

# JDBC



# JDBC Overview

- Stands for Java Database Connectivity
- A standard interface for accessing data sources
  - normally databases (works with Excel)
  - based on X/Open SQL Call Level Interface (CLI)
  - allows databases packages to be replaced without affecting application code
    - assuming no database specific features are needed and the databases are SQL-92 entry-level compliant
- A base for building higher level tools and APIs such as JavaBlend (see p. 4)
- Defined by
  - interfaces and classes in the java.sql package
- Implemented by
  - data source specific JDBC drivers
    - accept JDBC calls and perform operations using the API of the specific database
  - the JDBC-ODBC bridge
    - a special JDBC driver
    - included with JDK



# JDBC Overview (Cont'd)

- Pure Java JDBC drivers can be downloaded along with applets that use them
- Drivers that use native methods cannot be used by applets
  - these drivers must be installed on each client and used from Java applications
- Many vendors have endorsed JDBC
  - Borland, Gupta, **IBM**, **Informix**, Intersolv, **Object Design**, **Oracle**, RogueWave, SAS, SCO, **Sybase**, Symantec, Visigenic, WebLogic, and more
- To claim full JDBC compliance, drivers must
  - be SQL-92 entry-level compliant
  - contain all classes and methods in the JDBC API



# JDBC Overview (Cont'd)

- Databases are specified with URL syntax

protocol:subprotocol:data-source

↑                    ↑                    ↙

always "jdbc"      used to select appropriate driver      driver specific string for  
("odbc" for JDBC-ODBC bridge)      locating a data source

- Mapping database types to Java types
  - maps most SQL data types to Java data types
  - types that have no direct mapping are represented with
    - special Java classes
      - ex. Date, Time
    - binary large objects (BLOBs)
      - for images, sounds, and documents
- Javasoft's JavaBlend will automate mapping records in relational database tables to Java objects
  - will allow transparent database access
    - like serialization allows transparent reading and writing of objects
  - JDBC code to perform database queries and updates will be generated



# Why Doesn't Java Use ODBC Instead of JDBC?

- What is ODBC?
  - written by Microsoft
  - provides access to most popular relational databases
- Problems with using ODBC directly
  - Relies on C code which violates Java security
    - applets can't use it
  - Translating ODBC into pure Java would be difficult due to its heavy use of pointers
  - ODBC is harder to learn than JDBC
    - complex, rarely used operations coexist with common ones
    - must learn a lot in order to use basic functionality
    - with JDBC, uncommon operations are supported by separate interfaces from those that provide basic functionality
- Design of JDBC
  - based on ODBC and the X/OPEN SQL Call Level Interface
  - makes it easy for ODBC developers to learn
- JDBC-ODBC Bridge
  - allows access to ODBC databases from Java applications



# Ways to Utilize JDBC

- Two Tier
  - Java applet communicates directly with a database on the web server from which the applet was downloaded
    - requires a 100% Java database driver so it can be downloaded
  - Java application communicates directly with a database on any server
    - JDBC driver doesn't have to be pure Java but must be on the client
- Three Tier
  - middle tier can provide
    - a higher-level API, not just SQL
    - control over database access
    - performance advantages
      - ex. load balancing and caching frequently accessed data
  - Java applet communicates with a Java application on the web server from which the applet was downloaded (via sockets, RMI, or CORBA) which communicates directly with a database on any server
  - Java application communicates with a Java application on any server (via sockets, RMI, or CORBA) which communicates directly with a database on any server



# How JDBC Deals With Non-Standard Database

- Databases that are not SQL-92 entry-level compliant are supported by JDBC in three ways
- Database metadata
  - used to determine the capabilities of a database at run-time
- Query strings
  - any query string can be passed to a database driver
- ODBC-style escape clauses
  - supports common diversions from the SQL-92 standard



# Types of JDBC Drivers That Applets Cannot Use

- Type 1 - JDBC-ODBC Bridge
  - client translates JDBC calls to database independent ODBC
  - server translates ODBC calls to database specific calls
  - applets **can't** directly use this since it uses native methods
    - could access a middle tier on the web server that uses this
  - requires bridge software on clients
- Type 2 - Native-protocol, not pure Java
  - client translates JDBC calls to database specific calls
  - server has no translation to perform
  - applets **can't** directly use this since it uses native methods
    - could access a middle tier on the web server that uses this
  - requires JDBC driver software on clients



