JDBC and Database Programming in Java

Agenda

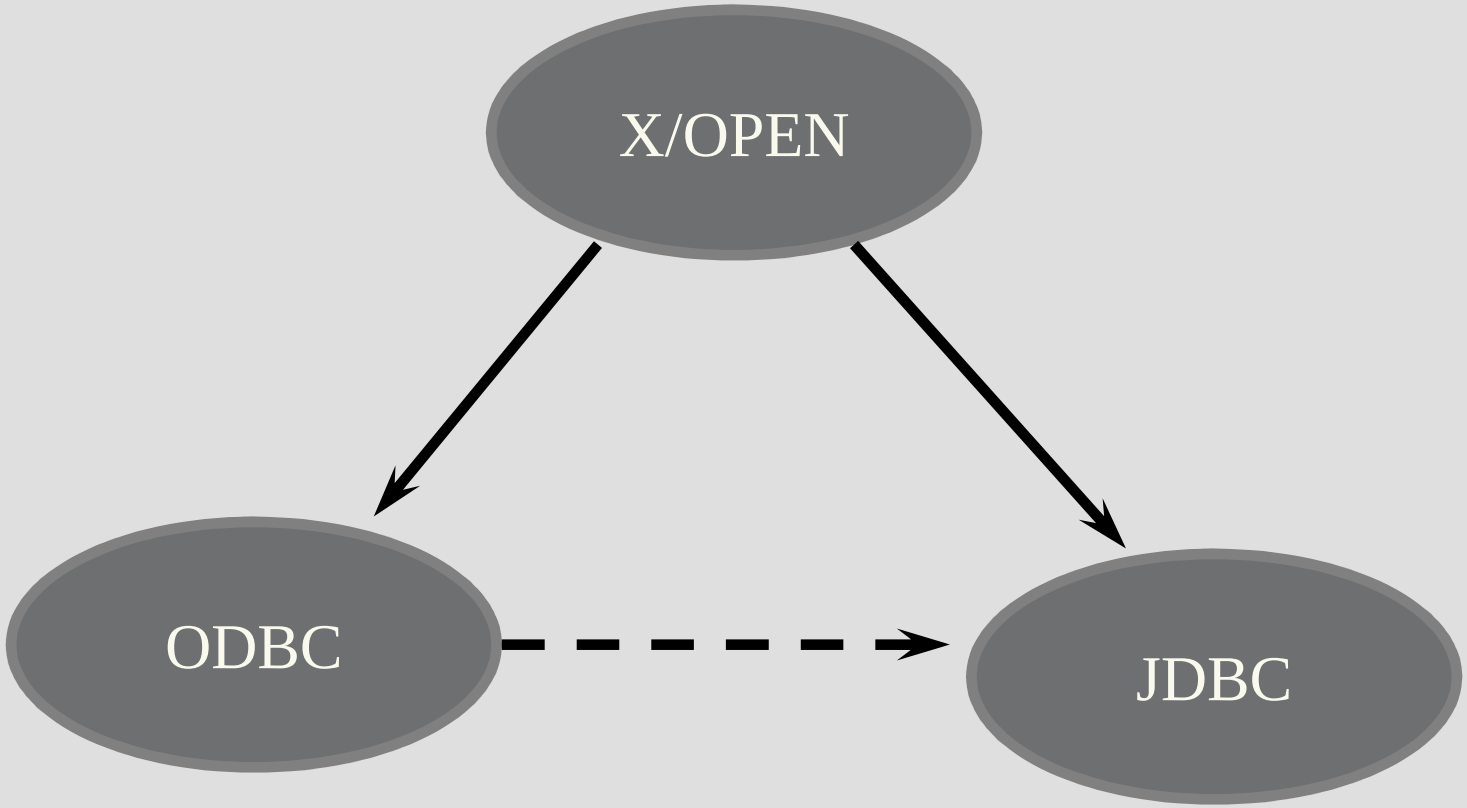
- Overview of JDBC
- JDBC APIs

JDBC Overview

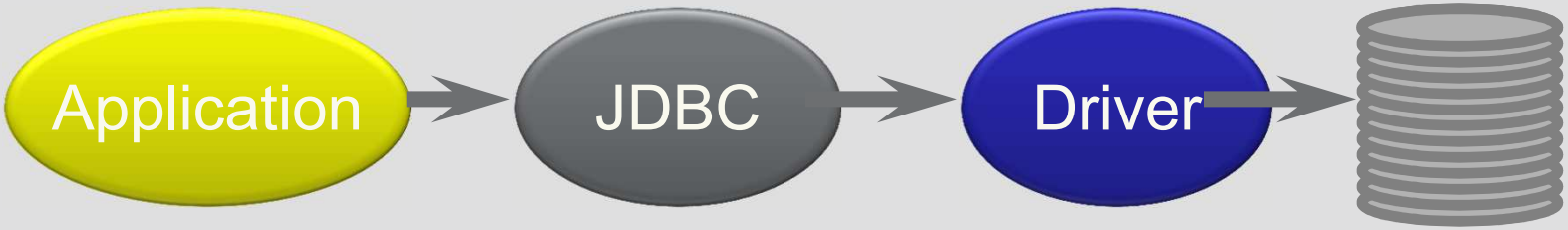
JDBC Goals

- SQL-Level
- 100% Pure Java
- Keep it simple
- High-performance
- Leverage existing database technology
  - why reinvent the wheel?
- Use strong, static typing wherever possible
- Use multiple methods to express multiple functionality

