PROGRAMMING III
OOP. JAVA LANGUAGE

COURSE 6

PREVIOUS COURSE CONTENT

- **Inheritance**
  - Abstract classes
  - Interfaces
  - instanceof operator
- **Nested classes**
- **Enumerations**

COUSE CONTENT

- Exceptions

- Database access

ERRORS

- What are errors?

ERRORS

- **What are errors?**

  - The state or condition of being wrong in conduct or judgement

  - A measure of the estimated difference between the observed or calculated value of a quantity and its true value

ERRORS

- **Errors Types**
  - Syntax errors
    - arise because the rules of the language have not been followed. They are detected by the compiler.
  - Runtime errors
    - occur while the program is running if the environment detects an operation that is impossible to carry out.
  - Logic errors
    - occur when a program doesn't perform the way it was intended to.

EXCEPTIONS

- **What is an exceprion**
  - A situation leading to an imposibility of finishing an operation
- **How to handle an exception**
  - Provide mechanism that allows communication between the method that is detcting an exceptional condition, while is performing an operation, and the fuctions/objects/modules that are clients of that method and wish to handle dinamicaly the situation
  - Exception handeling systems
    - allows user to signal exceptions and associate handlers (set system into a coherent state) to entities

JAVA EXCEPTIONS

- **Java exception**
  - Is an object that describes an error condition occurred in the code
- **What happens when a exception occures**
  - An object representing that exception is created and thrown in the method that caused the exception.
  - That method may choose to handle the exception itself, or pass it on.
  - Exceptions break the normal flow of control. When an exception occurs, the statement that would normally execute next is not executed.
- **At some point, the exception should be caught and processed.**

THROWING EXCETIONS

- Use the throw statement to *throw* an exception object

- Example

```
public class BankAccount {
    public void withdraw(double amout) {
        if (amount > balance)  {
            IllegalArgumentException ex
            = new IllegalArgumentException (
                 Amount exceeds balance”);
            throw ex;
        } balance = balance – amount;
    }
}
```

THROWING EXCETIONS

- **When an exception is thrown, the current method terminates immediately.**
- **Throw exceptions only in exceptional cases.**
- **Do not abuse of exception throwing**
  - Use exception just to exit a deeply nested loop or a set of recursive method calls.

TREATING EXCEPTIONS

- Every exception should be handled

- If an exception has no handler, an error message is printed, and the program terminates.

- A method that is ready to handle a particular exception type, contains the statements that can cause the exception inside a try block, and the handler inside a catch clause

TREATING EXECEPTIONS

- Example

```
try {
 System.out.println(“What is your name?”);
 String name = console.readLine();
 System.out.println(“Hello. “ + name + “!”);

} catch(IOException ex){
 ex.printStackTrace(); // should handle exception
 System.exit(1);
}
```

