JDBC and Database Programming in Java

HSBC Technology and Services

HSBC
The world's local bank

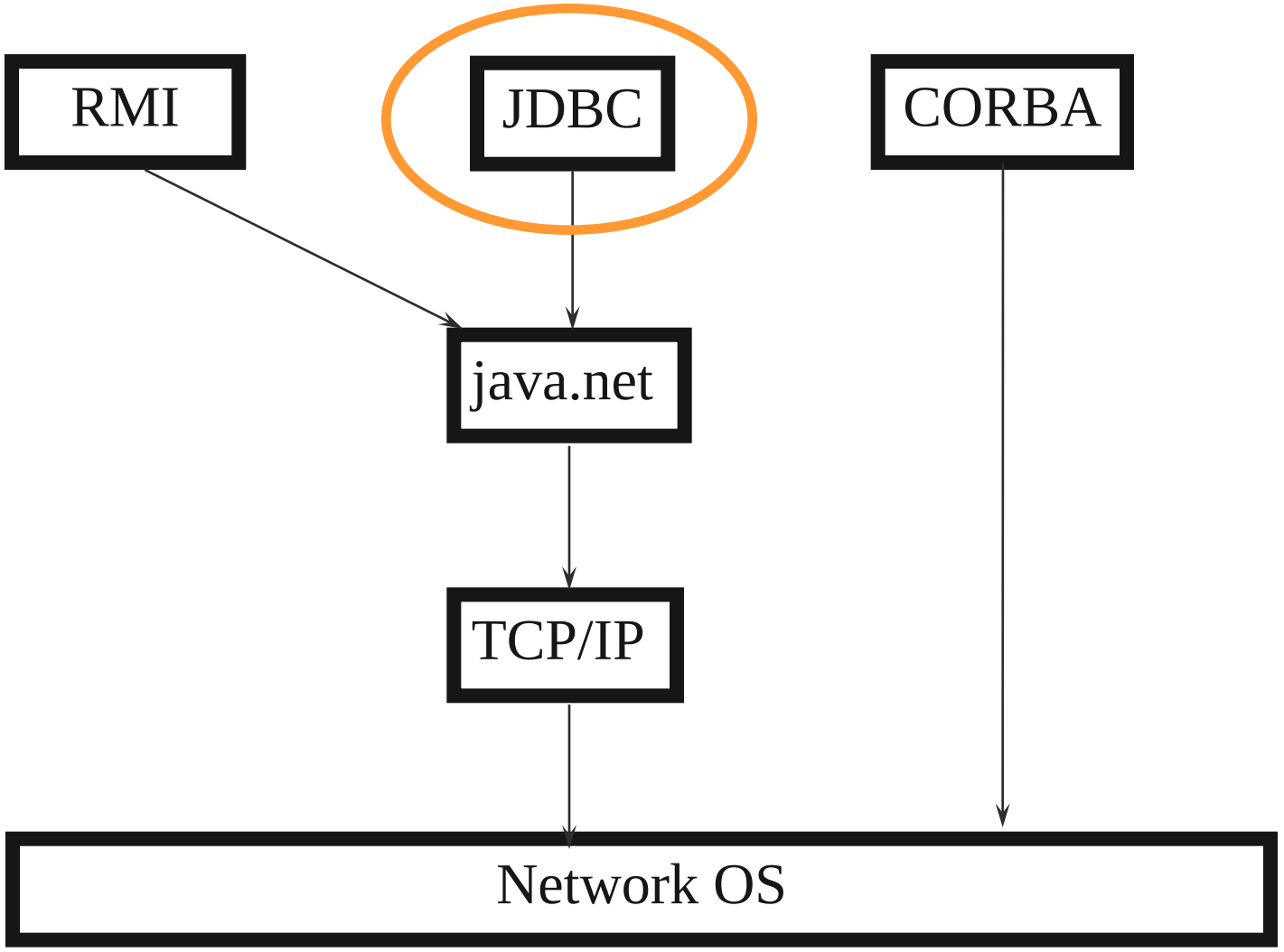
Introduction

- Database Access in Java
- Find out any relevant background and interest of the audience
  - SQL gurus?
  - Visual Basic Database Forms?