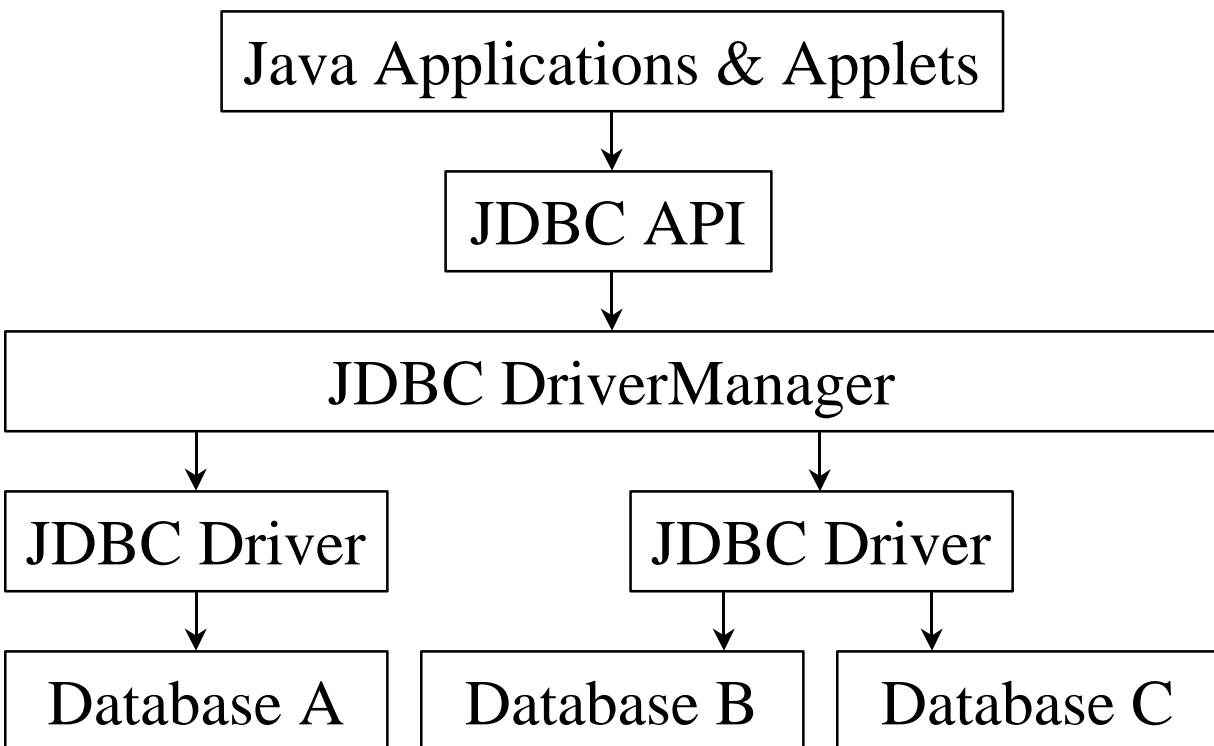


# Types of JDBC Drivers That Applets Can Use

- Type 3 - JDBC-Network protocol, pure Java
  - client translates JDBC calls to database independent network protocol (who defined this?)
  - server translates network protocol to one or more database specific protocols
  - applets **can** use this
- Type 4 - Native-protocol, pure Java
  - client translates JDBC calls to database specific network protocol
  - applets **can** use this
  - most efficient



