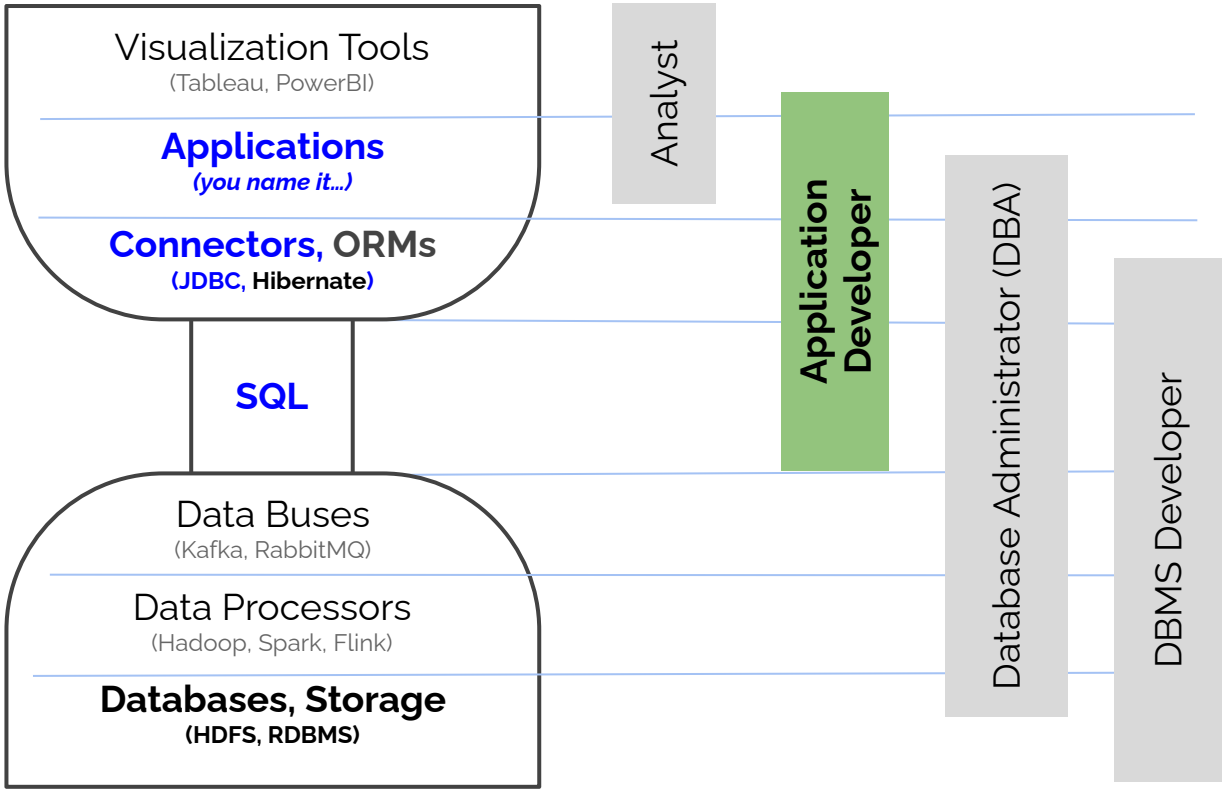
CSC 365

Introduction to Database Systems

SQL / Data Management Hourglass

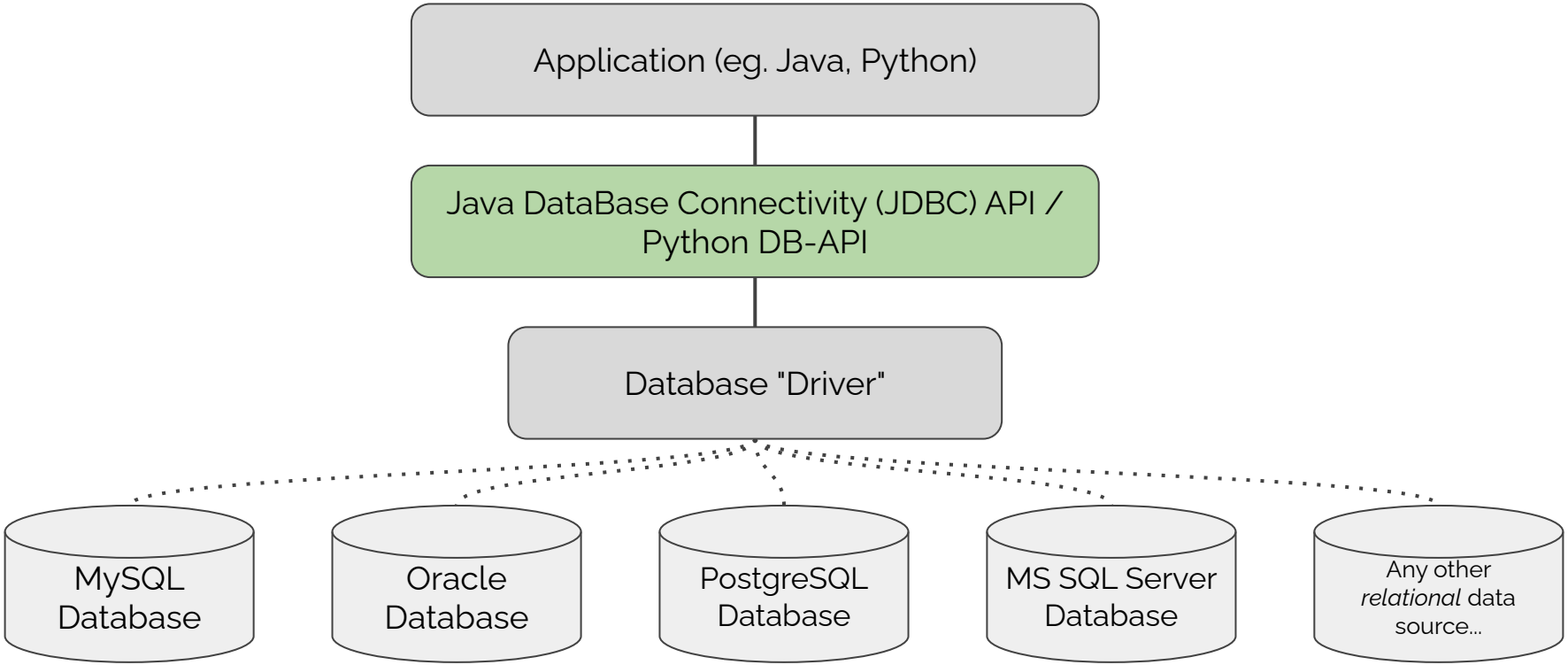
CAL POLY



- **Large Amounts of Data**: much larger than main memory (but perhaps not quite *Big*®™...)
- **High Performance**: thousands of tasks per second
- **Available**: no downtime / outages
- **Easy to Use**: powerful operations on large amounts of data
- **Safe & Reliable**: maintains consistency of data, no data loss
- **Multi-User**: simultaneous users operating concurrently on the same data
- **Persistent**: data is long-lived, retained between program executions
- **General Purpose**: common tools and techniques for many problem domains

Application Data Base Connectivity

CAL POLY



Connector / Python Examples

CAL POLY

Google Colab - Connector / Python Example

Google Colab - Food/Flavor Summary

Data Access APIs

Java Database Connectivity (JDBC)

Python Database API Specification

Microsoft Open Database Connectivity (ODBC) (C language, predates JDBC)

Microsoft Object Linking and Embedding (OLE DB)

PHP Data Objects

JDBC Overview

A Java application can use a standard set of classes and methods (the **JDBC API**) to interact with *any* relational data source.