Agenda

- Overview of Databases and Java
- Overview of JDBC
- JDBC APIs
- Other Database Techniques

Overview



Vocabulary

- Glossary of terms
- Define the terms as used in this subject

Part I: Overview of Databases and Java

Databases in the Enterprise

- All corporate data stored in DB
- SQL standardizes format (sort of)

Why Java?

- Write once, run anywhere
  - Multiple client and server platforms
- Object-relational mapping
  - databases optimized for searching/indexing
  - objects optimized for engineering/flexibility
- Network independence
  - Works across Internet Protocol
- Database independence
  - Java can access any database vendor
- Ease of administration
  - zero-install client

Database Architectures

- Two-tier
- Three-tier
- N-tier

Two-Tier Architecture

- Client connects directly to server
- e.g. HTTP, email
- Pro:
  - simple
  - client-side scripting offloads work onto the client
- Con:
  - fat client
  - inflexible

Three-Tier Architecture

- Application Server sits between client and database

Three-Tier Pros

- flexible: can change one part without affecting others
- can connect to different databases without changing code
- specialization: presentation / business logic / data management
- can cache queries
- can implement proxies and firewalls

Three-Tier Cons

- higher complexity
- higher maintenance
- lower network efficiency
- more parts to configure (and buy)

N-Tier Architecture

- Design your application using as many “tiers” as you need
- Use Object-Oriented Design techniques
- Put the various components on whatever host makes sense
- Java allows N-Tier Architecture, especially with RMI and JDBC

Database Technologies

- Hierarchical
  - obsolete (in a manner of speaking)
  - any specialized file format can be called a hierarchical DB
- Relational (aka SQL) (RDBMS)
  - row, column
  - most popular
- Object-relational DB (ORDBMS)
  - add inheritance, blobs to RDB
  - NOT object-oriented -- “object” is mostly a marketing term
- Object-oriented DB (OODB)
  - data stored as objects
  - high-performance for OO data models

Relational Databases

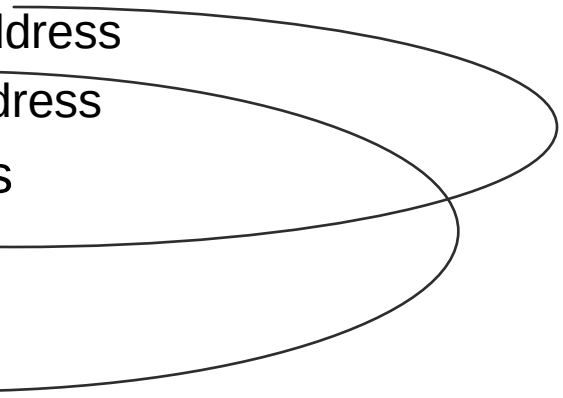
- invented by Dr. E.F.Codd
- data stored in *records* which live in *tables*
- maps *row* (record) to *column* (field) in a single *table*
- “relation” (as in “relational”) means row to column (not table to table)

Joining Tables

- you can associate tables with one another
- allows data to nest
- allows arbitrarily complicated data structures
- not object-oriented

Join example

- People
  - name
  - homeaddress
  - workaddress
- Addresses
  - id
  - street
  - state
  - zip



SQL

- Structured Query Language
- Standardized syntax for “querying” (accessing) a relational database
- Supposedly database-independent
- Actually, there are important variations from DB to DB