JDBC Ancestry



JDBC Architecture

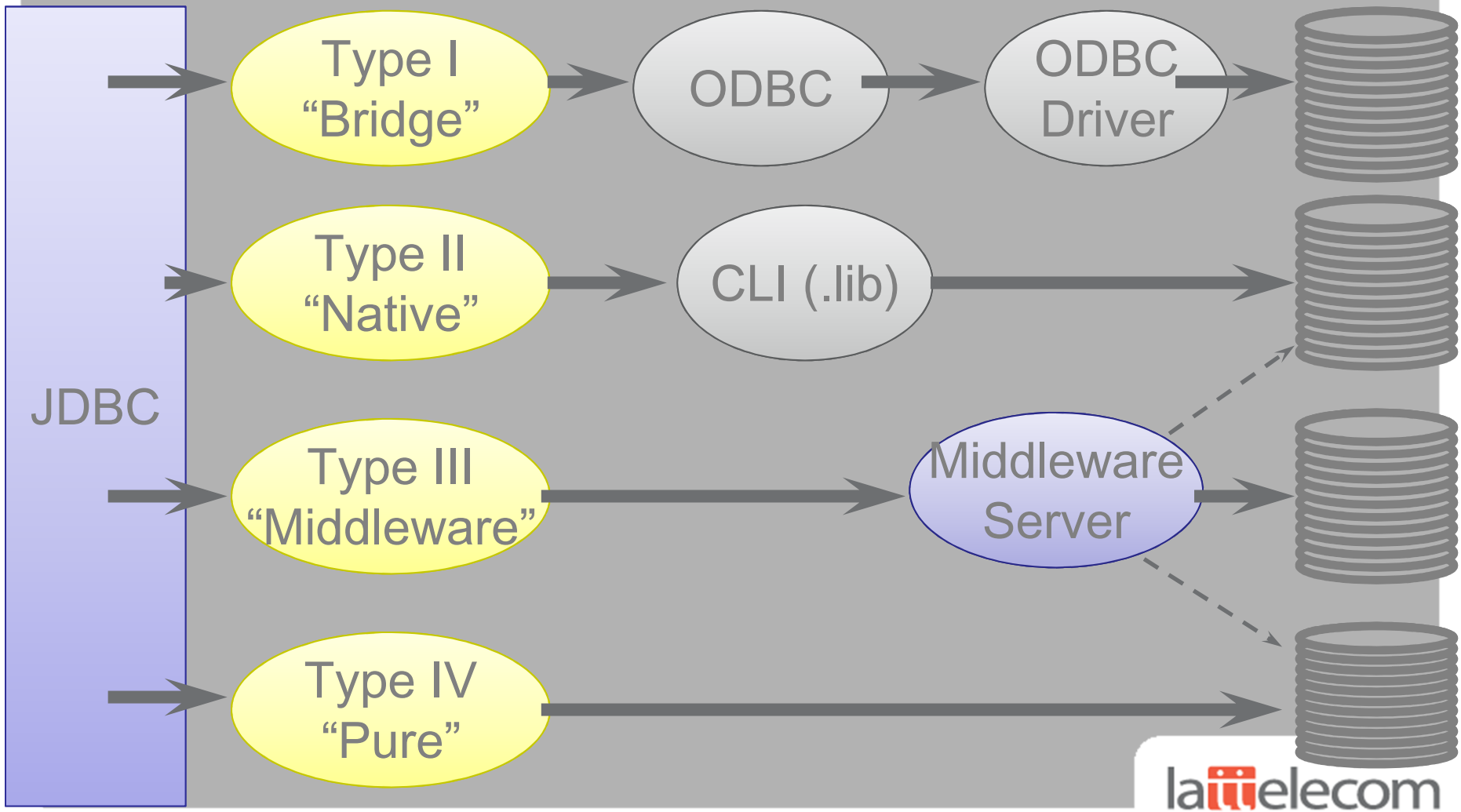


- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- Can have more than one driver -> more than one database
- Ideal: can change database engines without changing any application code

JDBC Drivers

- Type I: “Bridge”
- Type II: “Native”
- Type III: “Middleware”
- Type IV: “Pure”

JDBC Drivers (Fig.)



Type I Drivers

- Use bridging technology
- Requires installation/configuration on client machines
- Not good for Web
- e.g. ODBC Bridge

Type II Drivers

- Native API drivers
- Requires installation/configuration on client machines
- Used to leverage existing CLI libraries
- Usually not thread-safe
- Mostly obsolete now

Type III Drivers

- Calls middleware server, usually on database host
- Very flexible -- allows access to multiple databases using one driver
- Only need to download one driver
- But it’s another server application to install and maintain

Type IV Drivers

- 100% Pure Java -- the Holy Grail
- Use Java networking libraries to talk directly to database engines
- Only disadvantage: need to download a new driver for each database engine
- e.g. Oracle

Related Technologies

- ODBC
  - Requires configuration (odbc.ini)
- RDO, ADO
  - Requires Win32
- OODB
  - e.g. ObjectStore from ODI
- JPA, ORM
  - maps objects to tables transparently (more or less)

JDBC APIs

java.sql

- JDBC is implemented via classes in the java.sql package

Loading a Driver Directly

```
Driver d = new foo.bar.MyDriver();
Connection c = d.connect(...);
```

- Not recommended, use DriverManager instead
- Useful if you know you want a particular driver

DriverManager

- DriverManager tries all the drivers
- Uses the first one that works
- When a driver class is first loaded, it registers itself with the DriverManager
- Therefore, to register a driver, just load it!

