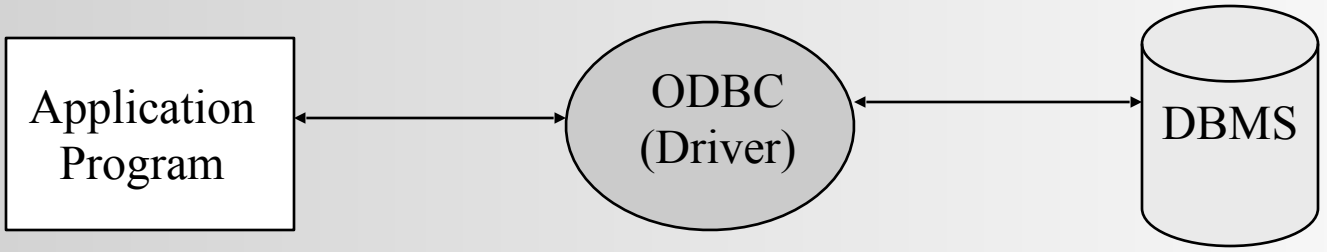
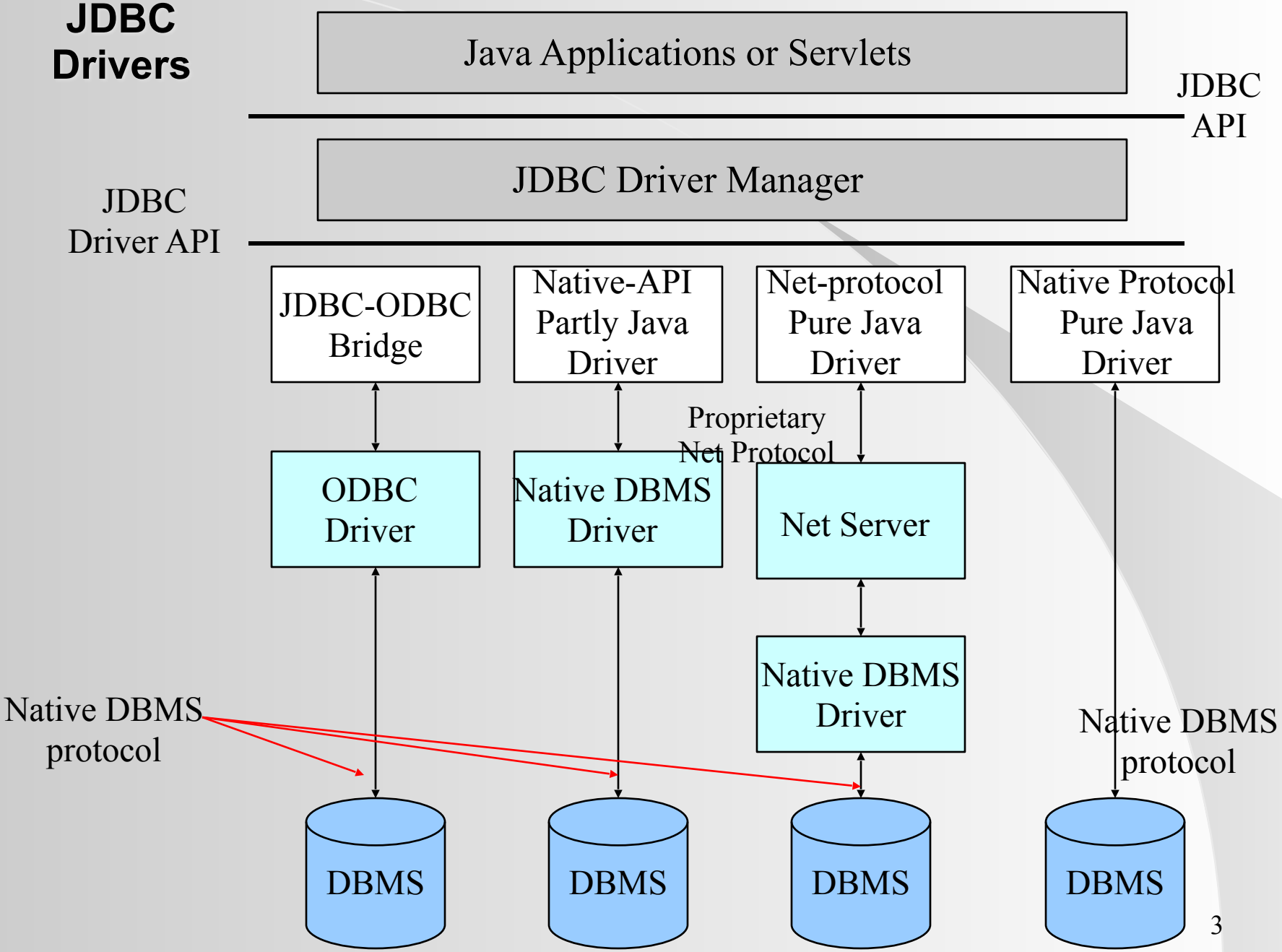
Java Database Connectivity

