CO324: Network and Web Application Design

Network Programming

Finite Statement Machines

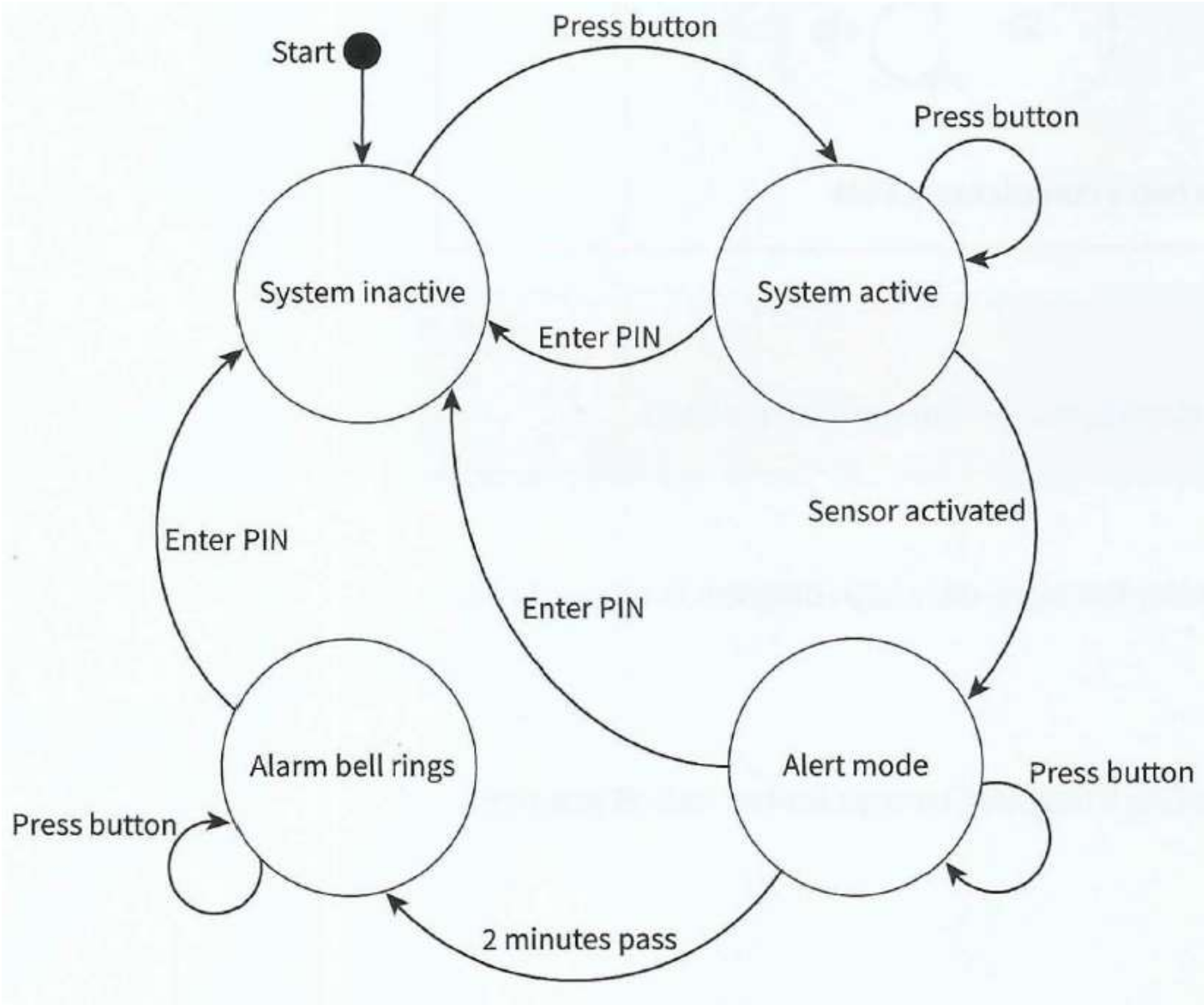
- Finite state machines (FSM) can be used to implement communication protocols.
- Machines are devices that respond to a set of stimuli (events) by generating predictable responses (actions) based on a history of prior events (current state).