DBMS-specific details (networking protocols, connection mechanisms, data conversion, etc.) are handled by a **JDBC Driver**. A JDBC Driver implementation is typically provided by the DBMS vendor/developer.

JDBC Driver

To make use of the JDBC API, an appropriate JDBC Driver (provided as a JAR file) must be added to the Java application's classpath.

MySQL driver, known as Connector/J may be downloaded from:
https://dev.mysql.com/downloads/connector/j/

JDBC Driver / Classpath

Once you have the JDBC Driver, there are a few ways to add it to your working Java classpath. Two simple options, using the MySQL Connector/J driver:

Option 1: Set the CLASSPATH environment variable:

```
% export CLASSPATH=$CLASSPATH:mysql-connector-java-8.0.16.jar:.
% java MyMainClass
```

Option 2: Use the `-cp` command line switch when running `java`:

```
% java -cp mysql-connector-java-8.0.16.jar:. MyMainClass
```

JDBC URLs

A **JDBC URL** specifies database location and other connection parameters. Each vendor/driver extends the URL structure in its own way. Generally:

```
jdbc:<vendor|driver>:<server>[:<port>]/<database name>[?<param1=val1>...]
```

MySQL Example:

```
jdbc:mysql://mysql.labthreesixfive.com/<databse name>?autoReconnect=true
```

Core JDBC Classes (package `java.sql`)

CAL POLY

`DriverManager`	Service for managing JDBC drivers and establishing connections to a database
`Connection`	Represents a connection (session) with a specific database server.
`Statement`	Used to execute SQL statements and retrieve results. All SQL statements are executed within the context of a connection.
`ResultSet`	Represents a database result as a table of data. Maintains an internal cursor that allows you to move through the results.
`PreparedStatement`	A precompiled (and possibly parameterized) SQL statement that may be reused.
`SQLException`	`Exception` subclass that provides detail about database-related errors.

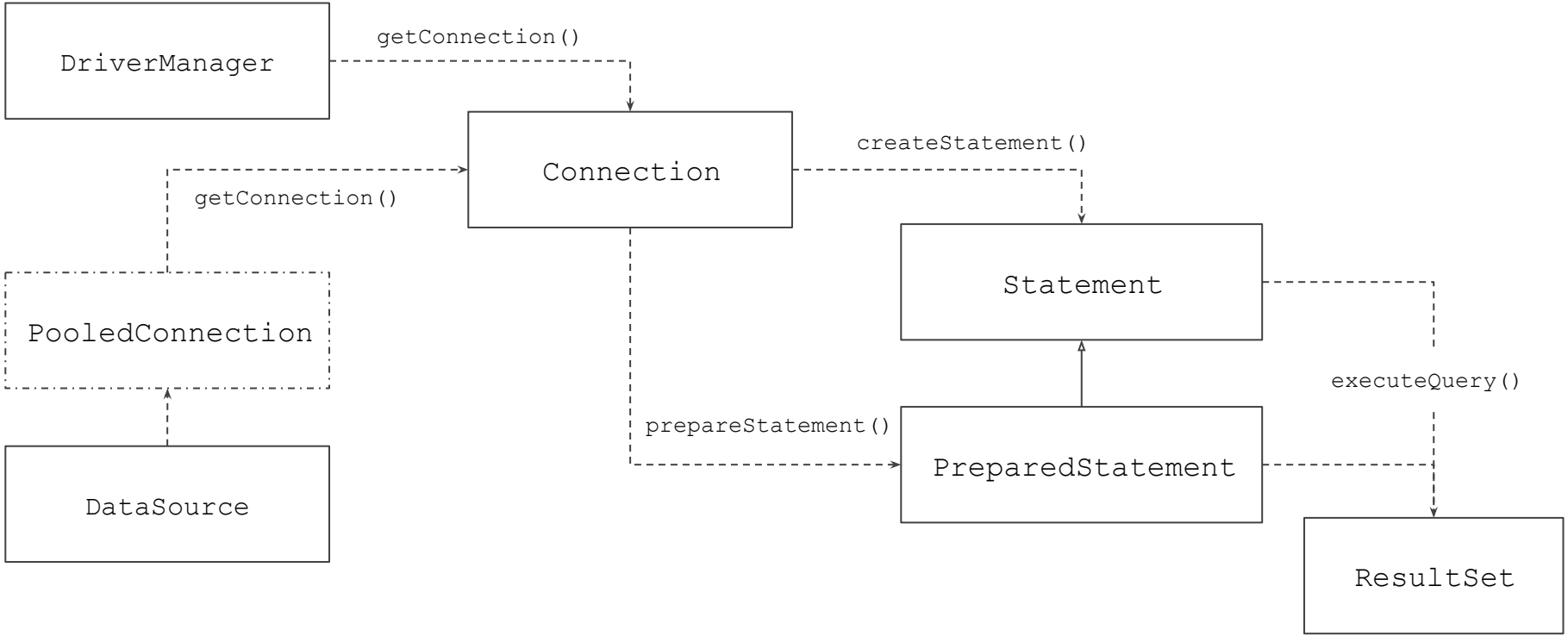
Extended JDBC Classes (package `javax.sql`)

