JDBC Access

The Sql package

http://java.sun.com/docs/books/tutorial/jdbc/

JDBC

- From any Java application you can access a DB through the JDBC. (package java.sql)

You must use only ANSI SQL-2 standard.

No special references to JDBC has to be done in the code.

SW Layers to get access to a DB

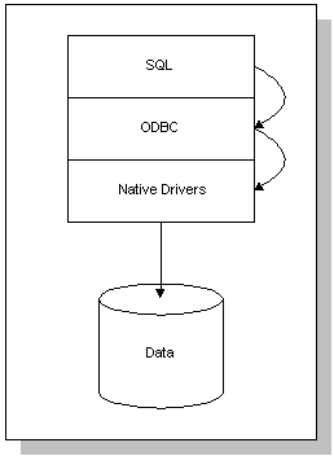
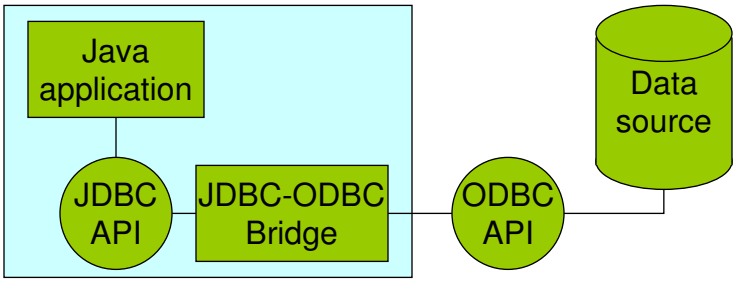