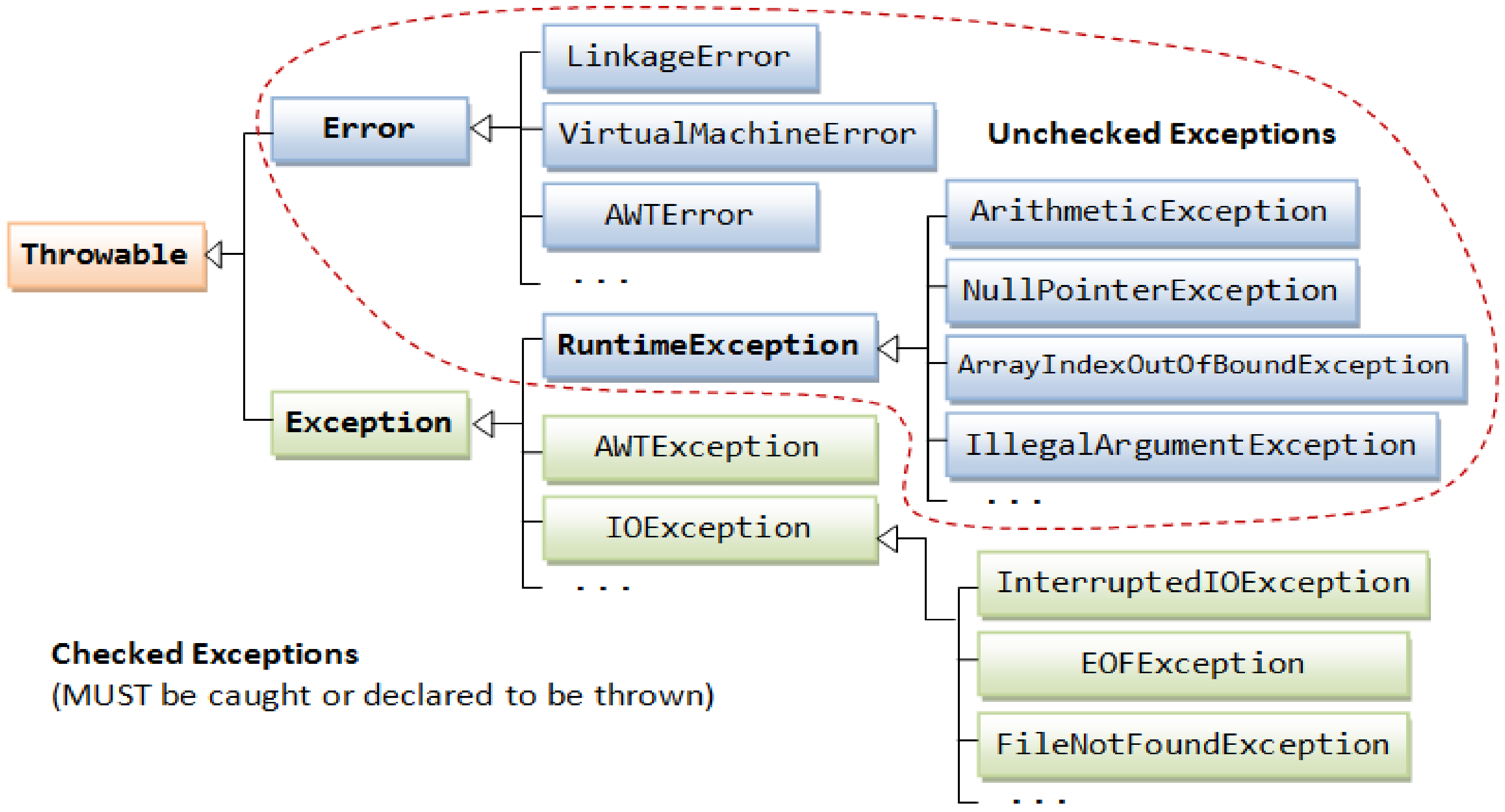
EXCEPTIONS FLOW

- **What happens instead depends on:**
  - whether the exception is caught,
  - where it is caught,
  - what statements are executed in the ‘catch block’,
  - and whether you have a ‘finally block’

EXCEPTIONS HIERACHY

- **Java organizes exceptions in inheritance tree:**
    - Throwable
        - superclass for all exceptions
    - Error
        - are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle
    - Exception
        - RuntimeException
        - TooManyListenersException
        - IOException
        - AWTException
- **OBS**
    - The code you write should throw only exceptions, not errors.
    - Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

EXCEPTIONS HIERACHY



Checked Exceptions
(MUST be caught or declared to be thrown)

EXCEPTIONS HIERACHY

- **Exceptions Type**
    - Unchecked exceptions
        - Error and RuntimeException
        - Are not checked by the compiler, and hence, need not be caught or declared to be thrown in your program

    - Checked exceptions
        - They are checked by the compiler and must be caught or declared to be thrown

CATCHING AN EXCEPTION

❑**Synatax**

```
try {
         // statement that could throw an exception
} catch (<exception type> e) {
        // statements that handle the exception
} catch (<exception type> e) { //e higher in hierarchy
        // statements that handle the exception
} finally {
        // release resources
}
```