CAL POLY

`DataSource`	Operates as a connection "factory," supporting several connection modes: basic, pooled, distributed transaction.
`RowSet`	Disconnected, serializable version of a `ResultSet` (extends ResultSet class, operates without an active connection with the database, versus ResultSet's connected cursor mode)
`PooledConnection`	Connection with support for pooling (sharing a single connection across multiple clients, to avoid the cost of re-establishing a connection for each client)
`XAConnection`	Connection that provides support for distributed transactions (a topic for another lecture/course, or several…)

JDBC - High-level Overview

CAL POLY



General Sequence of JDBC Application Actions

1. Connection to the database
2. Construct SQL statement (as a `String`)
3. Start a transaction (implicitly or explicitly)
4. Send SQL statement to the DBMS
5. Receive result
6. Commit (or rollback) transaction (sometimes implicit)
7. Close connection

Note: Loading a JDBC Driver

Documentation and examples will often include the following code to "load" a JDBC driver:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException ex) { ... }
```

As of JDBC 4.0 / Java SE 6, this step is no longer required. However, it may be useful to allow your program to detect and gracefully handle presence or absence of a specific JDBC driver on the Java classpath.

Connecting to a Database

CAL POLY

```
String jdbcUrl = System.getenv("APP_JDBC_URL");
String dbUsername = System.getenv("APP_JDBC_USER");
String dbPassword = System.getenv("APP_JDBC_PW");

Connection conn = DriverManager.getConnection(jdbcUrl, dbUsername, dbPassword);
```

It is possible to specify url, username and password as literal string values. However, hard-coding these sensitive values has significant downsides: (1) It is all too easy to mistakenly check credentials into source control, and (2) Changing credentials requires a rebuild. Instead, it is advisable to externalize these configuration parameters. For example, credentials may be passed via environment variables. Example Linux commands:

```
% export APP_JDBC_URL=jdbc:mysql://my-host/thedb
% export APP_JDBC_USER=myusername
% export APP_JDBC_PW=extrasecret
% java MyDatabaseApplication
```

Executing SQL - DML

CAL POLY

```
// Connection established on previous slide
Statement stmt = conn.createStatement();

boolean exRes = stmt.execute("ALTER TABLE hp_goods ADD AvailUntil DATE");
```

Use `execute()` to run *any* type of SQL statement (typically DDL; there are other specialized methods for queries and DML) This method returns a boolean: `true` indicates that the result is a ResultSet object, which may be retrieved by calling `getResultSet()`; `false` indicates that there is an update count or no results (`getUpdateCount()` allows you to retrieve the record count, if any)

Executing SQL - SELECT

CAL POLY

