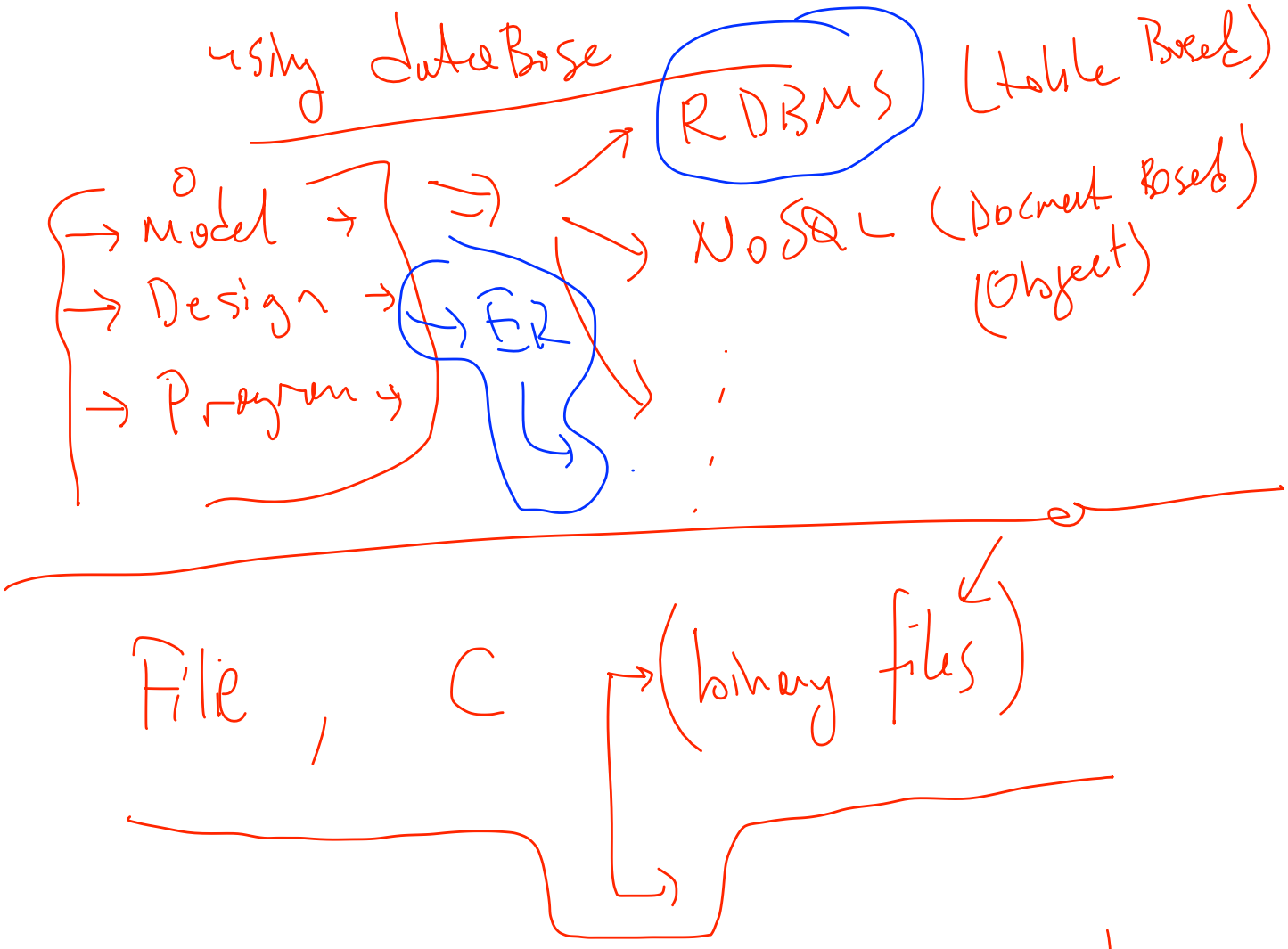


* What is the purpose of storing information in a File?

✗ RAM ⇒ ✓

→ Grocery ⇒ → File ← Single

→ University → Many GPA

→ DataBase Systems

Table all the

headers — email — ⇐ phonebook ⇒ — Customers

ID	Name	Surname	Address	Contact Number	TC
1	Ahmet	Keser	Kadiköy	1234	✓
2					

→ Key

GUID ⇒ * Primary Key — Foreign key

Relation "officially"

1 – 1 → person — married → person

1 – N → cars — owner → person

N ← 1

N – M → books ⟵ reserved ⟶ students

N — M

N – 1

FR

Vendors

↓ DBSystem ⇒

* Microsoft SQL Server (free)
* Oracle DB
* IBM DB2
* Progress

↓ free

→ MySQL

SQL

Structured Query Language

→ Mongo DBs

* Google
* Unilang
* Library

Access DB

Cloud (SaaS)

MSq My

Intro SQL

DDL
→ create DB
object
+ Table
+ Relations
+ Transactions
+ View
+ functions
+ Stored Procedures
+ triggers
+ Index
Key

Oracle

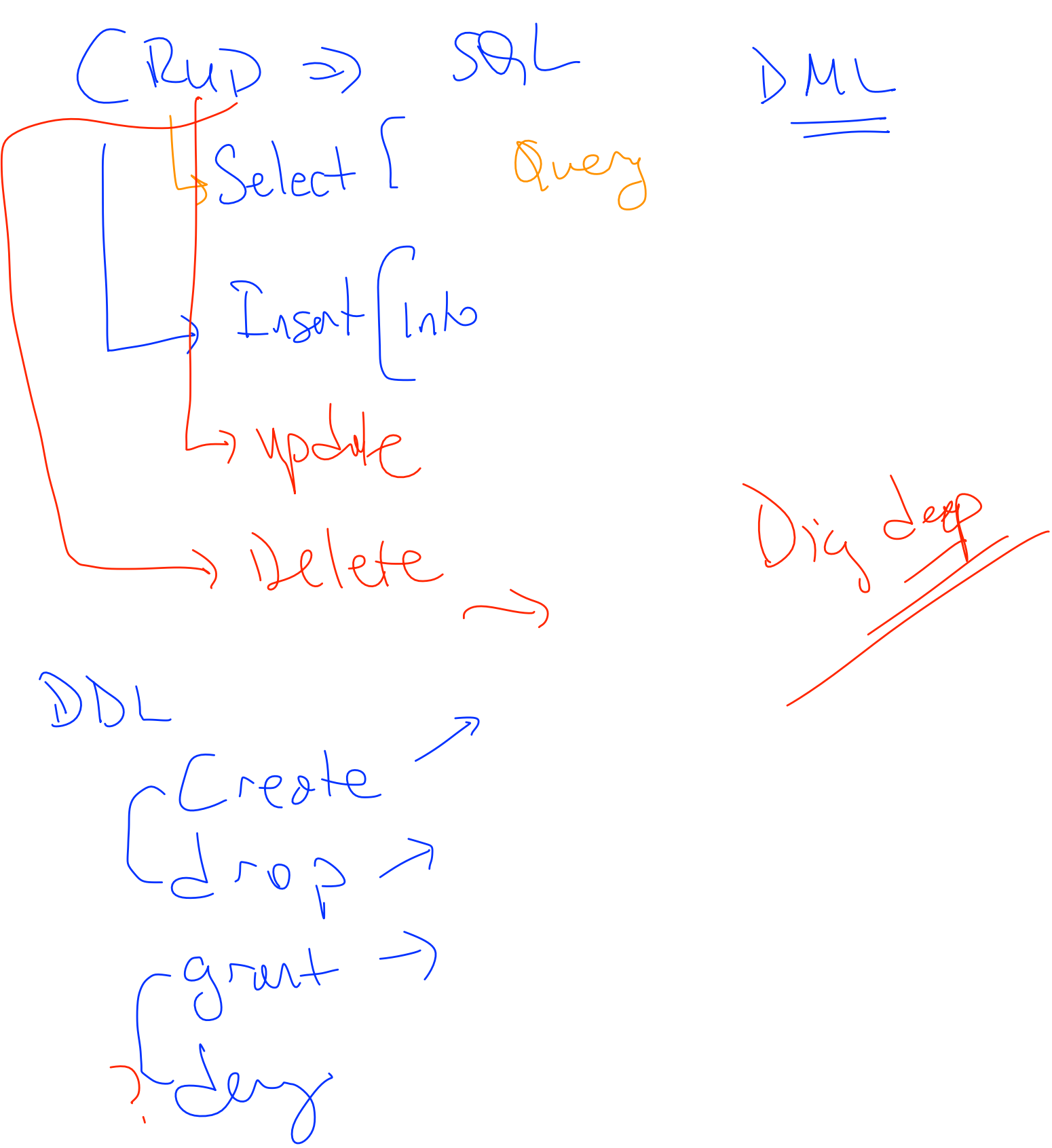
DML
→ Query
→ update
List
retrieve
read
Change
Insert
delete