- ❑ **At most one catch block executes**
- ❑ **finally block always executes once, whether there's an error or not**

CATCHING AN EXCEPTION

- **When an exception occurs, the nested try/catch statements are searched for a catch parameter matching the exception class**
- **A parameter is said to match the exception if it:**
  - is the same class as the exception; or
  - is a superclass of the exception; or
  - if the parameter is an interface, the exception class implements the interface.
- **The first try/catch statement that has a parameter that matches the exception has its catch statement executed.**
- **After the catch statement executes, execution resumes with the finally statement, then the statements after the try/catch statement.**

CATCHING AN EXCEPTION

- **Catching More Than One Type of Exception with One Exception Handler**
  - from Java 1.7
  - single catch block can handle more than one type of exception
  - separate each exception type with a vertical bar (|)
  - Usefull
    - same behaviour for multiple catch
  - Example

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

THROWING EXCEPTIONS

- **Syntax**
  - from method body
    - throw new Exceprion()
  - method prototype
    - throws Exception1, Exception2, ..., ExceptionN

- **If a method body throws an exception and is not threated in the body the thrown exception has to be added at method prototype**
- **Example**

```
public void foo(int i) throws IOException, RuntimeException {
    if ( i == 1) throw new IOException();
    if ( i == 2) throw new RuntimeException();
    System.out.println(“No exeception is thrown”);
}
```

TRY-WITH-RESOURCES STATEMENT

- **try statement that declares one or more resources**
- **A resource is an object that must be closed after the program is finished with it.**
  - Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable
- **Syntax**

```
try (/*Resourse declaration and initialization*/){
   //resource utilization
} catch(Exception e) { .. }
```

TRY-WITH-RESOURCES STATEMENT

- **Example**
  - before java 1.7

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws
IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

  - java 1.7

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

CUSTOM EXCEPTION CLASS

- For example if we want to withdraw mony from an accout

```
public class BankAccount {
    public void withdraw(double amout) {
            if (amount > balance) {
                    IllegalArgumentException ex
                    = new IllegalArgumentException (
                            Amount exceeds balance”);
                    throw ex;
            } balance = balance – amount;
    }
}
```

- What if we would like to throw a more speific error for the application?

CUSTOM EXCEPTION CLASS

- **How define a custom exception class**
  - class that extends Exception
  - add constructors
    - default
    - one parameter: the error message
    - two parameteres: the error message, an another Exception
  - add other elemts that help to explain better the exeception

- **Example**
  - public class MyException extends Exception{
  -     public MyException(){super();}
  -     public MyException(String msg){super(msg);}
  -     public MyException(String msg, Exception e){super(msg,e);}
  - }

CUSTOM EXCEPTION CLASS

- **When to create custom exception classes?**
  - Use exeption classes offered by API whenever possible
  - Write your excepion class if
    - You need an exception type that is not represented by those in Java platform
    - It helps users if they could differentiate yourexceptions from those thrown by classes written by other vendors
    - You want to pass more than just a string to the exception handler

INFORMATION ABOUT THROWN EXCEPTIONS

- **getMessage()**
  - Returns the detail message string of this throwable.
- **printStackTrace()**
  - Prints this throwable and its backtrace to the standard error stream.
- **printStackTrace(PrintStream s)**
  - Prints this throwable and its backtrace to the specified print stream.
- **printStackTrace(PrintWriter s)**
  - Prints this throwable and its backtrace to the specified print writer.