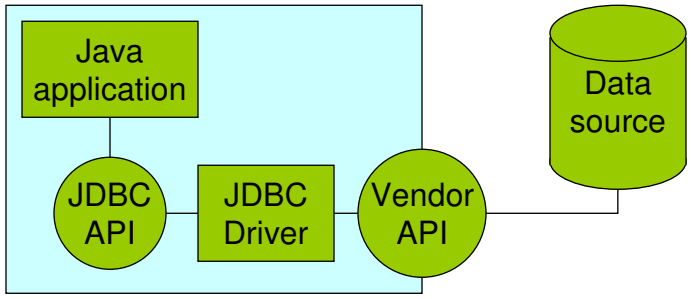
Figure 7.3 Database Access Layers

Type 1 – JDBC-ODBC Bridge

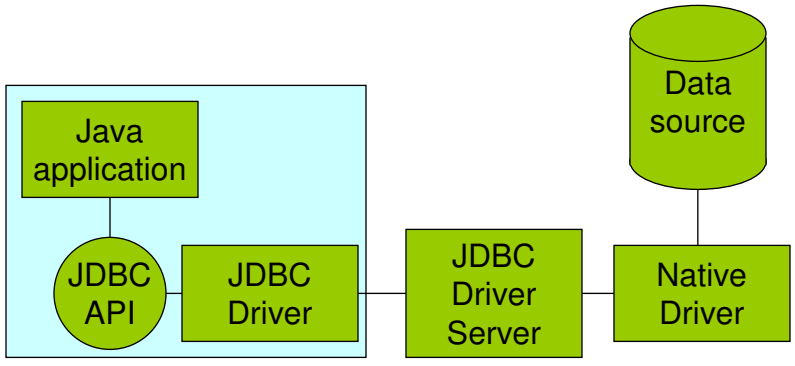


The standard JDK includes
sun.jdbc.odbc.JdbcOdbcDriver

Type 2 – Part Java, Part Native

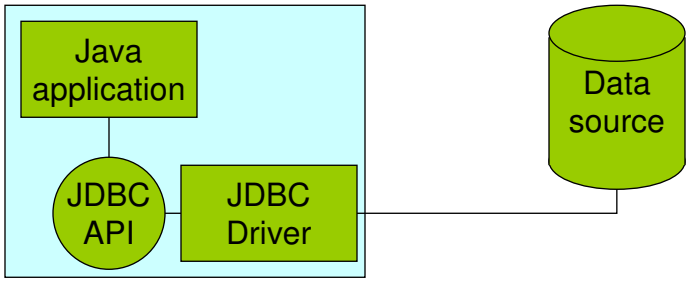


Type 3 – Intermediate DB Access Server



See http://industry.java.com/products/jdbc/drivers

Type 4 – Pure Java



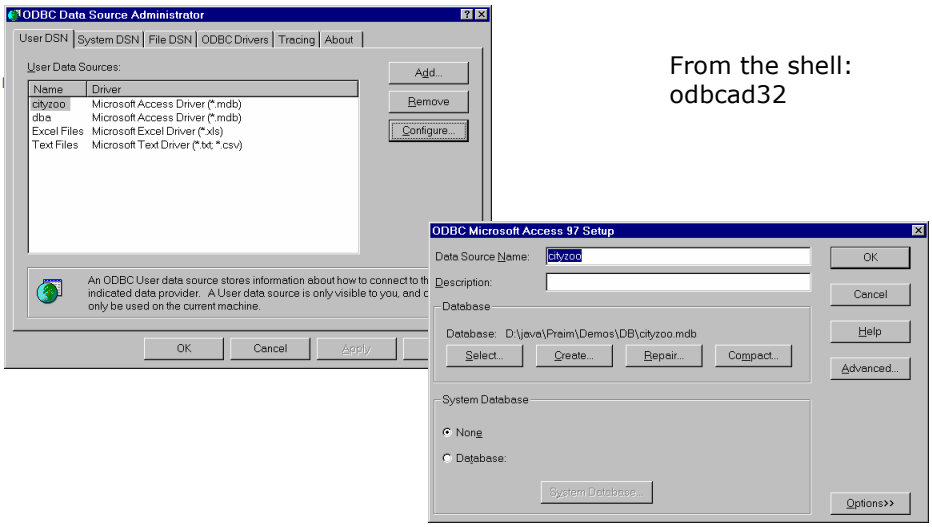
JDBC - Installation

A) Install a driver on your machine.
Your driver should include instructions for installing it. For JDBC drivers written for specific DBMSs, installation consists of just copying the driver onto your machine; there is no special configuration needed.

B) The JDBC-ODBC Bridge driver is not quite as easy to set up. If you download either the Windows versions of JDK, you will automatically get the JDBC-ODBC Bridge driver, which does not itself require any special configuration. ODBC, however, does. If you do not already have ODBC on your machine, you will need to see your ODBC driver vendor for information on installation and configuration.

.

Setting the ODBC Control Panel



From the shell:
odbcad32

JDBC - Steps

- A) Load the driver.
  B) Open a connection.
  C) Create Statement.
  D) Retrieve Values.

Always catch exceptions!
JDBC lets you see the warnings and exceptions generated by your DBMS and by the Java compiler. To see exceptions, you can have a catch block print them out.

.

Reminder: Class.forName

- static Class **forName**(String className)

  Returns the Class object associated with the class or interface with the given string name.

  Typical use:

  ```
  Object o=Class.forName("java.lang.String").newInstance();
  ```

  is equivalent to:

  ```
  Object o=new String();
  ```

JDBC – Steps – 1 LOAD THE DRIVER

- Loading the driver or drivers you want to use is very simple and involves just one line of code.
  If you want to use the JDBC-ODBC Bridge driver, the following code will load it:

  ```
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  ```

  Your driver documentation will give you the class name to use. For instance, if the class name is jdbc.DriverXYZ , you would load the driver with the following line of code:

  ```
  Class.forName("jdbc.DriverXYZ");
  ```

  You do not need to create an instance of a driver and register it with the DriverManager because calling Class.forName will do that for you automatically. If you were to create your own instance, you would be creating an unnecessary duplicate, but it would do no harm.

  .

JDBC – Steps – 2 LOAD THE DRIVER

```
Connection con = DriverManager.getConnection(url, "myLogin",
"myPassword");
```

This step is also simple, with the hardest thing being what to supply for url . If you are using the JDBC-ODBC Bridge driver, the JDBC URL will start with jdbc:odbc: . The rest of the URL is generally your data source name or database system.

If you are using a JDBC driver developed by a third party, the documentation will tell you what subprotocol to use, that is, what to put after jdbc: in the JDBC URL. For example, if the driver developer has registered the name acme as the subprotocol, the first and second parts of the JDBC URL will be jdbc:acme: . The driver documentation will also give you guidelines for the rest of the JDBC URL. This last part of the JDBC URL supplies information for identifying the data source.

JDBC – Steps – 3 CREATE STATEMENT

A Statement object is what sends your SQL statement to the DBMS.

For a SELECT statement, the method to use is executeQuery .
For statements that create or modify tables, the method to use is executeUpdate.

