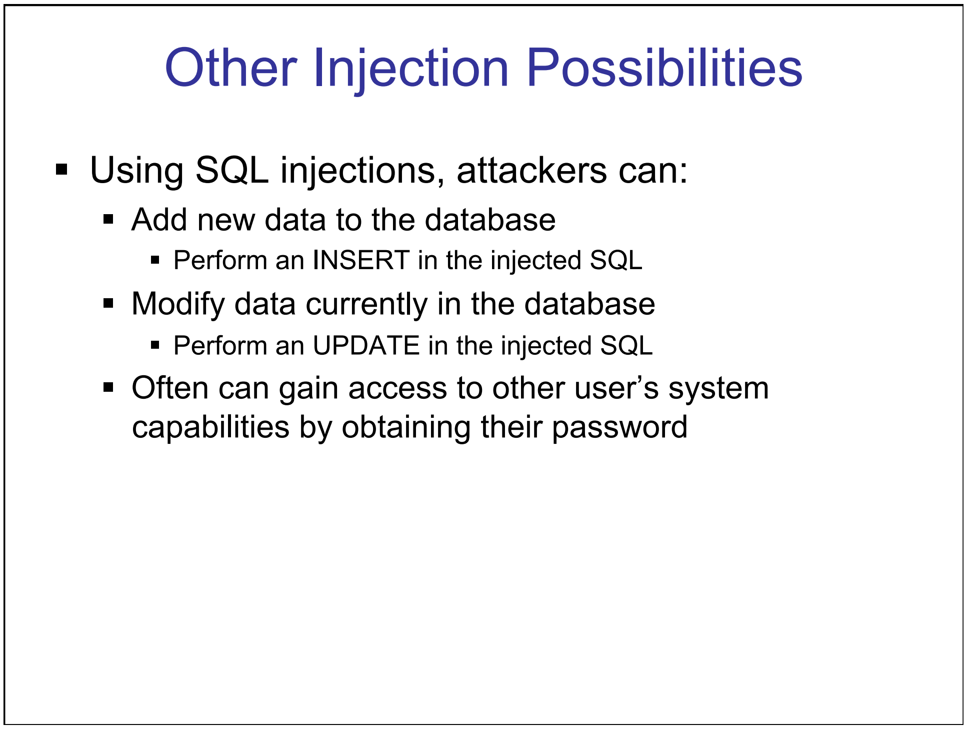
- What if the attacker had instead entered:

```
blah‘; DROP TABLE prodinfo; --
```

- Results in the following SQL:

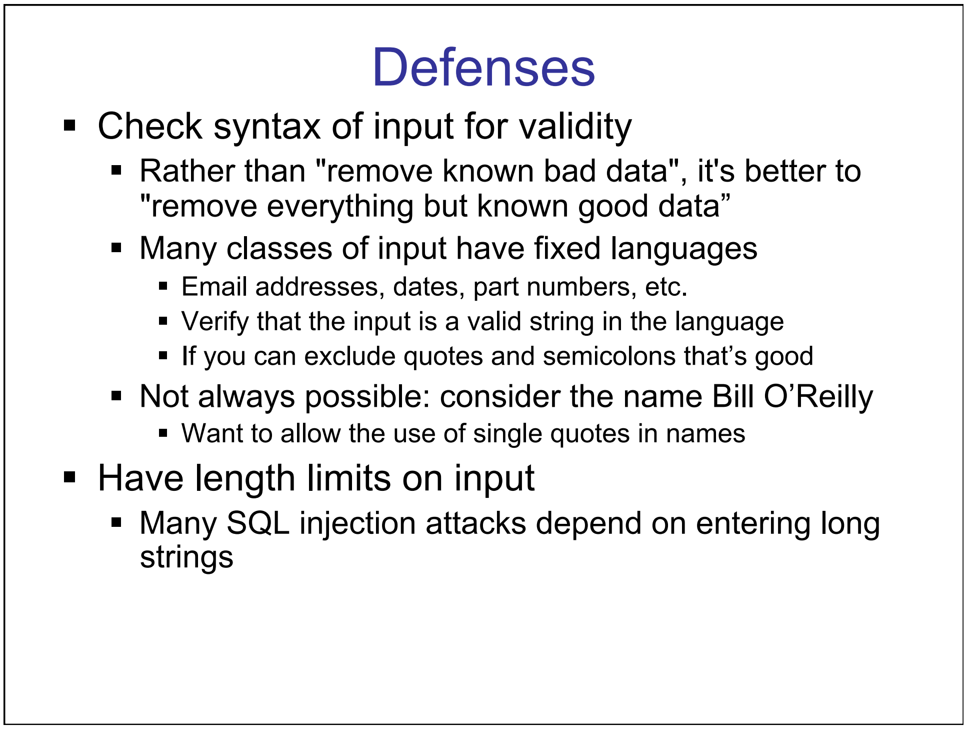
```
SELECT prodinfo FROM prodtable WHERE prodname = ‘blah’;
DROP TABLE prodinfo; --’
```