COUSE CONTENT

- **Exceptions**

- <u>**Database access**</u>

JDBC

- **JDBC - Java Data Base Conectivity**

- **Standard Java API for database-independent connectivity between the Java programming language, and a wide range of relational databases**

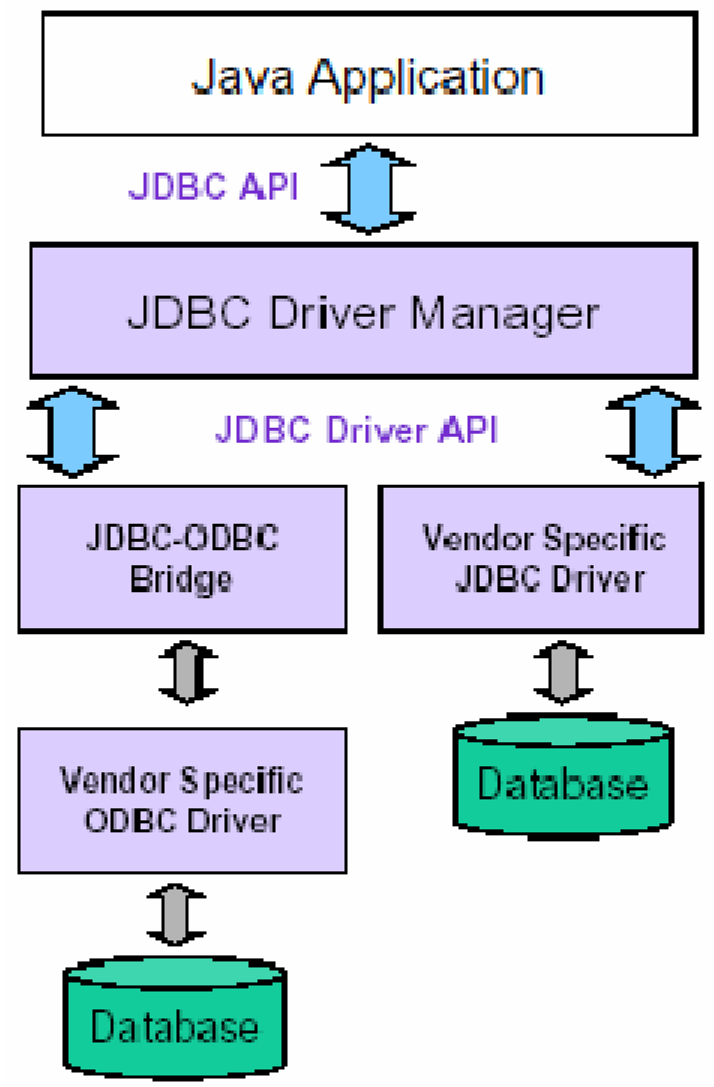
- **java.sql package**

- **Versions**
  - from Java 1.1
  - Java 1.4 & 1.5 - JDBC 3
  - Java 1.6 - JDBC 4

JDBC

- ❑ Database access is the same for all database vendors

- ❑ The JVM uses a JDBC driver to translate generalized JDBC calls into vendor specific database calls



JDBC ADVANTAGES

- **Simplified**
  - Easy to install and maintain
  - No supplimentary configuration files

- **Nonetwork configurations**
  - No configuration is required
  - Requires a suitable driver to connect

- **Full access to medatada**
  - inclyde API to obtain metadata about database and tables

- **No installation**

DRIVERS EXAMPLES

- **Oracle**
  - oracle.jdbc.driver.OracleDriver
- **MySQL**
  - com.mysql.jdbc.Driver
- **Sybase**
  - com.sybase.jdbc.SybDriver
- **SQL Server**
  - com.microsoft.jdbc.sqlserer.SQLServerDriver
- **DB2**
  - com.ibm.db2.jdbc.net.DB2Driver