```
Statement stmt = con.createStatement();
stmt.executeUpdate("CREATE TABLE COFFEES " +
     "(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
     "SALES INTEGER, TOTAL INTEGER)");
```

Typically you would put the SQL statement in a String (called let's say createTableCoffees), and then use
`stmt.executeUpdate(createTableCoffees);`

JDBC – Steps – 4 RETRIEVING VALUES

JDBC returns results in a ResultSet object.
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
In order to access the names and prices, we will go to each row and retrieve the values according to their types. The method next moves what is called a cursor to the next row and makes that row (called the current row) the one upon which we can operate. Since the cursor is initially positioned just above the first row of a ResultSet object, the first call to the method next moves the cursor to the first row and makes it the current row. Successive invocations of the method next move the cursor down one row at a time from top to bottom. Note that with the JDBC 2.0 API, you can move the cursor backwards, to specific positions, and to positions relative to the current row in addition to moving the curs or forward.

```
String query = "SELECT COF_NAME, PRICE FROM COFFEES"; ResultSet rs =
stmt.executeQuery(query);
while (rs.next()) {
        String s = rs.getString("COF_NAME");
        float n = rs.getFloat("PRICE");
        System.out.println(s + " " + n);
}
```

JDBC – Data Types

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x						
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x						
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x						
getBoolean	x	x	x	x	x	x	x	x	x	X	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x
getBytes														X	X	x			
getDate											x	x	x				X		x
getTime											x	x	x					X	x
getTimestamp											x	x	x				x	x	X
getAsciiStream											x	x	X	x	x	x			
getUnicodeStream											x	x	X	x	x	x			
getBinaryStream														x	x	X			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

JDBC – Prepared statements

If you want to execute a Statement object many times, it will normally reduce execution time to use a PreparedStatement object instead.

The main feature of a PreparedStatement object is that, unlike a Statement object, it is given an SQL statement when it is created.

The advantage to this is that in most cases, this SQL statement will be sent to the DBMS right away, where it will be compiled. As a result, the PreparedStatement object contains not just an SQL statement, but an SQL statement that has been precompiled.
This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement 's SQL statement without having to compile it first.

```
PreparedStatement updateSales = con.prepareStatement( "UPDATE
COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 75);
```

JDBC – Callable statements

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task. Stored procedures are used to encapsulate a set of operations or queries to execute on a database server. For example, operations on an employee database (hire, fire, promote, lookup) could be coded as stored procedures executed by application code. Stored procedures can be compiled and executed with different parameters and results, and they may have any combination of input, output, and input/output parameters.
Stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities.

If you want to call stored procedures, you must use a CallableStatement (subclass of PreparedStatement).

WARNING: stored procedures move the business logic WITHIN THE DB!

JDBC – Callable statements

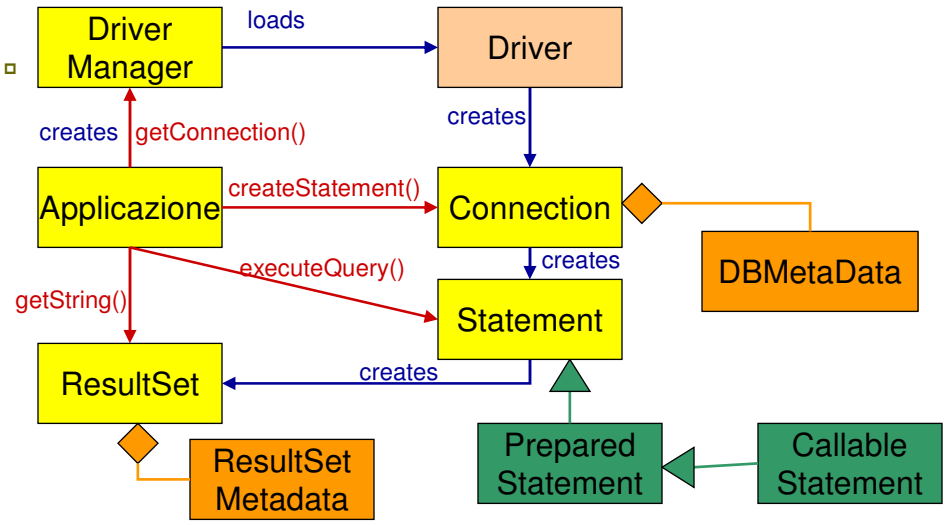
The following SQL statement creates a stored procedure:
create procedure SHOW_SUPPLIERS as select SUPPLIERS.SUP_NAME, COFFEES.COF_NAME from SUPPLIERS, COFFEES where SUPPLIERS.SUP_ID = COFFEES.SUP_ID order by SUP_NAME

Let us assume that the above has been put into a String called createProcedure

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

CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

The java.sql Object Model



The JDBC object model: the MetaData.

The Meta-Data Interfaces

- **Meta data is data about data. Java gives meta-data interfaces: java.sql.ResultSetMetaData e java.sql.DatabaseMetaData.**

ResultSetMetaData interface allows getting info about a ResultSet. For instance, ResultSetMetaData gives info on the number of columns and on the types.

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
ResultSetMetaData rsmd = rs.getMetaData();
int numberOfColumns = rsmd.getColumnCount();
```