SQL Syntax

`INSERT INTO table ( field1, field2 ) VALUES ( value1, value2 )`

- inserts a new record into the named table

`UPDATE table SET ( field1 = value1, field2 = value2 ) WHERE condition`

- changes an existing record or records

`DELETE FROM table WHERE condition`

- removes all records that match condition

`SELECT field1, field2 FROM table WHERE condition`

- retrieves all records that match condition

Transactions

- Transaction = more than one statement which must all succeed (or all fail) together
- If one fails, the system must reverse all previous actions
- Also can’t leave DB in inconsistent state halfway through a transaction
- COMMIT = complete transaction
- ROLLBACK = abort

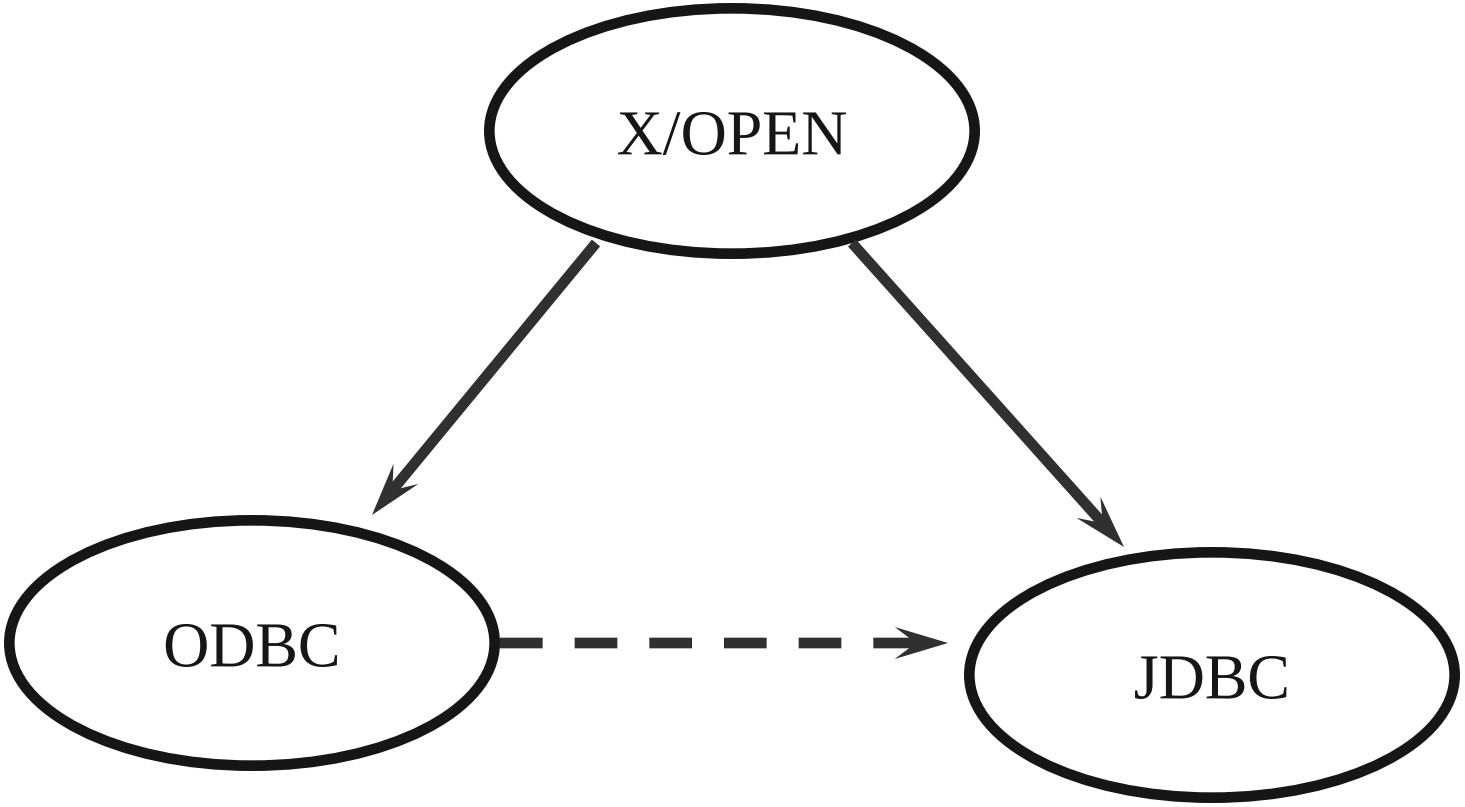
}

Part II: JDBC Overview

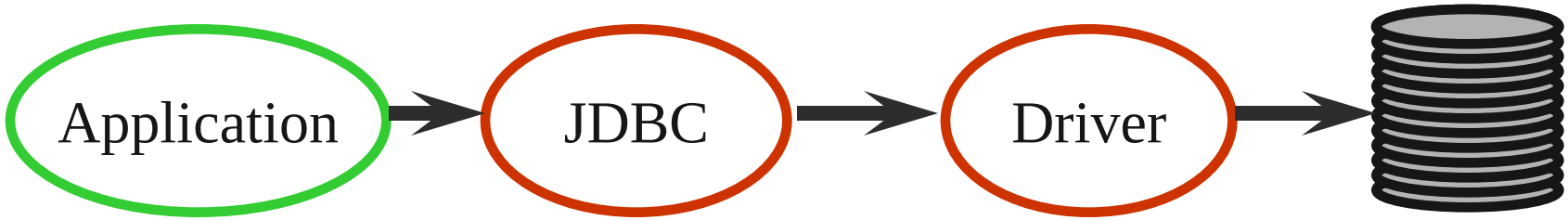
JDBC Goals

- 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