BASIC STEPS TO USE A DATABASE IN JAVA

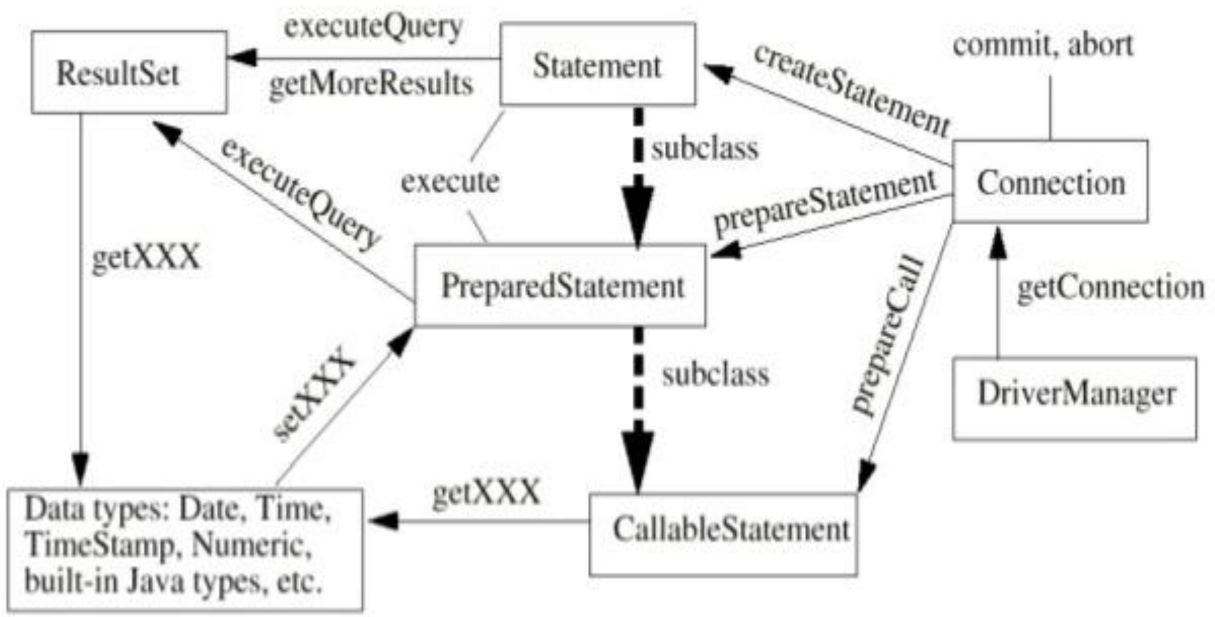
1.Establish a connection

2.Create JDBC Statements

3.Execute SQL Statements

[4.GET ResultSet]

5.Close connections



ESTABLISH A CONNECTION

- **Driver Manager**

  - The purpose of the java.sql.DriverManger class in JDBC is to provide a common access layer on top of different database drivers used in an application

  - DriverManager requires that each driver required by the application must be registered before use, so that the DriverManager is aware of it

ESTABLISH A CONNECTION

- **Driver Manager**

  - Load the database driver using ClassLoader
    - Before Java 1.7
      - Class.forName (“oracle.jdbc.driver.OracleDriver”);

    - From Java 1.7
      - Driver is load automaticaly when the jar is add into classpath

ESTABLISH A CONNECTION

- **Connection creation**
  - Connection connection = DriverManager.getConnection (“jdbc:mysql://localhost/databasename”, uid, passwd);
- **Every database is identified by a URL**
  - jdbc:pointbase: //host.domain.com: 9092 /data/file
    - DB protocol
    - Machine holding the DB
    - Database Port
    - The path to the database on the machine
- **Given a URL, DriverManager looks for the driver that can talk to the corresponding database**
- **DriverManager tries all registered drivers, until a suitable one is found**

CREATE JDBC STATEMENTS

- **There are 3 different types of statements that are supported**
  - Statement
    - A basic SQL statement
  - PreparedStatement
    - A precompiled SQL statement
  - CallableStatement
    - Access to stored procedures

- **Just like a connection, we should close the statement when we are done with it**

CREATE JDBC STATEMENTS

- **Query operation**
  - Statement stmt = null;
  - String query = " SELECT * FROM CITY WHERE country='”+country+”'”;
  - stmt = connection.createStatement();
  - ResultSet rs = stmt.executeQuery(query);