see http://java.sun.com/j2se/1.4.2/docs/api/java/sql/ResultSetMetaData.html

The JDBC object model: the MetaData.

DatabaseMetaData interface allows getting info about a the Database.

- **For instance its getCatalogs() method retrieves the catalog names available in this database while getDatabaseProductName() retrieves the name of this database product.**

A user for this interface is commonly a tool that needs to discover how to deal with the underlying DBMS. LIMITED SUPPORT BY TOYS LIKE MS Access!

The class DatabaseMetaData is used by IDEs like JBuilder.

Simple Database Access Using the JDBC Interfaces

- **Scrivere una applicazione di Database usando solo chiamate JDBC comporta i seguenti passi:**

1.Chiedi al DriverManager una implementazione di Connection.
2.Chiedi alla Connection uno Statement o una sottoclasse Statement per eseguire il tuo SQL.
3.Per le sottoclassi di Statement, lega i parametri da passare alla prepared statement.
4.Esegui lo statement.
5.Per le queries, processa il result set ritornato dalla query. Ripetilo per tutti i result set finche' ce ne sono.
6.Per gli altri other statements, leggi il numero di righe toccate.
7.Chiudi lo statement.
8.Processa cosi' tutti gli statement che servono e poi chiudi la connessione.

Esempio

```
package first;
import java.lang.*;
import java.util.*;
import java.sql.*;
import sun.jdbc.odbc.*;
import java.io.*;

public class first
{

   public static void main(String arg[]) {
   int id;
   float amount;
   java.sql.Date dt;
   String companyName;
   String result;
   String item_desc;

     try {
//connect to ODBC database
        Class.forName(
           sun.jdbc.odbc.JdbcOdbcDriver");
        String url = "jdbc:odbc:cityzoo";
        // connect
        Properties p = new Properties();
        p.put("user", "");
        p.put("password","");
        Connection con =
        DriverManager.getConnection(url,p);
```

```
// create Statement object
    Statement stmt = con.createStatement();
    String sqlselect =
       "Select item_nbr, wholesale_cost, "
     + " item_desc, company_name"
     +" from retail_item,company"
     + " where wholesale_cost<9 and"
     + " company.company_id=retail_item.company_id"
     + " order by wholesale_cost";
// run query
    ResultSet rs = stmt.executeQuery(sqlselect);
// process results
    while(rs.next()) {
      result  = "";
      id      = rs.getInt(1);
      amount  = rs.getFloat(2);
      //dt      = rs.getDate(2);
      item_desc   = rs.getString(3);
      companyName   = rs.getString(4);
      result = "#"+result.valueOf(id) + " $";
      result+= result.valueOf(amount) + " <";
      result+= item_desc+"> <"+companyName+">";
      System.out.println("Values are: " + result);
    }
//close connection
    con.close();
  }
  catch(Exception e) {
   System.out.println(e.getMessage());
  }
  try {
   Thread.sleep(20*1000);
  } catch (Exception e) {}
 }
}
```