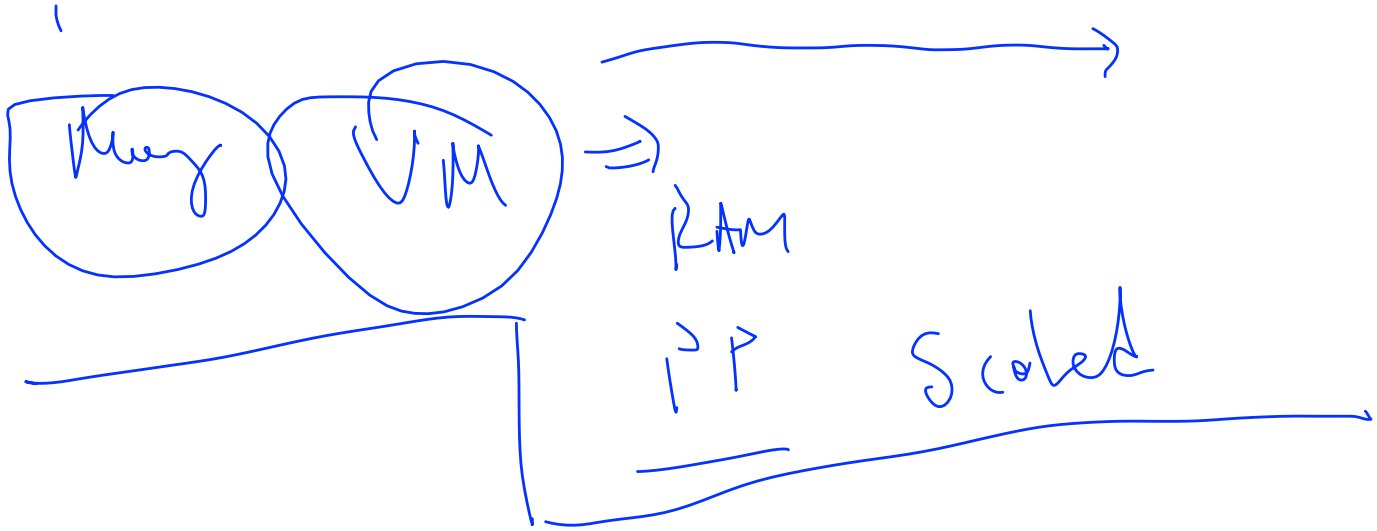
CRUD
Create
Read
update
delete

tree like
B+

sorted

Hash
index
f(0) = H


CRUD ⇒ SQL
DML
Select [Query
Insert [Into
Update
Delete
Dig deep
DDL
[Create
[Drop
[grant
[deny



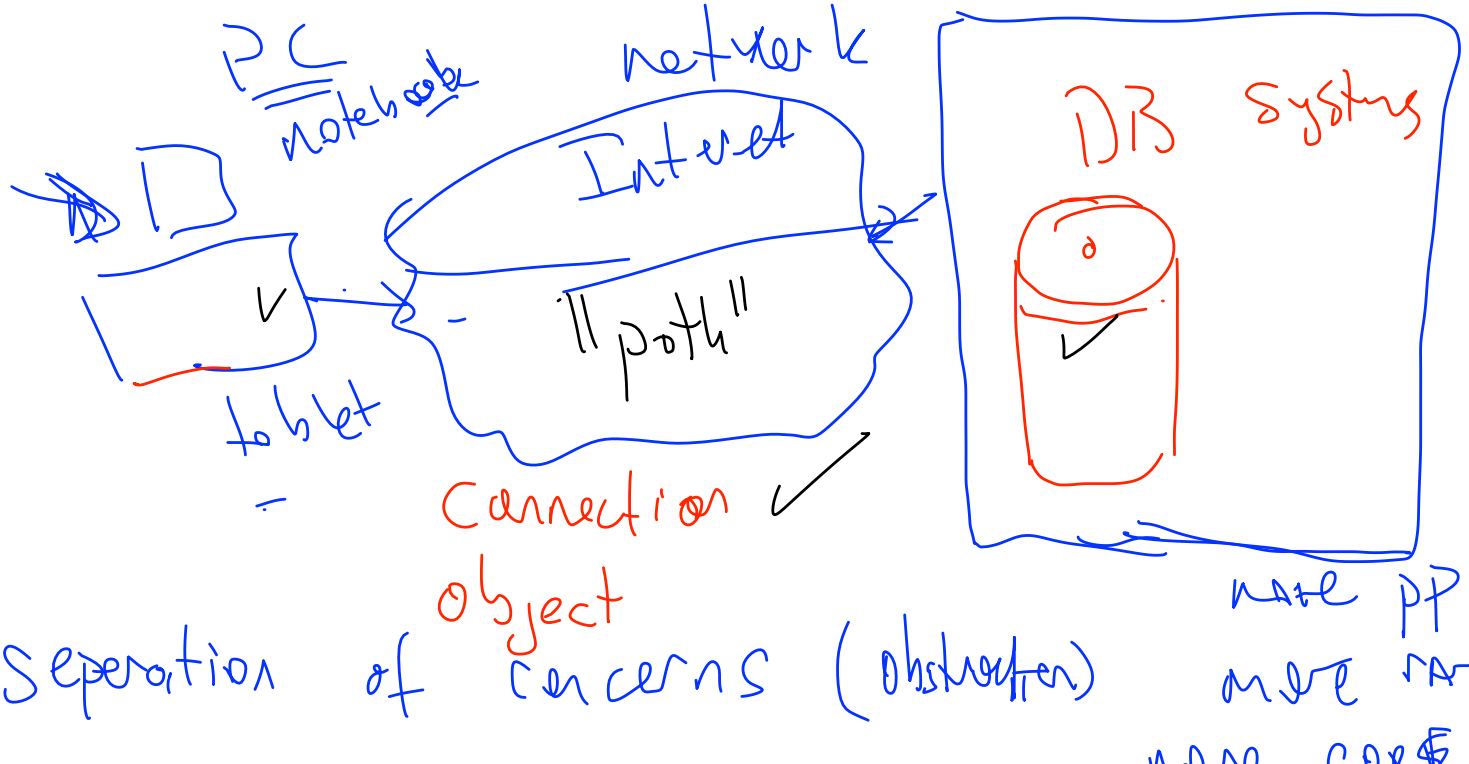
Java technologies ⇒

to Access a DB system

JDBC ⇒

Java DataBase Connectivity

end user



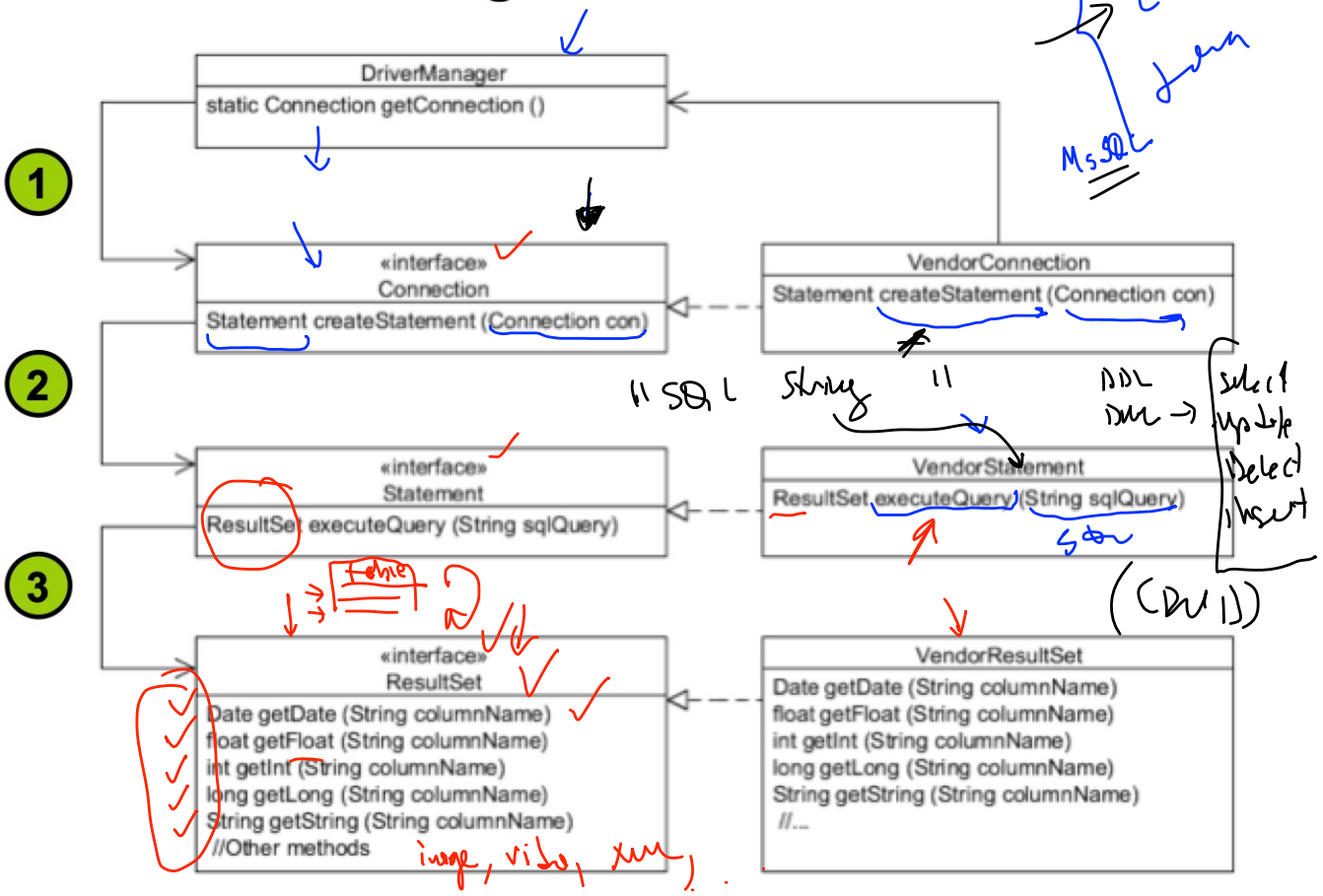
Separation of concerns (abstraction)