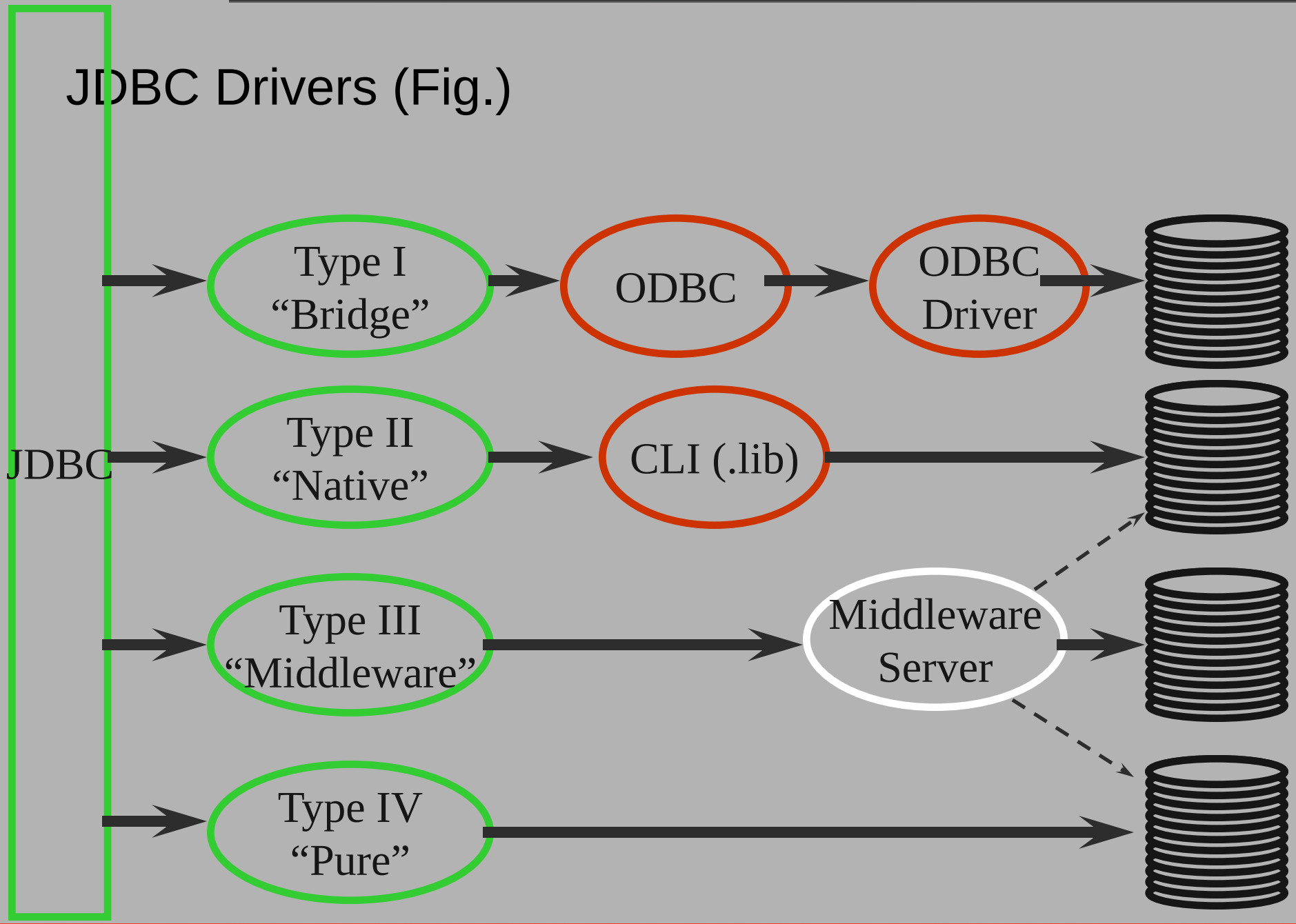


- Java code calls JDBC library
- JDBC loads a *driver*
- Driver talks to a particular database
- Can have more than one driver -> more than one database
- Ideal: can change database engines without changing any application code

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
- e.g. Intersolv Oracle Driver, WebLogic drivers

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