JDBC 2.0

package javax.sql

See http://java.sun.com/docs/books/tutorial/jdbc/jdbc2dot0/index.html

New extensions

- **Moving the Cursor in Scrollable Result Sets**
- **Making Updates to Updatable Result Sets**
- **Support for SQL3 Data types**

JDBC – SQL3 Data Types

SQL3 type	getXXX method	setXXX method	updateXXX method
BLOB	`getBlob`	`setBlob`	updateBlob
CLOB	`getClob`	`setClob`	updateClob
ARRAY	`getArray`	`setArray`	updateArray
Structured type	`getObject`	`setObject`	updateObject
REF (structured type)	`getRef`	`setRef`	updateRef

Transactions

Introduction

1. getConnection/setConnection

Bank

```
package transactions_1;
import java.sql.*;
public class Bank {

 public Connection getConnection(String jdbcDriverName,
                                 String jdbcURL) {
  try {
    Class.forName(jdbcDriverName);
    return DriverManager.getConnection(jdbcURL);
  } catch (ClassNotFoundException ex) { ex.printStackTrace();
  } catch (SQLException ex) { ex.printStackTrace(); }
  return null;
 }

 public void releaseConnection(Connection conn) {
   if (conn!=null)
     try {
       conn.close();
     } catch (SQLException ex) { ex.printStackTrace(); }
 }
```

2. deposit/withdraw

Bank

```
public void deposit(int account, double amount, Connection conn)
    throws SQLException{
  String sql="UPDATE Account SET Balance = Balance + "+ amount+
          "WHERE AccountId = "+account;
  Statement stmt=conn.createStatement();
  stmt.executeQuery(sql);
  System.out.println("Deposited "+amount+" to account "+account);
 }

 public void withdraw(int account, double amount, Connection conn)
    throws SQLException{
  String sql="UPDATE Account SET Balance = Balance - "+ amount+
          "WHERE AccountId = "+account;
  Statement stmt=conn.createStatement();
  stmt.executeQuery(sql);
  System.out.println("Withdrew "+amount+" from account "+
          account);
 }
```

3. printBalance

Bank

```
public void printBalance(Connection conn) {
   ResultSet rs=null;
   Statement stmt=null;
   try {
     stmt=conn.createStatement();
     rs=stmt.executeQuery("SELECT * FROM Account");
     while (rs.next())
       System.out.println("Account "+rs.getInt(1)+
                          " has a balnce of "+rs.getDouble(2));
   } catch (SQLException ex) { ex.printStackTrace(); }
   finally {
     try {
       if (rs!=null)
         rs.close();
       if (stmt!=null)
         stmt.close();
     } catch (SQLException ex) { ex.printStackTrace(); }
   }
 }
```

4. trasferFunds

Bank

```
public void transferFunds(int fromAccount, int toAccount,
                          double amount, Connection conn){
   Statement stmt=null;
   try {
     withdraw(fromAccount, amount, conn);
     deposit(toAccount,amount,conn);
   }
   catch (SQLException ex) {
     System.out.println("An error occured!");
     ex.printStackTrace();
   }
 }
```

Bank

5. main

```
public static void main(String[] args) {
  if (args.length <3) {
    System.exit(1);
  }
  Connection conn=null;
  Bank bank = new Bank();
  try {
    conn=bank.getConnection(args[0],args[1]);
    bank.transferFunds(1,2,Double.parseDouble(args[2]),conn);
    bank.printBalance(conn);
  } catch (NumberFormatException ex) { ex.printStackTrace();
  } finally {bank.releaseConnection(conn);}
 }
}
```

Using transactions

- **When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.**

con.setAutoCommit(false);

Committing a Transaction

Once auto-commit mode is disabled, no SQL statements will be committed until you call the method commit explicitly. All statements executed after the previous call to the method commit will be included in the current transaction and will be committed (or rolled back) together as a unit.