  - Note how the comment (--) consumes the final quote.
- Causes the entire table to be deleted
  - Depends on knowledge of table name
  - This is sometimes exposed to the user in debug code called during a database error
  - Use non-obvious table names, and never expose them to user



Other Injection Possibilities

- Using SQL injections, attackers can:
  - Add new data to the database
    - Perform an INSERT in the injected SQL
  - Modify data currently in the database
    - Perform an UPDATE in the injected SQL
  - Often can gain access to other user’s system capabilities by obtaining their password



Defenses

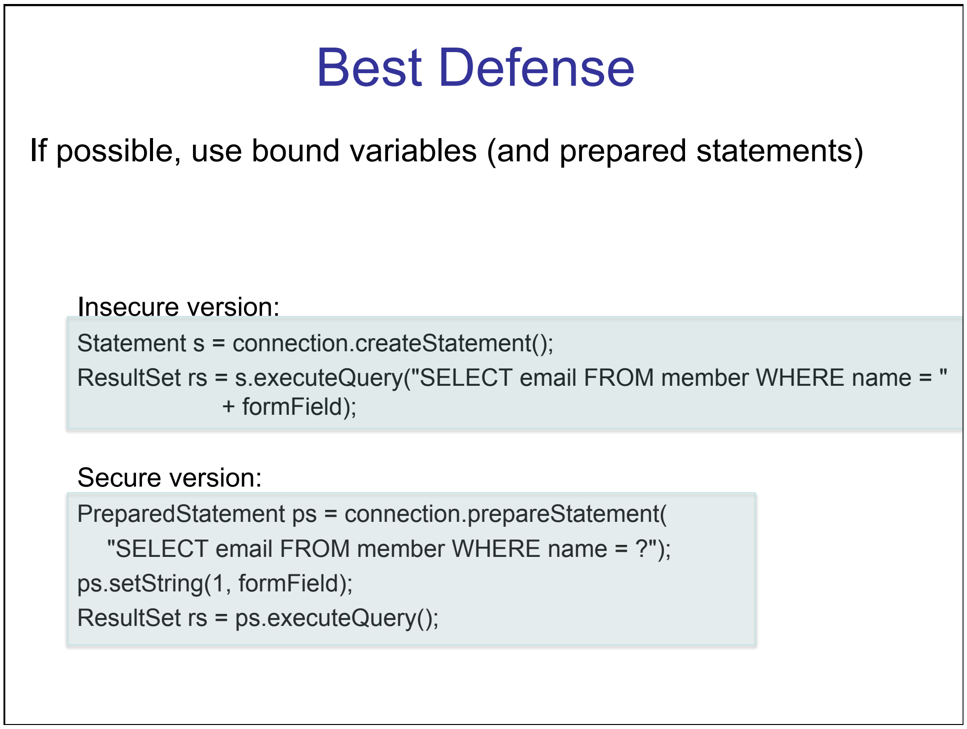
- Check syntax of input for validity
  - Rather than "remove known bad data", it's better to "remove everything but known good data”
  - Many classes of input have fixed languages
    - Email addresses, dates, part numbers, etc.
    - Verify that the input is a valid string in the language
    - If you can exclude quotes and semicolons that’s good
  - Not always possible: consider the name Bill O’Reilly
    - Want to allow the use of single quotes in names
- Have length limits on input
  - Many SQL injection attacks depend on entering long strings

More Defenses

- Scan query string for undesirable word combinations that indicate SQL statements
  - INSERT, DROP, etc.
  - If you see these, you can check against SQL syntax to see if they represent a statement or valid user input
- Limit database permissions and segregate users
  - If you're only reading the database, connect to database as a user that only has read permissions
  - Never connect as a database administrator in your web application
- Configure database error reporting
  - Default error reporting often gives away information that is valuable for attackers (table name, field name, etc.)
  - Configure so that this information is never exposed to a user

Even More Defenses

- Use provided functions for escaping strings
  - MySQL's library function `mysql_real_escape_string` prepends backslashes to the following characters: *\x00, \n, \r, \, ', " and \x1a.*
- Not a silver bullet!
  - Consider:
    - `SELECT fields FROM table WHERE id = 23 OR 1=1`
    - No quotes here!



Best Defense

If possible, use bound variables (and prepared statements)

Insecure version:

```
Statement s = connection.createStatement();
ResultSet rs = s.executeQuery("SELECT email FROM member WHERE name = "
                    + formField);
```

Secure version:

```
PreparedStatement ps = connection.prepareStatement(
    "SELECT email FROM member WHERE name = ?");
ps.setString(1, formField);
ResultSet rs = ps.executeQuery();
```