Example

Consider the following description of an intruder detection system.

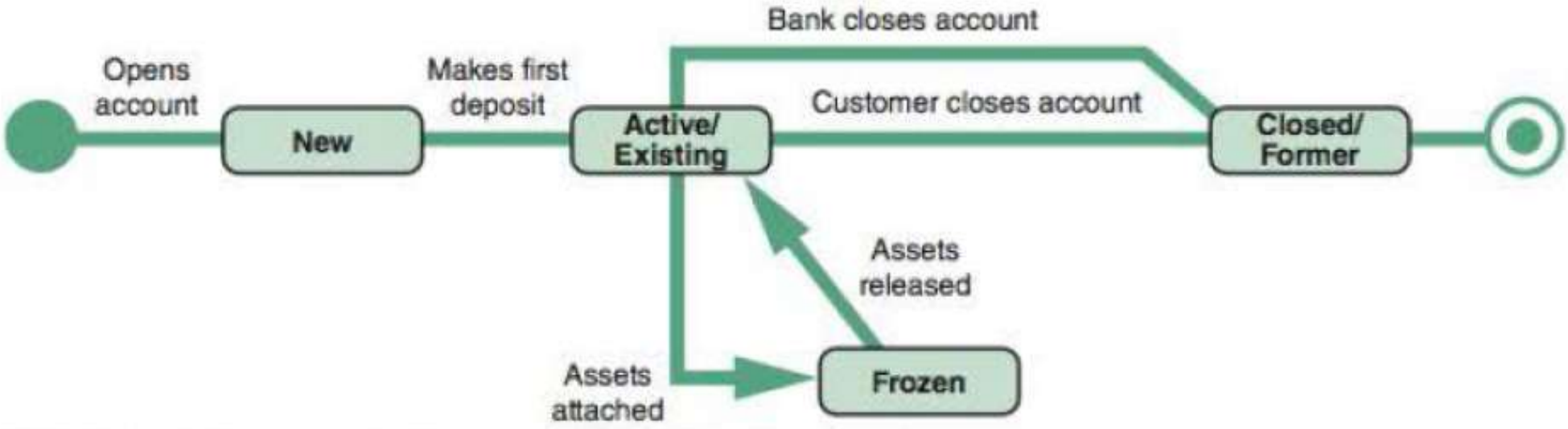
The system has a battery power supply. The system is activated when the start button is pressed. Pressing the start button when the system is active has no effect. To de-activate the system, the operator must enter a PIN. The system goes into alert mode when a sensor is activated. The system will stay in alert mode for two minutes. If the system has not been de-activated within two minutes an alarm bell will ring.

Draw state diagram for the system. State any assumptions made to complete the diagram.


Start
Press button
Press button
System inactive
System active
Enter PIN
Enter PIN
Sensor activated
Enter PIN
Alarm bell rings
Alert mode
Press button
Press button
2 minutes pass