```
// Connection "conn" established on earlier slide
Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT * FROM hp_goods");
```

Use `executeQuery()` to run a `SELECT` statement. This method returns a `ResultSet` object, allowing you to iterate through resulting rows.

Reading Results

CAL POLY

```
// ResultSet rs from previous slide

while (rs.next()) {
    String flavor = rs.getString("Flavor");
    Srting food = rs.getString("Food");
    float price = rs.getFloat("price");
    System.out.format("%s %s: $%.2f %n", flavor, food, price);
}


// What about getString(<col index>) with SELECT *?
// Duplicate column names?
```

All `get`*`Type`*`()` methods accept either a column index (starting from 1) or a case-***in***sensitive column name.

SQL / Java Data Type Mapping

ANSI SQL Data Type	Java Type
`CHAR, VARCHAR`	`String`
`INTEGER, FLOAT, DOUBLE`	`int, float, double`
`DECIMAL(p,s)`	`java.math.BigDecimal`
`BIT`	`boolean`
`DATE`	`java.sql.Date` (subclass of `java.util.Date`)
`TIME`	`java.sql.Time` (subclass of `java.util.Date`)
`DATETIME`	`java.sql.Timestamp` (subclass of `java.util.Date`)
`BINARY`	`byte[]`

Scrollable / Updateable ResultSets

By default, a `ResultSet` object is *not updatable* and has an internal cursor that moves *forward only*. You may iterate through only once and only from the first row to the last row.

It is possible to create `ResultSet` objects that are scrollable and/or updatable.

Both options may be specified in the two-argument method:
Connection.createStatement(int resultSetType, int resultSetConcurrency)

Scrollable ResultSet Options

- `ResultSet.TYPE_FORWARD_ONLY` - Internal cursor may move only forward using the `next()` method on `ResultSet` *(this is the default)*

- `ResultSet.TYPE_SCROLL_INSENSITIVE` - Scrollable (via `next()`, `previous()`, `first()`, `last()`, `absolute()`) This type of `ResultSet` is generally *not sensitive to changes* to underlying data.

- `ResultSet.TYPE_SCROLL_SENSITIVE` - Scrollable and generally sensitive to changes to underlying data.

Scrollable ResultSets & Performance

- Using a scrollable `ResultSet` can impact performance
  - Depends on database and connection settings / features

- The scrollable option should be used with caution, and only in cases where you are *sure* that the underlying database offers full support.

Updateable ResultSets

- `ResultSet.CONCUR_READ_ONLY` - `ResultSet` may NOT be updated
  - *(the is the default mode)*

- `ResultSet.CONCUR_UPDATABLE` - Updates are allowed through the `ResultSet`

Specifying `CONCUR_UPDATABLE` *does not guarantee* that updates will be allowed. Not all databases/drivers support this feature. Also, the `SELECT` must include the primary key and must reference just one table.

JDBC objects (such as: `Connection`, `Statement`, and `ResultSet`) allocate and hold resources (file descriptors, local/remote sockets, database connections, etc.)

To avoid *resource leaks*, it is important to call `close()` when you are finished using an object, taking particular care to do so even when exceptions occur.

Exception Handling / Resource Management


```
// Legacy approach (pre- Java 1.7)
Connection conn = null;
Statement stmt = null;
try {
   conn = DriverManager.getConnection(...);
   stmt = conn.createStatement();
   // execute SQL, read results, etc.
} catch (SQLException e) {
   // handle error
} finally {
  if (stmt != null) { stmt.close(); }
  if (conn != null) { conn.close(); }
}
```

Try-with-resources / AutoCloseable (Java 1.7+):

