Classic Approaches

1. Use a preprocessor
   - Usually for older languages (e.g., C and C++)
   - I’ll show an example or two of this, just for context

2. Use a library (API)
   - Usually the only option for languages with APIs
   - Often several options per language
The Preprocessor Approach (1 / 2)

A common C program line:  
```
#include <stdio.h>
```

But that is not C; rather, it’s a ________________.

A preprocessor can be used to expand DBMS commands, thus saving us coding:

1. Insert preprocessor statements into program code
2. Execute the DBMS’ preprocessor
3. Compile & link the program
4. Execute the application

The Preprocessor Approach (2 / 2)

Two varieties of preprocessed SQL statements:

1. Embedded SQL
   - SQL statements are hard-coded (static)
2. Dynamic SQL
   - Arguments added to an SQL statement shell
Cursors

A Problem:

How large will your query’s result be?

(That is, how much memory do we need to hold what the DBMS is going to return to us?)

The Solution:

Preprocessor Examples

See the Sample C & Postgres programs on class webpage!

“Embedded and Dynamic SQL APIs in Postgres”

Advantage:

- Can be adapted to any programming language

Disadvantages:

- Several preprocessor directives to learn.
- Very little abstraction (e.g., cursors are explicit)
The Library Approach

Advantage:

- Just another API; use it like any other API

Disadvantage:

- Might be a 3rd party add-on; needs to be installed

ODBC vs. JDBC

ODBC:

- An early 1990s Microsoft API to connect C programs to DBMSes
- ODBC stands for “Open Database Connectivity”
- Recently (2018) updated by Microsoft to support hierarchical and semistructured data

JDBC:

- Sun Microsystems’s (now Oracle’s) 1997 Java API based on ODBC
- JDBC stands for . . .
Core capabilities:

Some related technologies:

- SQLJ — a preprocessor–based Java language extension
- Java Persistence API (JPA) — supplies object persistence
- Java Data Objects (JDO) — ditto

Using JDBC (1 / 4)

1. Establish connection to a data source
   
   (a) import java.sql.*
   
   (b) Load the driver (names vary by DBMS)
       
       - Add your DBMS’ JAR file to your classpath:
         
         - Oracle 11: ojdbc14.jar

       - Call Class.forName() to initialize the driver class:
         
         - Oracle 11: oracle.jdbc.OracleDriver
Using JDBC (2 / 4)

(c) Connect to the DBMS

Connection dbConnect = DriverManager.getConnection (
"jdbc:oracle:thin:@host.foo.bar.com:1234:oracle",
"username", "password" );

where:

- The first argument is the DB URL. Parts:
  - thin is the type of driver
  - host.foo.bar.com is the DBMS server
  - 1234 is the port number
  - oracle is the sid (system ID)
- username is the user's DBMS login
- password is the user's DBMS password

Using JDBC (3 / 4)

2. Send SQL statements to that source

Create a Statement object:

Statement stmt = dbConnect.createStatement();

Ask it to execute the SQL query:

ResultSet answer = stmt.executeQuery ( 
"SELECT sno, status FROM s" );
Using JDBC (4 / 4)

3. Process returned results and messages

JDBC uses cursors, too, but the details are implicit.

Before the first read, test `answer.next()`:

If true, a tuple is available

Then, fetch field values by type. E.g.:

```
answer.getString("sno")
answer.getInt("status")
```

---

Accessing MetaData with JDBC

First, get a `ResultSetMetaData` object by calling:

```
rsmd = answer.getMetaData()
```

Then, fetch the metadata you want to see. E.g.:

```
rsmd.getColumnCount()              // returns degree
rsmd.getColumnName()               // returns attr. name
rsmd.getColumnDisplaySize()        // returns width
```

A final ‘FYI’: To get a result’s cardinality, call in sequence:

```
answer.last()           // moves to last tuple
answer.getRow()         // to get current row number
```