State Transition Diagrams

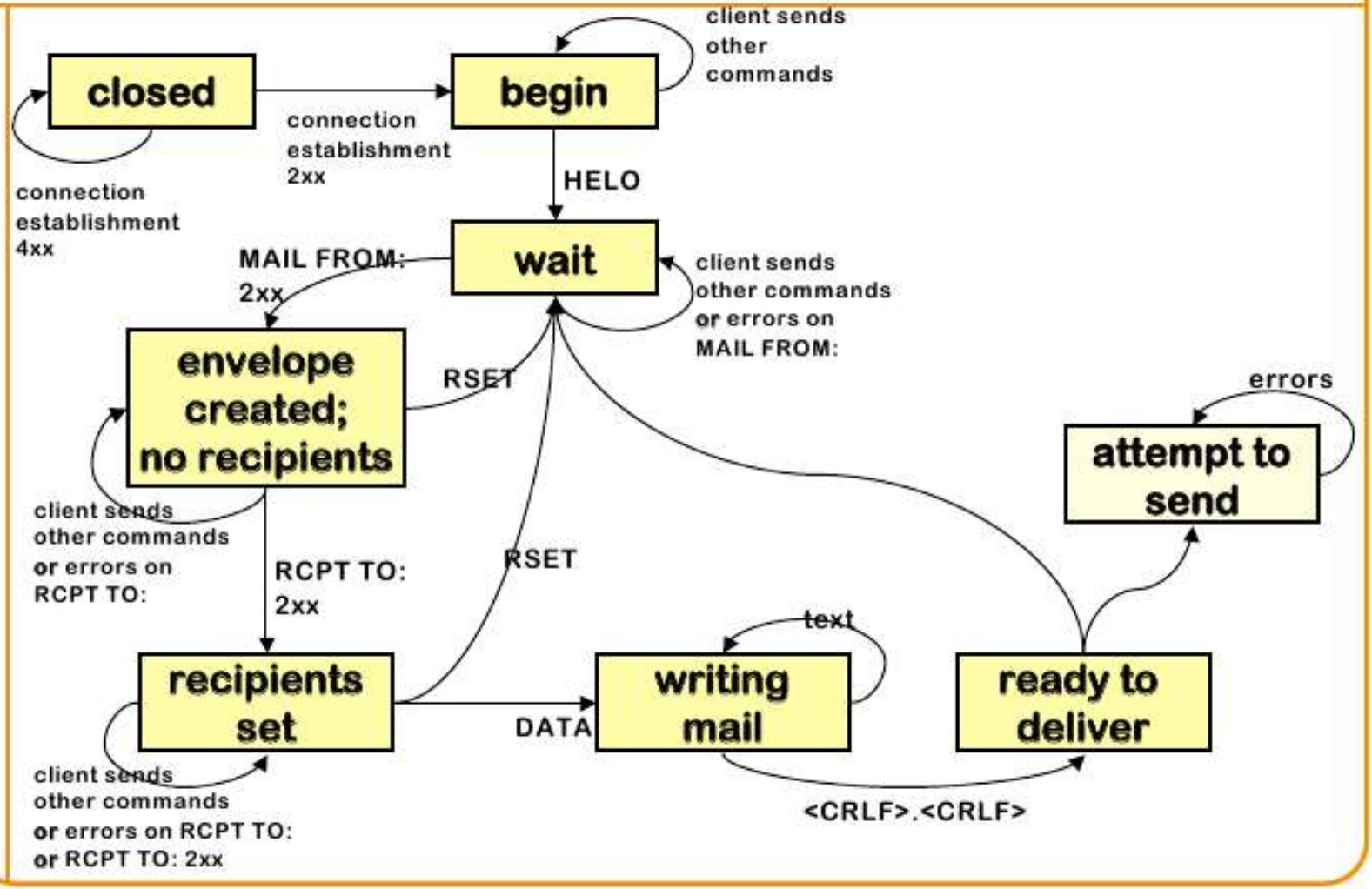
- A state transition diagram shows how an object changes from one state to another, depending on events that affect the object.
- In a state transition diagram, the states appear as rounded rectangles with the state names inside. The small circle to the left is the initial state, or the point where the object first interacts with the system.
- Reading from left to right, the lines show direction and describe the action or event that causes a transition from one state to another. The circle at the right with a hollow border is the final state.


Bank closes account
Opens account
Makes first deposit
Customer closes account
New
Active/ Existing
Closed/ Former
Assets released
Assets attached
Frozen