Registering a Driver

- statically load driver

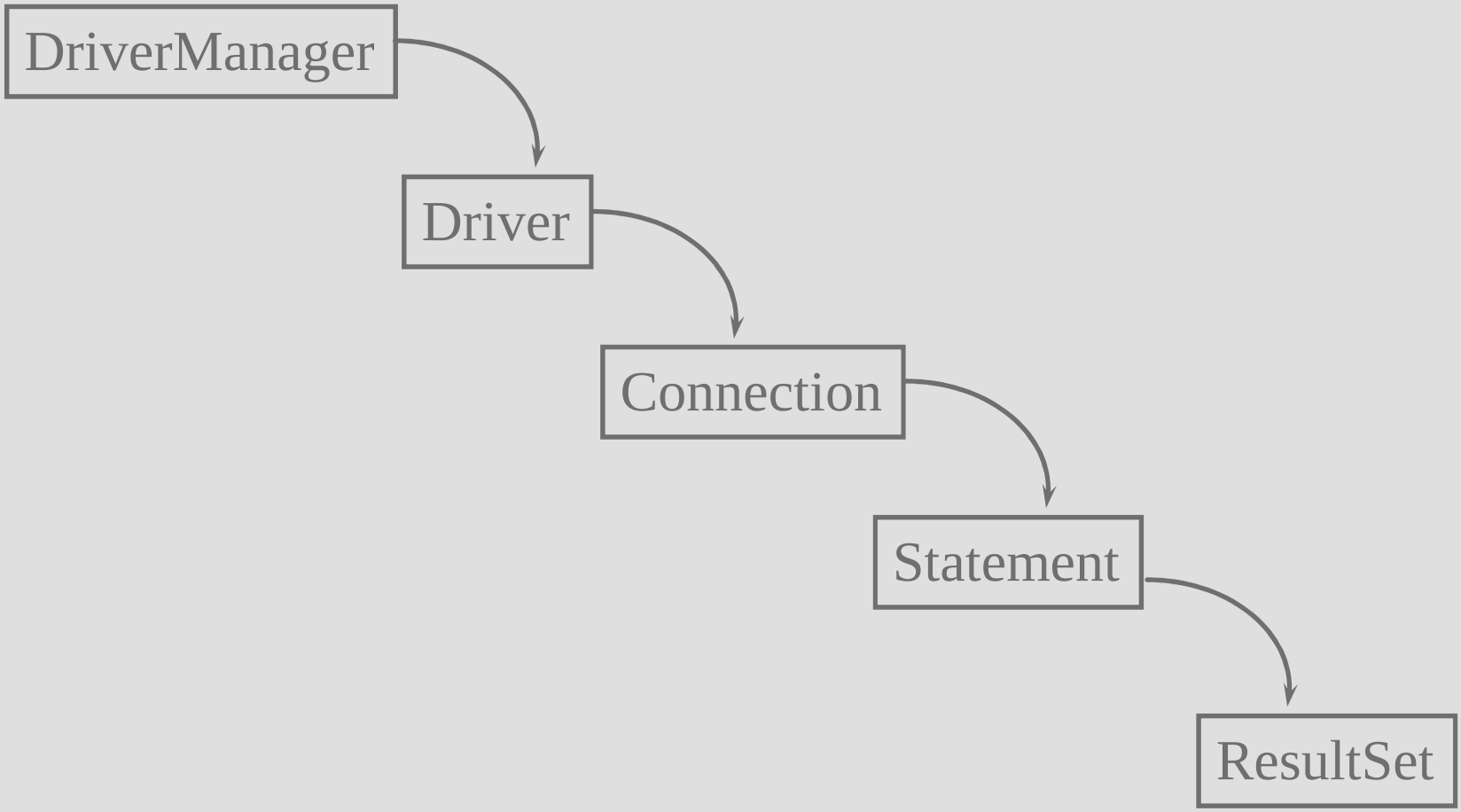
```
Class.forName("foo.bar.MyDriver");
Connection c =
  DriverManager.getConnection(...);
```

- or use the `jdbc.drivers` system property

JDBC Object Classes

- DriverManager
  - Loads, chooses drivers
- Driver
  - connects to actual database
- Connection
  - a series of SQL statements to and from the DB
- Statement
  - a single SQL statement
- ResultSet
  - the records returned from a Statement

JDBC Class Usage



JDBC URLs

`jdbc:`*`subprotocol`*`:`*`source`*

- each driver has its own subprotocol
- each subprotocol has its own syntax for the source

`jdbc:odbc:`*`DataSource`*

  - e.g. `jdbc:odbc:Northwind`

`jdbc:msql:`*`//host[:port]/database`*

  - e.g. `jdbc:msql://foo.nowhere.com:4333/accounting`

DriverManager

```
Connection getConnection
    (String url, String user, String
    password)
```

- Connects to given JDBC URL with given user name and password
- Throws java.sql.SQLException
- returns a Connection object

Connection

- A Connection represents a session with a specific database.
- Within the context of a Connection, SQL statements are executed and results are returned.
- Can have multiple connections to a database
  - NB: Some drivers don't support serialized connections
  - Fortunately, most do (now)
- Also provides "metadata" -- information about the database, tables, and fields
- Also methods to deal with transactions

Obtaining a Connection

```
String url   = "jdbc:odbc:Northwind";
try {
   Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection con = DriverManager.getConnection(url);
}
catch (ClassNotFoundException e)
   { e.printStackTrace(); }
catch (SQLException e)
   { e.printStackTrace(); }
```

Connection Methods

`Statement createStatement()`

- returns a new Statement object

`PreparedStatement prepareStatement(String sql)`

- returns a new PreparedStatement object

`CallableStatement prepareCall(String sql)`

- returns a new CallableStatement object

- Why all these different kinds of statements?
  Optimization.