Java Database Connectivity - JDBC

- Java API (java.sql) that enables a Java program to execute SQL (Structured Query Language) statements and therefore interacts with any SQL-compliant database
- SQL is a standardized language to request information from a database, and the common SQL statements are CREATE, INSERT, UPDATE, DELETE and SELECT
- JDBC allows a DB application to run on different platform and interacts with all DBMSs that support SQL, i.e. access data across incompatible DBMSs on different platforms
- How about Microsoft's ODBC (Open Database Connectivity)?
  - ODBC connects programs with DBMSs by means of database drivers, which are C interfaces



JDBC Drivers



JDBC Driver Types

		Interface Openness	
		Vendor-independent	Vendor-specific
Interface Locality	Client-side native	Type 1: JDBC-ODBC bridge	Type 2: Native-API partly Java driver
	Remote network listener	Type 3: Net-protocol pure Java driver	Type 4: Native-protocol pure java driver

JDBC Drivers

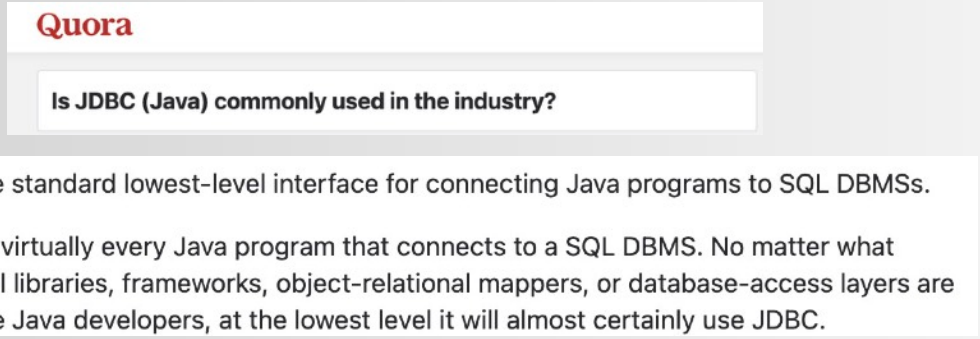
- 4 types of JDBC drivers:

1. *JDBC-ODBC bridge with ODBC driver*: there is a bridging JDBC-ODBC API, providing JDBC access to the DB via ODBC driver; the *JDBC-ODBC bridge* translates JDBC calls to ODBC (use native, non-Java code)
2. *Native-API Partly Java driver*: translates JDBC calls into a client API provided by the DB vendor (e.g. Oracle Call Interface OCI), which wraps around native code library
3. *Net-protocol All-Java driver*: translates JDBC calls to DBMS-independent network protocol which is then translated into a DBMS protocol by a server.
   Middleware approach: can interface to multiple database types
4. *Native-protocol All-Java driver*: translates JDBC calls into DBMS network protocol, i.e. talks directly to a DB using network protocol

JDBC is popular

- Java is still poplar in industry (including finance, IT, and bank).