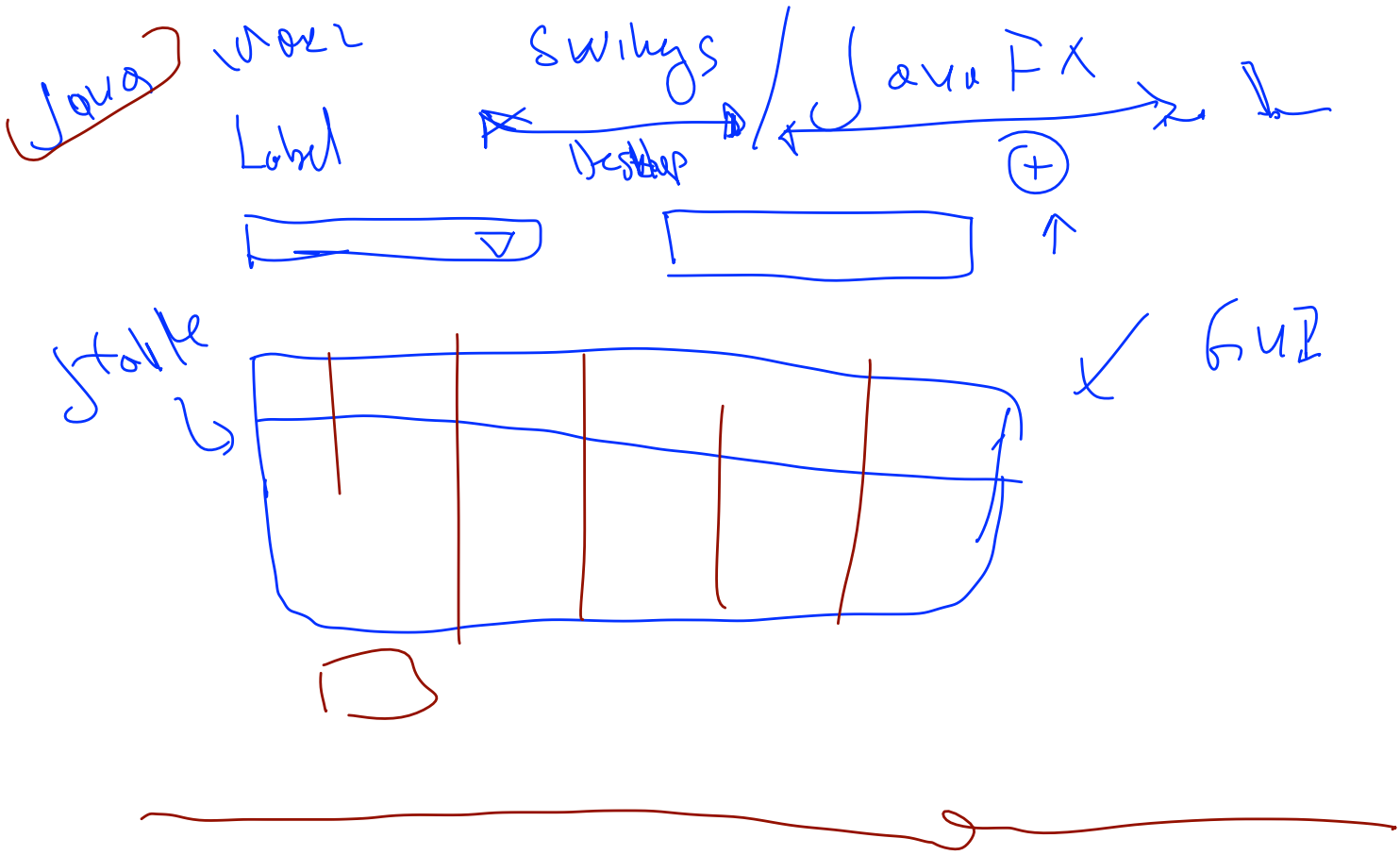
more pp
more ram
more cores

14

Building Database Applications with JDBC

Using the JDBC API





Using a Vendor's Driver Class

The `DriverManager` class is used to get an instance of a Connection object, using the JDBC driver named in the JDBC URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

```
public Connection getConnection() throws SQLException {

    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", this.userName);
    connectionProps.put("password", this.password);

    if (this.dbms.equals("mysql")) {
        conn = DriverManager.getConnection(
                   "jdbc:" + this.dbms + "://" +
                   this.serverName +
                   ":" + this.portNumber + "/",
                   connectionProps);
    } else if (this.dbms.equals("derby")) {
        conn = DriverManager.getConnection(
                   "jdbc:" + this.dbms + ":" +
                   this.dbName +
                   ";create=true",
                   connectionProps);
    }
    System.out.println("Connected to database");
    return conn;
}
```

Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

- `java.sql.Connection`: A connection that represents the session between your Java application and the database

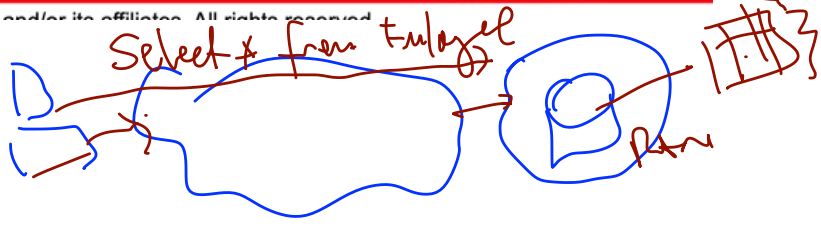
```
Connection con = DriverManager.getConnection(url, username,
    password);
```

- `java.sql.Statement`: An object used to execute a static SQL statement and return the result

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

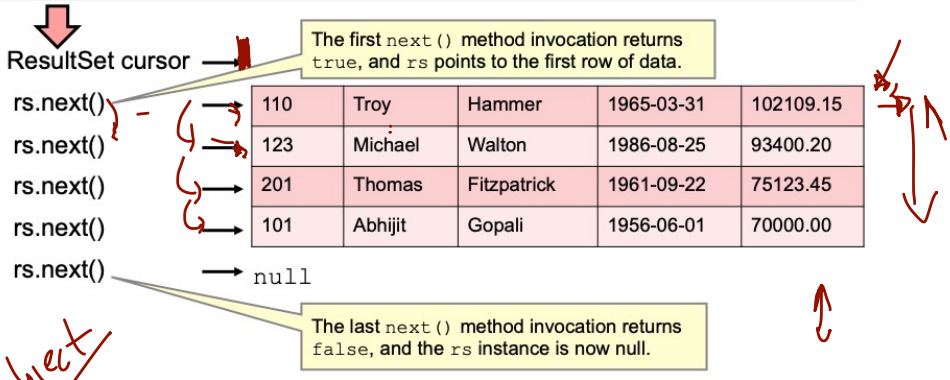
- `java.sql.ResultSet`: A object representing a database result set

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

Using a ResultSet Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```



ResultSet Objects

- ResultSet maintains a cursor to the returned rows. The cursor is initially pointing before the first row.
- The ResultSet.next() method is called to position the cursor in the next row.
- The default ResultSet is not updatable and has a cursor that points only forward.
- It is possible to produce ResultSet objects that are scrollable and/or updatable. The following code fragment, in which con is a valid Connection object, illustrates how to make a result set that is scrollable and insensitive to updates by others, and that is updatable:

```
Statement stmt
    = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                          ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

Note: Not all databases support scrollable result sets.

ResultSet has accessor methods to read the contents of each column returned in a row. ResultSet has a getter method for each type.

Putting It All Together

```
1  package com.example.text;
2
3  import java.sql.DriverManager;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.util.Date;
7
8  public class SimpleJDBCTest {
9
10     public static void main(String[] args) {
11         String url = "jdbc:derby://localhost:1527/EmployeeDB";
12         String username = "public";
13         String password = "tiger";
14         String query = "SELECT * FROM Employee";
15         try (Connection con =
16             DriverManager.getConnection (url, username, password);
17             Statement stmt = con.createStatement ();
18             ResultSet rs = stmt.executeQuery (query)) {
```

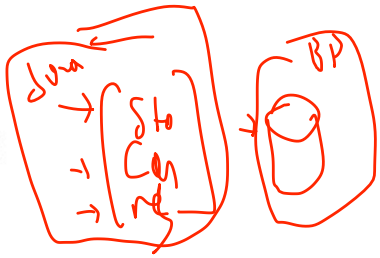
The hard-coded JDBC URL, username, and password is just for this simple example.

rows in the ResultSet.

```
19          while (rs.next()) {
20              int empID = rs.getInt("ID");
21              String first = rs.getString("FirstName");
22              String last = rs.getString("LastName");
23              Date birthDate = rs.getDate("BirthDate");
24              float salary = rs.getFloat("Salary");
25              System.out.println("Employee ID:    " + empID + "\n"
26              + "Employee Name: " + first + " " + last + "\n"
27              + "Birth Date:    " + birthDate + "\n"
28              + "Salary:        " + salary);
29          } // end of while
30      } catch (SQLException e) {
31          System.out.println("SQL Exception: " + e);
32      } // end of try-with-resources
33  }
34 }
```

This example is from the SimpleJDBCExample project.
Output:
run:
Employee ID: 110
Employee Name: Troy Hammer
Birth Date: 1965-03-31
Salary: 102109.15

etc.

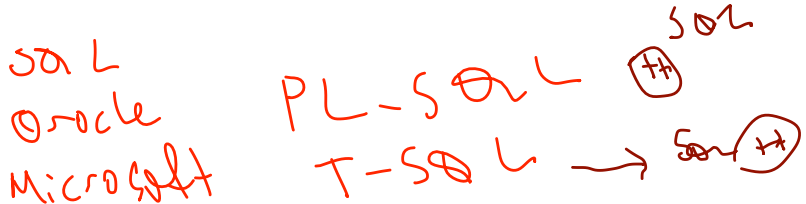


Writing Portable JDBC Code

The JDBC driver provides a programmatic "insulating" layer between your Java application and the database. However, you also need to consider SQL syntax and semantics when writing database applications.

- Most databases support a standard set of SQL syntax and semantics described by the American National Standards Institute (ANSI) SQL-92 Entry-level specification.
- You can programmatically check for support for this specification from your driver:

```
Connection con = DriverManager.getConnection(url, username,
    password);
DatabaseMetaData dbm = con.getMetaData();
if (dbm.supportsANSI92EntrySQL()) {
    // Support for Entry-level SQL-92 standard
}
```



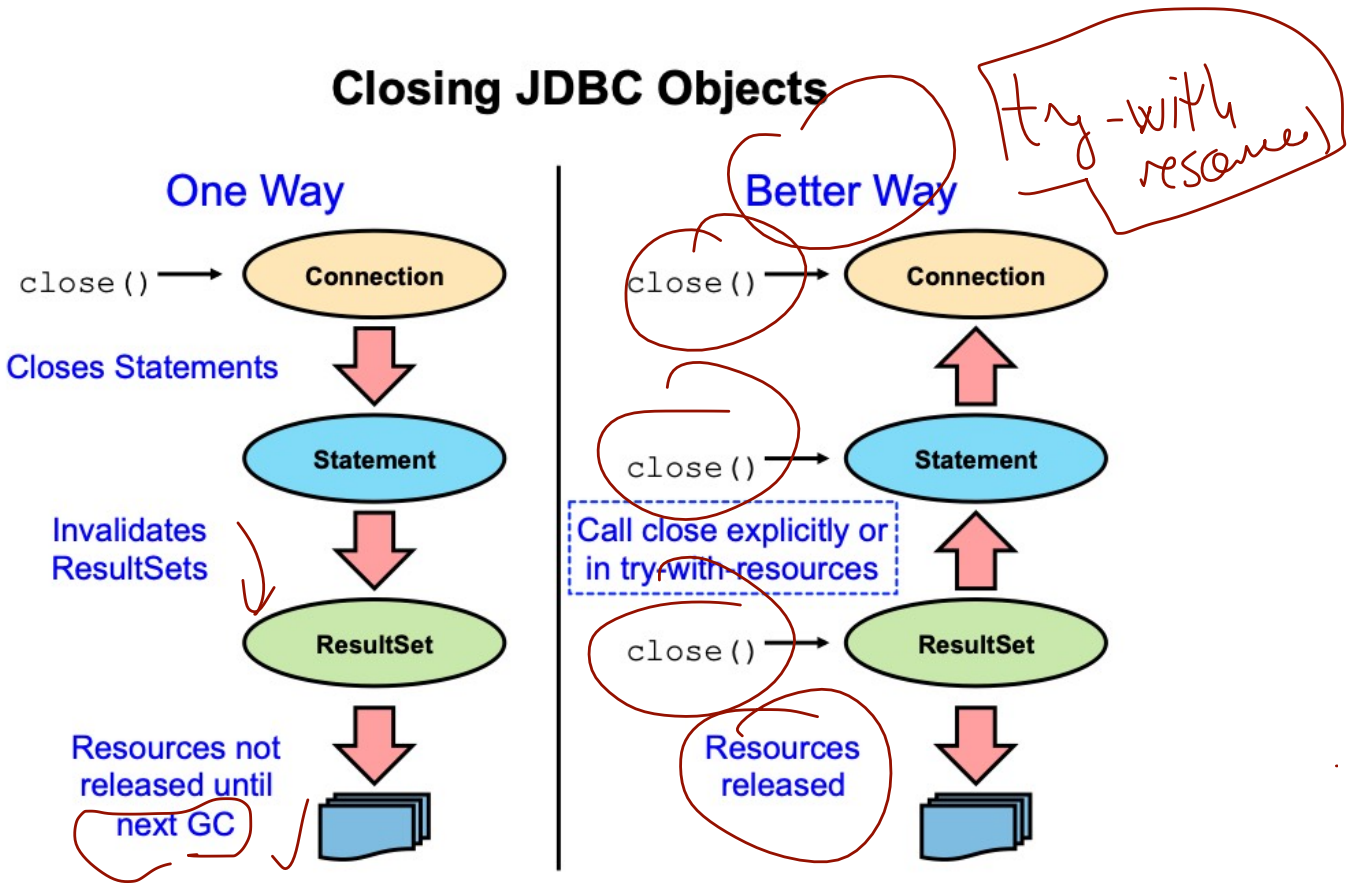
The SQLException Class

`SQLException` can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the `SQLException`s thrown:

```
1  catch(SQLException ex) {
2      while(ex != null) {
3          System.out.println("SQLState:   " + ex.getSQLState());
4          System.out.println("Error Code:" + ex.getErrorCode());
5          System.out.println("Message:    " + ex.getMessage());
6          Throwable t = ex.getCause();
7          while(t != null) {
8              System.out.println("Cause:" + t);
9              t = t.getCause();
10         }
11         ex = ex.getNextException();
12     }
13 }
```

Vendor-dependent state codes, error codes and messages

Closing JDBC Objects



The `try`-with-resources Construct

Given the following `try`-with-resources statement:

```
try (Connection con =
     DriverManager.getConnection(url, username, password);
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery (query)){
```

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
  - This interface includes one method: `void close()`.
- The close method is automatically called at the end of the try block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the try block, separated by semicolons.

try-with-resources: Bad Practice

It might be tempting to write `try`-with-resources more compactly:

```
try (ResultSet rs = DriverManager.getConnection(url, username,
password).createStatement().executeQuery(query)) {
```

- However, only the close method of `ResultSet` is called, which is not a good practice.
- Always keep in mind which resources you need to close when using `try`-with-resources.

Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the Statement object.

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

- Use the Statement instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery (query);
```

- Note that there are three Statement execute methods:

Method	Returns	Used for
`executeQuery(sqlString)`	`ResultSet`	`SELECT` statement
`executeUpdate(sqlString)`	`int` (rows affected)	`INSERT`, `UPDATE`, `DELETE`, or a DDL
`execute(sqlString)`	`boolean` (true if there was a `ResultSet`)	Any SQL command or commands

ORACLE

ResultSetMetaData

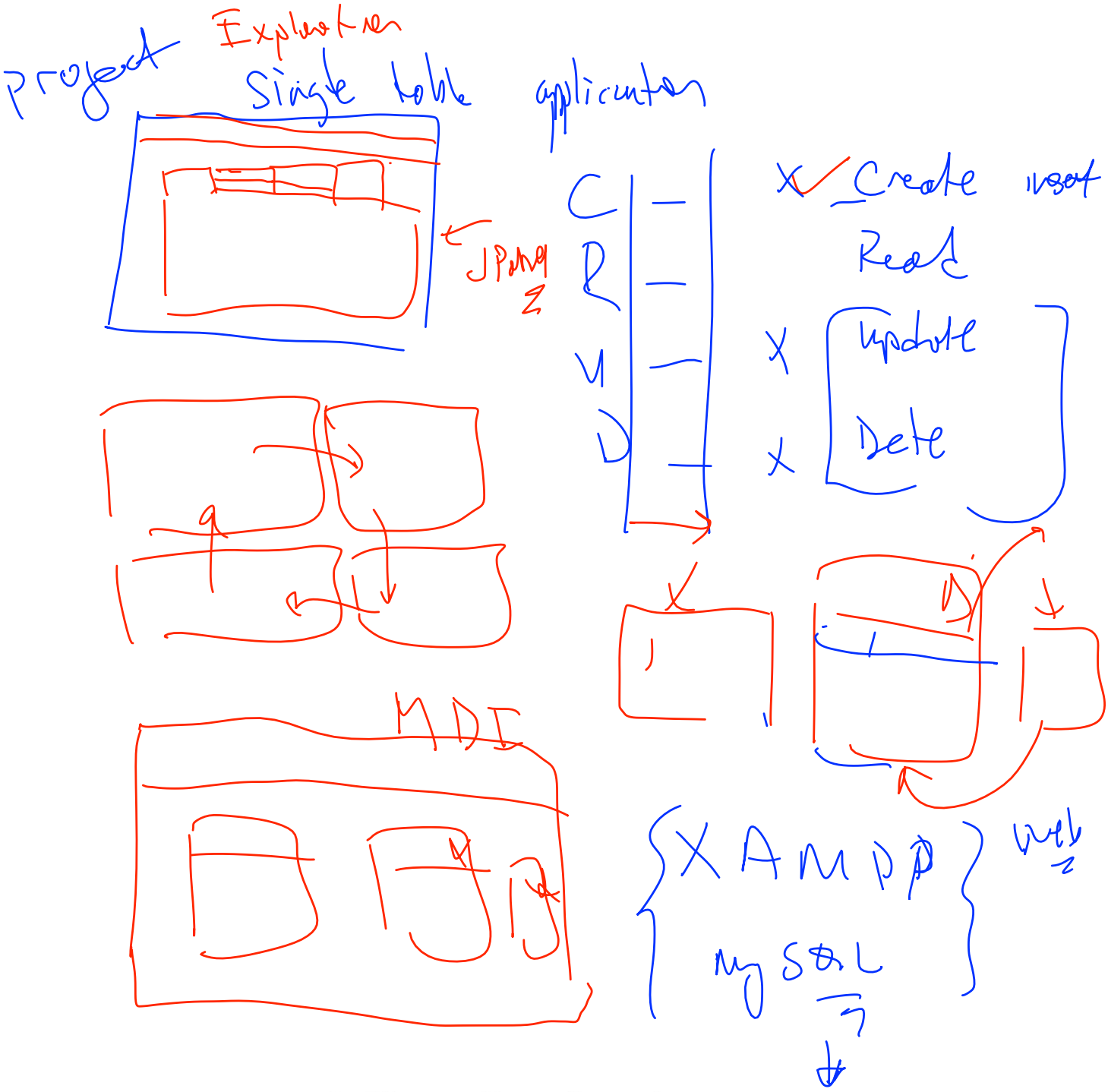
There may be a time where you need to dynamically discover the number of columns and their type.

```
1 int numCols = rs.getMetaData().getColumnCount();
2 String [] colNames = new String[numCols];
3 String [] colTypes = new String[numCols];
4 for (int i= 0; i < numCols; i++) {
5     colNames[i] = rs.getMetaData().getColumnName(i+1);
6     colTypes[i] = rs.getMetaData().getColumnTypeName(i+1);
7 }
8 System.out.println ("Number of columns returned: " + numCols);
9 System.out.println ("Column names/types returned: ");
10 for (int i = 0; i < numCols; i++) {
11     System.out.println (colNames[i] + " : " + colTypes[i]);
12 }
```

Note that these methods are indexed from 1, not 0.