Statement

- A Statement object is used for executing a static SQL statement and obtaining the results produced by it.

Statement Methods

`ResultSet executeQuery(String)`

- Execute a SQL statement that returns a single ResultSet.

`int executeUpdate(String)`

- Execute a SQL INSERT, UPDATE or DELETE statement. Returns the number of rows changed.

`boolean execute(String)`

- Execute a SQL statement that may return multiple results.

ResultSet

- A ResultSet provides access to a table of data generated by executing a Statement.
- Only one ResultSet per Statement can be open at once.
- The table rows are retrieved in sequence.
- A ResultSet maintains a cursor pointing to its current row of data.
- The 'next' method moves the cursor to the next row.

ResultSet Methods

- boolean next()
  - activates the next row
  - the first call to next() activates the first row
  - returns false if there are no more rows
- void close()
  - disposes of the ResultSet
  - allows you to re-use the Statement that created it
  - automatically called by most Statement methods

ResultSet Methods

- *Type* get*Type*(int columnIndex)
  - returns the given field as the given type
  - fields indexed starting at 1 (not 0)
- *Type* get*Type*(String columnName)
  - same, but uses name of field
  - less efficient
- int findColumn(String columnName)
  - looks up column index given column name

ResultSet Methods

- String getString(int columnIndex)
- boolean getBoolean(int columnIndex)
- byte getByte(int columnIndex)
- short getShort(int columnIndex)
- int getInt(int columnIndex)
- long getLong(int columnIndex)
- float getFloat(int columnIndex)
- double getDouble(int columnIndex)
- Date getDate(int columnIndex)
- Time getTime(int columnIndex)
- Timestamp getTimestamp(int columnIndex)

ResultSet Methods

- String getString(String columnName)
- boolean getBoolean(String columnName)
- byte getByte(String columnName)
- short getShort(String columnName)
- int getInt(String columnName)
- long getLong(String columnName)
- float getFloat(String columnName)
- double getDouble(String columnName)
- Date getDate(String columnName)
- Time getTime(String columnName)
- Timestamp getTimestamp(String columnName)

Sample Database

Employee ID	Last Name	First Name
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

SELECT Example

```
Connection con =
  DriverManager.getConnection(url,
  "alex", "8675309");
Statement st = con.createStatement();
ResultSet results =
  st.executeQuery("SELECT EmployeeID,
  LastName, FirstName FROM Employees");
```

SELECT Example (Cont.)

```
while (results.next()) {
  int id = results.getInt(1);
  String last = results.getString(2);
  String first = results.getString(3);
  System.out.println("" + id + ": " +
  first + " " + last);
}
st.close();
con.close();
```

Mapping Java Types to SQL Types

SQL type	Java Type
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Modifying the Database

- use executeUpdate if the SQL contains “INSERT” or “UPDATE”
- Why isn’t it smart enough to parse the SQL? Optimization.
- executeUpdate returns the number of rows modified
- executeUpdate also used for “CREATE TABLE” etc. (DDL)