Protocols as State Machines

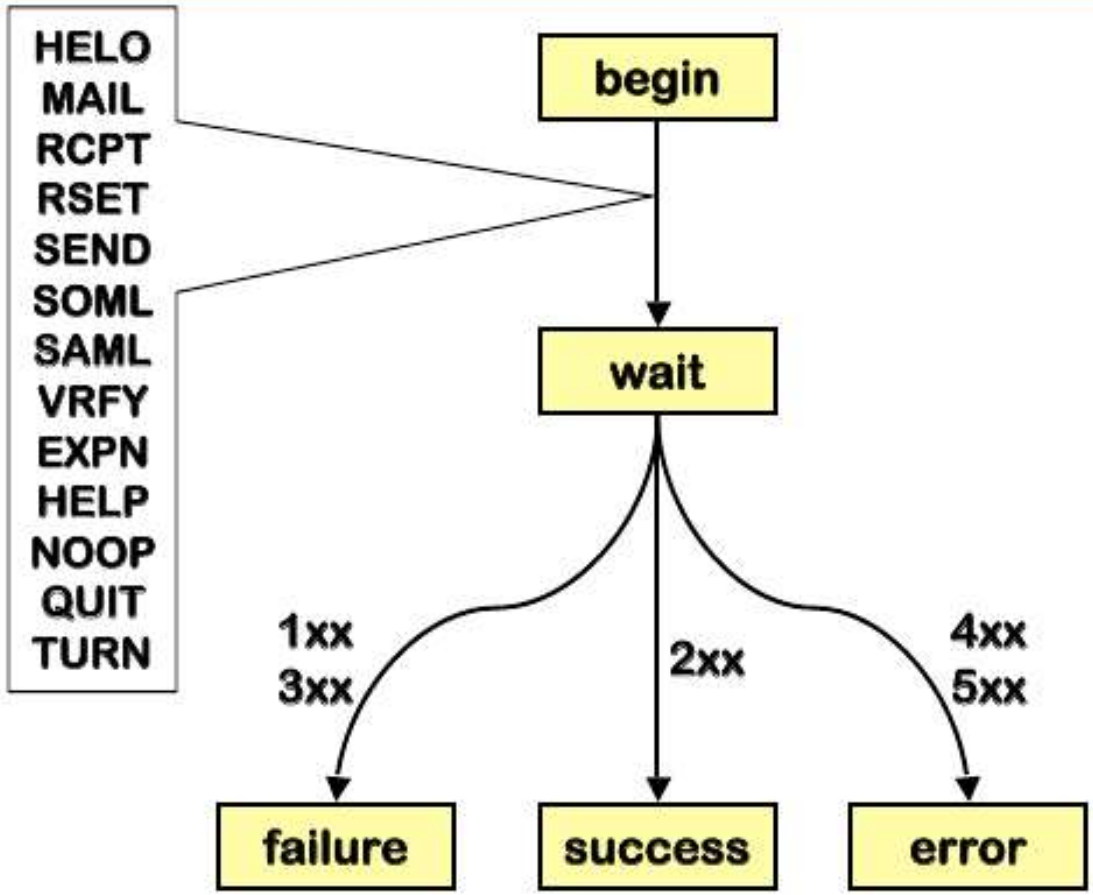
SMTP State Diagram

Server States



SMTP State Diagram

Command States



Deployment Consideration

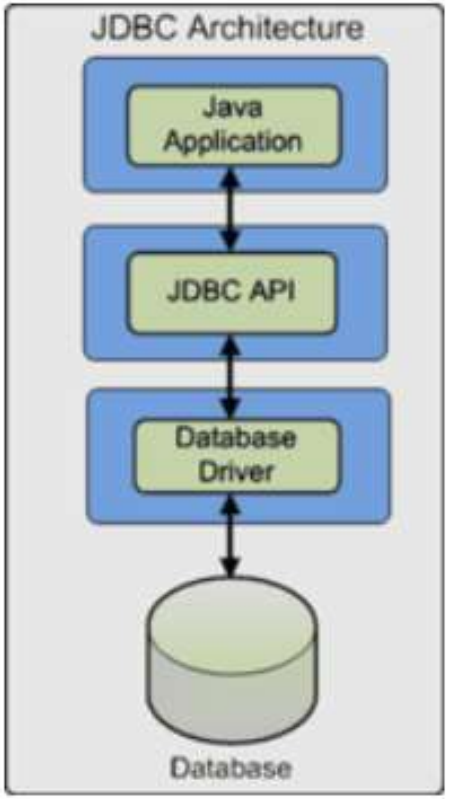
- Database Connectivity
- Cache control headers
- Scaling techniques (reverse proxies, message queues)
- Best practices

Java Database Connectivity Services

- Database management systems are a very important part of any information system.
- Java Database Connectivity or JDBC, is a Java-based data access technology which enables database-independent connectivity between Java applications and a wide range of databases.
- JDBC leverages existing database technologies and allows SQL-based database access.
- The process of accessing a database via this method involves establishing a connection to a database or tabular data source, sending SQL statements and retrieving and processing the results.