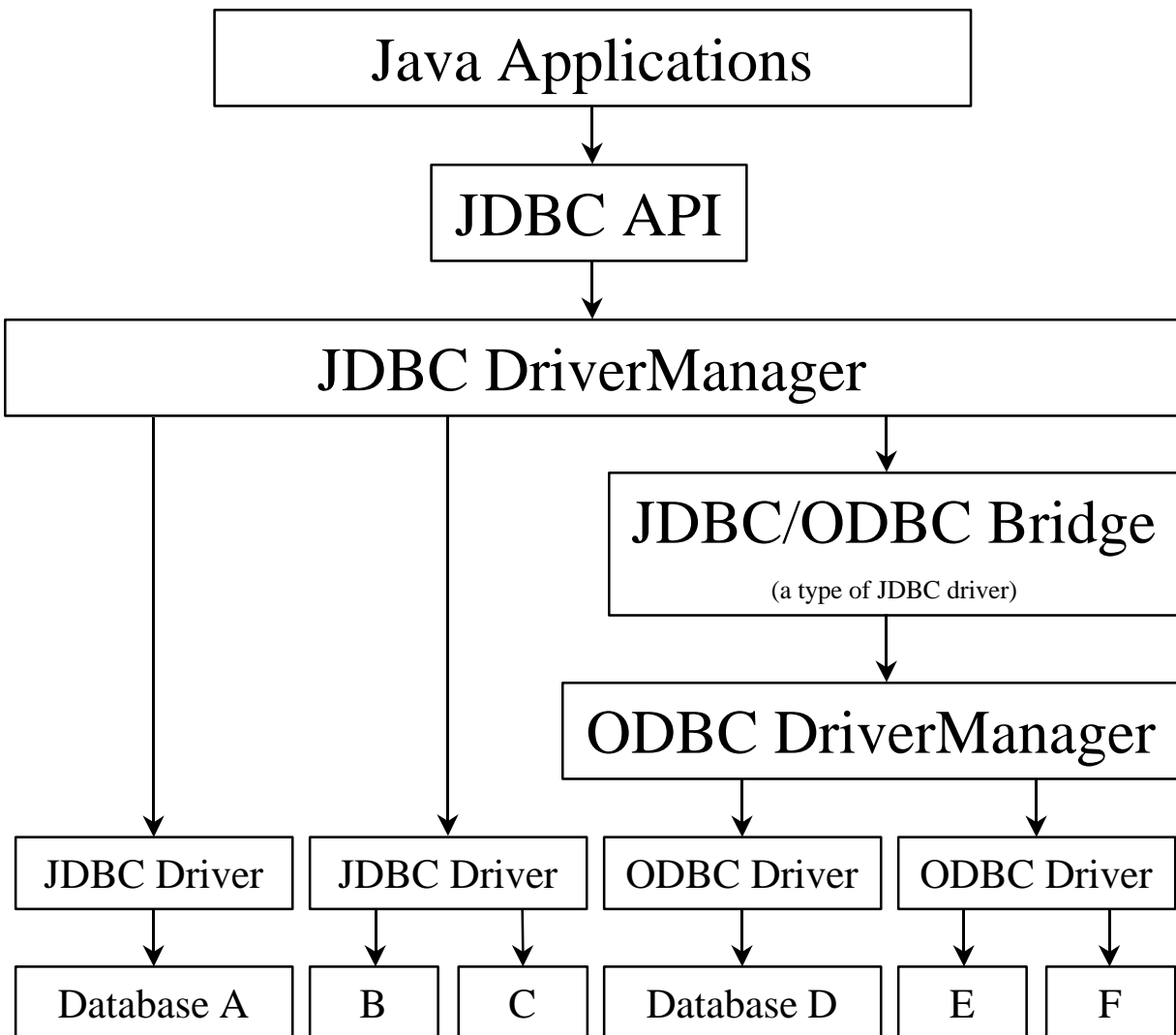
# JDBC Architecture



- When Java code requests a data source connection the DriverManager chooses the appropriate registered driver
  - determined from subprotocol in URL specification
    - ex. `jdbc:odbc:MySource`
  - see “JDBC Setup” on page 12



# Architecture With JDBC/ODBC Bridge



- Bridge developed by Javasoft and Intersolv
- Applets cannot use the bridge because it uses native methods



# JDBC Setup

- JDK1.1 includes
  - java.sql package
  - JDBC-ODBC Bridge
- Only setup required is to specify data sources
  - Under Win 95/NT the Driver Manager is configured in Settings...Control Panels...[32bit ]ODBC
    - UNIX Oracle has a similar registry in /etc/tnsnames.ora
  - To add a new data source under Win 95/NT
    - click the “Add ...” button
    - select an ODBC driver such as “Microsoft Access Driver” or “Microsoft Excel Driver”
    - click the “OK” button
    - enter a name and description for the data source
    - click the “Select...” button to select an existing database →  
OR  
click the “Create...” button to create a new database
      - use "Network..." button to select the remote drive containing the database
      - select a directory and enter a name for the new database
    - click the “OK” button
    - click the “Advanced...” button to specify a username and password for accessing the database
    - click the “Options>>” button to request
      - exclusive access to the database (one user at a time)
      - read-only access to the database
    - click the "OK" button

INSTRUCTOR Java Adv JDBC AddressBookAccess AddressBook.mdb
---



# SQL Overview

- Stands for Structured Query Language
- SQL keywords are case-insensitive
- Whether table and column names are case-sensitive is database dependent