Using transactions

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE
?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con.prepareStatement(
    "UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE
COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit(); // con.rollback();
con.setAutoCommit(true);
```

transferFunds – fixed version!

Bank

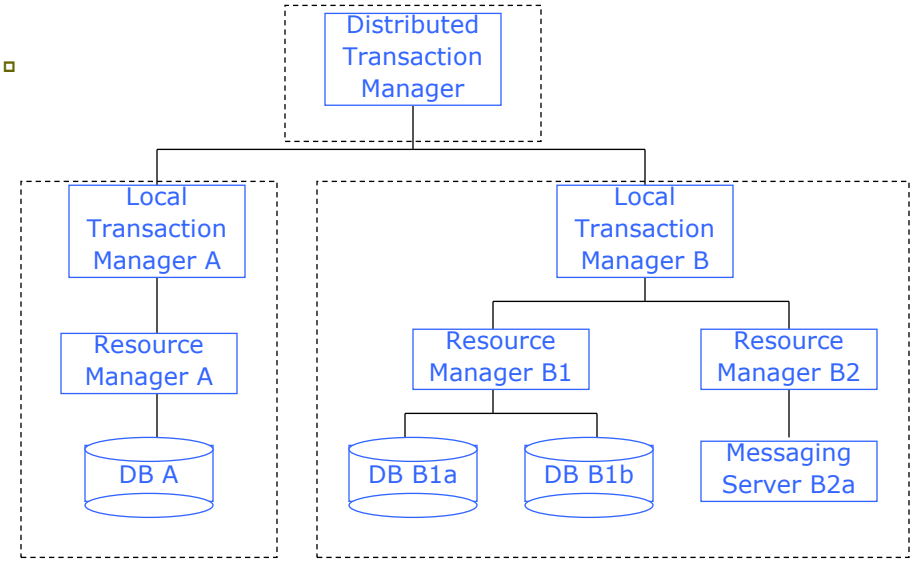
```
public void transferFunds(int fromAccount, int toAccount,
                          double amount, Connection conn){
  Statement stmt=null;
  try {
    conn.setAutoCommit(false);
    withdraw(fromAccount, amount, conn);
    deposit(toAccount,amount,conn);
    conn.commit();
  }
  catch (SQLException ex) {
    System.out.println("An error occured!");
    ex.printStackTrace();
    try {
      conn.rollback();
    } catch (SQLException e) { e.printStackTrace(); }
  }
}
```

TRANSACTION

Actors

- A ***transactional object*** (or ***transactional component***) is an application component that is involved in a transaction.
- A ***transaction manager*** is responsible for managing the transactional operations of the transactional components.
- A ***resource*** is a persistent storage from which you read or write.
- A ***resource manager*** manages a resource. Resource managers are responsible for managing all state that is permanent.
- The most popular interface for resource managers is the ***X/Open XA*** resource manager interface (a de facto standard): a deployment with heterogeneous resource managers from different vendors can interoperate.

Distributed Systems



Who begins a transaction?

Who begins a transaction? Who issues either a commit or abort? This is called demarcating transactional boundaries .

There are three ways to demarcate transactions:

- programmatically:
  you are responsible for issuing a *begin* statement and either a *commit* or an *abort* statement.
- declaratively,
  the EJB container *intercepts* the request and starts up a transaction automatically on behalf of your bean.
- client-initiated.
  write code to start and end the transaction from the client code outside of your bean.

Programmatic vs. declarative

programmatic transactions:

- ***your bean has full control over transactional boundaries.****For instance,you can use programmatic transactions to run a series of minitransactions within a bean method.*