- e.g. Symantec DBAnywhere

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, mSQL

JDBC Limitations

- No scrolling cursors
- No bookmarks

Related Technologies

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

}

Part III: 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

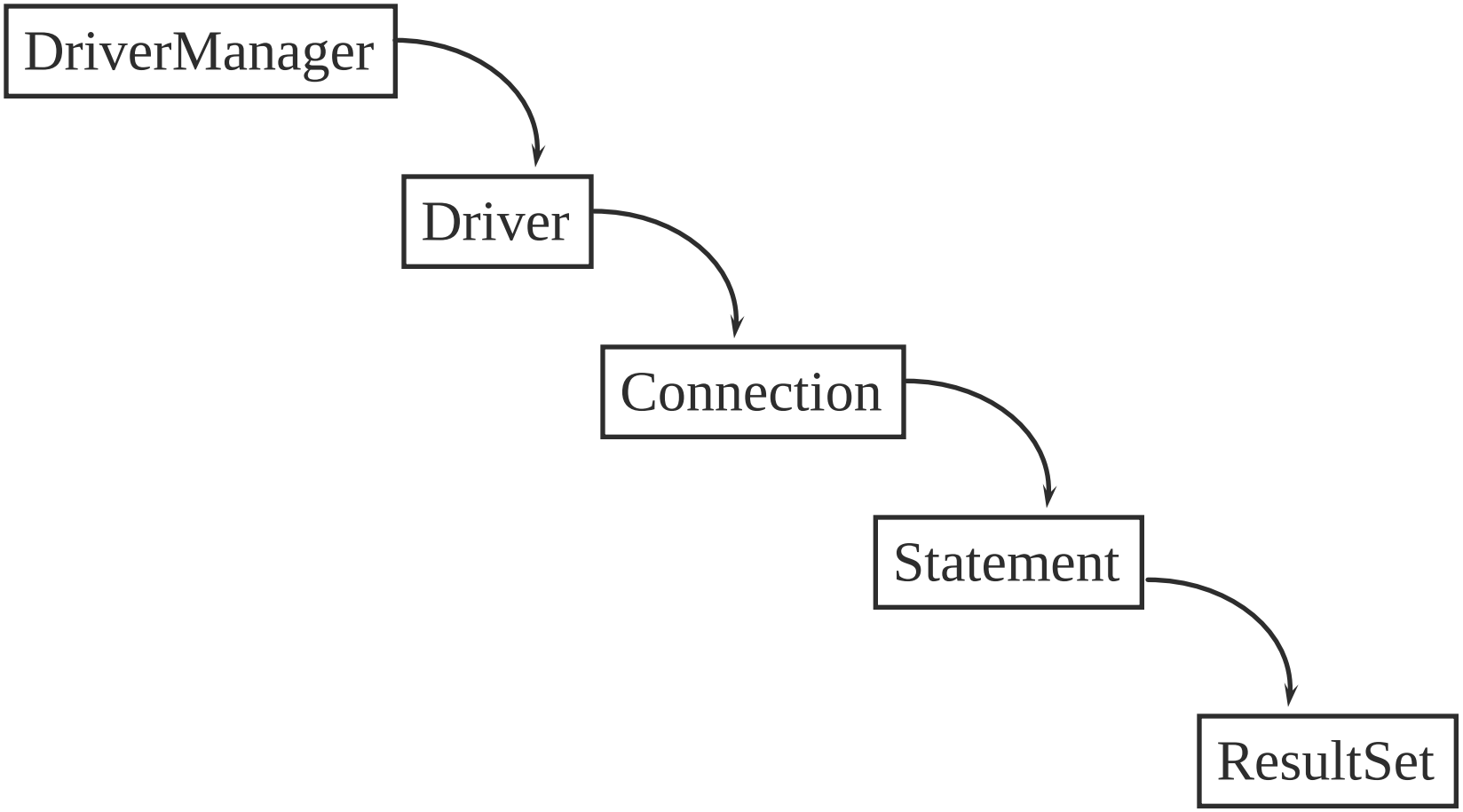
```
Class.forName("foo.bar.MyDriver");
Connection c = DriverManager.getConnection(...);
```