JDBC Architecture

- JDBC architecture involves Java Application, JDBC API, Database Driver, and Database System.
- JDBC database interaction process
  - Java Application calls JDBC library methods to connect and interact with the database.
  - JDBC communicates with the Database System via the correct Database Driver (e.g. MySQL database server uses the MySQL specific driver).
  - The results are transferred from the Database System to the Java Application via the driver and JDBC API.



Types of database drivers

- Direct-to-Database Pure Java Driver allows JDBC calls to be converted directly into the protocol used by the database.
  - The advantages of such a driver include good performance and no need to install special software on the client side or server.
  - Disadvantages involve reduced optimization for server operating system and the need for a different driver for each different database.
- Database Middleware Pure Java Driver allows JDBC calls to be converted into a middleware-specific protocol which is then converted into database specific protocol by a middleware server. Middleware server manages multiple database connectivity.
  - This type of driver can be used when multiple databases need to be connected to via the same JDBC driver. It is server-based, so there is no need for a JDBC driver on the client machine. For performance reasons, the server-side component is optimized for the operating system of the server machine.
  - The main disadvantage occurs when the middleware must run on different platforms as database-specific code must run on the middleware server.

Types of database drivers (cont.)

- Native API Partly Java Driver allows JDBC calls to be converted into calls on the client API for specific database systems (e.g., Oracle, Informix). Specific drivers must be loaded on each client machine.
  - Advantages for using this type of driver include flexibility (allows access to almost any database as long as ODBC drivers are available), significantly better performance and limited Java feature set.
  - As a disadvantage, this solution requires specific client library to be installed.
- JDBC-ODBS Bridge allows JDBC to accesses the database via Microsoft's ODBC drivers. ODBC drivers must be loaded on each client machine in order for the database interaction to be performed.
  - Advantages include flexibility, almost any database for which ODBC driver is installed can be accessed via this method.
  - Disadvantages are represented by performance overhead and the need to install an ODBC driver on the client machine which makes it unsuitable for applets.

JDBC Database Access: Steps

- Step 1—Load database driver, by loading the appropriate driver class. The current class instantiates the driver class and register it with the JDBC driver manager.

```
/*Use Class::forName(String classname)*/
// Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// Class.forName("connect.microsoft.MicrosoftDriver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- Step 2—Establish connection with the database. Create connection URL which includes “jdbc”, protocol, host, port number, database name. Host and port number are required for remote access. Connect to the database which uses DriverManager::getConnection(String URL, String username, String password) or DriverManager::getConnection(String URL). Return a reference to an object of type Connection and throw SQLException if a database access error occurs.

```
jdbcodbcURL = "jdbc:odbc:BankDatabase";
Connection link = getConnection(jdbcodbcURL);
```

JDBC Database Access: Steps (cont.)

- Step 3—Create a statement object. Use Connection class's method createStatement() and catch SQLException id thrown.

```
Statement st = link.createStatement();
```

- Step 4—Run the query statement.

```
/*create query*/
String query = "SELECT account, balance FROM
               Accounts";
/*execute query and retrieve the results*/
ResultSet resultSet = st.executeQuery(query);
/*throws SQLException*/
```

JDBC Database Access: Steps (cont.)

- Step 5—Run the update statement.

```
/*Create update statement*/
/*Uses SQL statements INSERT, UPDATE, DELETE*/
String ins =
  "INSERT INTO Accounts (1234, "John Doe",
   "02/03/2007")";
String upd =
  "UPDATE Accounts SET (account=1234,
   name="John Doe")";
String rem =
  "DELETE FROM Accounts WHERE account=1234";

/*Execute update*/
/*Returns an int representing the row count*/
/*Throws SQLException */
int result = st.executeUpdate(ins);
int result = st.executeUpdate(upd);
int result = st.executeUpdate(rem);
```

JDBC Database Access: Steps (cont.)

- Step 6—Process and display the database operation results.

```
/*Process returned data*/
/*Use ResultSet's methods
 *next() - moves cursor down to the next row
 *previous() - moves cursor up to the previous row
 *XXX getXXX(String colName)
 *returns the value of designated
 *column in current row;
 *XXX can be Byte, String, Int, Date, etc.
 *XXX getXXX(int colNo)
 *returns the value of designated
 *column in current row;
 *XXX can be Byte, String, Int, Date, etc.
 *Throw SQLException
 */
resultSet.next();
resultSet.getInt("account");
```

JDBC Database Access: Steps (cont.)

- Step 7—Close connection.