The ResultSetMetaData class is obtained from a ResultSet.
The getColumnCount returns the number of columns returned in the query that produced the ResultSet.
The getColumnName and getColumnTypeName methods return strings. These could be used to perform a dynamic retrieval of the column data.
Note: These methods use 1 to indicate the first column, not 0.
Given a query of "SELECT * FROM Employee" and the Employee data table from the practices, this fragment produces this result:

```
Number of columns returned: 5
Column names/types returned:
ID : INTEGER
FIRSTNAME : VARCHAR
LASTNAME : VARCHAR
BIRTHDATE : DATE
SALARY : REAL
```



Getting a Row Count

A common question when executing a query is: "How many rows were returned?"

```
1 public int rowCount(ResultSet rs) throws SQLException{
2     int rowCount = 0;
3     int currRow = rs.getRow();
4     // Valid ResultSet?
5     if (!rs.last()) return -1;
6     rowCount = rs.getRow();
7     // Return the cursor to the current position
8     if (currRow == 0) rs.beforeFirst();
9     else rs.absolute(currRow);
10    return rowCount;
11 }
```

Move the cursor to the last row, this method returns `false` if the `ResultSet` is empty.

Returning the row cursor to its original position before the call is a good practice.

- To use this technique, the `ResultSet` must be scrollable.

```
reads the results. Be aware that this technique requires locking control over the tables to ensure th
ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM EMPLOYEE");
rs.next();
int count = rs.getInt(1);
rs.stmt.excuteQuery ("SELECT * FROM EMPLOYEE");
// process results
```

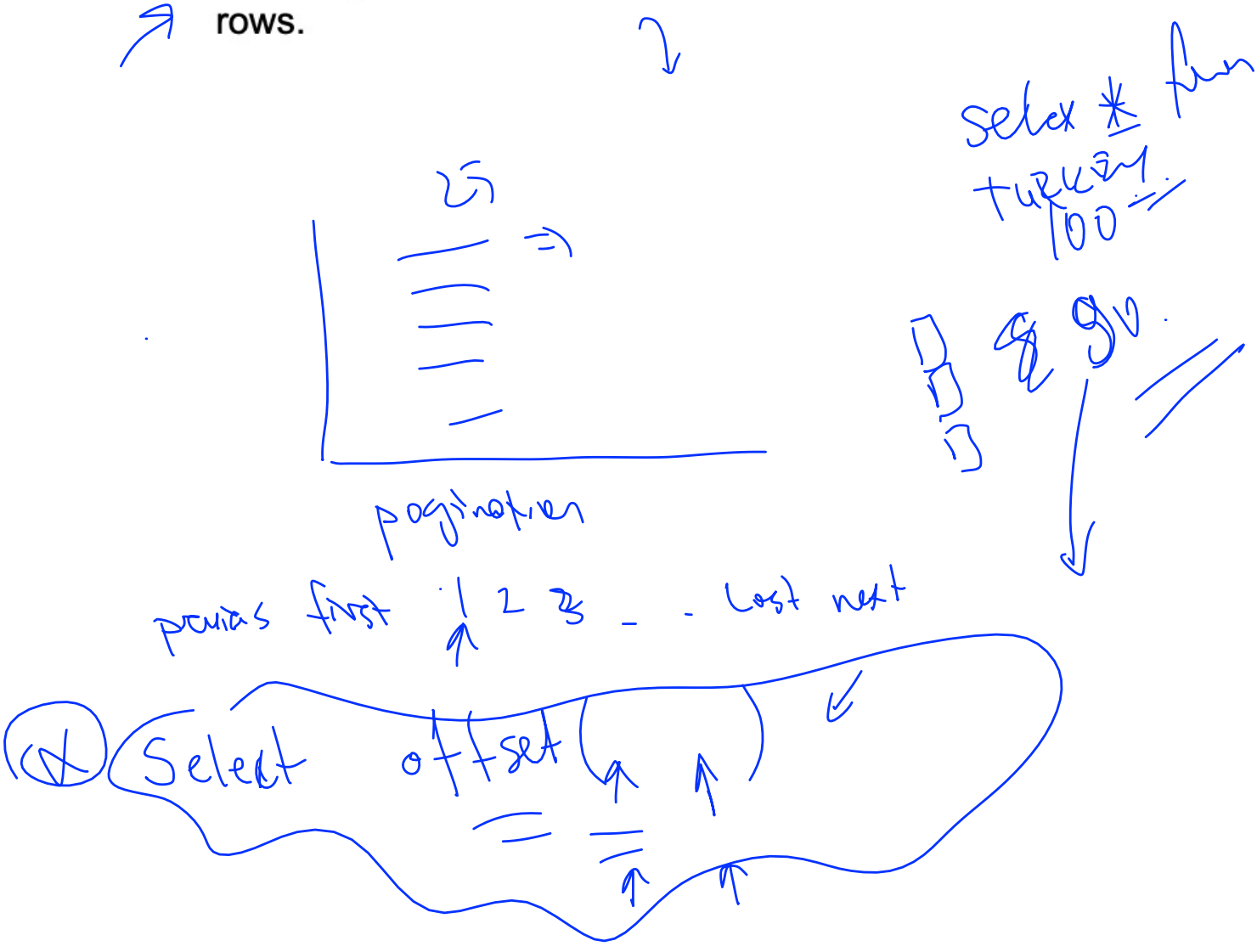
Controlling `ResultSet` Fetch Size

By default, the number of rows fetched at one time by a query is determined by the JDBC driver. You may wish to control this behavior for large data sets.

- For example, if you wanted to limit the number of rows fetched into cache to 25, you could set the fetch size:

```
rs.setFetchSize(25);
```

- Calls to `rs.next()` return the data in the cache until the 26th row, at which time the driver will fetch another 25 rows.



Using PreparedStatement

PreparedStatement is a subclass of Statement that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Parameter for substitution.

Substitutes value for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.
- PreparedStatement is useful when you have a SQL statements that you are going to execute multiple times.

Using CallableStatement

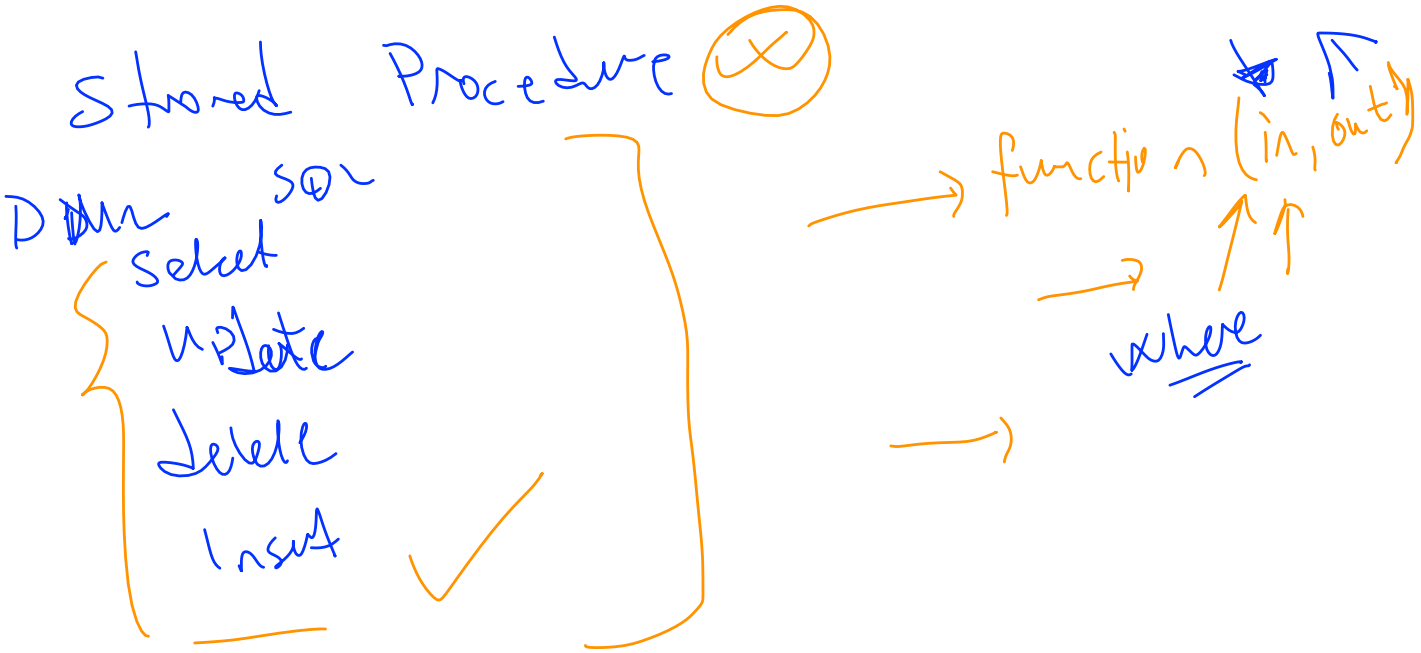
A CallableStatement allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt
       = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
          " Employees over the age of " + age);
```

The IN parameter is passed in to the stored procedure.

The OUT parameter is returned from the stored procedure.

- Stored procedures are executed on the database.



DDL

In the example shown in the slide, the stored procedure is declared using the following syntax:

```
CREATE PROCEDURE EmplAgeCount (IN age INTEGER, OUT num INTEGER) DYNAMIC RESULT SETS 0
LANGUAGE JAVA
EXTERNAL NAME 'DerbyStoredProcedure.countAge'
PARAMETER STYLE JAVA
READS SQL DATA;
```

A Java class is loaded into the Derby database using the following syntax:

```
CALL SQLJ.install_jar ('D:\temp\DerbyStoredProcedure.jar', 'PUBLIC.DerbyStoredProcedure', 0);
CALL syscs_util.syscs_set_database_property('derby.database.classpath', 'PUBLIC.DerbyStoredProcedure');
```

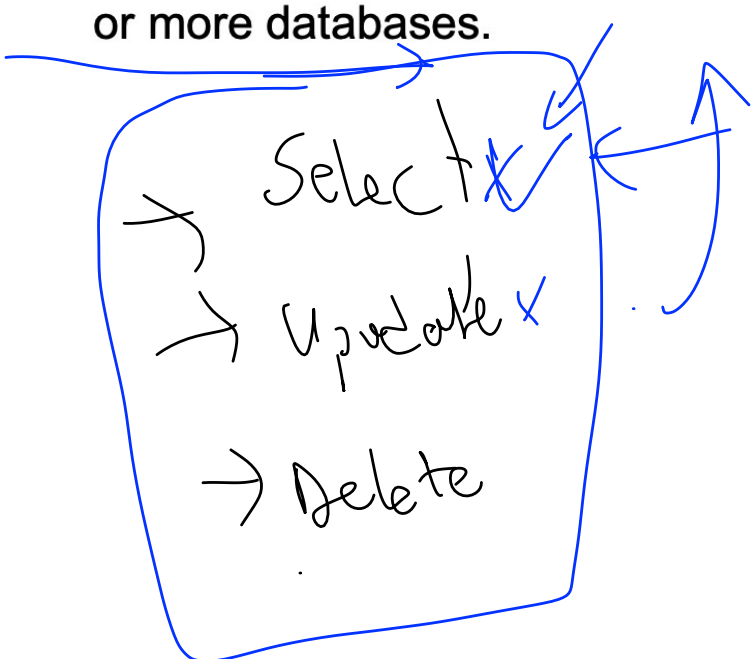
Java

```
public class DerbyStoredProcedure {
   public static void countAge (int age, int[] count) throws SQLException {
      String url = "jdbc:default:connection";
      Connection con = DriverManager.getConnection(url);
      String query = "SELECT COUNT(DISTINCT ID) " +
                     "AS count FROM Employee " +
                     "WHERE Birthdate <= ?";
      PreparedStatement ps = con.prepareStatement(query);
      Calendar now = Calendar.getInstance();
      now.add(Calendar.YEAR, (age*-1));
      Date past = new Date (now.getTimeInMillis());
      ps.setDate(1, past);
      ResultSet rs = ps.executeQuery();
      if (rs.next()) {
         count[0] = rs.getInt(1);
      } else {
         count[0] = 0;
      }
      con.close();
   }
}
```

SQL

What Is a Transaction?

- A transaction is a mechanism to handle groups of operations as though they were one.
- Either all operations in a transaction occur or none occur at all.
- The operations involved in a transaction might rely on one or more databases.



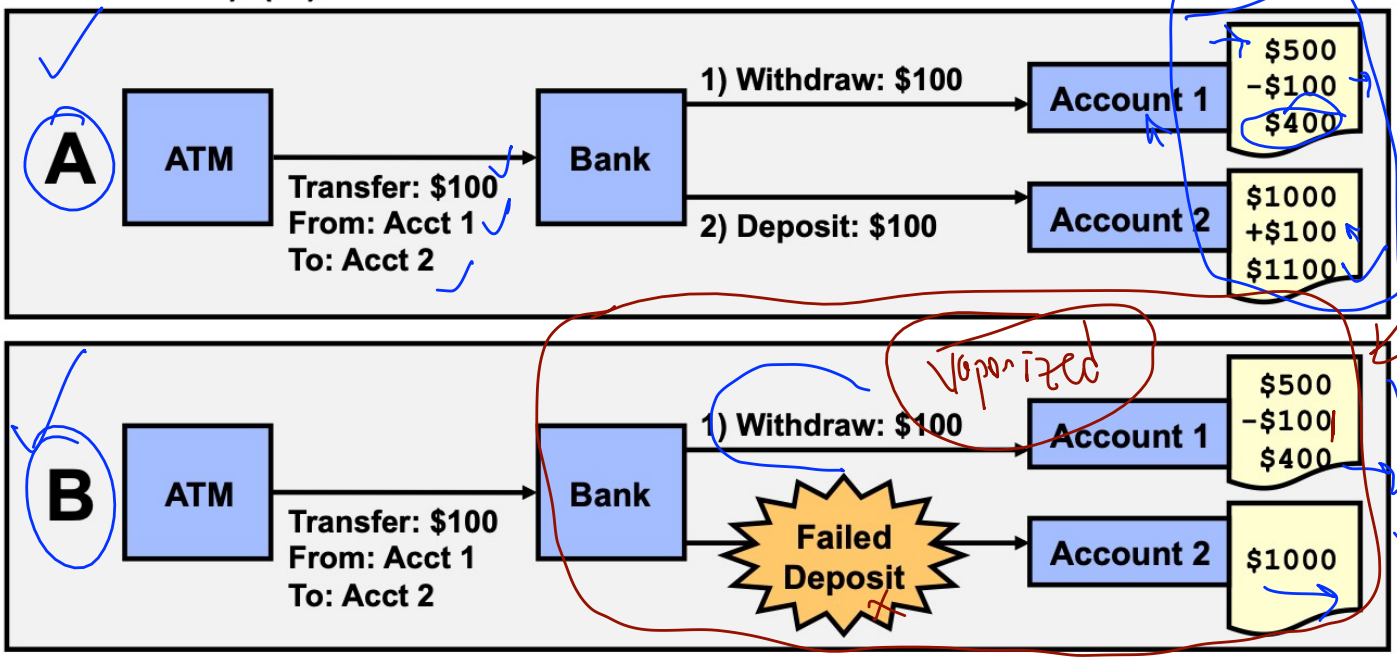
ACID Properties of a Transaction

A transaction is formally defined by the set of properties that is known by the acronym ACID.

- **Atomicity:** A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.
- **Consistency:** A transaction transforms a system from one consistent state to another consistent state.
- **Isolation:** Each transaction occurs independently of other transactions that occur at the same time.
- **Durability:** Completed transactions remain permanent, even during system failure.

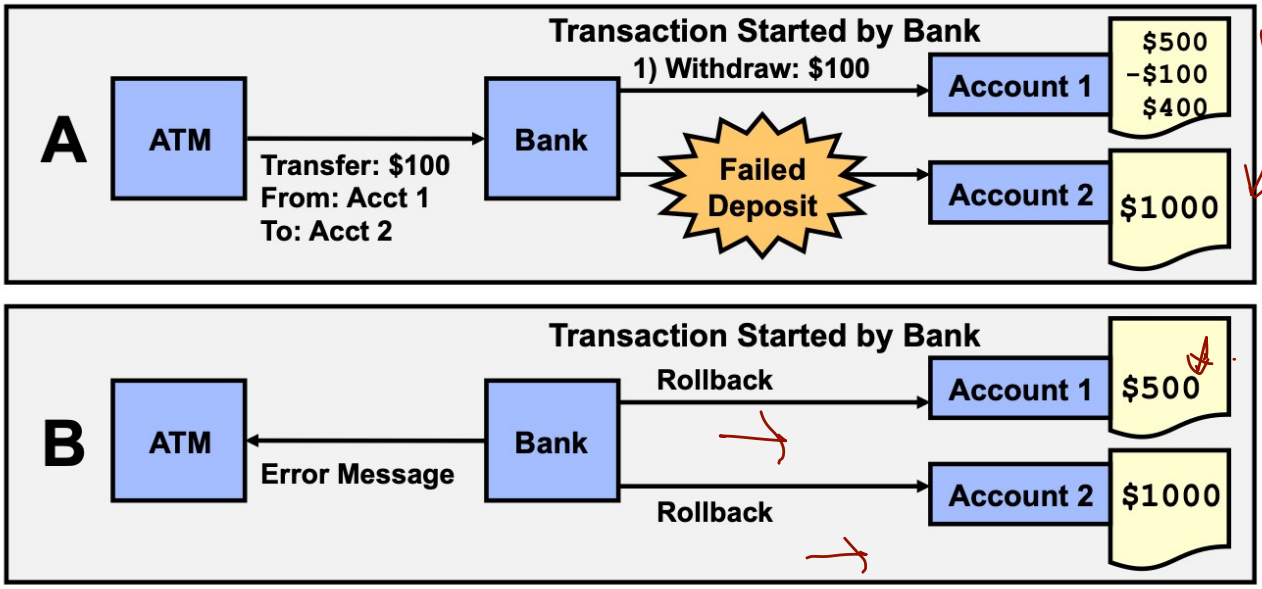
Transferring Without Transactions

- Successful transfer (A)
- Unsuccessful transfer (Accounts are left in an inconsistent state.) (B)



Unsuccessful Transfer with Transactions

- Changes within a transaction are buffered. (A)
- If a problem occurs, the transaction is rolled back to the previous consistent state. (B)



JDBC Transactions

By default, when a `Connection` is created, it is in auto-commit mode.

- Each individual SQL statement is treated as a transaction and automatically committed after it is executed.
- To group two or more statements together, you must disable auto-commit mode.

```
con.setAutoCommit (false);
```

- You must explicitly call the commit method to complete the transaction with the database.

```
con.commit();
```

- You can also programmatically roll back transactions in the event of a failure.