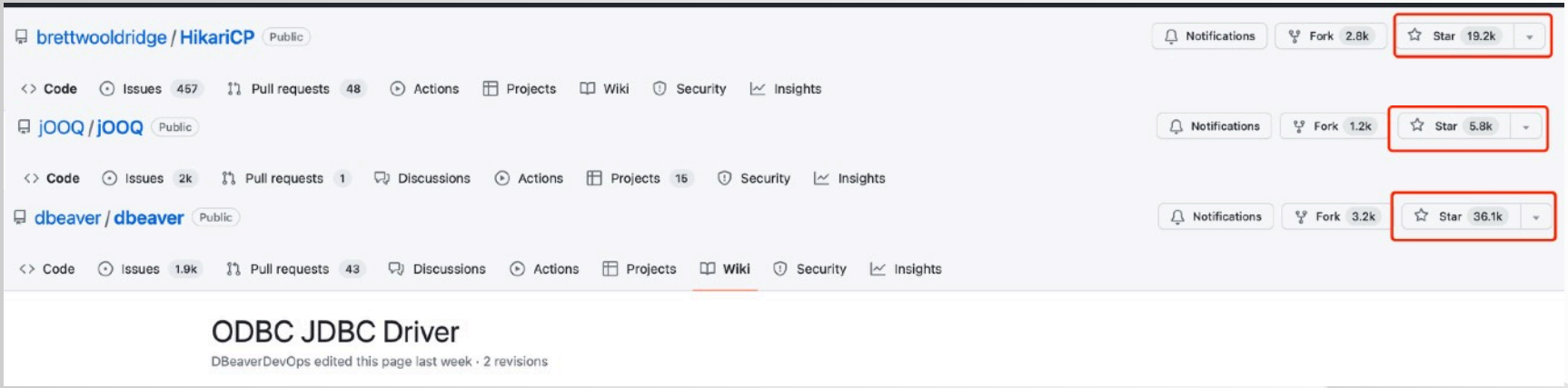
- JDBC is the default approach for using database in Java.



- https://www.cleveroad.com/blog/programming-languages-ranking/
- https://www.quora.com/Is-JDBC-Java-commonly-used-in-the-industry