- **insert/update/delete/create/alter/drop**
  - Statement stmt = connection.createStatement();
  - String sql = "UPDATE CITY SET population='”+ population +”' WHERE NAME='”+ cityName +”' AND PROVINCE=‘”+ province +”‘";
  - stmt.executeUpdate(sql);

RESULTSET

- **ResultSet objects provide access to the tables generated as results of executing a Statement queries**
- **Only one ResultSet per Statement can be open at the same time!**
- **The table rows are retrieved in sequence**
  - A ResultSet maintains a cursor pointing to its current row
  - The next() method moves the cursor to the next row

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 getType(int columnIndex)**
  - returns the given field as the given type
  - indices start at 1 and not 0!
- **Type getType(String columnName)**
  - same, but uses name of field
  - less efficient
- **Example:**
  - getString(columnIndex), getInt(columnName), getTime, getBoolean, getType,...
- **int findColumn(String columnName)**
  - looks up column index given column name

RESULTSET

- **JDBC 2.0 includes scrollable result sets. Additional methods included are : ‘first’, ‘last’, ‘previous’, and other methods.**
- **Example**

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.
executeQuery("select lname,salary from Employees");
// Print the result
while(rs.next()) {
    System.out.print(rs.getString(1) + ":");
    System.out.println(rs.getDouble(“salary"));
}
```

PREPARED STATEMENTS

- Prepared Statements are used for queries that are executed many times
- They are parsed (compiled) by the DBMS only once
- Column values can be set after compilation
- Instead of values, use ‘?’
- Hence, Prepared Statements can be though of as statements that contain placeholders to be substituted later with actual values

PREPARED STATEMENTS

- **Example**

```
String queryStr =
        "SELECT * FROM employee " +
        "WHERE mgr= ? and salary > ?";

PreparedStatement pstmt =    con.prepareStatement(queryStr);

pstmt.setString(1, "Xescu");
pstmt.setInt(2, 26000);

ResultSet rs = pstmt.executeQuery();
```

PREPARED STATEMENTS

- **Will this work?**
  - PreparedStatement pstmt =
    con.prepareStatement("select * from ?");
  - pstmt.setString(1, myFavoriteTableString);
- **No!!! A '?' can only be used to represent a column value**

PREPARED STATEMENTS

- **What is SQL Injection?**

- **Example**

```
Statement stmt = conn.createStatement("INSERT INTO
students VALUES('" + user + "')");
stmt.execute();
```

- What happens if user variable takes te following values
  - “Xescu”
  - “Xescu'); DELETE FROM students;” --

PREPARED STATEMENTS

- **What is SQL Injection?**
  - SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via “page input”.
- **Example**

```
Statement stmt = conn.createStatement("INSERT INTO
students VALUES('" + user + "')");
stmt.execute();
```

  - What happens if user variable takes te following values
    - “Xescu”
    - “Xescu'); DELETE FROM students;” --

PREPARED STATEMENTS

- **What is SQL Injection?**
  - SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via “page input”.
- **Recomandation**
  - use prepared statement
  - use '?' to add user input into a SQL statement

PREPARED STATEMENTS

- **Timeout**
  - Use setQueryTimeOut(int seconds) of Statement to set a timeout for the driver to wait for a statement to be completed
  - If the operation is not completed in the given time, an SQLException is thrown

PREPARED STATEMENTS

how to map sql types to java 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

NULL VALUES

- **In SQL, NULL means the field is empty**
- **Not the same as 0 or ""**
- **In JDBC, you must explicitly ask if the last-read field was null**
  - ResultSet.wasNull(column)
- **For example, getInt(column) will return 0 if the value is either 0 or NULL!**
- **When inserting null values into placeholders of Prepared Statements:**
  - Use the method setNull(index, Types.sqlType) for primitive types (e.g. INTEGER, REAL);
  - You may also use the setType(index, null) for object types (e.g. STRING, DATE).

RESULTSET META-DATA

- **A ResultSetMetaData is an object that can be used to get information about the properties of the columns in a ResultSet object**

- **Example**
  - Display the column names of a resultset