```
con.rollback();
```

RowSet 1.1: RowSetProvider and RowSetFactory

The JDK 7 API specification introduces the new RowSet 1.1 API. One of the new features of this API is RowSetProvider.

- javax.sql.rowset.RowSetProvider is used to create a RowSetFactory object:

```
myRowSetFactory = RowSetProvider.newFactory();
```

- The default RowSetFactory implementation is:

```
com.sun.rowset.RowSetFactoryImpl
```

- RowSetFactory is used to create one of the RowSet 1.1 RowSet object types.

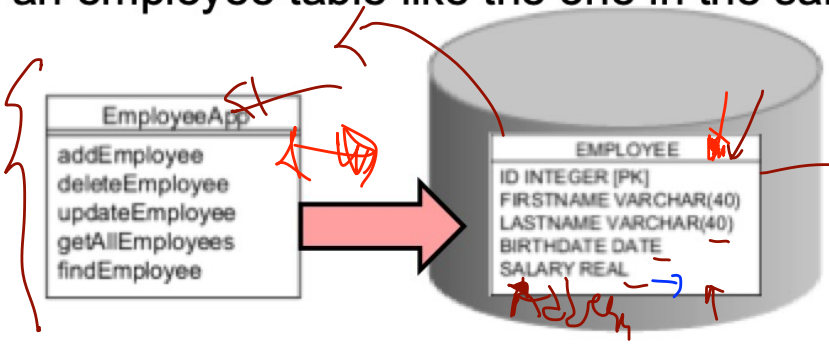
Using RowSet 1.1 RowSetFactory

RowSetFactory is used to create instances of RowSet implementations:

RowSet type	Provides
CachedRowSet	A container for rows of data that caches its rows in memory
FilteredRowSet	A RowSet object that provides methods for filtering support
JdbcRowSet	A wrapper around ResultSet to treat a result set as a JavaBeans component
JoinRowSet	A RowSet object that provides mechanisms for combining related data from different RowSet objects
WebRowSet	A RowSet object that supports the standard XML document format required when describing a RowSet object in XML

Data Access Objects

Consider an employee table like the one in the sample JDBC code.



- By combining the code that accesses the database with the "business" logic, the data access methods and the Employee table are tightly coupled.
- Any changes to the table (such as adding a field) will require a complete change to the application.
- Employee data is not encapsulated within the example application.

encapsulate that very

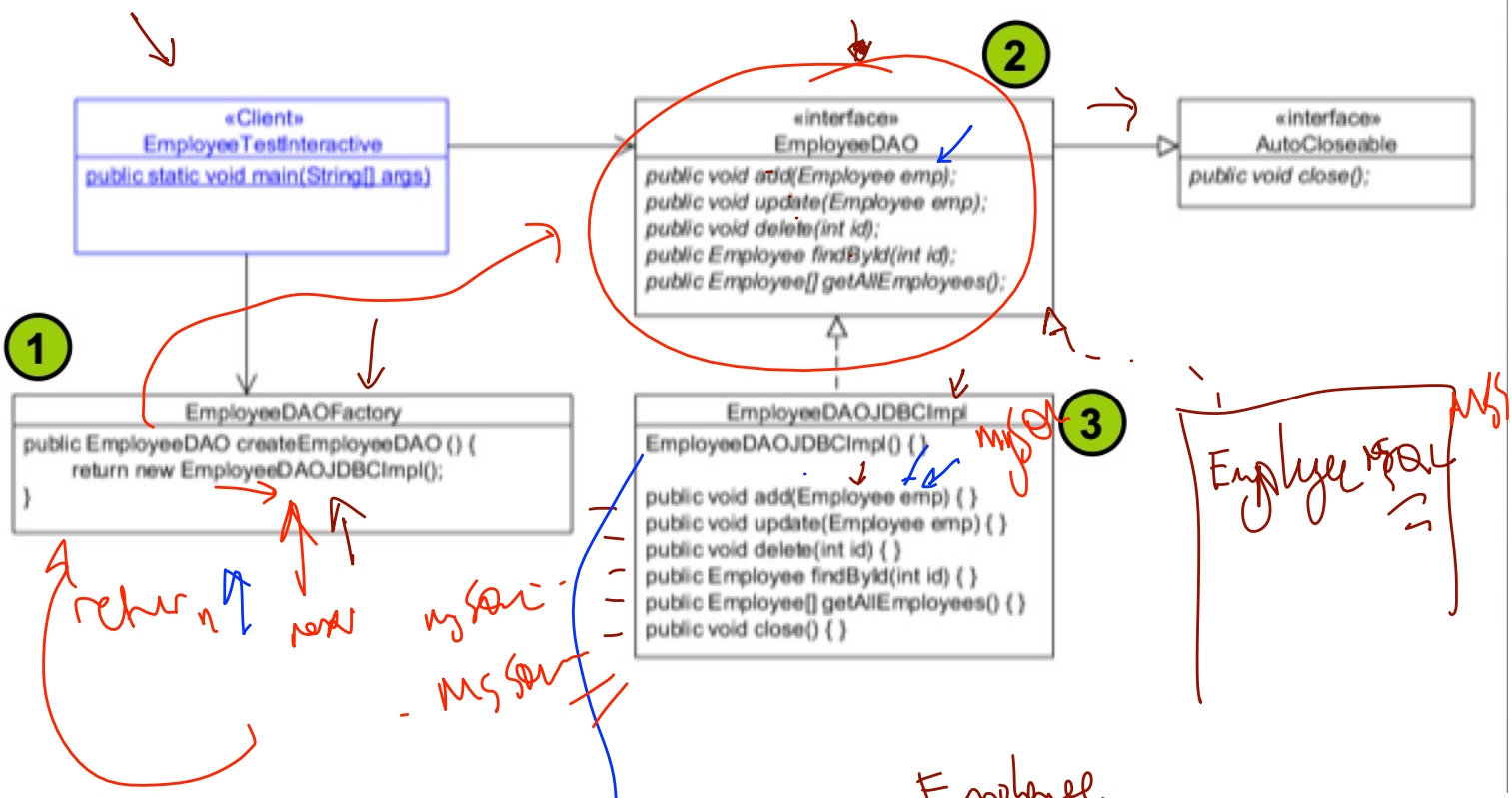
private

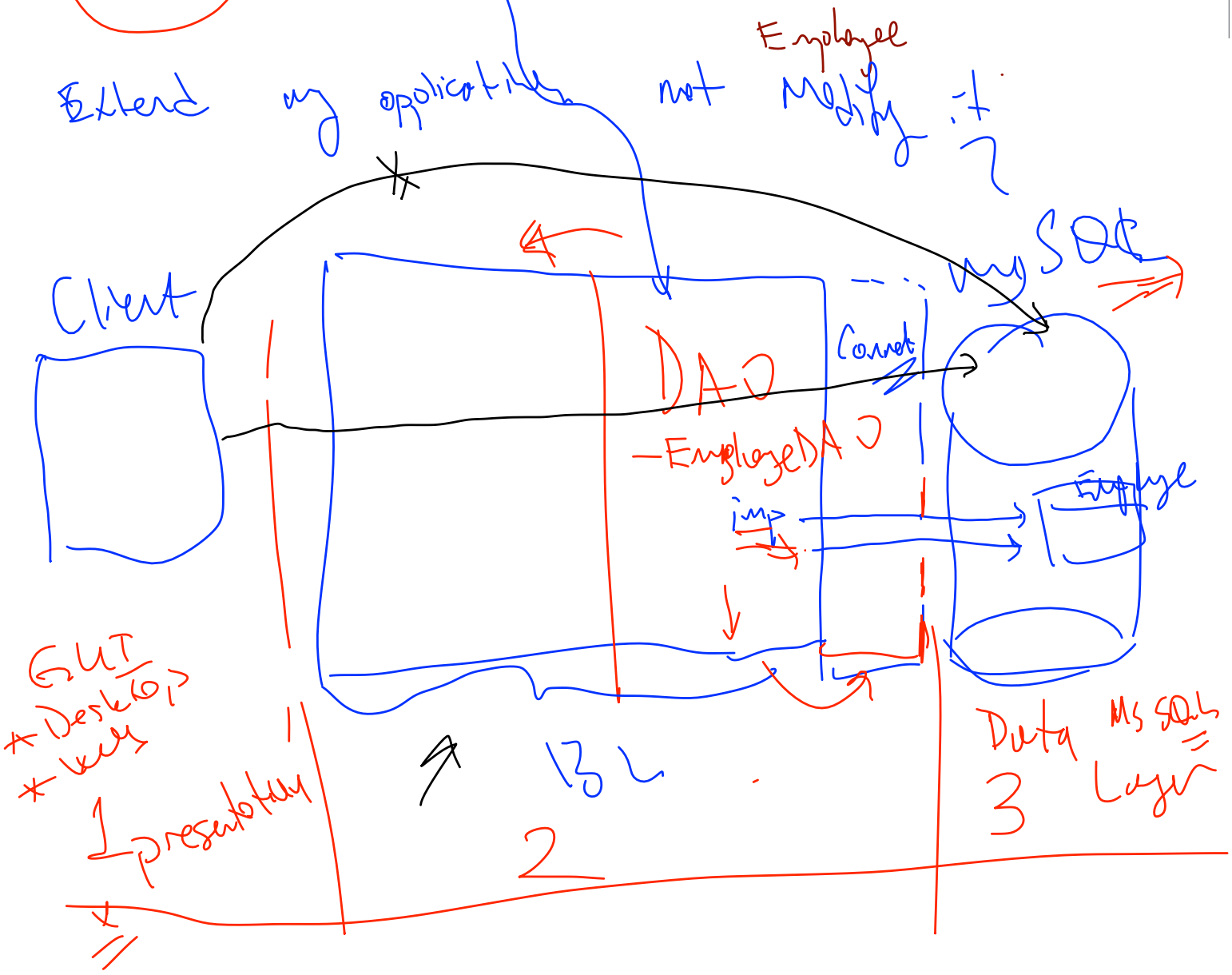
```
public class Employee {
    private int _ID;
    public void setID(int ID){
        this._ID = ID;
    }
    public int getID() {
        return _ID;
    }
}
```

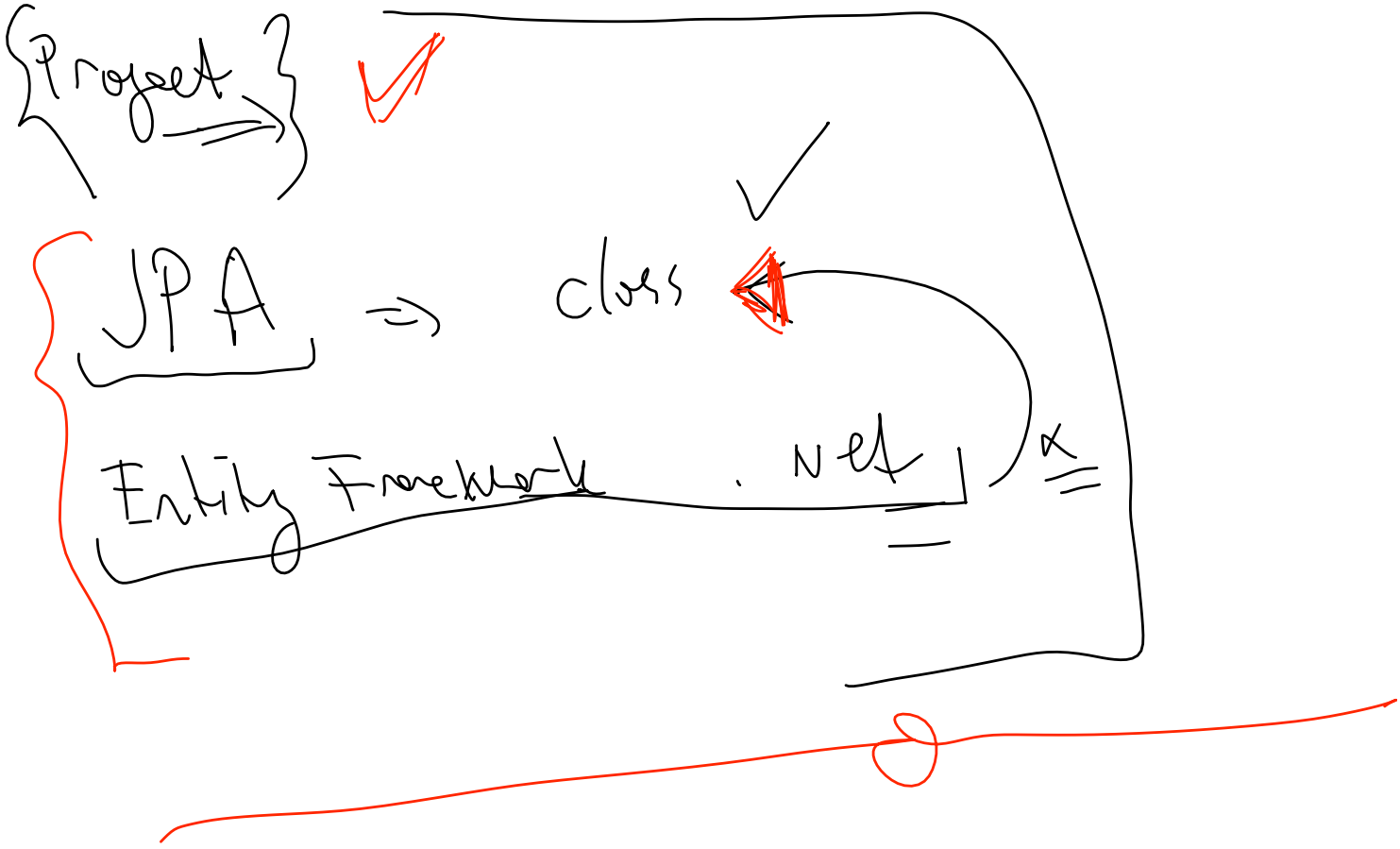
setter

getter

The Data Access Object Pattern



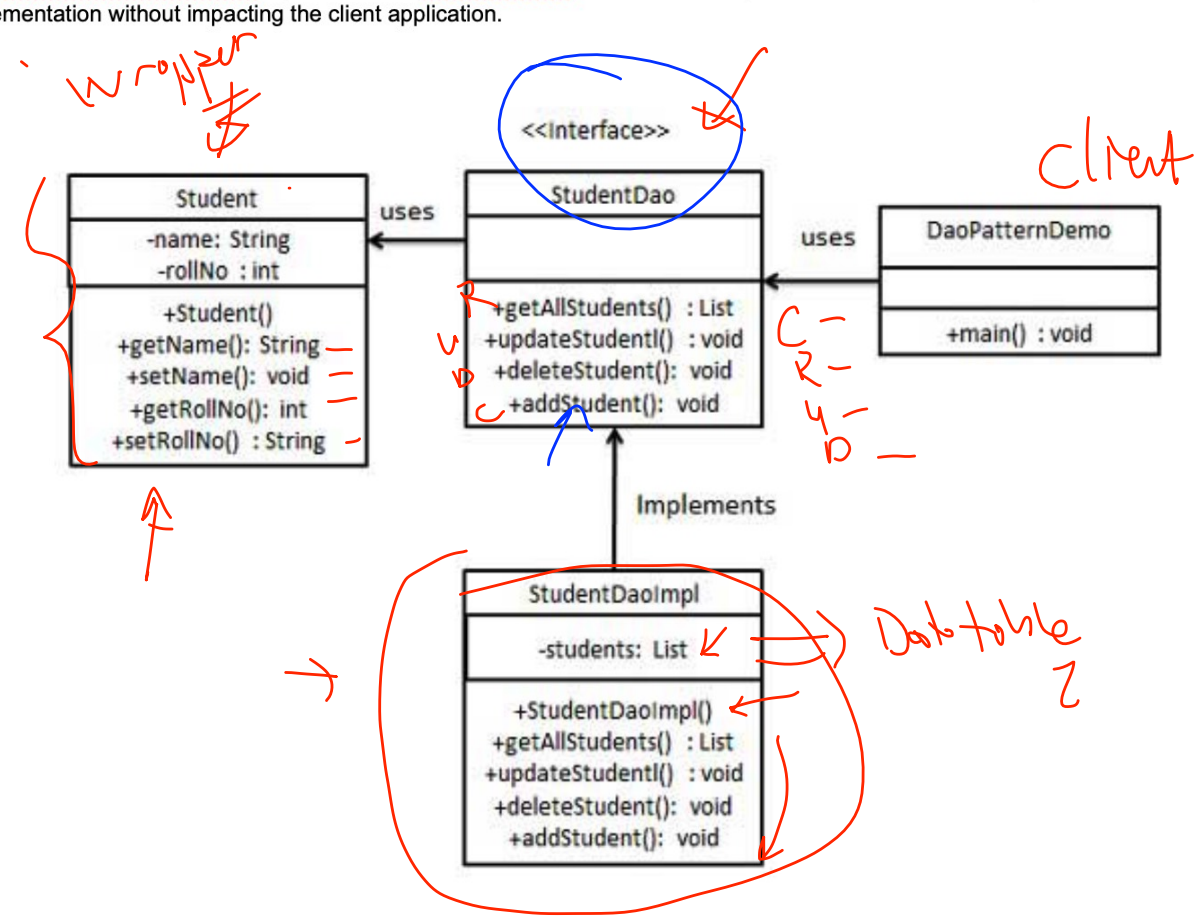




The Data Access Object and Factory Pattern

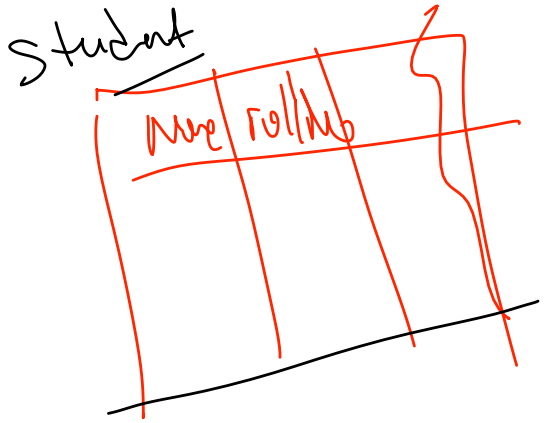
The purpose of a Data Access Object (DAO) is to separate database-related activities from the business model. In this design pattern, there are two techniques to insure future design flexibility.

1. A factory is used to generate instances (references) to an implementation of the EmployeeDAO interface. A factory makes it possible to insulate the developer using the DAO from the details about how a DAO implementation is instantiated. As you have seen, this same pattern was used to create an implementation where the data was stored in memory.
2. An EmployeeDAO interface is designed to model the behavior that you want to allow on the Employee data. Note that this technique of separating behavior from data demonstrates a *separation of concerns*. The EmployeeDAO interface encourages additional separation between the implementation of the methods required to support the DAO and references to EmployeeDAO objects.
3. The EmployeeDAOJDBCImpl implements the EmployeeDAO interface. The implementation class can be replaced with a different implementation without impacting the client application.



Student.java