JDBC is popular

- JDBC tools are continuously popular on GitHub

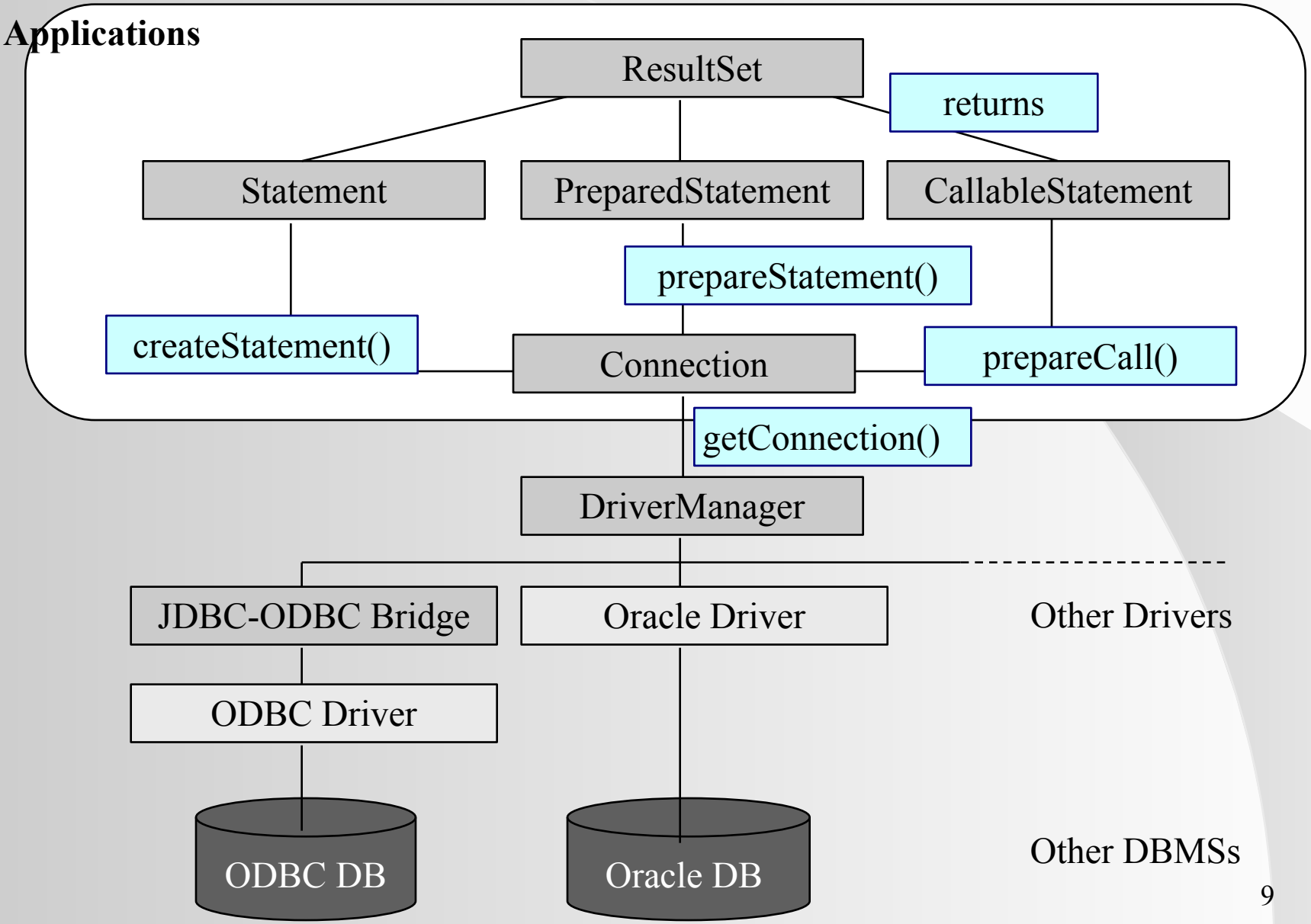


- (Almost) all Cloud providers highlight the support of JDBC.
  - AWS: https://github.com/awslabs/aws-advanced-jdbc-wrapper
  - Azure: https://learn.microsoft.com/en-us/azure/azure-sql/database/connect-query-java?view=azuresql
  - Google Cloud: https://github.com/GoogleCloudPlatform/cloud-sql-jdbc-socket-factory
  - Alibaba Cloud: https://www.alibabacloud.com/help/en/maxcompute/jdbc-driver
  - Huawei Cloud: https://doc.hcs.huawei.com/usermanual/dws/dws_01_0077.html

Using JDBC

- Which type of driver to use?
  - Speed
  - Reliability
  - Portability
  - Availability
- JDBC URL form: “jdbc:driver:database”, e.g.
  jdbc:odbc:datasource
  jdbc:oracle:thin@site:port:database
- Steps:
  1. Load a specific JDBC driver
  2. Establish a connection to a specific database (URL)
  3. Send SQL queries and obtain results (if any)
  4. Process query results
  5. Close connection

JDBC-Database Interaction



Step1. Load Drivers

- The JDBC java.sql.sql.DriverManager uses the Java system property java.sql.Drivers to identify the classes containing JDBC driver
  - The driver must be registered with JDBC DriverManager before its class can be loaded; or
  - The driver is automatically registered when its class is loaded
- 2 ways to load the drivers:
  1. Instantiate the driver:

  VendorDriver driver = new VendorDriver();
  2. Load the driver class:

  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
- When need a driver to connect to a DB, DriverManager will try the registered drivers one by one until it finds a driver which can handle the URL

Step 2. Connect to a Database

- Use the static method DriverManager.getConnection( ) to create a Connection object to a database at a designated URL:

```
Connection con = DriverManager.getConnection(
    “jdbc.odbc:organizationdb”, userid, password);
```

Step 3. SQL Queries and Results - Statement

- To make a query, need to create object from one of the following 3 interfaces:

1. Statement interface:
   - An interface to execute SQL statements: Statement objects are created by Connection.createStatement:

```
Statement statement = con.createStatement( );
```