```
ResultSetMetaData rsmd = rs.getMetaData();
int numcols = rsmd.getColumnCount();

for (int i = 1 ; i <= numcols; i++) {
            System.out.print(rsmd.getColumnLabel(i)+" ");
}
```

SQL EXCEPTIONS

- **An SQLException is actually a list of exceptions**
- **Methods**
  - A description of the error - SQLException.getMessage
  - A SQLState code - SQLException.getSQLState
    - These codes and their respective meanings have been standardized by ISO/ANSI and Open Group (X/Open), although some codes have been reserved for database vendors to define for themselves. This String object consists of five alphanumeric characters. Retrieve this code by calling the method SQLException.getSQLState.
  - An error code - SQLException.getErrorCode.
    - This is an integer value identifying the error that caused the SQLException instance to be thrown. Its value and meaning are implementation-specific and might be the actual error code returned by the underlying data source.
  - A cause.
    - A SQLException instance might have a causal relationship, which consists of one or more Throwable objects that caused the SQLException instance to be thrown.
    - To navigate this chain of causes, recursively call the method SQLException.getCause until a null value is returned.
  - A reference to any chained exceptions.
    - If more than one error occurs, the exceptions are referenced through this chain. Retrieve these exceptions by calling the method SQLException.getNextException on the exception that was thrown.

SQL EXCEPTIONS

- **Display all information stored into SQL exception**

```
public static  void dispaySQLExceptions(SQLException ex) {
        while (ex != null) {
                System.out.println("SQL State:" + ex.getSQLState());
                System.out.println("Error Code:" + ex.getErrorCode());
                System.out.println("Message:" + ex.getMessage());
                Throwable t = ex.getCause();
                while (t != null) {
                        System.out.println("Cause:" + t);
                        t = t.getCause();
                }
                ex = ex.getNextException();
        }
    }
```

TRANSACTIONS AND JDBC

- **Transaction: more than one statement that must all succeed (or all fail) together**
  - e.g., updating several tables due to customer purchase
- **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 = cancel all actions**

TRANSACTIONS AND JDBC

- **Transactions are not explicitly opened and closed**
- **The connection has a state called AutoCommit mode**
  - if AutoCommit is true, then every statement is automatically committed
  - if AutoCommit is false, then every statement is added to an ongoing transaction
- **Default: true**
- **If you set AutoCommit to false, you must explicitly commit or rollback the transaction using Connection.commit() and Connection.rollback()**

TRANSACTIONS AND JDBC

- **Example for maaging manaly transactions**

```
 PreparedStatement updateSales = null, updateTotal = null;
 String updateString =  "update " + dbName + ".COFFEES " +   "set SALES = ? where COF_NAME = ?";
 String updateStatement =   "update " + dbName + ".COFFEES " +   "set TOTAL = TOTAL + ? " +   "where COF_NAME = ?";
try {
    con.setAutoCommit(false);
    updateSales = con.prepareStatement(updateString);
    updateTotal = con.prepareStatement(updateStatement);
    updateSales.setInt(1, 2); updateSales.setString(2, “DECAF”); updateSales.executeUpdate();
    updateTotal.setInt(1, 100); updateTotal.setString(2, “DECAF”);  updateTotal.executeUpdate();
    con.commit();
 } catch (SQLException e ) { //print exception
      if (con != null) {
        try {
            System.err.print("Transaction is being rolled back");
            con.rollback();
        } catch(SQLException excep) {  // print exceptiom  }
 }} finally {
    if (updateSales != null) {  updateSales.close();  }
    if (updateTotal != null) {   updateTotal.close();   }
    con.setAutoCommit(true);
 }
```

CLEANING UP AFTER YOURSELF

- **Remember to close the Connections, Statements, Prepared Statements and Result Sets**
  - con.close();
  - stmt.close();
  - pstmt.close();
  - rs.close()