```
/*Close Statement object*/
/*Use Statement::close()
 *Throws SQLException*/
st.close();

/*Close Connection
 *Use Connection::close()
 *Throws SQLException*/
link.close();
```

```
import java.sql.*;

/*JDBC application implementation*/
public class JDBCApp {

  /*define connection, statement and result set*/
  static Connection link;
  static Statement statement;
  static ResultSet results;

  /*main method to be invoked
   *when the application starts*/
  public static void main(String[] args)
  {
    System.out.println("Welcome to JDBC program!");
    try {
      /*load the driver*/
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      /*initiate the connection*/
      link = DriverManager.getConnection
        ("jdbc:odbc:BankDatabase","","");
    }
      catch(ClassNotFoundException e) {
      System.out.println("Error: Unable to load
                          driver!");
      System.exit(1);
    }
    catch(SQLException e) {
      System.out.println("Error: Connect to
                          database!");
      System.exit(1);
    }
```

```
try {
  /*create statement*/
  statement = link.createStatement();
  /*execute the statement and retrieve the result*/
  results =
  statement.executeQuery("SELECT * FROM Accounts");
}
catch(SQLException e) {
  System.out.println("Error: Cannot execute
                      query!");
  e.printStackTrace();
  System.exit(1);
}

try {
  /*process the results*/
  while (results.next()) {
    System.out.println();
    System.out.println("Account:"
    + results.getInt(1));
    System.out.println("Holder:  "
    + results.getString(2));
    System.out.println("Balance: "
    + results.getFloat(3));
    System.out.println("Modified: "
    + results.getDate(4));
  }
}
catch(SQLException e) {
  System.out.println("Error: Retrieving data!");
  e.printStackTrace();
  System.exit(1);
}
```

```
  try {
    /*close the statement and connection*/
    statement.close();
    link.close();
  }
  catch(SQLException e) {
    System.out.println("Error: Unable to
                        disconnect!");
    e.printStackTrace();
  }
}
```

JDBC Transactions

- Most databases require protection against errors that may leave the entire data storage in an inconsistent state.
  - For example, when multiple tables are to be updated as a result of an operation, it is important to make sure that either all operations succeed or none of them. Unfortunately, data processing, network and database errors may occur, and a mechanism should be in place to protect the database consistency.
- A transaction consists of a number of statements that are related logically and make sense only if all of them are successfully executed. By default, after each database statement is executed, the changes are automatically committed to the database (auto-commit is on).
- As a consequence, the changes are saved on the disk and are irreversible.

JDBC Transactions (cont.)

- Before a transaction that consists of two or more statements is performed, the auto-commit feature must be turned off (Connection::setAutoCommit(false)).
- This will prevent the changes to be saved to disk directly. When all the statements are executed successfully, the changes should be permanently recorded to the database by using a commit operation (Connection::commit());
- All changes from the last call to commit() method are recorded without being actually saved on the disk. If an error occurs, the database must be returned to the state before transaction started and this is made by using the rollback method (Connection::rollback()).
- When rollback is performed, all changes from the last call to commit() are dropped.

```
connection con =
  DriverManager.getConnection(url, username, passwd);

/*turn auto-commit off*/
con.setAutoCommit(false);

try {
  statement.executeUpdate(sqlStatement_1);
  ...
  statement.executeUpdate(sqlStatement_N);
  /*at this stages all the changes done
   *by running the above statements are recoded
   *but not saved on the disk*/

  /*commit changes as no error occurred*/
  con.commit();
}
catch (Exception e) {
  try{
    /*as an exception was thrown indicating an error*/
    /*and rollback is performed*/
    con.rollback();
  }
  catch (SQLException sqle)    {
    /*report problem rolling back*/
  }
}
finally {
  try{
    /*close connection*/
    con.close();
  }
  catch (SQLException sqle)    {
    /*report problem closing connection*/
  }
}
```