   - 3 execution methods:

a) executeQuery( ):

```
a) ResultSet rs = statement.executeQuery("SELECT * FROM CUSTOMERS");
b) System.out.println(tableHeader);
c) while ( rs.next( ) ) {
     System.out.println(rs.getString("Name"));
     System.out.println(rs.getString(2));
     System.out.println(rs.getString("ADDRESS"));
     System.out.println(rs.getString("PHONE"));
d) }
```

Execute the select command: select all fields

A column can be obtained by column name or column index

Step 3. SQL Queries and Results - Statement (cont)

b) executeUpdate( ):
   - Executes SQL statements that do not return results, e.g. UPDATE, DELETE
   - Returns an integer which indicates the number of rows that were altered

c) execute( ):
   - Used if do not know whether the SQL statements will return results or not
   - Returns a Boolean:
     - true: if there is a result, which can be obtained by getResultMethod()
     - false: otherwise
   - Can use getUpdateCount() to obtain number of rows updated

Step 3. SQL Queries and Results - PreparedStatement

2. PreparedStatement interface:
   - An interface to precompile SQL statements to increase performance:

   PreparedStatement pstatement = con.preparedStatement(
   "INSERT INTO CUSTOMERS (NAME,ID) VALUES (?,?)");
   - The PreparedStatement object is created with a SQL statement, which is not executed
   - The SQL is compiled and can be run repeatedly
   - Precompilation is for performance improvement
   - Steps to execute the precompiled SQL statement:
     - Use the set( ) method to supply the SQL parameters
     - Use the executeUpdate( ) method to execute the SQL
     - E.g

       pstatement.clearParameters( );
       pstatement.setString(1, "HKUCSIS");
       pstatement.setString(2, csis@hku.hk);
       pstate.executeUpdate( );

Step 3. SQL Queries and Results - CallableStatement

3. CallableStatement interface:
   - An interface to access SQL stored procedures within the DB

CallableStatement cstatement = con.preparedCall("{call newBalance(?,?)}");
OR
CallableStatement cstatement = con.preparedCall("{? = call newBalance(?,?)}");

   - Stored procedure:
     - Procedural code embedded in the DB which can be called by other applications
     - Write once but invoked many times by different applications
   - Assume newBalance(cust_id,balance) is a stored procedure

```
CallableStatement cs = con.preparedCall("{call newBalance(?,?)}");
cs.registerOutParameter(2,Types.FLOAT);
cs.setInt(1,accountID);
cs.setFloat(2,balance);
cs.execute();
System.out.println("New balance is" + cs.getFloat(2));
```

Parameter 2 is an output parameter

Set values to the 2 input parameters

Step 4. Close Connections

- Close to release all DB resources:

```
rs.close( );
statement.close( );
con.close( );
```

JDBC Transaction - Auto-commit in JDBC

- By default, JDBC commits each SQL statement as it is sent to the database, called auto-commit

```
class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
Statement statement = con.createStatement( );
ResultSet rs = statement.executeQuery("SELECT * FROM CUSTOMERS");
System.out.println(tableHeader);
while ( rs.next( ) ) {
        System.out.println(rs.getString("Name"));
        System.out.println(rs.getString(2));
        System.out.println(rs.getString("ADDRESS"));
        System.out.println(rs.getString("PHONE"));
}
PreparedStatement pstatement = con.preparedStatement(
     "INSERT INTO CUSTOMERS (NAME,ID) VALUES (?,?)" );
pstatement.clearParameters( );
pstatement.setString(1, "HKUCSIS");
pstatement.setString(2, csis@hku.hk);
pstate.executeUpdate( );
```

Transaction with Multiple SQL Statements

- Set up a connection object to issue a series of changes that have no effect on the database until you send a “commit”
- Each connection is *separate*, and a commit on one connection has no effect on the statements on another
- To support transaction with multiple SQL statements, you need to turn auto-commit off by

  connection.setAutoCommit(false);

Whey do you need transaction with multiple SQL statements?