- 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.

} Why all these different kinds of queries? Optimization.

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.
  - you can’t rewind

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)

isNull

- In SQL, NULL means the field is empty
- Not the same as 0 or “”
- In JDBC, you must explicitly ask if a field is null by calling ResultSet.isNull(column)

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

Database Time

- Times in SQL are notoriously unstandard
- Java defines three classes to help
- java.sql.Date
  - year, month, day
- java.sql.Time
  - hours, minutes, seconds
- java.sql.Timestamp
  - year, month, day, hours, minutes, seconds, nanoseconds
  - usually use this one

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)

}

INSERT example

Transaction Management

- Transactions are not explicitly opened and closed
- Instead, the connection has a state called *AutoCommit* mode
- if *AutoCommit* is true, then every statement is automatically committed
- default case: true

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()

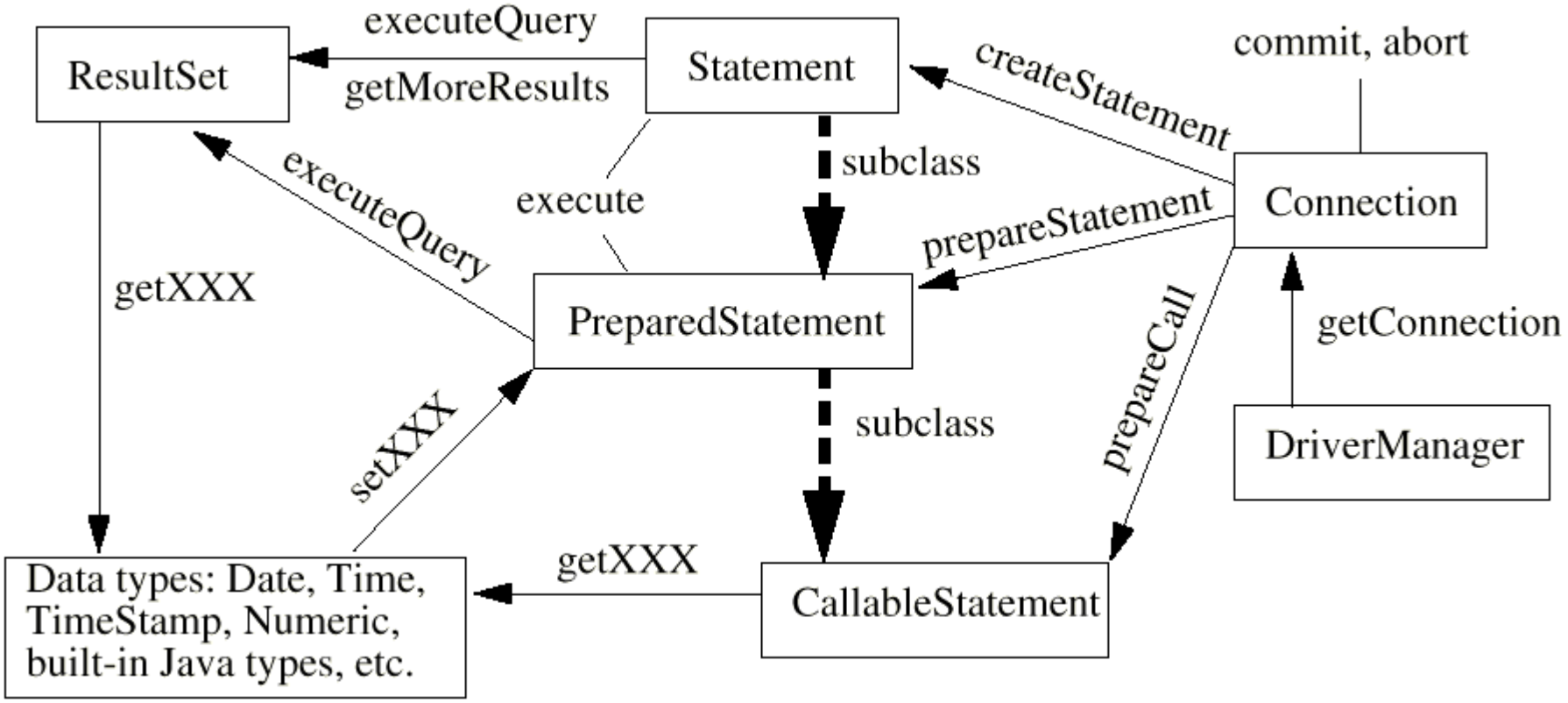
Connection Managers

- Hint: for a large threaded database server, create a Connection Manager object
- It is responsible for maintaining a certain number of open connections to the database
- When your applications need a connection, they ask for one from the CM's pool
- Why? Because opening and closing connections takes a long time
- Warning: the CM should always setAutoCommit(false) when a connection is returned

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



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…

}

JavaBlend: Java to Relational Mapping

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

Summary

- State what has been learned
- Define ways to apply training
- Request feedback of training session

Where to get more information

- Other training sessions
- Reese, *Database Programming with JDBC and Java* (O'Reilly)
- http://java.sun.com/products/jdbc/
- http://java.sun.com/products/java-blend/
- http://www.purpletech.com/java/ (Author's site)

About HSBC Technology and Services

HSBC Technology and Services (HTS) is a pivotal part of the Group and seamlessly integrates technology platforms and operations with an aim to re-define customer experience and drive down unit cost of production. Its solutions connect people, devices and networks across the globe and combine domain expertise, process skills and technology to deliver unparalleled business value, thereby enabling HSBC to stay ahead of competition by addressing market changes quickly and developing profitable customer relationships.

Presenter's Contact Details:
Name: Lorem ipsum
Role: Lorem ipsum
Direct: +00 00 000
Email: loremipsum@hsbc.com

Name: Lorem ipsum
Role: Lorem ipsum
Direct: +00 00 000
Email: loremipsum@hsbc.com