# SQL Data Definition Language (DDL) Commands

- To create a table

```
CREATE TABLE table-name (  
    column-name type {modifiers},  
    . . . ,  
    column-name type {modifiers})
```

valid column types and modifiers  
may be database dependent

- To delete a table

```
DROP TABLE table-name
```

- To add a column to a table

```
ALTER TABLE table-name  
ADD COLUMN column-name type {modifiers}
```

- To delete a column from a table

```
ALTER TABLE table-name  
DROP column-name
```



# SQL Data Manipulation Language (DML) Commands

- To add a row into a table

```
INSERT INTO table-name  
(column-name, ..., column-name)  
VALUES (value, ..., value)
```

not needed if all values  
are supplied in order

- To modify rows in a table

```
UPDATE table-name  
SET column-name = value, ..., column-name = value  
WHERE condition
```

- To delete rows in a table

```
DELETE FROM table-name  
WHERE condition
```

- *condition* specifies affected rows
- use LIKE to compare strings
- use relational operators to compare #'s

- To select rows in a table

```
SELECT column-name, ..., column-name  
FROM table-name(s)  
WHERE condition
```

the where clause can be  
omitted to operate on  
every row

- selected rows are returned in a ResultSet
- use \* in place of column names to select all columns
- can perform "joins" using multiple table names separated by commas



# JDBC Example

```
import java.io.*;
import java.sql.*;
import sun.jdbc.odbc.*;

public class AddressBookDB {
    public static void main(String[] args) {
        try {
            // Load the JDBC-ODBC driver.
            // This creates a single instance of the driver.
            // The drivers static initializer passes this instance
            // to DriverManager.registerDriver().
            // There are other ways to accomplish this
            // but this method is the most common.
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // Direct driver log information to a file for debugging.
            try {
                FileOutputStream fos = new FileOutputStream("db.log");
                // Note: The PrintStream class has been deprecated.
                //       PrintWriter replaces it. However, we can't
                //       use that because setLogStream still requires
                //       a PrintStream.
                PrintStream ps = new PrintStream(fos);
                DriverManager.setLogStream(ps);
            } catch (IOException ioe) { // if the file cannot be opened
                System.err.println(ioe);
            }
        }
    }
}
```





# JDBC Example (Cont'd)

```
// Connect to a database.
// DriverManager.getConnection() checks each loaded driver
// in order and selects the first one that is able to
// process the specified database.
// AddressBookDB is associated with a specific database
// in Settings...Control Panel...32bit ODBC.
// No username or password is required for this database.
String url = "jdbc:odbc:AddressBookDB";
Connection con =
    DriverManager.getConnection(url, "", "");

// Create a Statement object for executing SQL statements
// against the database.
Statement stmt = con.createStatement();

String sql;
ResultSet rs;
int rowCount;

// Delete an existing table from the database.
sql = "DROP TABLE Person";
stmt.executeUpdate(sql);

// Add a new table to the database.
// Access95 doesn't support the types CHAR and VARCHAR.
sql = "CREATE TABLE Person (" +
    "lastName TEXT CONSTRAINT C1 PRIMARY KEY, " +
    "firstName TEXT, " +
    "street TEXT, " +
    "city TEXT, " +
    "state TEXT, " +
    "zipCode TEXT, " +
    "email TEXT)";
stmt.executeUpdate(sql);
```

name of data source, not a database

username

password

name of the constraint (can be anything)



# JDBC Example (Cont'd)


```
// Add a record to a table in the database specifying
// fields in the order they are defined in the database.
sql = "INSERT INTO Person " +
      "VALUES ('Volkmann', 'Mark', '10 Galaxy Dr.', " +
            "'St. Peters', 'MO', '63376', " +
            "'mvolkman@mail.win.org')";
stmt.executeUpdate(sql);

// Add another record to a table in the database
// specifying a subset of the fields in a different order.
sql = "INSERT INTO Person " +
      "(email, firstName, lastName, street, state) " +
      "VALUES ('gosling@eng.sun.com', 'Jimmy', " +
            "'Gosling', '123 Sun Ave.', 'CA')";
stmt.executeUpdate(sql);

// Change fields within a database record.
sql = "UPDATE Person " +
      "SET firstName = 'James', city = 'Palo Alto' " +
      "WHERE lastName = 'Gosling';
rowCount = stmt.executeUpdate(sql);

// Select records from the database.
sql = "SELECT state, email FROM Person " +
      "WHERE lastName='Volkmann';
rs = stmt.executeQuery(sql); // returns a ResultSet object
```