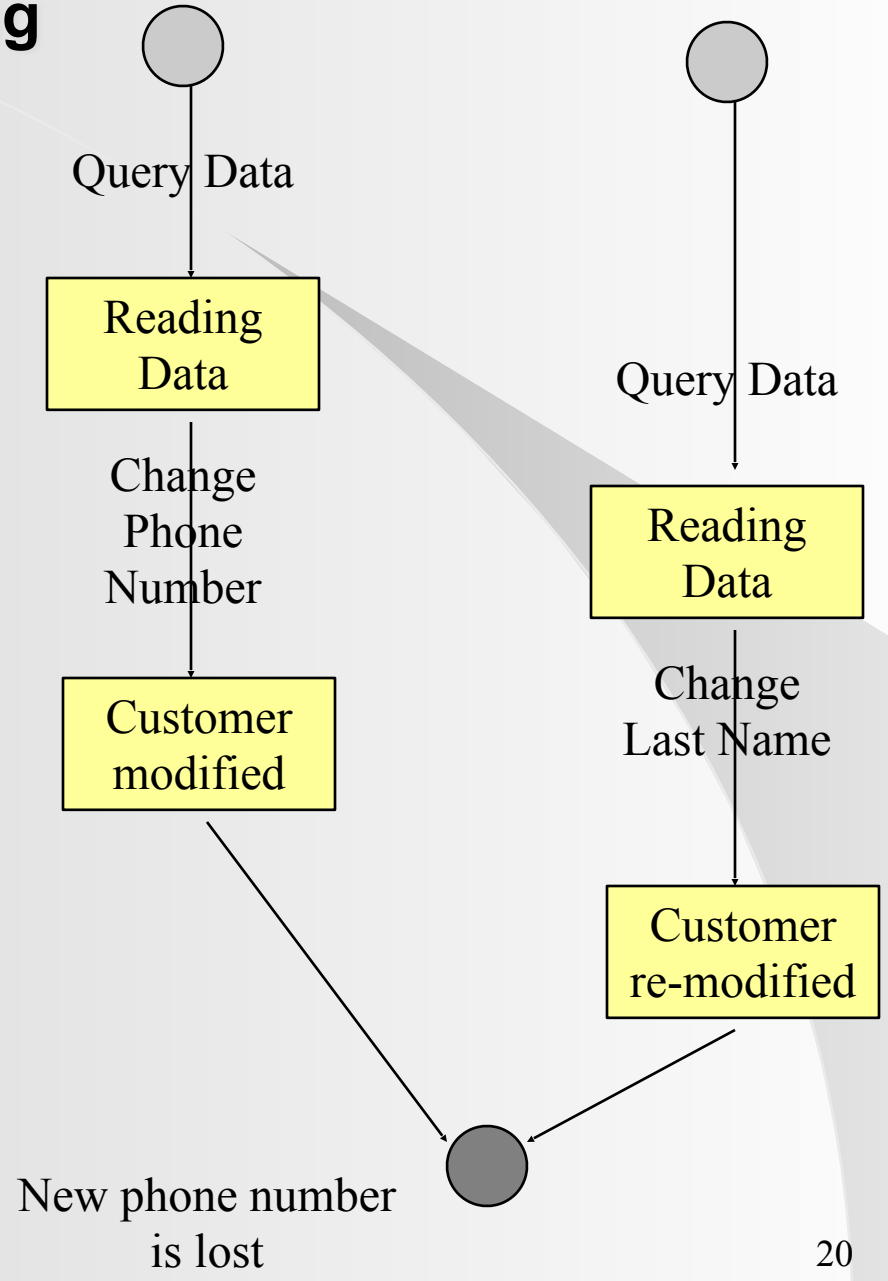
Basic Transaction Management

```
public void transfer(Account targ, float amt) {
        Connection conn = null;
        conn.setAutoCommit(false);
try {
                PreparedStatement stmt;
                conn = dataSource.getConnection();
                stmt = conn.prepareStatement("UPDATE Account SET balance
    = ?" +
                                                "WHERE id = ?");
                stmt.setFloat(1, balance-amt);
                stmt.setInt(2, id);
                stmt.executeUpdate();
                stmt.setFloat(1, targ.balance + a
                stmt.setInt(2, targ.id);
                stmt.executeUpdate();
                balance = balance – amt;
                targ.balance = targ.balance + am
                conn.commit();
}
```

```
catch (SQLExeception e) {
    try { conn.rollback(); }
    catch (SQLException e) { }
}
finally {
    if ( conn != null ) {
        try { conn.close(); }
        catch ( SQLException e ) { }
    }
} }
```