```
public class Student {
   private String name;
   private int rollNo;

   Student(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getRollNo() {
      return rollNo;
   }

   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```



Step 2

Create Data Access Object Interface.

StudentDao.java

```
import java.util.List;

public interface StudentDao {
   public List<Student> getAllStudents();
   public Student getStudent(int rollNo);
   public void updateStudent(Student student);
   public void deleteStudent(Student student);
}
```

Step 3

Create concrete class implementing above interface.

StudentDaoImpl.java

```
import java.util.ArrayList;
import java.util.List;

public class StudentDaoImpl implements StudentDao {

   //list is working as a database
   List<Student> students;

   public StudentDaoImpl(){
      students = new ArrayList<Student>();
      Student student1 = new Student("Robert",0);
      Student student2 = new Student("John",1);
      students.add(student1);
      students.add(student2);
   }
   @Override
   public void deleteStudent(Student student) {
      students.remove(student.getRollNo());
      System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database
   }

   //retrive list of students from the database
   @Override
   public List<Student> getAllStudents() {
      return students;
   }

   @Override
   public Student getStudent(int rollNo) {
      return students.get(rollNo);
   }

   @Override
   public void updateStudent(Student student) {
      students.get(student.getRollNo()).setName(student.getName());
      System.out.println("Student: Roll No " + student.getRollNo() + ", updated in the databa
   }
}
```

Use the *StudentDao* to demonstrate Data Access Object pattern usage.

DaoPatternDemo.java

```
public class DaoPatternDemo {
   public static void main(String[] args) {
      StudentDao studentDao = new StudentDaoImpl();

      //print all students
      for (Student student : studentDao.getAllStudents()) {
         System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + stude
      }

      //update student
      Student student =studentDao.getAllStudents().get(0);
      student.setName("Michael");
      studentDao.updateStudent(student);

      //get the student
      studentDao.getStudent(0);
      System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.
   }
}
```