# of records  
updated



# JDBC Example (Cont'd)

Besides calling getObject on a ResultSet you can also call getByte, getBytes, getDate, getDouble, getFloat, getInt, getLong, getShort, getString, getTime, and getTimeStamp. All of these accept one argument that is either a column index or a column name.

```
// Process the result set.
ResultSetMetaData rsmd = rs.getMetaData();
int columns = rsmd.getColumnCount();
while (rs.next()) {
    for (int pos = 1; pos <= columns; ++pos) {
        Object obj = rs.getObject(pos);
        System.out.println
            (rsmd.getColumnLabel(pos) + " = " + obj +
             " (type = " + obj.getClass().getName() + ")");
    }
}
```

data describing  
the results

see note 1 at bottom

see note 2  
at bottom

```
// Delete records from a table in the database.
sql = "DELETE FROM Person WHERE lastName='Volkmann'";
rowCount = stmt.executeUpdate(sql);
```

# of records  
deleted

```
// Remove a column from a table.
sql = "ALTER TABLE Person DROP street";
stmt.executeUpdate(sql);
```

```
// Add a column to a table.
sql = "ALTER TABLE Person ADD COLUMN birthDate DATE";
stmt.executeUpdate(sql);
```

- 1) moves cursor to the next row;  
previous(), which would require support for scrollable cursors in JDBC drivers, is not implemented
- 2) pos is the column number in current row of the result set;  
can also use column name but that is less efficient



# JDBC Example (Cont'd)

catches for the try block that starts at the top of main()

```
// ClassNotFoundException is thrown if the ODBC-JDBC Bridge
// is not found.
} catch (ClassNotFoundException e) {
    System.err.println(e);
} catch (SQLException e) { // catch-all for database exceptions
    System.out.println("Message = " + e.getMessage());
    System.out.println("SQLState = " + e.getSQLState());
    System.out.println("Vendor Code = " + e.getErrorCode());
} finally {
    // Close the database connection.
    con.close();
}
} // end of main()
} // end of class
```

XOpen SQLState defined in the Xopen SQL specification

- Can also access a chain of exceptions that led to this one with `SQLException.getNextException()`
- Some JDBC drivers may generate non-fatal errors represented by `java.sql.SQLWarning` and `java.sql.DataTruncation`. Get these with calls to `Connection.getWarnings()` until it returns null.

- Statements, ResultSets, and Connections are all automatically closed when they are garbage collected.
- Closing a Statement frees associated ResultSets.
- Closing a ResultSet frees associated result objects.
- Some databases may limit active resources such as Statements, ResultSets, and Connections so it may be necessary to explicitly close them before reusing them.



# Statement Class

## Execute Methods

- `stmt.executeQuery(String sql)`
  - returns a `ResultSet` object
  - used for these SQL statements
    - `SELECT`
- `stmt.executeUpdate(String sql)`
  - returns the number of rows affected
  - used for these SQL statements
    - `INSERT` - for inserting one row in a table
    - `UPDATE` - for modifying one or more rows in a table
    - `DELETE` - for deleting one or more rows in a table
    - `CREATE TABLE` - returns zero
    - `DROP TABLE` - returns zero
    - `ALTER TABLE` - returns zero

these were used in the previous example code



# Statement Class

## Execute Methods (Cont'd)

- `stmt.execute(String sql)`
    - for SQL statements that may be a query or an update
    - used for stored procedures that result in
      - more than one `ResultSet`
      - more than one row count
      - both `ResultSet`s and row counts
    - returns a boolean indicating whether results were obtained
- methods  
in the  
Statement  
class
- use `getMoreResults()` and `getResultSet()` to get all `ResultSet`s
  - use `getUpdateCount()` to get each row count
    - returns -1 when there are no more



# PreparedStatement

- Provide efficient repeated execution of SQL statements that differ only by their parameters
- When a PreparedStatement object is created, the statement is sent to the DBMS and compiled

- **Creating a PreparedStatement**

```
PreparedStatement updatePhone =  
    con.prepareStatement  
        ("UPDATE Person SET phone = ? WHERE name = ?");
```

- question marks indicate where parameters will be inserted

- **Executing a PreparedStatement**

```
updatePhone.setString(1, "(123)456-7890");  
updatePhone.setString(2, "Doe, John");  
updatePhone.executeUpdate();
```

- there are `set?( )` methods for all Java types
- parameters retain their values until changed or `clearParameters( )` is called
- also supports `execute( )` and `executeQuery( )`