“Real” Transaction Handling

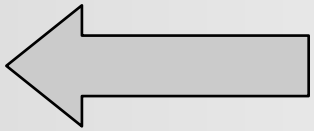
1. Read data from the database
2. Display the data
3. Modify the data
4. Update the database with the modified data



Pessimistic Concurrency

- Assume that some other transactions will make changes to your data behind your back

1. Start transaction
2. Read data from the database
3. Display the data
4. Modify the data
5. Update the database with the modified data
6. End transaction



- What's the problem?
  - Huge bottleneck
- Improved version:
  - Use SELECT FOR UPDATE just prior to actually sending changes to the database using the values returned from the first query
  - If the rows no longer match the original select, you will receive no rows and lock no resources in the DB

Pessimistic Concurrency Example

```
SELECT firstName, lastName, phone, birthday FROM Customer WHERE id = ?;

// User input

SELECT firstName, lastName, phone, birthday FROM Customer
WHERE id = ? AND firstName = ? AND lastName = ? AND phone = ? AND birthday = ?
FOR UPDATE

UPDATE Customer
SET firstName = ? AND lastName = ? AND phone = ? AND birthday = ?
WHERE id = ?
```

- Relatively long-lived transaction
- Placing columns that are probably not indexed in a SELECT statement

Optimistic Concurrency

- Assume no other transaction attempt to make changes to your data behind your back: use values from the original row in the WHERE clause

```
SELECT firstName, lastName, phone, birthday FROM Customer
WHERE id = ?

// make changes here

UPDATE Customer
SET firstName = ? AND lastName = ? AND phone = ? AND birthday = ?
WHERE id = ? AND firstName = ? AND lastName = ? AND phone = ? AND birthday
    = ?
```