```
try (Connection conn = DriverManager.getConnection();
     Statement stmt = conn.createStatement()) {
     // execute SQL, read results, etc.
} catch (SQLException e) {
     // handle error (log, rethrow, rollback, etc.)
}
// finally {} block not required, .close() methods called automatically,
// even if an Exception occurs within the try {} block
```

Generating SQL within Java

CAL POLY

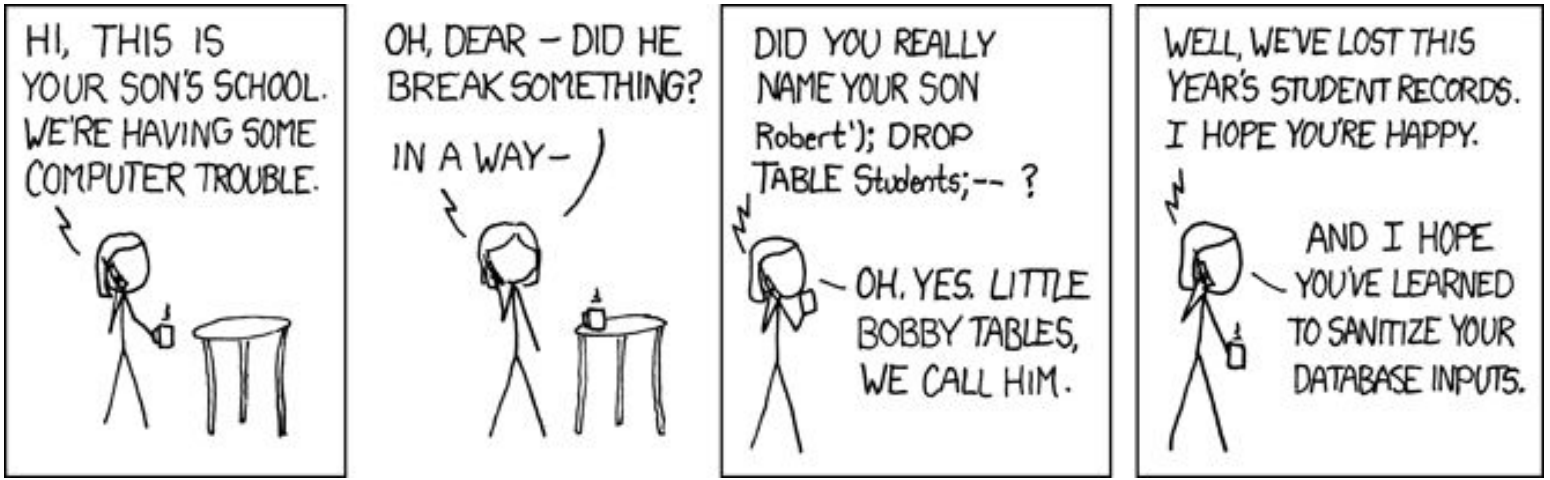
SQL statements sent to JDBC methods are regular Java `String` objects, and may be constructed using normal Java string concatenation syntax.

```
System.out.print("Enter a flavor: ");
String flavor = scanner.nextLine();
System.out.print("Until what date will " + flavor + " be available (YYYY-MM-DD)? ");
String availUntilDate = scanner.nextLine();
String updateSql = "UPDATE hp_goods SET AvailUntil = '" + availUntilDate + "' " +
                   "WHERE Flavor = '" + flavor + "'";
Statement stmt = con.createStatement();
int rowCount = stmt.executeUpdate(updateSql);
System.out.format("Updated %d records for %s pastries%n", rowCount, flavor);
// Never (ever) write database code like this!
```

SQL Injection

CAL POLY

The previous slide is a demonstration of a security vulnerability known as SQL Injection. A malicious user can craft input values to change the behavior of a dynamically-constructed SQL statement in harmful ways. Just ask Bobby Tables:



PreparedStatement

CAL POLY

JDBC's `PreparedStatement` supports precompilation (for fast repeated execution) and parameterization (for security & easy reuse of similar statements):

```
// (1) Prepare/precompile SQL statement (parameterized with ? placeholders)
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE hp_goods SET AvailUntil = ? WHERE Flavor = ?");