  When using programmatic transactions,always try to complete your transactions in the same method that you began them.Doing otherwise results in spaghetti code where it is difficult to track the transactions;the performance decreases because the transaction is held open longer.

declarative transactions:

your entire bean method must either run under a transaction or not run under a transaction*.*
Transactions are simpler! (just declare them in the descriptor)

Client-initiated

Client initiated transactions:

- *A nontransactional remote client calls an enterprise bean that performs its own transactions The bean succeeds in the transaction,but the network or application server crashes before the result is returned to a remote client.The remote client would receive a Java RMI RemoteException indicating a network error,but would not know whether the transaction that took place in the enterprise bean was a success or a failure.*

With client-controlled transactions, if anything goes wrong,the client will know about it.

The downside to client-controlled transactions is that if the client is located far from the server, ***transactions are likely to take a longer time and the efficiency will suffer****.*

Transactions

ACID

The ACID Properties

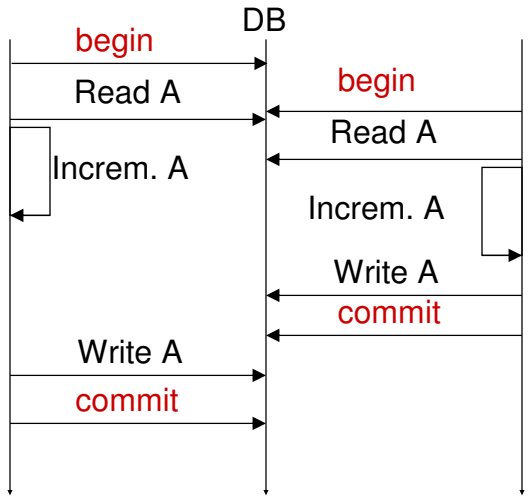
Atomicity *guarantees that many operations are bundled together and appear as one contiguous unit of work .*

Consistency *guarantees that a transaction leaves the system 's state to be consistent after a transaction completes.*

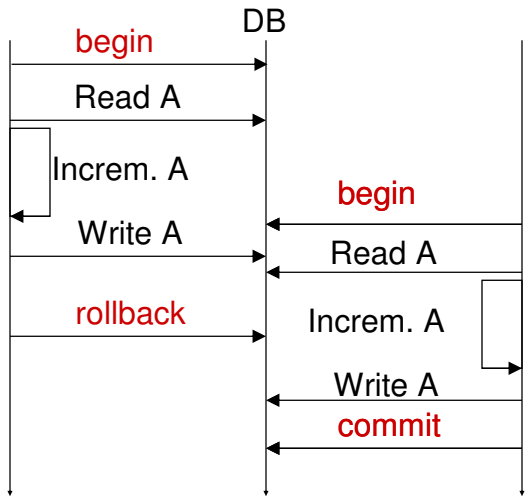
Isolation *protects concurrently executing transactions from seeing eachother 's incomplete results.*

Durability *guarantees that updates to managed resources,such as database records,survive failures. (Recoverable resources keep a transactional log for exactly this purpose.If the resource crashes,the permanent data can be reconstructed by reapplying the steps in the log.)*

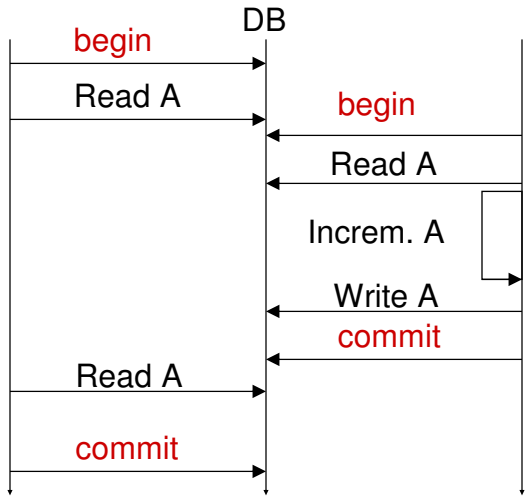
Lost Update



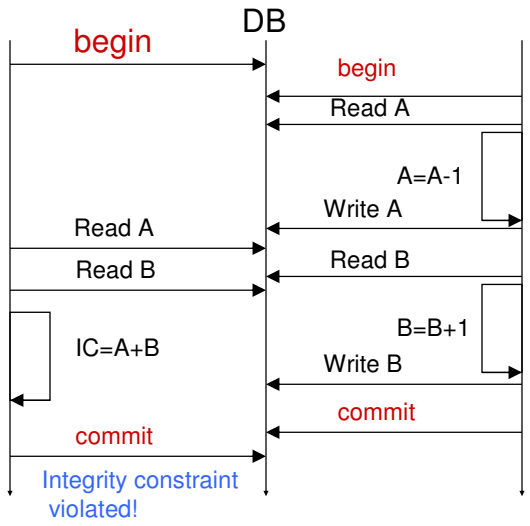
Dirty Read



Unrepeatable Read



Phantom Read (ghost update)



Integrity Constraint: A+B=100

Isolation levels

ISOLATION LEVEL	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	NO	SI	SI
REPEATABLE READ	NO	NO	SI
SERIALIZABLE	NO	NO	NO

Default level for many DBMS

24