- Very likely matching against unindexed columns

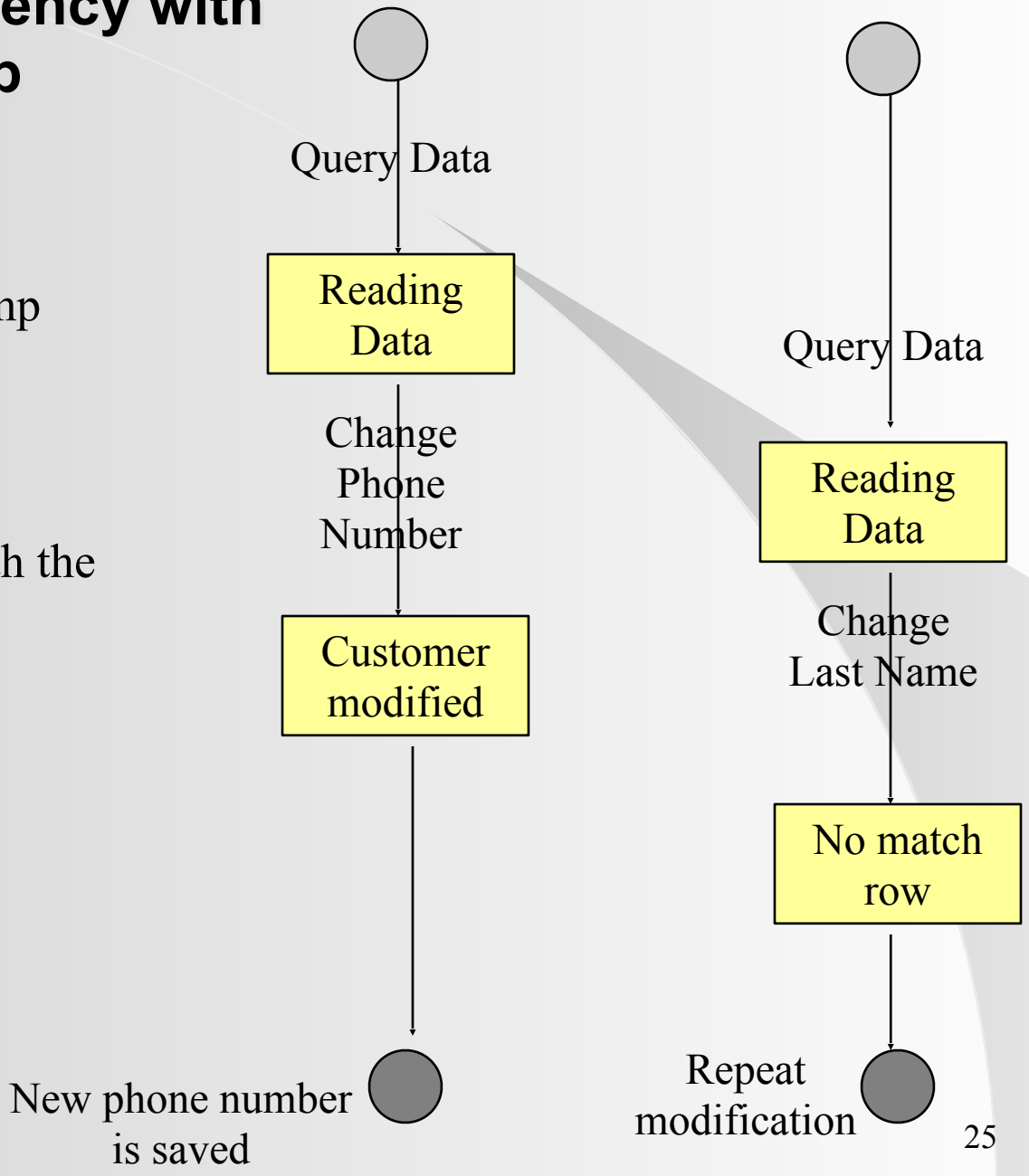
Optimistic Concurrency with Timestamp

```
try {
    PreparedStatement stmt = null;
    long ts;
    conn = ds.getConnection();
    stmt = conn.prepareStatement("UPDATE Customer SET firstName = ?," +
            "lastName=?, phone=?, birthday=?, lastUpdateTS=?" +
            "WHERE id=? AND lastUpdateTS=?");
    stmt.setString(1, firstName);
    stmt.setString(2, lastName);
    stmt.setString(3, phone);
    stmt.setString(4, birthDate);
    stmt.setLong(5, ts=System.currentTimeMillis());
    stmt.setInt(6, id);
    stmt.setLong(7, lastUpdateTS);
    stmt.executeUpdate();
    lastUpdateTS = ts;
```

```
catch (SQLExeception e) {
   e.printStackTrace();
   // no rollback needed
}
finally {
   if ( conn != null ) {
      try { conn.close(); }
      catch ( SQLException e ) { }
   }
} }
```

Optimistic Concurrency with Timestamp

1. Read data with timestamp from the database
2. Display the data
3. Modify the data
4. Update the database with the modified data and new timestamp



Batch Transactions

- Batch transactions are sets of transactions that occur on the server independent of user interaction, e.g. the monthly process that calculates interest for a saving account
- JDBC 2.0 introduced a batch processing mechanism: store multiple statements on the client to be sent over to the DB as a group (batch)
- The more updates you hold on the client, the faster the batch processing occurs, but the number of updates is limited by:
  - Amount of memory you have
  - The size of the database transaction log
  - Complexity of recovery processing: determine when was the last update and then recover from the failure point

Batch Transactions Example

```
PreparedStatement stmt;
conn.setAutoCommit(false);
stmt = conn.prepareStatement("UPDATE account SET balance = ? WHERE id=?");

for ( int i=0; i<accts.length; i++ ) {
    int[] rows;
    while ( (i%10 != 9) && (i<accts.length) ) {
            accts[i].calculateInterest();
            stmt.setDouble(1, accts[i].getBalance());
            stmt.setInt(2, accts[i].getId());
            stmt.addBatch();
            i++;
    }
    rows = stmt.executeBatch();
}
```