// (2) Set parameters (indexed starting at 1)
pstmt.setDate(1, java.sql.Date.valueOf(availDt));  // Note: setDate() expects java.sql.Date
pstmt.setString(2, flavor);

// (3) Execute the statement
int rowCount = pstmt.executeUpdate();
```

PreparedStatement with NULL

CAL POLY

```
// To set the AvailUntil column to NULL:
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE hp_goods SET AvailUntil = ? WHERE Flavor = ?");
pstmt.setDate(1, null);
pstmt.setString(2, flavor);
int rowCount = pstmt.executeUpdate();


// PreparedStatement also supports a setNull() method:
pstmt.setNull(1, java.sql.Types.DATE);


// Some JDBC drivers have quirks in this area
```

Prepared Statement Reuse

CAL POLY

```
String ins = "INSERT INTO bank_ledger (id, amount, date, descr) VALUES (?, ?, CURRENT_TIMESTAMP, ?)"

try (PreparedStatement pstmt = conn.prepareStatement(ins)) {
    pstmt.setLong(1, 101);
    pstmt.setFloat(2, -50);
    pstmt.setString(3, "Transfer from savings to checking");
    int rowCount = pstmt.executeUpdate();
    pstmt.clearParameters();   // allows re-use of the *same* PreparedStatement object
    pstmt.setLong(1, 102);
    pstmt.setFloat(2, 50);
    pstmt.setString(3, "Transfer from checking");
    int rowCount2 = pstmt.executeUpdate();
    pstmt.clearParameters();
    pstmt.setLong(1, 103);
    pstmt.setFloat(2, -40);
    pstmt.setString(3, "ATM Withdrawal");
    int rowCount3 = pstmt.executeUpdate();
} catch (SQLException e) {
    // Handle exception appropriately
}
```

Batch Mode

In the previous example, each `INSERT` statement would be sent separately to the database, resulting in multiple network round trips. When executing many SQL statements, it is more efficient to use *batch mode*.

```
try (PreparedStatement pstmt = con.prepareStatement(ins)) {
    for (Transfer t : bankTransfers) {
        pstmt.setLong(1, t.getId());
        // ...
        pstmt.addBatch();
    }
    int[] rowCounts = pstmt.executeBatch(); // one array entry for each addBatch() call
}
```

With an open `Connection`, we can execute one or more SQL commands using `Statement` or `PreparedStatement`

Important to consider:

- Cost of establishing & closing connections
- Repeated execution of the same SQL command with different parameters
- Transactions

Transactions in JDBC

By default, a JDBC `Connection` is in **auto-commit** mode. In auto-commit mode, *each individual SQL statement* is treated as a transaction and is committed when executed.

For control over transaction boundaries, use `setAutoCommit(false)` along with the `commit()` and `rollback()` methods on a `Connection`.

Transactions in JDBC - Example

CAL POLY

```
String ins = "INSERT INTO bank_ledger (id, amount, date, descr) VALUES (?, ?, CURRENT_TIMESTAMP, ?)"

conn.setAutoCommit(false);  // disable auto-commit, this Connection now allows transactional control

try (PreparedStatement pstmt = conn.prepareStatement(ins)) {
    pstmt.setLong(1, 101);
    pstmt.setFloat(2, -50);
    pstmt.setString(3, "Transfer from savings to checking");
    int rowCount = pstmt.executeUpdate();
    pstmt.clearParameters();

    pstmt.setLong(1, 102);
    pstmt.setFloat(2, 50);
    pstmt.setString(3, "Transfer from checking");
    int rowCount2 = pstmt.executeUpdate();

    conn.commit();
} catch (SQLException e) {
    conn.rollback();
}
```

JDBC Summary

The Java DataBase Connectivity (JDBC) API offers a database-independent way to construct Java applications that make use of relational databases and SQL.

JDBC is a relatively low-level API. There are many libraries and APIs that build on JDBC and offer a higher level of abstraction.