# CallableStatements

- For calling with stored procedures
  - groups of SQL statements stored in compiled form on database servers
  - can have IN, OUT, and INOUT parameters
  - syntax for creating differs by database

- Example

```
CREATE PROCEDURE PersonsByState AS
    SELECT name, city, phone
    FROM Person WHERE state = ? ORDER BY name
```

- can contain more than one SQL statement
- To add a stored procedure to the database

```
Statement stmt = con.createStatement();
stmt.executeUpdate("CREATE PROCEDURE ...");
```





# CallableStatements (Cont'd)

- To call a stored procedure

```
CallableStatement cStmt =  
    con.prepareStatement("{call PersonsByState(?)}");  
cStmt.setString(1, "MO");  
ResultSet resultSet = cStmt.executeQuery();
```

- CallableStatement extends PreparedStatement so ? parameters can be used in the same way
- also supports execute() and executeUpdate()
  - execute() is useful when there is more than one statement in the stored procedure since it allows more than one row count and more than one ResultSet to be obtained

- Three forms of calling stored procedures

these create a  
CallableStatement object  
that still must be executed

- no parameters or return value

```
con.prepareStatement("{call procedure-name}")
```

- parameters but no return value

```
con.prepareStatement("{call procedure-name(?, ?, ...)}")
```

- parameters and a return value

```
con.prepareStatement  
    ("{? = call procedure-name[(?, ?, ...)]}")
```



# CallableStatements (Cont'd)

- Usually only IN parameters are used
- The types of OUT and INOUT parameters must be registered before a CallableStatement can be executed

```
cStmt.registerOutParameter(1, java.sql.Types.VARCHAR);
```
- To set the values of IN or INOUT parameters
  - use the `set?()` methods inherited from `PreparedStatement`
- To get the values of OUT or INOUT parameters
  - use the `get?()` methods in `CallableStatement`
- For more detail see the Addison Wesley book "JDBC Database Access with Java"



# Transactions

- A database transaction is group of database operations that must
  - all complete successfully then commit
  - OR
  - all rollback (undo changes they caused)



# Transactions (Cont'd)

- The `java.sql.Connection` interface provides methods for implementing transactions
  - by default each JDBC statement calls `con.commit()` before completing
    - where `con` is a `Connection` object
  - disable this by calling `con.setAutoCommit(false)`
  - add calls to `con.commit()` where appropriate
    - such as at the end of a try block
  - add calls to `con.rollback()` where appropriate
    - such as in corresponding catch blocks
  - database records are locked when they are read, not just when they are modified
  - `commit()` makes database modifications permanent and releases all database locks associated with the transaction
  - `rollback()` discards all changes made in transaction and releases all locks associated with the transaction
  - after `commit()` or `rollback()` complete, a new transaction is automatically started



# Transaction Isolation Levels

- Resolve attempts to access locked records
- Select one of five levels with `con.setTransactionIsolation(int level);`
  - see constants on page 31
- Specific databases may not support all five
- Determine which are supported with

```
DatabaseMetaData metadata = con.getMetaData();
int level = metadata.getDefaultTransactionIsolation();
boolean supported =
    metadata.supportsTransactionIsolationLevel(int level);
```



# Transaction Isolation Levels (Cont'd)

- Situations under which a locked record could be read
  - dirty read
    - reading modified, locked records
    - don't know if the modification will be committed
  - non-repeatable read
    - reading unmodified, locked records
    - the transaction that holds the lock may modify and commit before it releases the lock
  - phantom read
    - reading newly inserted records, locked records
    - the transaction that holds the lock may rollback, not saving the new record



# Transaction Isolation Levels (Cont'd)

D N P (Dirty, Non-repeatable, Phantom)

- The levels (ordered from least to most restrictive)

na – TRANSACTION\_NONE

- transactions are not supported
- every operation is immediately committed

1 0 1 – TRANSACTION\_READ\_UNCOMMITTED

- dirty and phantom reads allowed
- non-repeatable reads disallowed

0 1 1 – TRANSACTION\_READ\_COMMITTED

- non-repeatable and phantom reads allowed
- dirty reads disallowed

0 0 1 – TRANSACTION\_REPEATABLE\_READ

- dirty and non-repeatable reads disallowed
- phantom reads allowed

0 0 0 – TRANSACTION\_SERIALIZABLE

- locked records cannot be read
- dirty, non-repeatable, and phantom reads disallowed

Other combinations of D, N, & P don't make sense.