setAutoCommit

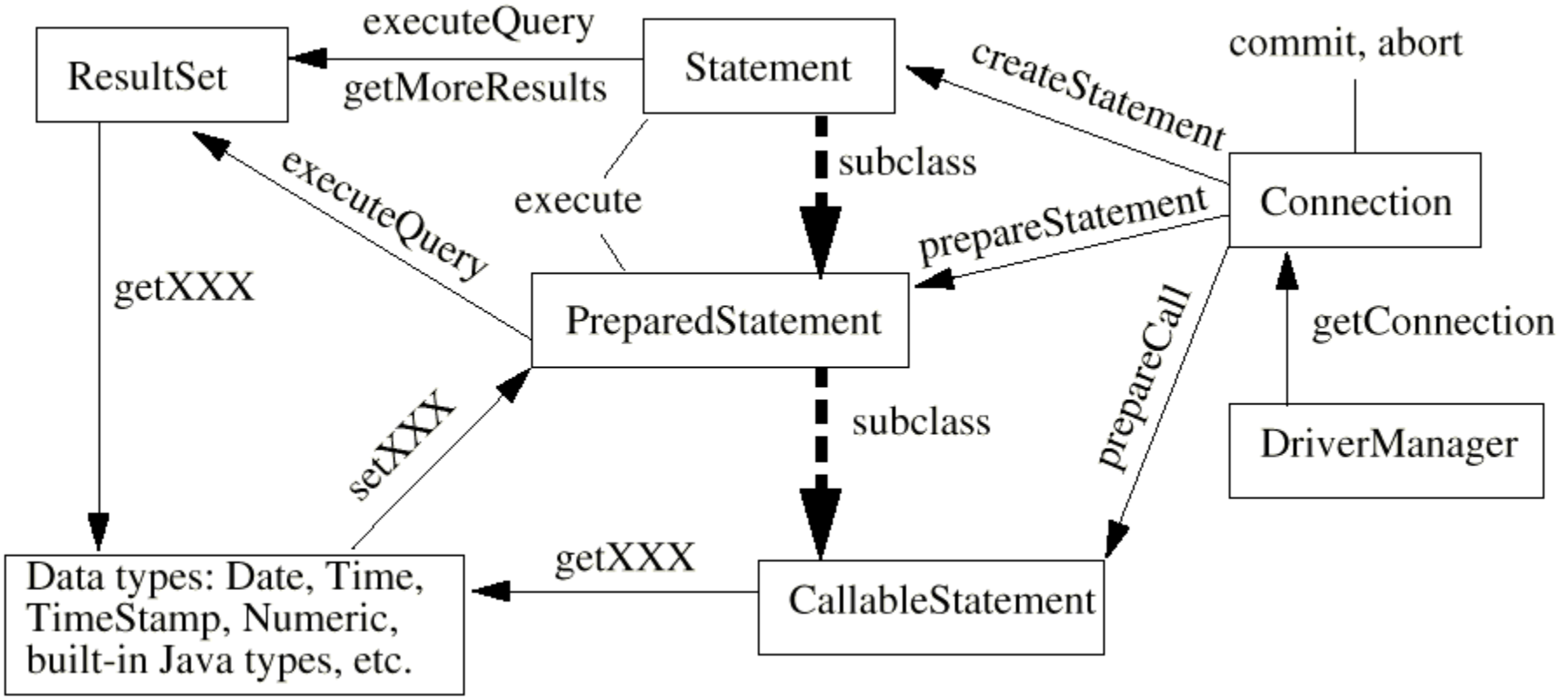
Connection.setAutoCommit(boolean)

- if *AutoCommit* is false, then every statement is added to an ongoing transaction
- you must explicitly commit or rollback the transaction using Connection.commit() and Connection.rollback()

Optimized Statements

- Prepared Statements
  - SQL calls you make again and again
  - allows driver to optimize (compile) queries
  - created with Connection.prepareStatement()
- Stored Procedures
  - written in DB-specific language
  - stored inside database
  - accesed with Connection.prepareCall()

JDBC Class Diagram



Whoa!

Metadata

- Connection:
  - DatabaseMetaData getMetaData()
- ResultSet:
  - ResultSetMetaData getMetaData()

ResultSetMetaData

- What's the number of columns in the ResultSet?
- What's a column's name?
- What's a column's SQL type?
- What's the column's normal max width in chars?
- What's the suggested column title for use in printouts and displays?
- What's a column's number of decimal digits?
- Does a column's case matter?
- Is the column a cash value?
- Will a write on the column definitely succeed?
- Can you put a NULL in this column?
- Is a column definitely not writable?
- Can the column be used in a where clause?
- Is the column a signed number?
- Is it possible for a write on the column to succeed?
- and so on...

DatabaseMetaData

- What tables are available?
- What's our user name as known to the database?
- Is the database in read-only mode?
- If table correlation names are supported, are they restricted to be different from the names of the tables?
- and so on…

JDBC 2.0

- Scrollable result set
- Batch updates
- Advanced data types
  - Blobs, objects, structured types
- Rowsets
  - Persistent JavaBeans
- JNDI
- Connection Pooling
- Distributed transactions via JTS

Where to get more information

- Other training sessions
- Reese, *Database Programming with JDBC and Java* (O'Reilly)
- http://java.sun.com/products/jdbc/
- http://docs.oracle.com/javase/tutorial/jdbc/index.html